O'Mega:
Optimal Monte-Carlo
Event Generation Amplitudes

Thorsten Ohl[*]

Institut für Theoretische Physik und Astrophysik
Julius-Maximilians-Universität Würzburg
Emil-Hilb-Weg 22, 97074 Würzburg, Germany

Jürgen Reuter[†]

DESY Theory Group, Notkestr. 85, 22603 Hamburg, Germany

Wolfgang Kilian[c,‡]

Theoretische Physik 1
Universität Siegen
Walter-Flex-Str. 3, 57068 Siegen, Germany

with contributions from Christian Speckner[d,§]
as well as Christian Schwinn et al.

unpublished draft, printed 10/05/2023, 10:10

Abstract

. . .

[*]ohl@physik.uni-wuerzburg.de, http://physik.uni-wuerzburg.de/ohl
[†]juergen.reuter@desy.de
[‡]kilian@physik.uni-siegen.de
[§]cnspeckn@googlemail.com

Contents

—1—
Introduction

1.1 Complexity

There are

$$P(n) = \frac{2^n - 2}{2} - n = 2^{n-1} - n - 1 \tag{1.1}$$

independent internal momenta in a n-particle scattering amplitude [1]. This grows much slower than the number

$$F(n) = (2n-5)!! = (2n-5) \cdot (2n-7) \cdot \ldots \cdot 3 \cdot 1 \tag{1.2}$$

of tree Feynman diagrams in vanilla ϕ^3 (see table 1.1). There are no known corresponding expressions for theories with more than one particle type. However, empirical evidence from numerical studies [1, 2] as well as explicit counting results from O'Mega suggest

$$P^*(n) \propto 10^{n/2} \tag{1.3}$$

while he factorial growth of the number of Feynman diagrams remains unchecked, of course.

The number of independent momenta in an amplitude is a better measure for the complexity of the amplitude than the number of Feynman diagrams, since there can be substantial cancellations among the latter. Therefore it should be possible to express the scattering amplitude more compactly than by a sum over Feynman diagrams.

1.2 Ancestors

Some of the ideas that O'Mega is based on can be traced back to HELAS [5]. HELAS builts Feynman amplitudes by recursively forming off-shell 'wave functions' from joining external lines with other external lines or off-shell 'wave functions'.

The program Madgraph [6] automatically generates Feynman diagrams and writes a Fortran program corresponding to their sum. The amplitudes are calculated by calls to HELAS [5]. Madgraph uses one straightforward optimization: no statement is written more than once. Since each statement corresponds to a collection of trees, this optimization is very effective for up to four particles in the final state. However, since the amplitudes are

n	$P(n)$	$F(n)$
4	3	3
5	10	15
6	25	105
7	56	945
8	119	10395
9	246	135135
10	501	2027025
11	1012	34459425
12	2035	654729075
13	4082	13749310575
14	8177	316234143225
15	16368	7905853580625
16	32751	213458046676875

Table 1.1: The number of ϕ^3 Feynman diagrams $F(n)$ and independent poles $P(n)$.

given as a sum of Feynman diagrams, this optimization can, by design, *not* remove the factorial growth and is substantially weaker than the algorithms of [1, 2] and the algorithm of O'Mega for more particles in the final state.

Then ALPHA [1] (see also the slightly modified variant [2]) provided a numerical algorithm for calculating scattering amplitudes and it could be shown empirically, that the calculational costs are rising with a power instead of factorially.

1.3 Architecture

1.3.1 General purpose libraries

Functions that are not specific to O'Mega and could be part of the O'Caml standard library

ThoList : (mostly) simple convenience functions for lists that are missing from the standard library module *List* (section F, p. 682)

Product : effcient tensor products for lists and sets (section N, p. 730)

Combinatorics : combinatorical formulae, sets of subsets, etc. (section Q, p. 740)

1.3.2 O'Mega

The non-trivial algorithms that constitute O'Mega:

DAG : Directed Acyclical Graphs (section 4, p. 29)

Topology : unusual enumerations of unflavored tree diagrams (section 3, p. 16)

Momentum : finite sums of external momenta (section 5, p. 43)

Fusion : off shell wave functions (section 15, p. 195)

Omega : functor constructing an application from a model and a target (section 24, p. 640)

1.3.3 Abstract interfaces

The domains and co-domains of functors (section 16, p. 251)

Coupling : all possible couplings (not comprensive yet)

Model : physical models

Target : target programming languages

1.3.4 Models

(section **??**, p. **??**)

$Modellib_SM.QED$: Quantum Electrodynamics

$Modellib_SM.QCD$: Quantum Chromodynamics (not complete yet)

$Modellib_SM.SM$: Minimal Standard Model (not complete yet)

etc.

1.3.5 Targets

Any programming language that supports arithmetic and a textual representation of programs can be targeted by O'Caml. The implementations translate the abstract expressions derived by *Fusion* to expressions in the target (section 20, p. 427).

$Target_Fortran$: Fortran95 language implementation, calling subroutines

Other targets could come in the future: `C`, `C++`, O'Caml itself, symbolic manipulation languages, etc.

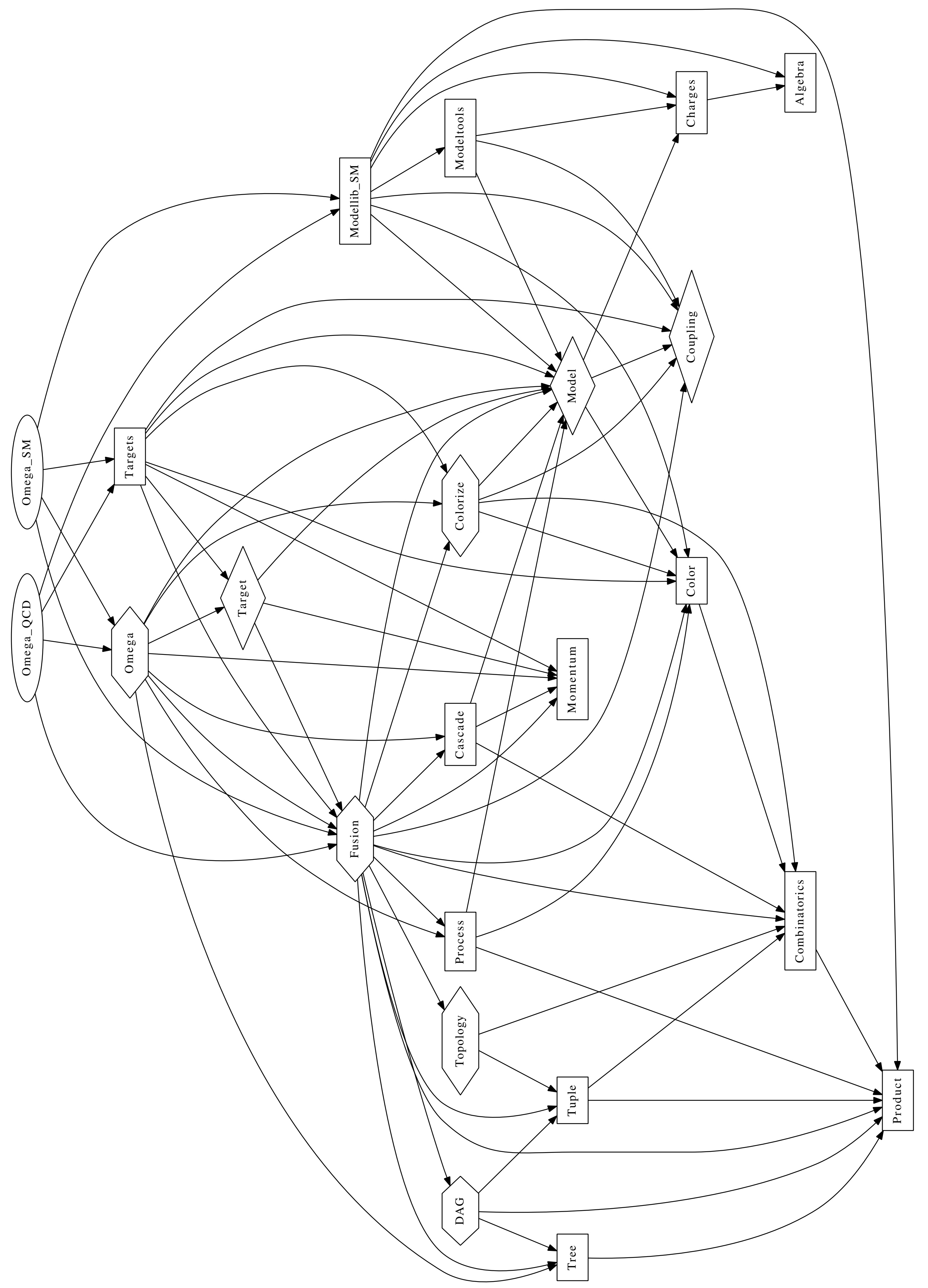


Figure 1.1: Module dependencies in O'Mega.

1.3.6 Applications

(section 24, p. 640)

1.4 The Big To Do Lists

1.4.1 Required

All features required for leading order physics applications are in place.

1.4.2 Useful

1. select allowed helicity combinations for massless fermions
2. Weyl-Van der Waerden spinors
3. speed up helicity sums by using discrete symmetries
4. general triple and quartic vector couplings
5. diagnostics: count corresponding Feynman diagrams more efficiently for more than ten external lines
6. recognize potential cascade decays (τ, b, etc.)
   - warn the user to add additional
   - kill fusions (at runtime), that contribute to a cascade
7. complete standard model in R_ξ-gauge
8. groves (the simple method of cloned generations works)

1.4.3 Future Features

1. investigate if unpolarized squared matrix elements can be calculated faster as traces of densitiy matrices. Unfortunately, the answer apears to be *no* for fermions and *up to a constant factor* for massive vectors. Since the number of fusions in the amplitude grows like $10^{n/2}$, the number of fusions in the squared matrix element grows like 10^n. On the other hand, there are $2^{\#\text{fermions}+\#\text{massless vectors}} \cdot 3^{\#\text{massive vectors}}$ terms in the helicity sum, which grows *slower* than $10^{n/2}$. The constant factor is probably also not favorable. However, there will certainly be asymptotic gains for sums over gauge (and other) multiplets, like color sums.
2. compile Feynman rules from Lagrangians
3. evaluate amplitues in O'Caml by compiling it to three address code for a virtual machine

   ```
   type mem  =  scalar array × spinor array × spinor array × vector array
   type instr  =
         | VSS of int × int × int
         | SVS of int × int × int
         | AVA of int × int × int
         ...
   ```

   this could be as fast as [1] or [2].
4. a virtual machine will be useful for for other target as well, because native code appears to become to large for most compilers for more than ten external particles. Bytecode might even be faster due to improved cache locality.
5. use the virtual machine in O'Giga

1.4.4 Science Fiction

1. numerical and symbolical loop calculations with O'Tera: O'Mega Tool for Evaluating Renormalized Amplitudes

—2— Tuples and Polytuples

2.1 Interface of Tuple

The *Tuple.Poly* interface abstracts the notion of tuples with variable arity. Simple cases are binary polytuples, which are simply pairs and indefinite polytuples, which are nothing but lists. Another example is the union of pairs and triples. The interface is very similar to *List* from the O'Caml standard library, but the *Tuple.Poly* signature allows a more fine grained control of arities. The latter provides typesafe linking of models, targets and topologies.

```
module type Mono =
  sig
    type α t
```

The size of the tuple, i. e. *arity* (*a1*, *a2*, *a3*) = 3.

```
    val arity : α t → int
```

The maximum size of tuples supported by the module. A negative value means that there is no limit. In this case the functions *power* and *power_fold* may raise the exception *No_termination*.

```
    val max_arity : unit → int

    val compare : (α → α → int) → α t → α t → int

    val for_all : (α → bool) → α t → bool

    val map : (α → β) → α t → β t
    val iter : (α → unit) → α t → unit
    val fold_left : (α → β → α) → α → β t → α
    val fold_right : (α → β → β) → α t → β → β
```

We have applications, where no sensible intial value can be defined:

```
    val fold_left_internal : (α → α → α) → α t → α
    val fold_right_internal : (α → α → α) → α t → α

    val map2 : (α → β → γ) → α t → β t → γ t

    val split : (α × β) t → α t × β t
```

The distributive tensor product expands a tuple of lists into list of tuples, e. g. for binary tuples:

$$\mathit{product}\,([x_1;x_2],[y_1;y_2]) = [(x_1,y_1);(x_1,y_2);(x_2,y_1);(x_2,y_2)] \tag{2.1}$$

NB: *product_fold* is usually much more memory efficient than the combination of *product* and *List.fold_right* for large sets.

```
    val product : α list t → α t list
    val product_fold : (α t → β → β) → α list t → β → β
```

For homogeneous tuples the *power* function could trivially be built from *product*, e. g.:

$$\mathit{power}\,[x_1;x_2] = \mathit{product}\,([x_1;x_2],[x_1;x_2]) = [(x_1,x_1);(x_1,x_2);(x_2,x_1);(x_2,x_2)] \tag{2.2}$$

but it is also well defined for polytuples, e. g. for pairs and triples

$$\mathit{power}\,[x_1;x_2] = \mathit{product}\,([x_1;x_2],[x_1;x_2]) \cup \mathit{product}\,([x_1;x_2],[x_1;x_2],[x_1;x_2]) \tag{2.3}$$

For tuples and polytuples with bounded arity, the *power* and *power_fold* functions terminate. In polytuples with unbounded arity, the the *power* function raises *No_termination* unless a limit is given by ?*truncate*. *power_fold* also raises *No_termination*, but could be changed to run until the argument function raises an exception. However, if we need this behaviour, we should probably implement *power_iter* instead.

```
    val power : ?truncate :int → α list → α t list
    val power_fold : ?truncate :int → (α t → β → β) → α list → β → β
```

We can also identify all (poly)tuples with permuted elements and return only one representative, e. g.:

$$sym\_power\,[x_1; x_2] = [(x_1, x_1); (x_1, x_2); (x_2, x_2)] \tag{2.4}$$

NB: this function has not yet been implemented, because O'Mega only needs the more efficient special case *graded_sym_power*.
If a set X is graded (i. e. there is a map $\phi : X \to \mathbf{N}$, called *rank* below), the results of *power* or *sym_power* can canonically be filtered by requiring that the sum of the ranks in each (poly)tuple has one chosen value. Implementing such a function directly is much more efficient than constructing and subsequently disregarding many (poly)tuples. The elements of rank n are at offset $(n - 1)$ in the array. The array is assumed to be *immutable*, even if O'Caml doesn't support immutable arrays. NB: *graded_power* has not yet been implemented, because O'Mega only needs *graded_sym_power*.

```
    type α graded = α list array
    val graded_sym_power : int → α graded → α t list
    val graded_sym_power_fold : int → (α t → β → β) → α graded →
      β → β
```

We hope to be able to avoid the next one in the long run, because it mildly breaks typesafety for arities. Unfortunately, we're still working on it . . .

```
    val to_list : α t → α list
```

The next one is only used for Fermi statistics in the obsolescent *Fusion_vintage* module below, but can not be implemented if there are no binary tuples. It must be retired as soon as possible.

```
    val of2_kludge : α → α → α t
  end

module type Poly =
  sig
    include Mono
    exception Mismatched_arity
    exception No_termination
  end

module type Binary =
    sig
      include Poly (∗ should become Mono! ∗)
      val of2 : α → α → α t
    end
module Binary : Binary

module type Ternary =
    sig
      include Mono
      val of3 : α → α → α → α t
    end
module Ternary : Ternary

type α pair_or_triple = T2 of α × α | T3 of α × α × α

module type Mixed23 =
    sig
      include Poly
      val of2 : α → α → α t
      val of3 : α → α → α → α t
    end
module Mixed23 : Mixed23

module type Nary =
    sig
      include Poly
      val of2 : α → α → α t
      val of3 : α → α → α → α t
      val of_list : α list → α t
    end
module Unbounded_Nary : Nary
```

It seemed like a good idea, but hardcoding *max_arity* here prevents optimizations for processes with fewer external particles than *max_arity*. For *max_arity* $\geq$ 8 things become bad! Need to implement a truncating version of *power* and *power_fold*.

```
module type Bound = sig val max_arity : unit → int end
module Nary (B : Bound) : Nary
```

For compleneteness sake, we could add most of the *List* signature

- val *length* : α t → int
- val *hd* : α t → α
- val *nth* : α t → int → α
- val *rev* : α t → α t
- val *rev_map* : (α → β) → α t → β t
- val *iter2* : (α → β → unit) → α t → β t → unit
- val *rev_map2* : (α → β → γ) → α t → β t → γ t
- val *fold_left2* : (α → β → γ → α) → α → β t → γ t → α
- val *fold_right2* : (α → β → γ → γ) → α t → β t → γ → γ
- val *exists* : (α → bool) → α t → bool
- val *for_all2* : (α → β → bool) → α t → β t → bool
- val *exists2* : (α → β → bool) → α t → β t → bool
- val *mem* : α → α t → bool
- val *memq* : α → α t → bool
- val *find* : (α → bool) → α t → α
- val *find_all* : (α → bool) → α t → α list
- val *assoc* : α → (α × β) t → β
- val *assq* : α → (α × β) t → β
- val *mem_assoc* : α → (α × β) t → bool
- val *mem_assq* : α → (α × β) t → bool
- val *combine* : α t → β t → (α × β) t
- val *sort* : (α → α → int) → α t → α t
- val *stable_sort* : (α → α → int) → α t → α t

but only if we ever have too much time on our hand . . .

2.2 Implementation of *Tuple*

```
module type Mono =
  sig
    type α t
    val arity : α t → int
    val max_arity : unit → int
    val compare : (α → α → int) → α t → α t → int
    val for_all : (α → bool) → α t → bool
    val map : (α → β) → α t → β t
    val iter : (α → unit) → α t → unit
    val fold_left : (α → β → α) → α → β t → α
    val fold_right : (α → β → β) → α t → β → β
    val fold_left_internal : (α → α → α) → α t → α
    val fold_right_internal : (α → α → α) → α t → α
    val map2 : (α → β → γ) → α t → β t → γ t
    val split : (α × β) t → α t × β t
    val product : α list t → α t list
    val product_fold : (α t → β → β) → α list t → β → β
    val power : ?truncate :int → α list → α t list
    val power_fold : ?truncate :int → (α t → β → β) → α list → β → β
    type α graded = α list array
    val graded_sym_power : int → α graded → α t list
    val graded_sym_power_fold : int → (α t → β → β) → α graded →
      β → β
    val to_list : α t → α list
    val of2_kludge : α → α → α t
  end

module type Poly =
  sig
    include Mono
    exception Mismatched_arity
    exception No_termination
  end
```

2.2.1 Typesafe Combinatorics

Wrap the combinatorical functions with varying arities into typesafe functions with fixed arities. We could provide specialized implementations, but since we *know* that *Impossible* is *never* raised, the present approach is just as good (except for a tiny inefficiency).

```
exception Impossible of string
let impossible name = raise (Impossible name)

let choose2 set =
  List.map (function [x; y] → (x, y) | _ → impossible "choose2")
      (Combinatorics.choose 2 set)

let choose3 set =
  List.map (function [x; y; z] → (x, y, z) | _ → impossible "choose3")
    (Combinatorics.choose 3 set)
```

2.2.2 Pairs

```
module type Binary =
    sig
      include Poly (* should become Mono! *)
      val of2 : α → α → α t
    end

module Binary =
  struct

    type α t = α × α

    let arity _ = 2
    let max_arity () = 2

    let of2 x y = (x, y)
    let compare cmp (x1, y1) (x2, y2) =
      let cx = cmp x1 x2 in
      if cx ≠ 0 then
        cx
      else
        cmp y1 y2

    let for_all p (x, y) = p x ∧ p y

    let map f (x, y) = (f x, f y)
    let iter f (x, y) = f x; f y
    let fold_left f init (x, y) = f (f init x) y
    let fold_right f (x, y) init = f x (f y init)
    let fold_left_internal f (x, y) = f x y
    let fold_right_internal f (x, y) = f x y

    exception Mismatched_arity
    let map2 f (x1, y1) (x2, y2) = (f x1 x2, f y1 y2)

    let split ((x1, x2), (y1, y2)) = ((x1, y1), (x2, y2))

    let product (lx, ly) =
      Product.list2 (fun x y → (x, y)) lx ly
    let product_fold f (lx, ly) init =
      Product.fold2 (fun x y → f (x, y)) lx ly init

    let power ?truncate l =
      match truncate with
      | None → product (l, l)
      | Some n →
          if n ≥ 2 then
            product (l, l)
          else
            invalid_arg "Tuple.Binary.power:␣truncate␣<␣2"

    let power_fold ?truncate f l =
      match truncate with
      | None → product_fold f (l, l)
      | Some n →
          if n ≥ 2 then
            product_fold f (l, l)
          else
            invalid_arg "Tuple.Binary.power_fold:␣truncate␣<␣2"
```

In the special case of binary fusions, the implementation is very concise.

```
    type α graded = α list array

    let fuse2 f set (i, j) acc =
      if i = j then
        List.fold_right (fun (x, y) → f x y) (choose2 set.(pred i)) acc
      else
        Product.fold2 f set.(pred i) set.(pred j) acc

    let graded_sym_power_fold rank f set acc =
      let max_rank = Array.length set in
      List.fold_right (fuse2 (fun x y → f (of2 x y)) set)
        (Partition.pairs rank 1 max_rank) acc

    let graded_sym_power rank set =
      graded_sym_power_fold rank (fun pair acc → pair :: acc) set []

    let to_list (x, y) = [x; y]

    let of2_kludge = of2

    exception No_termination
  end
```

2.2.3 Triples

```
module type Ternary =
    sig
      include Mono
      val of3 : α → α → α → α t
    end

module Ternary =
  struct

    type α t = α × α × α

    let arity _ = 3
    let max_arity () = 3

    let of3 x y z = (x, y, z)

    let compare cmp (x1, y1, z1) (x2, y2, z2) =
      let cx = cmp x1 x2 in
      if cx ≠ 0 then
        cx
      else
        let cy = cmp y1 y2 in
        if cy ≠ 0 then
          cy
        else
          cmp z1 z2

    let for_all p (x, y, z) = p x ∧ p y ∧ p z

    let map f (x, y, z) = (f x, f y, f z)
    let iter f (x, y, z) = f x; f y; f z
    let fold_left f init (x, y, z) = f (f (f init x) y) z
    let fold_right f (x, y, z) init = f x (f y (f z init))
    let fold_left_internal f (x, y, z) = f (f x y) z
    let fold_right_internal f (x, y, z) = f x (f y z)

    exception Mismatched_arity
    let map2 f (x1, y1, z1) (x2, y2, z2) = (f x1 x2, f y1 y2, f z1 z2)

    let split ((x1, x2), (y1, y2), (z1, z2)) = ((x1, y1, z1), (x2, y2, z2))

    let product (lx, ly, lz) =
      Product.list3 (fun x y z → (x, y, z)) lx ly lz
    let product_fold f (lx, ly, lz) init =
      Product.fold3 (fun x y z → f (x, y, z)) lx ly lz init

    let power ?truncate l =
      match truncate with
      | None → product (l, l, l)
      | Some n →
          if n ≥ 3 then
            product (l, l, l)
          else
            invalid_arg "Tuple.Ternary.power:␣truncate␣<␣3"

    let power_fold ?truncate f l =
      match truncate with
      | None → product_fold f (l, l, l)
      | Some n →
          if n ≥ 3 then
            product_fold f (l, l, l)
          else
            invalid_arg "Tuple.Ternary.power_fold:␣truncate␣<␣3"

    type α graded = α list array
    let fuse3 f set (i, j, k) acc =
      if i = j then begin
        if j = k then
          List.fold_right (fun (x, y, z) → f x y z) (choose3 set.(pred i)) acc
        else
          Product.fold2 (fun (x, y) z → f x y z)
            (choose2 set.(pred i)) set.(pred k) acc
      end else begin
        if j = k then
          Product.fold2 (fun x (y, z) → f x y z)
            set.(pred i) (choose2 set.(pred j)) acc
        else
          Product.fold3 (fun x y z → f x y z)
            set.(pred i) set.(pred j) set.(pred k) acc
      end

    let graded_sym_power_fold rank f set acc =
      let max_rank = Array.length set in
      List.fold_right (fuse3 (fun x y z → f (of3 x y z)) set)
        (Partition.triples rank 1 max_rank) acc

    let graded_sym_power rank set =
      graded_sym_power_fold rank (fun pair acc → pair :: acc) set []

    let to_list (x, y, z) = [x; y; z]

    let of2_kludge _ = failwith "Tuple.Ternary.of2_kludge"

  end
```

2.2.4 Pairs and Triples

```
type α pair_or_triple = T2 of α × α | T3 of α × α × α

module type Mixed23 =
    sig
      include Poly
      val of2 : α → α → α t
      val of3 : α → α → α → α t
    end

module Mixed23 =
  struct

    type α t = α pair_or_triple

    let arity = function
      | T2 _ → 2
      | T3 _ → 3
    let max_arity () = 3

    let of2 x y = T2 (x, y)
    let of3 x y z = T3 (x, y, z)

    let compare cmp m1 m2 =
      match m1, m2 with
      | T2 _, T3 _ → −1
      | T3 _, T2 _ → 1
      | T2 (x1, y1), T2 (x2, y2) →
          let cx = cmp x1 x2 in
          if cx ≠ 0 then
            cx
          else
            cmp y1 y2
      | T3 (x1, y1, z1), T3 (x2, y2, z2) →
          let cx = cmp x1 x2 in
```

```
        if cx ≠ 0 then
          cx
        else
          let cy = cmp y1 y2 in
          if cy ≠ 0 then
            cy
          else
            cmp z1 z2

let for_all p = function
  | T2 (x, y) → p x ∧ p y
  | T3 (x, y, z) → p x ∧ p y ∧ p z

let map f = function
  | T2 (x, y) → T2 (f x, f y)
  | T3 (x, y, z) → T3 (f x, f y, f z)

let iter f = function
  | T2 (x, y) → f x; f y
  | T3 (x, y, z) → f x; f y; f z

let fold_left f init = function
  | T2 (x, y) → f (f init x) y
  | T3 (x, y, z) → f (f (f init x) y) z

let fold_right f m init =
  match m with
  | T2 (x, y) → f x (f y init)
  | T3 (x, y, z) → f x (f y (f z init))

let fold_left_internal f m =
  match m with
  | T2 (x, y) → f x y
  | T3 (x, y, z) → f (f x y) z

let fold_right_internal f m =
  match m with
  | T2 (x, y) → f x y
  | T3 (x, y, z) → f x (f y z)

exception Mismatched_arity
let map2 f m1 m2 =
  match m1, m2 with
  | T2 (x1, y1), T2 (x2, y2) → T2 (f x1 x2, f y1 y2)
  | T3 (x1, y1, z1), T3 (x2, y2, z2) → T3 (f x1 x2, f y1 y2, f z1 z2)
  | T2 _, T3 _ | T3 _, T2 _ → raise Mismatched_arity

let split = function
  | T2 ((x1, x2), (y1, y2)) → (T2 (x1, y1), T2 (x2, y2))
  | T3 ((x1, x2), (y1, y2), (z1, z2)) → (T3 (x1, y1, z1), T3 (x2, y2, z2))

let product = function
  | T2 (lx, ly) → Product.list2 (fun x y → T2 (x, y)) lx ly
  | T3 (lx, ly, lz) → Product.list3 (fun x y z → T3 (x, y, z)) lx ly lz
let product_fold f m init =
  match m with
  | T2 (lx, ly) → Product.fold2 (fun x y → f (T2 (x, y))) lx ly init
  | T3 (lx, ly, lz) →
      Product.fold3 (fun x y z → f (T3 (x, y, z))) lx ly lz init

exception No_termination

let power_fold23 f l init =
  product_fold f (T2 (l, l)) (product_fold f (T3 (l, l, l)) init)

let power_fold2 f l init =
  product_fold f (T2 (l, l)) init
```

```
    let power_fold ?truncate f l init =
      match truncate with
      | None → power_fold23 f l init
      | Some n →
          if n ≥ 3 then
            power_fold23 f l init
          else if n = 2 then
            power_fold2 f l init
          else
            invalid_arg "Tuple.Mixed23.power_fold:␣truncate␣<␣2"

    let power ?truncate l =
      power_fold ?truncate (fun m acc → m :: acc) l []

    type α graded = α list array

    let graded_sym_power_fold rank f set acc =
      let max_rank = Array.length set in
      List.fold_right (Binary.fuse2 (fun x y → f (of2 x y)) set)
        (Partition.pairs rank 1 max_rank)
        (List.fold_right (Ternary.fuse3 (fun x y z → f (of3 x y z)) set)
           (Partition.triples rank 1 max_rank) acc)

    let graded_sym_power rank set =
      graded_sym_power_fold rank (fun pair acc → pair :: acc) set []

    let to_list = function
      | T2 (x, y) → [x; y]
      | T3 (x, y, z) → [x; y; z]

    let of2_kludge = of2

  end
```

2.2.5 ... and All The Rest

```
module type Nary =
    sig
      include Poly
      val of2 : α → α → α t
      val of3 : α → α → α → α t
      val of_list : α list → α t
    end

module Nary (A : sig val max_arity : unit → int end) =
  struct

    type α t = α × α list

    let arity (_, y) = succ (List.length y)

    let max_arity () =
      try A.max_arity () with _ → −1

    let of2 x y = (x, [y])
    let of3 x y z = (x, [y; z])

    let of_list = function
      | x :: y → (x, y)
      | [] → invalid_arg "Tuple.Nary.of_list:␣empty"

    let compare cmp (x1, y1) (x2, y2) =
      let c = cmp x1 x2 in
      if c ≠ 0 then
        c
      else
        ThoList.compare ~cmp y1 y2
```

```
let for_all p (x, y) = p x ∧ List.for_all p y

let map f (x, y) = (f x, List.map f y)
let iter f (x, y) = f x; List.iter f y
let fold_left f init (x, y) = List.fold_left f (f init x) y
let fold_right f (x, y) init = f x (List.fold_right f y init)
let fold_left_internal f (x, y) = List.fold_left f x y
let fold_right_internal f (x, y) =
  match List.rev y with
  | [] → x
  | y0 :: y_sans_y0 →
      f x (List.fold_right f (List.rev y_sans_y0) y0)

exception Mismatched_arity
let map2 f (x1, y1) (x2, y2) =
  try (f x1 x2, List.map2 f y1 y2) with
  | Invalid_argument _ → raise Mismatched_arity

let split ((x1, x2), y12) =
  let y1, y2 = List.split y12 in
  ((x1, y1), (x2, y2))

let product (xl, yl) =
  Product.list (function
    | x :: y → (x, y)
    | [] → failwith "Tuple.Nary.product") (xl :: yl)

let product_fold f (xl, yl) init =
  Product.fold (function
    | x :: y → f (x, y)
    | [] → failwith "Tuple.Nary.product_fold") (xl :: yl) init

exception No_termination

let truncated_arity ?truncate () =
  let ma = max_arity () in
  match truncate with
  | None → ma
  | Some n →
      if n < 2 then
        invalid_arg "Tuple.Nary.power:␣truncate␣<␣2"
      else if ma ≥ 2 then
        min n ma
      else
        n

let power_fold ?truncate f l init =
  let ma = truncated_arity ?truncate () in
  if ma > 0 then
    List.fold_right
      (fun n → product_fold f (l, ThoList.clone l (pred n)))
      (ThoList.range 2 ma) init
  else
    raise No_termination

let power ?truncate l =
  power_fold ?truncate (fun t acc → t :: acc) l []

type α graded = α list array

let fuse_n f set partition acc =
  let choose (n, r) =
    Printf.printf "chose:␣n=%d␣r=%d␣len=%d\n"
      n r (List.length set.(pred r));
    Combinatorics.choose n set.(pred r) in
  Product.fold (fun wfs → f (List.concat wfs))
    (List.map choose (ThoList.classify partition)) acc
let fuse_n f set partition acc =
  let choose (n, r) = Combinatorics.choose n set.(pred r) in
  Product.fold (fun wfs → f (List.concat wfs))
    (List.map choose (ThoList.classify partition)) acc
```

graded_sym_power_fold is well defined for unbounded arities as well: derive a reasonable replacement from *set*. The length of the flattened *set* is an upper limit, of course, but too pessimistic in most cases.

```
    let graded_sym_power_fold rank f set acc =
      let max_rank = Array.length set in
      let degrees = ThoList.range 2 (max_arity ()) in
      let partitions =
        ThoList.flatmap
          (fun deg → Partition.tuples deg rank 1 max_rank) degrees in
      List.fold_right (fuse_n (fun wfs → f (of_list wfs)) set) partitions acc

    let graded_sym_power rank set =
      graded_sym_power_fold rank (fun pair acc → pair :: acc) set []

    let to_list (x, y) = x :: y

    let of2_kludge = of2

  end

module type Bound = sig val max_arity : unit → int end
module Unbounded_Nary = Nary (struct let max_arity () = −1 end)
```

—3— Topologies

3.1 Interface of Topology

```
module type T =
  sig
```

partition is a collection of integers, with arity one larger than the arity of α *children* below. These arities can one fixed number corresponding to homogeneous tuples or a collection of tupes or lists.

```
    type partition
```

partitions n returns the union of all $[n_1; n_2; \ldots; n_d]$ with $1 \leq n_1 \leq n_2 \leq \ldots \leq n_d \leq \lfloor n/2 \rfloor$ and

$$\sum_{i=1}^{d} n_i = n \tag{3.1}$$

for d from 3 to $d_{\max}$, where $d_{\max}$ is a fixed number for each module implementating T. In particular, if type *partition* = *int* × *int* × *int*, then *partitions* n returns all (n_1, n_2, n_3) with $n_1 \leq n_2 \leq n_3$ and $n_1+n_2+n_3 = n$.

```
    val partitions : int → partition list
```

A (poly)tuple as implemented by the modules in *Tuple*:

```
    type α children
```

keystones externals returns all keystones for the amplitude with external states *externals* in the vanilla scalar theory with a

$$\sum_{3 \leq k \leq d_{\max}} \lambda_k \phi^k \tag{3.2}$$

interaction. One factor of the products is factorized. In particular, if

```
    type α children = α Tuple.Binary.t = α × α,
```

then *keystones externals* returns all keystones for the amplitude with external states *externals* in the vanilla scalar $\lambda\phi^3$-theory.

```
    val keystones : α list → (α list × α list children list) list
```

The maximal depth of subtrees for a given number of external lines.

```
    val max_subtree : int → int
```

Only for diagnostics:

```
    val inspect_partition : partition → int list
  end
```

```
module Binary : T with type α children = α Tuple.Binary.t
module Ternary : T with type α children = α Tuple.Ternary.t
module Mixed23 : T with type α children = α Tuple.Mixed23.t
module Nary : functor (B : Tuple.Bound) →
  (T with type α children = α Tuple.Nary(B).t)
```

3.1.1 Diagnostics: Counting Diagrams and Factorizations for $\sum_n \lambda_n\phi^n$

The number of diagrams for many particles can easily exceed the range of native integers. Even if we can not calculate the corresponding amplitudes, we want to check combinatorical factors. Therefore we code a functor that can use arbitray implementations of integers.

```
module type Integer =
  sig
    type t
    val zero : t
    val one : t
    val ( + ) : t → t → t
    val ( − ) : t → t → t
    val ( × ) : t → t → t
    val ( / ) : t → t → t
    val pred : t → t
    val succ : t → t
    val ( = ) : t → t → bool
    val ( ≠ ) : t → t → bool
    val ( < ) : t → t → bool
    val ( ≤ ) : t → t → bool
    val ( > ) : t → t → bool
    val ( ≥ ) : t → t → bool
    val of_int : int → t
    val to_int : t → int
    val to_string : t → string
    val compare : t → t → int
    val factorial : t → t
  end
```

Of course, native integers will provide the fastest implementation:

```
module Int : Integer
```

```
module type Count =
  sig
    type integer
```

diagrams f d n returns the number of tree diagrams contributing to the n-point amplitude in vanilla scalar theory with

$$\sum_{3\le k\le d \land f(k)} \lambda_k \phi^k \tag{3.3}$$

interaction. The default value of f returns true for all arguments.

```
    val diagrams : ?f : (integer → bool) → integer → integer → integer
    val diagrams_via_keystones : integer → integer → integer
```

$$\frac{1}{S(n_k, n-n_k)}\frac{1}{S(n_1,n_2,\ldots,n_k)}\binom{n_1+n_2+\ldots+n_k}{n_1,n_2,\ldots,n_k} \tag{3.4}$$

```
    val keystones : integer list → integer
```

diagrams_via_keystones d n must produce the same results as *diagrams d n*. This is shown explicitely in tables 3.2, 3.3 and 3.4 for small values of d and n. The test program in appendix V can be used to verify this relation for larger values.

```
    val diagrams_per_keystone : integer → integer list → integer
  end
```

```
module Count : functor (I : Integer) → Count with type integer = I.t
```

3.1.2 Emulating HELAC

We can also proceed á la [2].

```
module Helac : functor (B : Tuple.Bound) →
  (T with type α children = α Tuple.Nary(B).t)
```

n	*partitions* n
4	(1,1,2)
5	(1,2,2)
6	(1,2,3), (2,2,2)
7	(1,3,3), (2,2,3)
8	(1,3,4), (2,2,4), (2,3,3)
9	(1,4,4), (2,3,4), (3,3,3)
10	(1,4,5), (2,3,5), (2,4,4), (3,3,4)
11	(1,5,5), (2,4,5), (3,3,5), (3,4,4)
12	(1,5,6), (2,4,6), (2,5,5), (3,3,6), (3,4,5), (4,4,4)
13	(1,6,6), (2,5,6), (3,4,6), (3,5,5), (4,4,5)
14	(1,6,7), (2,5,7), (2,6,6), (3,4,7), (3,5,6), (4,4,6), (4,5,5)
15	(1,7,7), (2,6,7), (3,5,7), (3,6,6), (4,4,7), (4,5,6), (5,5,5)
16	(1,7,8), (2,6,8), (2,7,7), (3,5,8), (3,6,7), (4,4,8), (4,5,7), (4,6,6), (5,5,6)

Table 3.1: *partitions* n for moderate values of n.

The following has never been tested, but it is no rocket science and should work anyway …

```
module Helac_Binary : T with type α children = α Tuple.Binary.t
```

3.2 Implementation of Topology

```
module type T =
  sig
    type partition
    val partitions : int → partition list
    type α children
    val keystones : α list → (α list × α list children list) list
    val max_subtree : int → int
    val inspect_partition : partition → int list
  end
```

3.2.1 Factorizing Diagrams for ϕ^3

```
module Binary =
  struct
    type partition = int × int × int
    let inspect_partition (n1, n2, n3) = [n1; n2; n3]
```

One way [1] to lift the degeneracy is to select the vertex that is closest to the center (see table 3.1):

$$\textit{partitions} : n \to \left\{(n_1,n_2,n_3)\,|\,n_1+n_2+n_3=n \land n_1 \le n_2 \le n_3 \le \lfloor n/2 \rfloor\right\} \tag{3.5}$$

Other, less symmetric, approaches are possible. The simplest of these is: choose the vertex adjacent to a fixed external line [2]. They will be made available for comparison in the future.
An obvious consequence of $n_1+n_2+n_3=n$ and $n_1 \le n_2 \le n_3$ is $n_1 \le \lfloor n/3 \rfloor$:

```
    let rec partitions' n n1 =
      if n1 > n / 3 then
        []
      else
        List.map (fun (n2, n3) → (n1, n2, n3))
          (Partition.pairs (n − n1) n1 (n / 2)) @ partitions' n (succ n1)

    let partitions n = partitions' n 1


    type α children = α Tuple.Binary.t
```

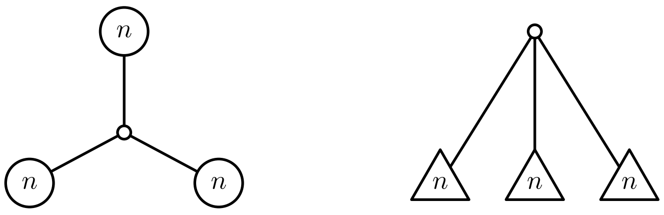


Figure 3.1: Topologies with a blatant three-fold permutation symmetry, if the number of external lines is a multiple of three

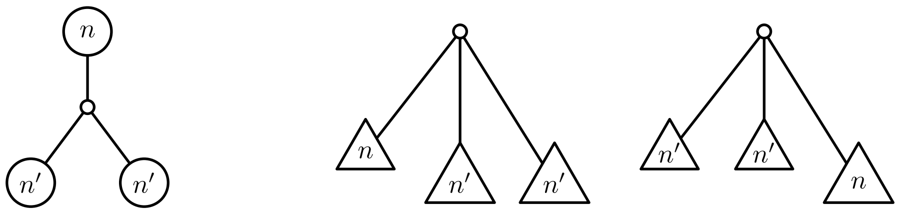


Figure 3.2: Topologies with a blatant two-fold symmetry.

There remains one peculiar case, when the number of external lines is even and $n_3 = n_1 + n_2$ (cf. figure 3.3). Unfortunately, this reflection symmetry is not respected by the equivalence classes. E. g.

$$\{1\}\{2,3\}\{4,5,6\} \mapsto \big\{\{4\}\{5,6\}\{1,2,3\};\{5\}\{4,6\}\{1,2,3\};\{6\}\{4,5\}\{1,2,3\}\big\} \tag{3.6}$$

However, these reflections will always exchange the two halves and a representative can be chosen by requiring that one fixed momentum remains in one half. We choose to filter out the half of the partitions where the element p appears in the second half, i. e. the list of length *n3*.

Finally, a closed expression for the number of Feynman diagrams in the equivalence class (n_1, n_2, n_3) is

$$N(n_1,n_2,n_3) = \frac{(n_1+n_2+n_3)!}{S(n_1,n_2,n_3)} \prod_{i=1}^{3} \frac{(2n_i-3)!!}{n_i!} \tag{3.7}$$

where the symmetry factor from the above arguments is

$$S(n_1,n_2,n_3) = \begin{cases} 3! & \text{for } n_1 = n_2 = n_3 \\ 2\cdot 2 & \text{for } n_3 = 2n_1 = 2n_2 \\ 2 & \text{for } n_1 = n_2 \vee n_2 = n_3 \\ 2 & \text{for } n_1 + n_2 = n_3 \end{cases} \tag{3.8}$$

Indeed, the sum of all Feynman diagrams

$$\sum_{\substack{n_1+n_2+n_3=n \\ 1\le n_1\le n_2\le n_3\le\lfloor n/2\rfloor}} N(n_1,n_2,n_3) = (2n-5)!! \tag{3.9}$$

can be checked numerically for large values of $n = n_1 + n_2 + n_3$, verifying the symmetry factor (see table 3.2).

P. M. claims to have seen similar formulae in the context of Young tableaux. That's a good occasion to read the new edition of Howard's book ...

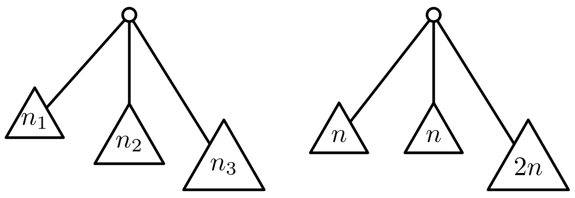


Figure 3.3: If $n_3 = n_1 + n_2$, the apparently asymmetric topologies on the left hand side have a non obvious two-fold symmetry, that exchanges the two halves. Therefore, the topologies on the right hand side have a four fold symmetry.

n	$(2n-5)!!$	$\sum N(n_1,n_2,n_3)$
4	3	$3\cdot(1,1,2)$
5	15	$15\cdot(1,2,2)$
6	105	$90\cdot(1,2,3)+15\cdot(2,2,2)$
7	945	$630\cdot(1,3,3)+315\cdot(2,2,3)$
8	10395	$6300\cdot(1,3,4)+1575\cdot(2,2,4)+2520\cdot(2,3,3)$
9	135135	$70875\cdot(1,4,4)+56700\cdot(2,3,4)+7560\cdot(3,3,3)$
10	2027025	$992250\cdot(1,4,5)+396900\cdot(2,3,5)$
		$+354375\cdot(2,4,4)+283500\cdot(3,3,4)$
11	34459425	$15280650\cdot(1,5,5)+10914750\cdot(2,4,5)$
		$+4365900\cdot(3,3,5)+3898125\cdot(3,4,4)$
12	654729075	$275051700\cdot(1,5,6)+98232750\cdot(2,4,6)$
		$+91683900\cdot(2,5,5)+39293100\cdot(3,3,6)$
		$+130977000\cdot(3,4,5)+19490625\cdot(4,4,4)$

Table 3.2: Equation (3.9) for small values of n.

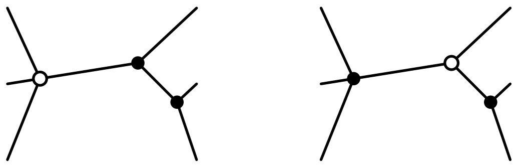


Figure 3.4: Degenerate $(1,1,1,3)$ and $(1,2,3)$.

Return a list of all inequivalent partitions of the list l in three lists of length $n1$, $n2$ and $n3$, respectively. Common first lists are factored. This is nothing more than a typedafe wrapper around *Combinatorics.factorized_keystones*.

```
  exception Impossible of string
  let tuple_of_list2 = function
    | [x1; x2] → Tuple.Binary.of2 x1 x2
    | _ → raise (Impossible "Topology.tuple_of_list")

  let keystone (n1, n2, n3) l =
    List.map (fun (p1, p23) → (p1, List.rev_map tuple_of_list2 p23))
      (Combinatorics.factorized_keystones [n1; n2; n3] l)

  let keystones l =
    ThoList.flatmap (fun n123 → keystone n123 l) (partitions (List.length l))

  let max_subtree n = n / 2

end
```

3.2.2 Factorizing Diagrams for $\sum_n \lambda_n \phi^n$

Mixed ϕ^n adds new degeneracies, as in figure 3.4. They appear if and only if one part takes exactly half of the external lines and can relate central vertices of different arity.

```
module Nary (B : Tuple.Bound) =
  struct
    type partition = int list
    let inspect_partition p = p

    let partition d sum =
      Partition.tuples d sum 1 (sum / 2)

    let rec partitions′ d sum =
      if d < 3 then
        []
```

n	$\sum$	$\sum$
4	4	$1\cdot(1,1,1,1)+3\cdot(1,1,2)$
5	25	$10\cdot(1,1,1,2)+15\cdot(1,2,2)$
6	220	$40\cdot(1,1,1,3)+45\cdot(1,1,2,2)+120\cdot(1,2,3)+15\cdot(2,2,2)$
7	2485	$840\cdot(1,1,2,3)+105\cdot(1,2,2,2)+1120\cdot(1,3,3)+420\cdot(2,2,3)$
8	34300	$5250\cdot(1,1,2,4)+4480\cdot(1,1,3,3)+3360\cdot(1,2,2,3)$
		$+105\cdot(2,2,2,2)+14000\cdot(1,3,4)$
		$+2625\cdot(2,2,4)+4480\cdot(2,3,3)$
9	559405	$126000\cdot(1,1,3,4)+47250\cdot(1,2,2,4)+40320\cdot(1,2,3,3)$
		$+5040\cdot(2,2,2,3)+196875\cdot(1,4,4)$
		$+126000\cdot(2,3,4)+17920\cdot(3,3,3)$
10	10525900	$1108800\cdot(1,1,3,5)+984375\cdot(1,1,4,4)+415800\cdot(1,2,2,5)$
		$+1260000\cdot(1,2,3,4)+179200\cdot(1,3,3,3)+78750\cdot(2,2,2,4)$
		$+100800\cdot(2,2,3,3)+3465000\cdot(1,4,5)+1108800\cdot(2,3,5)$
		$+984375\cdot(2,4,4)+840000\cdot(3,3,4)$

Table 3.3: $\mathcal{L}=\lambda_3\phi^3+\lambda_4\phi^4$

n	$\sum$	$\sum$
4	4	$1\cdot(1,1,1,1)+3\cdot(1,1,2)$
5	26	$1\cdot(1,1,1,1,1)+10\cdot(1,1,1,2)+15\cdot(1,2,2)$
6	236	$1\cdot(1,1,1,1,1,1)+15\cdot(1,1,1,1,2)+40\cdot(1,1,1,3)$
		$+45\cdot(1,1,2,2)+120\cdot(1,2,3)+15\cdot(2,2,2)$
7	2751	$21\cdot(1,1,1,1,1,2)+140\cdot(1,1,1,1,3)+105\cdot(1,1,1,2,2)$
		$+840\cdot(1,1,2,3)+105\cdot(1,2,2,2)+1120\cdot(1,3,3)+420\cdot(2,2,3)$
8	39179	$224\cdot(1,1,1,1,1,3)+210\cdot(1,1,1,1,2,2)+910\cdot(1,1,1,1,4)$
		$+2240\cdot(1,1,1,2,3)+420\cdot(1,1,2,2,2)+5460\cdot(1,1,2,4)$
		$+4480\cdot(1,1,3,3)+3360\cdot(1,2,2,3)+105\cdot(2,2,2,2)$
		$+14560\cdot(1,3,4)+2730\cdot(2,2,4)+4480\cdot(2,3,3)$

Table 3.4: $\mathcal{L}=\lambda_3\phi^3+\lambda_4\phi^4+\lambda_5\phi^5+\lambda_6\phi^6$

```
      else
        partition d sum @ partitions' (pred d) sum

    let partitions sum = partitions' (succ (B.max_arity ())) sum


    module Tuple = Tuple.Nary(B)
    type α children = α Tuple.t

    let keystones' l =
      let n = List.length l in
      ThoList.flatmap (fun p → Combinatorics.factorized_keystones p l)
        (partitions n)

    let keystones l =
      List.map (fun (bra, kets) → (bra, List.map Tuple.of_list kets))
        (keystones' l)

    let max_subtree n = n / 2

  end

module Nary4 = Nary (struct let max_arity () = 3 end)
```

3.2.3 Factorizing Diagrams for ϕ^4

```
module Ternary =
  struct
```

```
    type partition = int × int × int × int
    let inspect_partition (n1, n2, n3, n4) = [n1; n2; n3; n4]
    type α children = α Tuple.Ternary.t
    let collect4 acc = function
      | [x; y; z; u] → (x, y, z, u) :: acc
      | _ → acc
    let partitions n =
      List.fold_left collect4 [] (Nary4.partitions n)
    let collect3 acc = function
      | [x; y; z] → Tuple.Ternary.of3 x y z :: acc
      | _ → acc
    let keystones l =
      List.map (fun (bra, kets) → (bra, List.fold_left collect3 [] kets))
        (Nary4.keystones' l)
    let max_subtree = Nary4.max_subtree
  end
```

3.2.4 Factorizing Diagrams for $\phi^3 + \phi^4$

```
module Mixed23 =
  struct
    type partition =
      | P3 of int × int × int
      | P4 of int × int × int × int
    let inspect_partition = function
      | P3 (n1, n2, n3) → [n1; n2; n3]
      | P4 (n1, n2, n3, n4) → [n1; n2; n3; n4]
    type α children = α Tuple.Mixed23.t
    let collect34 acc = function
      | [x; y; z] → P3 (x, y, z) :: acc
      | [x; y; z; u] → P4 (x, y, z, u) :: acc
      | _ → acc
    let partitions n =
      List.fold_left collect34 [] (Nary4.partitions n)
    let collect23 acc = function
      | [x; y] → Tuple.Mixed23.of2 x y :: acc
      | [x; y; z] → Tuple.Mixed23.of3 x y z :: acc
      | _ → acc
    let keystones l =
      List.map (fun (bra, kets) → (bra, List.fold_left collect23 [] kets))
        (Nary4.keystones' l)
    let max_subtree = Nary4.max_subtree
  end
```

3.2.5 Diagnostics: Counting Diagrams and Factorizations for $\sum_n \lambda_n \phi^n$

```
module type Integer =
  sig
    type t
    val zero : t
    val one : t
    val ( + ) : t → t → t
    val ( − ) : t → t → t
    val ( × ) : t → t → t
    val ( / ) : t → t → t
    val pred : t → t
    val succ : t → t
    val ( = ) : t → t → bool
    val ( ≠ ) : t → t → bool
    val ( < ) : t → t → bool
    val ( ≤ ) : t → t → bool
    val ( > ) : t → t → bool
    val ( ≥ ) : t → t → bool
    val of_int : int → t
    val to_int : t → int
    val to_string : t → string
    val compare : t → t → int
    val factorial : t → t
  end
```

O'Caml's native integers suffice for all applications, but in appendix V, we want to use big integers for numeric checks in high orders:

```
module Int : Integer =
  struct
    type t = int
    let zero = 0
    let one = 1
    let ( + ) = ( + )
    let ( − ) = ( − )
    let ( × ) = ( × )
    let ( / ) = ( / )
    let pred = pred
    let succ = succ
    let ( = ) = ( = )
    let ( ≠ ) = ( ≠ )
    let ( < ) = ( < )
    let ( ≤ ) = ( ≤ )
    let ( > ) = ( > )
    let ( ≥ ) = ( ≥ )
    let of_int n = n
    let to_int n = n
    let to_string = string_of_int
    let compare = compare
    let factorial = Combinatorics.factorial
  end
```

```
module type Count =
  sig
    type integer
    val diagrams : ?f : (integer → bool) → integer → integer → integer
    val diagrams_via_keystones : integer → integer → integer
    val keystones : integer list → integer
    val diagrams_per_keystone : integer → integer list → integer
  end
```

```
module Count (I : Integer) =
  struct
    let description = ["(still␣inoperational)␣phi^n␣topology"]

    type integer = I.t
    open I
    let two = of_int 2
    let three = of_int 3
```

If $I.t$ is an abstract datatype, the polymorphic *Stdlib.min* can fail. Provide our own version using the specific comparison "$(\leq)$".

```
    let min x y =
      if x ≤ y then
        x
      else
        y
```

Counting Diagrams for $\sum_n \lambda_n \phi^n$

Classes of diagrams are defined by the number of vertices and their degrees. We could use fixed size arrays, but we will use a map instead. For efficiency, we also maintain the number of external lines and the total number of propagators.

```
module IMap = Map.Make (struct type t = integer let compare = I.compare end)

type diagram_class = { ext : integer; prop : integer; v : integer IMap.t }
```

The numbers of external lines, propagators and vertices are determined by the degrees and multiplicities of vertices:

$$E(\{n_3,n_4,\ldots\}) = 2 + \sum_{d=3}^{\infty}(d-2)n_d \tag{3.10a}$$

$$P(\{n_3,n_4,\ldots\}) = \sum_{d=3}^{\infty} n_d - 1 = V(\{n_3,n_4,\ldots\}) - 1 \tag{3.10b}$$

$$V(\{n_3,n_4,\ldots\}) = \sum_{d=3}^{\infty} n_d \tag{3.10c}$$

```
let num_ext v =
  List.fold_left (fun sum (d, n) → sum + (d − two) × n) two v

let num_prop v =
  List.fold_left (fun sum (_, n) → sum + n) (zero − one) v
```

The sum of all vertex degrees must be equal to the number of propagator end points. This can be verified easily:

$$2P(\{n_3,n_4,\ldots\}) + E(\{n_3,n_4,\ldots\}) = \sum_{d=3}^{\infty} dn_d \tag{3.11}$$

```
let add_degree map (d, n) =
  if d < three then
    invalid_arg "add_degree:␣d␣<␣3"
  else if n < zero then
    invalid_arg "add_degree:␣n␣<=␣0"
  else if n = zero then
    map
  else
    IMap.add d n map

let create_class v =
  { ext = num_ext v;
    prop = num_prop v;
    v = List.fold_left add_degree IMap.empty v }

let multiplicity cl d =
  if d ≥ three then
    try
      IMap.find d cl.v
    with
    | Not_found → zero
  else
    invalid_arg "multiplicity:␣d␣<␣3"
```

Remove one vertex of degree d, maintaining the invariants. Raises *Zero* if all vertices of degree d are exhausted.

```
exception Zero

let remove cl d =
  let n = pred (multiplicity cl d) in
  if n < zero then
    raise Zero
  else
```

```
    { $ext = cl.ext - (d - two)$;
      $prop = pred\ cl.prop$;
      $v$ = if $n = zero$ then
        $IMap.remove\ d\ cl.v$
      else
        $IMap.add\ d\ n\ cl.v$ }
```

Add one vertex of degree d, maintaining the invariants.

```
let $add\ cl\ d$ =
  { $ext = cl.ext + (d - two)$;
    $prop = succ\ cl.prop$;
    $v = IMap.add\ d\ (succ\ (multiplicity\ cl\ d))\ cl.v$ }
```

Count the number of diagrams. Any diagram can be obtained recursively either from a diagram with one ternary vertex less by insertion if a ternary vertex in an internal or external propagator or from a diagram with a higher order vertex that has its degree reduced by one:

$$\begin{aligned} D(\{n_3, n_4, \ldots\}) = {} & \\ & (P(\{n_3-1, n_4, \ldots\}) + E(\{n_3-1, n_4, \ldots\}))\, D(\{n_3-1, n_4, \ldots\}) \\ & + \sum_{d=4}^{\infty} (n_{d-1}+1) D(\{n_3, n_4, \ldots, n_{d-1}+1, n_d-1, \ldots\}) \end{aligned} \tag{3.12}$$

```
let rec $class\_size\ cl$ =
  if $cl.ext = two \lor cl.prop = zero$ then
    $one$
  else
    $IMap.fold$ (fun $d$ _ $s \to class\_size\_n\ cl\ d + s$) $cl.v$ ($class\_size\_3\ cl$)
```

Purely ternary vertices recurse among themselves:

```
and $class\_size\_3\ cl$ =
  try
    let $d' = remove\ cl\ three$ in
    $(d'.ext + d'.prop) \times class\_size\ d'$
  with
  | $Zero \to zero$
```

Vertices of higher degree recurse one step towards lower degrees:

```
and $class\_size\_n\ cl\ d$ =
  if $d > three$ then begin
    try
      let $d' = pred\ d$ in
      let $cl' = add\ (remove\ cl\ d)\ d'$ in
      $multiplicity\ cl'\ d' \times class\_size\ cl'$
    with
    | $Zero \to zero$
  end else
    $zero$
```

Find all $\{n_3, n_4, \ldots, n_d\}$ with

$$E(\{n_3, n_4, \ldots, n_d\}) - 2 = \sum_{i=3}^{c} l(i-2)n_i = sum \tag{3.13}$$

The implementation is a variant of *tuples* above.

```
let rec $distribute\_degrees'\ d\ sum$ =
  if $d < three$ then
    $invalid\_arg$ "distribute_degrees"
  else if $d = three$ then
    $[[(d,\ sum)]]$
  else
```

```
    distribute_degrees'' d sum (sum / (d − two))

and distribute_degrees'' d sum n =
  if n < zero then
    []
  else
    List.fold_left (fun ll l → ((d, n) :: l) :: ll)
      (distribute_degrees'' d sum (pred n))
      (distribute_degrees' (pred d) (sum − (d − two) × n))
```

Actually, we need to find all $\{n_3, n_4, \ldots, n_d\}$ with

$$E(\{n_3, n_4, \ldots, n_d\}) = sum \tag{3.14}$$

```
let distribute_degrees d sum = distribute_degrees' d (sum − two)
```

Finally we can count all diagrams by adding all possible ways of splitting the degrees of vertices. We can also count diagrams where *all* degrees satisfy a predicate f:

```
let diagrams ?(f = fun _ → true) deg n =
  List.fold_left (fun s d →
    if List.for_all (fun (d', n') → f d' ∨ n' = zero) d then
      s + class_size (create_class d)
    else
      s)
    zero (distribute_degrees deg n)
```

The next two are duplicated from *ThoList* and *Combinatorics*, in order to use the specific comparison functions.

```
let classify l =
  let rec add_to_class a = function
    | [] → [of_int 1, a]
    | (n, a') :: rest →
        if a = a' then
          (succ n, a) :: rest
        else
          (n, a') :: add_to_class a rest
  in
  let rec classify' cl = function
    | [] → cl
    | a :: rest → classify' (add_to_class a cl) rest
  in
  classify' [] l

let permutation_symmetry l =
  List.fold_left (fun s (n, _) → factorial n × s) one (classify l)

let symmetry l =
  let sum = List.fold_left (+) zero l in
  if List.exists (fun x → two × x = sum) l then
    two × permutation_symmetry l
  else
    permutation_symmetry l
```

The number of Feynman diagrams built of vertices with maximum degree $d_{\max}$ in a partition $N_{d,n} = \{n_1, n_2, \ldots, n_d\}$ with $n = n_1 + n_2 + \cdots + n_d$ and

$$\tilde{F}(d_{\max}, N_{d,n}) = \frac{n!}{|\mathcal{S}(N_{d,n})|\sigma(n_d, n)} \prod_{i=1}^{d} \frac{F(d_{\max}, n_i + 1)}{n_i!} \tag{3.15}$$

with $|\mathcal{S}(N)|$ the size of the symmetric group of N, $\sigma(n, 2n) = 2$ and $\sigma(n, m) = 1$ otherwise.

```
let keystones p =
  let sum = List.fold_left (+) zero p in
  List.fold_left (fun acc n → acc / (factorial n)) (factorial sum) p
    / symmetry p
let diagrams_per_keystone deg p =
  List.fold_left (fun acc n → acc × diagrams deg (succ n)) one p
```

We must find

$$F(d_{\max}, n) = \sum_{d=3}^{d_{\max}} \sum_{\substack{N=\{n_1,n_2,\ldots,n_d\} \\ n_1+n_2+\cdots+n_d=n \\ 1\le n_1\le n_2\le\cdots\le n_d\le\lfloor n/2\rfloor}} \tilde{F}(d_{\max}, N) \tag{3.16}$$

```
    let diagrams_via_keystones deg n =
      let module N = Nary (struct let max_arity () = to_int (pred deg) end) in
      List.fold_left
        (fun acc p → acc + diagrams_per_keystone deg p × keystones p)
        zero (List.map (List.map of_int) (N.partitions (to_int n)))

  end
```

3.2.6 Emulating HELAC

In [2], one leg is singled out:

```
module Helac (B : Tuple.Bound) =
  struct
    module Tuple = Tuple.Nary(B)

    type partition = int list
    let inspect_partition p = p

    let partition d sum =
      Partition.tuples d sum 1 (sum − d + 1)

    let rec partitions′ d sum =
      let d′ = pred d in
      if d′ < 2 then
        []
      else
        List.map (fun p → 1 :: p) (partition d′ (pred sum)) @ partitions′ d′ sum

    let partitions sum = partitions′ (succ (B.max_arity ())) sum

    type α children = α Tuple.t

    let keystones′ l =
      match l with
      | [] → []
      | head :: tail →
          [([head],
            ThoList.flatmap (fun p → Combinatorics.partitions (List.tl p) tail)
              (partitions (List.length l)))]

    let keystones l =
      List.map (fun (bra, kets) → (bra, List.map Tuple.of_list kets))
        (keystones′ l)

    let max_subtree n = pred n
  end
```

The following is not tested, but it is no rocket science either . . .

```
module Helac_Binary =
  struct
    type partition = int × int × int
    let inspect_partition (n1, n2, n3) = [n1; n2; n3]

    let partitions sum =
      List.map (fun (n2, n3) → (1, n2, n3))
        (Partition.pairs (sum − 1) 1 (sum − 2))
    type α children = α Tuple.Binary.t

    let keystones' l =
      match l with
      | [] → []
      | head :: tail →
          [([head],
            ThoList.flatmap (fun (_, p2, _) → Combinatorics.split p2 tail)
              (partitions (List.length l)))]

    let keystones l =
      List.map (fun (bra, kets) →
        (bra, List.map (fun (x, y) → Tuple.Binary.of2 x y) kets))
        (keystones' l)

    let max_subtree n = pred n

  end
```

—4— Directed Acyclical Graphs

4.1 Interface of DAG

This datastructure describes large collections of trees with many shared nodes. The sharing of nodes is semantically irrelevant, but can turn a factorial complexity to exponential complexity. Note that *DAG* implements only a very specialized subset of Directed Acyclical Graphs (DAGs).

4.1.1 Forests

A forest is a set of trees and we want to represent it efficiently by a DAG. However, we will not handle arbitrary forests here, but only such forests, where *all* subtrees of trees in the forest are also members of the forest.

In this case, we can represent a forest F over a set of nodes and a set of edges as a map from the set of nodes N to the direct product of the set of edges E and the set

$$t(N) = \bigcup_{n=0}^{\infty} N^{\times n} = \emptyset \cup N \cup N \times N \cup N \times N \times N \cup \ldots \tag{4.1}$$

of tuples of nodes augmented by a special element $\perp$ (“bottom”).

$$\begin{aligned} F : N &\to (E \times t(N)) \cup \{\perp\} \\ n &\mapsto \begin{cases} (e, (n'_1, n'_2, \ldots)) \\ \perp \end{cases} \end{aligned} \tag{4.2}$$

Nodes that are mapped to $\perp$ are called *leaf* nodes and nodes that do not appear in any $F(n)$ are called *root* nodes. There are as many trees in a given forest F as there are nodes. Our trees are Feynman tree diagrams and each forest F consists of one diagram and its subdiagrams.

For convenience, we require edges and nodes to be members of ordered sets. If the nodes are ordered, cycles can be detected easily

$$\forall n \in N : \Big(\big(F(n) = (e, x)\big) \Rightarrow \big(\forall n' \in x : n > n'\big) \Big) . \tag{4.3}$$

Note that this requirement does *not* exclude any trees. Even if we consider only topological equivalence classes with anonymous nodes and edges, we can always construct a canonical labeling and order from the children of the nodes. E. g. the depth of the tree beneath a node provides a suitable labeling for *all* forests. However, in practical applications, we will often have more efficient labelings and orders at our disposal.

The sematics of *compare* is expected to be compatible with *Pervasives.compare* (i. e. *Stdlib.compare* on O’Caml 4.08 and later):

$$\mathit{compare}(x, y) = \begin{cases} -1 & \text{for } x < y \\ 0 & \text{for } x = y \\ 1 & \text{for } x > y \end{cases} \tag{4.4}$$

```
module type Ord =
  sig
    type t
    val compare : t → t → int
  end

module type Forest =
  sig
```

```
    module Nodes : Ord
    type node = Nodes.t
    type edge
```

A tuple of nodes. The most general realization is type *children* = *node list*, but we use a *Tuple.Mono* or *Tuple.Poly* module for more specific implementations, where the number of nodes is bounded from below or above. For example to two for binary trees, as in ϕ^3 or QED. We can also have mixed arities (e. g. two and three for QCD) or even arbitrary arities. However, in most cases, there will be at least two children.

```
    type children
```

This type abbreviation and order allow to apply the *Set.Make* functor to $E \times t(N)$.

```
    type t = edge × children
```

In our implementation, we order by *children* and if they agree, we disambiguate by *edge*.

```
    val compare : t → t → int
```

Test a predicate for *all* children.

```
    val for_all : (node → bool) → t → bool
```

fold f (_, *children*) *acc* will calculate

$$f(x_1, f(x_2, \cdots f(x_n, acc))) \tag{4.5}$$

where the *children* are $\{x_1, x_2, \ldots, x_n\}$. There are slightly more efficient alternatives for fixed arity (in particular binary), but we want to be general.

```
    val fold : (node → α → α) → t → α → α
  end
```

We will use modules from *Tuple* to implement arity constraints for *Forest.children*.

```
module Forest : functor (PT : Tuple.Poly) →
  functor (N : Ord) → functor (E : Ord) →
      Forest with module Nodes = N and type edge = E.t
      and type node = N.t and type children = N.t PT.t
```

4.1.2 DAGs

A DAG will describe the recursive construction of one particle off-shell wave functions (1POW). The nodes are therefore the 1POWs and can be specified by a flavor and a momentum or a sum of external momenta. Just as in *Forest*, the edges are couplings and the leaf nodes are external on-shell wave functions. However, each node can now have more than one offspring, i. e. combination of edge and children or coupling and tuple of 1POWs. This factorizes the forest and optimizes the code by common subexpression elimination.

If $T(n, D)$ denotes the set of all binary trees with root n encoded in the DAG D, while

$$O(n, D) = \{(e_1, n_1, n_1'), \ldots, (e_k, n_k, n_k')\} \tag{4.6}$$

denotes the set of all *offspring* of n in D, and $\mathrm{tree}(e, t, t')$ denotes the binary tree formed by joining the binary trees t and t' with the label e, then

$$T(n, D) = \left\{ \mathrm{tree}(e_i, t_i, t_i') \,\middle|\, (e_i, t_i, t_i') \in \bigcup_{i=1}^{k} \{e_i\} \times T(n_i, D) \times T(n_i', D) \right\} \tag{4.7}$$

is the recursive definition of the binary trees encoded by the DAG D. It is obvious how this definitions translates to n-ary trees (including trees with mixed arity).

```
module type T =
  sig
```

When implementing modules of type T, the type *node* will be a *Ord.t* that allows us use *Map.Make* and *Set.Make* to construct maps. In a functor *Forest* → T, the order from *Forest.Node* will be used for ordering *node* in T. In particular, the equality of nodes in *add_node*, *add_offspring*, *harvest*, etc. below will be determined by *Forest.Node.compare*.

```
    type node
```

For *edge*, we need no additional structure.

```
type edge
```

In the description of the function we assume for definiteness DAGs of binary trees with type *children* = *node* × *node*. However, we will also have implementations with type *children* = *node list* below.
Other possibilities include type *children* = *V3* of *node* × *node* | *V4* of *node* × *node* × *node*. There's probable never a need to use sets with logarithmic access, but it would be easy to add.

```
type children
type t
```

The empty DAG.

```
val empty : t
```

add_node n dag returns the DAG *dag* with the node *n*. If the node *n* already exists in *dag*, *dag* is returned unchanged. Otherwise *n* is added without offspring.

```
val add_node : node → t → t
```

add_offspring n (*e*, (*n1*, *n2*)) *dag* returns the DAG *dag* with the node *n* and its offspring *n1* and *n2* with edge label *e*. Each node can have an arbitrary number of offspring, but identical offspring are added only once. In order to prevent cycles, *add_offspring* requires both $n > n1$ and $n > n2$ in the ordering of *node*s in the *Forest* that the DAG represents. The nodes *n1* and *n2* are added as by *add_node*. NB: Adding the nodes *n1* and *n2* even if they are sterile is not necessary for our applications. But even though it slows down the code by a few percent, it is desirable for consistency and allows much more concise implementations of *iter_nodes* and *fold_nodes* below.

```
val add_offspring : node → edge × children → t → t
exception Cycle
```

Just like *add_offspring*, but does not check for potential cycles.

```
val add_offspring_unsafe : node → edge × children → t → t
```

is_node n dag returns true iff *n* is a node in *dag*.

```
val is_node : node → t → bool
```

is_sterile n dag returns true iff *n* is a node in *dag*, but has no offspring.

```
val is_sterile : node → t → bool
```

is_offspring n (*e*, (*n1*, *n2*)) *dag* returns true iff *n1* and *n2* are offspring of *n* with label *e* in *dag*.

```
val is_offspring : node → edge × children → t → bool
```

There is no function val *offspring* : *node* → (*edge* × *children*) *list* to extract the structure of the DAG explicitely. Instead, we export a functional interface that allows us to transform a DAG and to evaluate the expression encoded by the DAG.
Note that the following functions can run into infinite recursion if the DAG given as argument contains cycles. The usual functionals for processing all nodes (including sterile) . . .

```
val iter_nodes : (node → unit) → t → unit
val map_nodes : (node → node) → t → t
val fold_nodes : (node → α → α) → t → α → α
```

. . . and all parent/offspring relations. Note that *map* requires *two* functions: one for the nodes and one for the edges and children. This is so because a change in the definition of node is *not* propagated automatically to where it is used as a child.

```
val iter : (node → edge × children → unit) → t → unit
val map : (node → node) →
  (node → edge × children → edge × children) → t → t
val fold : (node → edge × children → α → α) → t → α → α
```

Note that in it's current incarnation, *fold add_offspring dag empty* copies *only* the fertile nodes, while *fold add_offspring dag* (*fold_nodes add_node dag empty*) includes sterile ones, as does *map* (fun *n* → *n*) (fun *n ec* → *ec*) *dag*.

Return the DAG as a list of lists.

val $lists : t \rightarrow (node \times (edge \times children)\ list)\ list$

dependencies dag node returns a canonically sorted *Tree2.t* of all nodes reachable from *node*.

val $dependencies : t \rightarrow node \rightarrow (node,\ edge)\ Tree2.t$

harvest dag n roots returns the DAG *roots* enlarged by all nodes in *dag* reachable from n.

val $harvest : t \rightarrow node \rightarrow t \rightarrow t$

harvest_list dag nodes returns the part of the DAG *dag* that is reachable from the *nodes*.

val $harvest\_list : t \rightarrow node\ list \rightarrow t$

size dag returns the number of nodes in the DAG *dag*.

val $size : t \rightarrow int$

eval f mul_edge mul_nodes add null unit root dag interprets the part of *dag* beneath *root* as an algebraic expression:

- each node is evaluated by $f : node \rightarrow \alpha$
- each set of children is evaluated by iterating the binary $mul\_nodes : \alpha \rightarrow \gamma \rightarrow \gamma$ on the values of the nodes, starting from $unit$: γ
- each offspring relation $(node, (edge, children))$ is evaluated by applying $mul\_edge : node \rightarrow edge \rightarrow \gamma \rightarrow \delta$ to *node*, *edge* and the evaluation of *children*.
- all offspring relations of a *node* are combined by iterating the binary $add : \delta \rightarrow \alpha \rightarrow \alpha$ starting from $null : \alpha$

In our applications, we will always have $\alpha = \gamma = \delta$, but the more general type is useful for documenting the relationships. The memoizing variant *eval_memoized f mul_edge mul_nodes add null unit root dag* requires some overhead, but can be more efficient for complex operations.

val $eval : (node \rightarrow \alpha) \rightarrow (node \rightarrow edge \rightarrow \gamma \rightarrow \delta) \rightarrow$
  $(\alpha \rightarrow \gamma \rightarrow \gamma) \rightarrow (\delta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \gamma \rightarrow node \rightarrow t \rightarrow \alpha$
val $eval\_memoized : (node \rightarrow \alpha) \rightarrow (node \rightarrow edge \rightarrow \gamma \rightarrow \delta) \rightarrow$
  $(\alpha \rightarrow \gamma \rightarrow \gamma) \rightarrow (\delta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \gamma \rightarrow node \rightarrow t \rightarrow \alpha$

forest root dag expands the *dag* beneath *root* into the equivalent list of trees *Tree.t*. *children* are represented as list of nodes.

A sterile node n is represented as $Tree.Leaf\ ((n,\ None),\ n)$, cf. page 766. There might be a better way, but we need to change the interface and semantics of *Tree* for this.

val $forest : node \rightarrow t \rightarrow (node \times edge\ option,\ node)\ Tree.t\ list$
val $forest\_memoized : node \rightarrow t \rightarrow (node \times edge\ option,\ node)\ Tree.t\ list$

count_trees n dag returns the number of trees with root n encoded in the DAG *dag*, i. e. $|T(n,D)|$. NB: the current implementation is very naive and can take a *very* long time for moderately sized DAGs that encode a large set of trees.

val $count\_trees : node \rightarrow t \rightarrow int$

end

module $Make\ (F : Forest)$:
  T with type $node = F.node$ and type $edge = F.edge$
  and type $children = F.children$

4.1.3 Graded Sets, Forests & DAGs

A graded ordered[1] set is an ordered set with a map *rank* into another ordered set (often the non-negative integers). Note that it is *not* required that the grading respects the ordering, i. e. $x < y \not\Rightarrow rank\ x < rank\ y$.

[1]We don't appear to have use for graded unordered sets.

Conceptionally, there is some overlap with *Bundle* (cf. section O.1), if we interpret the set of ranks as the base of the bundle and *rank* as the projection π. We might want to unify the structures. But note that in the case of *Bundle*, the intuition is that the base is a subset of the bundle, i. e. each element of the base is an element of a fiber. In the case of a grading, the set of ranks can be completely disjoint from the original set.

```
module type Graded_Ord =
  sig
    include Ord
    module G : Ord
    val rank : t → G.t
  end
```

For all ordered sets, there are two canonical gradings: a *Chaotic* grading that assigns the same rank (e. g. *unit*) to all elements and the *Discrete* grading that uses the identity map as grading.

```
module type Grader = functor (O : Ord) → Graded_Ord with type t = O.t
module Chaotic : Grader
module Discrete : Grader
```

A graded forest is just a forest in which the nodes form a graded ordered set.

Module type substitions for avoiding the repetition here will come with O'Caml 4.13. Until then, we're lucky that the signature is short …

```
module type Graded_Forest =
  sig
    module Nodes : Graded_Ord
    type node = Nodes.t
    type edge
    type children
    type t = edge × children
    val compare : t → t → int
    val for_all : (node → bool) → t → bool
    val fold : (node → α → α) → t → α → α
  end

module type Forest_Grader = functor (G : Grader) → functor (F : Forest) →
  Graded_Forest with type Nodes.t = F.node
  and type node = F.node
  and type edge = F.edge
  and type children = F.children
  and type t = F.t

module Grade_Forest : Forest_Grader
```

Finally, a graded DAG is a DAG in which the nodes form a graded ordered set and the subsets with a given rank can be accessed cheaply.

```
module type Graded =
  sig
    include T
    type rank
    val rank : node → rank
    val ranks : t → rank list
    val min_max_rank : t → rank × rank
    val ranked : rank → t → node list
  end

module Graded (F : Graded_Forest) :
    Graded with type node = F.node and type edge = F.edge
    and type children = F.children and type rank = F.Nodes.G.t

module Test : sig val suite : OUnit.test end
```

4.2 Implementation of DAG

```
module type Ord =
  sig
    type t
    val compare : t → t → int
  end

module type Forest =
  sig
    module Nodes : Ord
    type node = Nodes.t
    type edge
    type children
    type t = edge × children
    val compare : t → t → int
    val for_all : (node → bool) → t → bool
    val fold : (node → α → α) → t → α → α
  end

module type T =
  sig
    type node
    type edge
    type children
    type t
    val empty : t
    val add_node : node → t → t
    val add_offspring : node → edge × children → t → t
    exception Cycle
    val add_offspring_unsafe : node → edge × children → t → t
    val is_node : node → t → bool
    val is_sterile : node → t → bool
    val is_offspring : node → edge × children → t → bool
    val iter_nodes : (node → unit) → t → unit
    val map_nodes : (node → node) → t → t
    val fold_nodes : (node → α → α) → t → α → α
    val iter : (node → edge × children → unit) → t → unit
    val map : (node → node) →
      (node → edge × children → edge × children) → t → t
    val fold : (node → edge × children → α → α) → t → α → α
    val lists : t → (node × (edge × children) list) list
    val dependencies : t → node → (node, edge) Tree2.t
    val harvest : t → node → t → t
    val harvest_list : t → node list → t
    val size : t → int
    val eval : (node → α) → (node → edge → γ → δ) →
      (α → γ → γ) → (δ → α → α) → α → γ → node → t → α
    val eval_memoized : (node → α) → (node → edge → γ → δ) →
      (α → γ → γ) → (δ → α → α) → α → γ → node → t → α
    val forest : node → t → (node × edge option, node) Tree.t list
    val forest_memoized : node → t → (node × edge option, node) Tree.t list
    val count_trees : node → t → int
   end

module type Graded_Ord =
  sig
    include Ord
    module G : Ord
    val rank : t → G.t
  end

module type Grader = functor (O : Ord) → Graded_Ord with type t = O.t
module type Graded_Forest =
  sig
    module Nodes : Graded_Ord
    type node = Nodes.t
    type edge
    type children
    type t = edge × children
    val compare : t → t → int
    val for_all : (node → bool) → t → bool
    val fold : (node → α → α) → t → α → α
  end

module type Forest_Grader = functor (G : Grader) → functor (F : Forest) →
  Graded_Forest with type Nodes.t = F.node
  and type node = F.node
  and type edge = F.edge
  and type children = F.children
  and type t = F.t
```

4.2.1 The Forest Functor

```
module Forest (PT : Tuple.Poly) (N : Ord) (E : Ord) :
    Forest with module Nodes = N and type edge = E.t
    and type node = N.t and type children = N.t PT.t =
  struct
    module Nodes = N
    type edge = E.t
    type node = N.t
    type children = node PT.t
    type t = edge × children

    let compare (edge1, children1) (edge2, children2) =
      let c = PT.compare N.compare children1 children2 in
      if c ≠ 0 then
        c
      else
        E.compare edge1 edge2

    let for_all f (_, nodes) = PT.for_all f nodes
    let fold f (_, nodes) acc = PT.fold_right f nodes acc

  end
```

4.2.2 Gradings

```
module Chaotic (O : Ord) =
  struct
    include O
    module G =
      struct
        type t = unit
        let compare _ _ = 0
      end
    let rank _ = ()
  end

module Discrete (O : Ord) =
  struct
    include O
    module G = O
    let rank x = x
  end
```

```
module Fake_Grading (O : Ord) =
  struct
    include O
    exception Impossible of string
    module G =
      struct
        type t = unit
        let compare _ _ = raise (Impossible "G.compare")
      end
    let rank _ = raise (Impossible "G.compare")
  end

module Grade_Forest (G : Grader) (F : Forest) =
  struct
    module Nodes = G(F.Nodes)
    type node = Nodes.t
    type edge = F.edge
    type children = F.children
    type t = F.t
    let compare = F.compare
    let for_all = F.for_all
    let fold = F.fold
  end
```

A subset of *Map.S*, with graded keys. The map is implemented as a two level map with the outer map from the rank of the key to a map from all key of this rank to the values. Thus we can find query the minimal and maximal ranks and find all keys with a given rank without having to scan the entire map.

```
module type Graded_Map =
  sig
```

We implement the subset of *Map.S* from the standard library that we need in our applications. The semantics is identical to *Map.S* so we don't need to duplicate the documentation. It would be trivial to implement the rest, if we ever need it.

```
    type key
    type α t
    val empty : α t
    val add : key → α → α t → α t
    val find : key → α t → α
    val mem : key → α t → bool
    val iter : (key → α → unit) → α t → unit
    val fold : (key → α → β → β) → α t → β → β
```

Here come the additional functions dealing with the *rank*. All could be implemented by inspecting all keys in a map, but the keeping track of the grading makes them much more efficient.

```
    type rank
```

Return a list of all ranks in a map. The application should not rely on the fact that the list is sorted.

```
    val ranks : α t → rank list
```

Return the minimal and maximal rank in the map, according to the order of *rank*.

```
    val min_max_rank : α t → rank × rank
```

Return all keys with the given *rank*.

```
    val ranked : rank → α t → key list

  end

module type Graded_Map_Maker = functor (O : Graded_Ord) →
  Graded_Map with type key = O.t and type rank = O.G.t
```

Nested $\alpha \to \beta$ *opt* functions cry out for the monadic binding operators introduced by O'Caml 4.08.

```
module Graded_Map (O : Graded_Ord) :
    Graded_Map with type key = O.t and type rank = O.G.t =
  struct
    module M1 = Map.Make(O.G)
    module M2 = Map.Make(O)

    type key = O.t
    type rank = O.G.t

    type (+α) t = α M2.t M1.t

    let empty = M1.empty

    let map2_of_rank rank map1 =
      match M1.find_opt rank map1 with
      | None → M2.empty
      | Some map2 → map2

    let add key data map1 =
      let rank = O.rank key in
      M1.add rank (M2.add key data (map2_of_rank rank map1)) map1

    let find key map1 =
      M2.find key (M1.find (O.rank key) map1)

    let mem key map1 =
      M2.mem key (map2_of_rank (O.rank key) map1)

    let iter f map1 =
      M1.iter (fun rank → M2.iter f) map1

    let fold f map1 acc1 =
      M1.fold (fun rank → M2.fold f) map1 acc1
```

The set of ranks and its minimum and maximum should be maintained explicitely!

```
    module S1 = Set.Make(O.G)

    let ranks map =
      M1.fold (fun key data acc → key :: acc) map [ ]

    let rank_set map =
      M1.fold (fun key data → S1.add key) map S1.empty

    let min_max_rank map =
      let s = rank_set map in
      (S1.min_elt s, S1.max_elt s)

    module S2 = Set.Make(O)

    let keys map =
      M2.fold (fun key data acc → key :: acc) map [ ]

    let sorted_keys map =
      S2.elements (M2.fold (fun key data → S2.add key) map S2.empty)

    let ranked rank map1 =
      keys (map2_of_rank rank map1)

  end
```

4.2.3 The DAG Functor

Currently, we are *not* using the grading in O'Mega. It seemed to be an interesting idea for structuring DAGs, but we have not yet come up with a real use case . . .

```
module Maybe_Graded (GMM : Graded_Map_Maker) (F : Graded_Forest) =
  struct

    module G = F.Nodes.G
```

```
type node = F.node
type rank = G.t
type edge = F.edge
type children = F.children
```

If we get tired of graded DAGs, we just have to replace *Graded_Map* by *Map* here and remove *ranked* below and gain a tiny amount of simplicity and efficiency.

```
module Parents = GMM(F.Nodes)
module Offspring = Set.Make(F)

type t = Offspring.t Parents.t

let rank = F.Nodes.rank
let ranks = Parents.ranks
let min_max_rank = Parents.min_max_rank
let ranked = Parents.ranked

let empty = Parents.empty

let add_node node dag =
  if Parents.mem node dag then
    dag
  else
    Parents.add node Offspring.empty dag

let add_offspring_unsafe node offspring dag =
  let offsprings =
    try Parents.find node dag with Not_found → Offspring.empty in
  Parents.add node (Offspring.add offspring offsprings)
    (F.fold add_node offspring dag)

exception Cycle

let add_offspring node offspring dag =
  if F.for_all (fun n → F.Nodes.compare n node < 0) offspring then
    add_offspring_unsafe node offspring dag
  else
    raise Cycle

let is_node node dag =
  Parents.mem node dag

let is_sterile node dag =
  try
    Offspring.is_empty (Parents.find node dag)
  with
  | Not_found → false

let is_offspring node offspring dag =
  try
    Offspring.mem offspring (Parents.find node dag)
  with
  | Not_found → false

let iter_nodes f dag =
  Parents.iter (fun n _ → f n) dag

let iter f dag =
  Parents.iter (fun node → Offspring.iter (f node)) dag

let map_nodes f dag =
  Parents.fold (fun n → Parents.add (f n)) dag Parents.empty

let map fn fo dag =
  Parents.fold (fun node offspring →
    Parents.add (fn node)
      (Offspring.fold (fun o → Offspring.add (fo node o))
         offspring Offspring.empty)) dag Parents.empty
```

```
let fold_nodes f dag acc =
  Parents.fold (fun n _ → f n) dag acc

let fold f dag acc =
  Parents.fold (fun node → Offspring.fold (f node)) dag acc
```

Note that in it's current incarnation, *fold add_offspring dag empty* copies *only* the fertile nodes, while *fold add_offspring dag* (*fold_nodes add_node dag empty*) includes sterile ones, as does *map* (fun *n* → *n*) (fun *n ec* → *ec*) *dag*.

```
let dependencies dag node =
  let rec dependencies' node' =
    let offspring = Parents.find node' dag in
    if Offspring.is_empty offspring then
      Tree2.leaf node'
    else
      Tree2.cons
        (Offspring.fold
           (fun o acc →
             (fst o,
              node',
              F.fold (fun wf acc' → dependencies' wf :: acc') o []) :: acc)
           offspring [])
  in
  dependencies' node

let lists dag =
  List.sort (fun (n1, _) (n2, _) → F.Nodes.compare n1 n2)
    (Parents.fold (fun node offspring l →
      (node, Offspring.elements offspring) :: l) dag [])

let size dag =
  Parents.fold (fun _ _ n → succ n) dag 0

let rec harvest dag node roots =
  Offspring.fold
    (fun offspring roots' →
      if is_offspring node offspring roots' then
        roots'
      else
        F.fold (harvest dag)
          offspring (add_offspring_unsafe node offspring roots'))
    (Parents.find node dag) (add_node node roots)

let harvest_list dag nodes =
  List.fold_left (fun roots node → harvest dag node roots) empty nodes
```

Build a closure once, so that we can recurse faster:

```
let eval f mule muln add null unit node dag =
  let rec eval' n =
    if is_sterile n dag then
      f n
    else
      Offspring.fold
        (fun (e, _ as offspring) v0 →
          add (mule n e (F.fold muln' offspring unit)) v0)
        (Parents.find n dag) null
  and muln' n = muln (eval' n) in
  eval' node

let count_trees node dag =
  eval (fun _ → 1) (fun _ _ p → p) ( × ) (+) 0 1 node dag

let build_forest evaluator node dag =
  evaluator (fun n → [Tree.leaf (n, None) n])
    (fun n e p → List.map (fun p' → Tree.cons (n, Some e) p') p)
    (fun p1 p2 → Product.fold2 (fun n nl pl → (n :: nl) :: pl) p1 p2 [])
    (@) [] [[]] node dag

let forest = build_forest eval
```

At least for *count_trees*, the memoizing variant *eval_memoized* is considerably slower than direct recursive evaluation with *eval*.

```
    let eval_offspring f mule muln add null unit dag values (node, offspring) =
      let muln' n = muln (Parents.find n values) in
      let v =
        if is_sterile node dag then
          f node
        else
          Offspring.fold
            (fun (e, _ as offspring) v0 →
              add (mule node e (F.fold muln' offspring unit)) v0)
            offspring null
      in
      (v, Parents.add node v values)

    let eval_memoized' f mule muln add null unit dag =
      let result, _ =
        List.fold_left
          (fun (v, values) → eval_offspring f mule muln add null unit dag values)
          (null, Parents.empty)
          (List.sort (fun (n1, _) (n2, _) → F.Nodes.compare n1 n2)
             (Parents.fold
                (fun node offspring l → (node, offspring) :: l) dag [])) in
      result

    let eval_memoized f mule muln add null unit node dag =
      eval_memoized' f mule muln add null unit
        (harvest dag node empty)

    let forest_memoized = build_forest eval_memoized

  end

module type Graded =
  sig
    include T
    type rank
    val rank : node → rank
    val ranks : t → rank list
    val min_max_rank : t → rank × rank
    val ranked : rank → t → node list
  end

module Graded (F : Graded_Forest) = Maybe_Graded(Graded_Map)(F)
```

The following is not a graded map, obviously. But it can pass as one by the typechecker for constructing non-graded DAGs.

```
module Fake_Graded_Map (O : Graded_Ord) :
    Graded_Map with type key = O.t and type rank = O.G.t =
  struct
    module M = Map.Make(O)
    type key = O.t
    type (+α) t = α M.t
    let empty = M.empty
    let add = M.add
    let find = M.find
    let mem = M.mem
    let iter = M.iter
    let fold = M.fold
```

We make sure that the remaining three are never called inside *DAG* and are not visible outside.

```
    type rank = O.G.t
    exception Impossible of string
    let ranks _ = raise (Impossible "ranks")
    let min_max_rank _ = raise (Impossible "min_max_rank")
    let ranked _ _ = raise (Impossible "ranked")
  end
```

We could also have used signature projection with a chaotic or discrete grading, but the *Graded_Map* can cost some efficiency. This is probably not the case for the current simple implementation, but future embellishment can change this. Therefore, the ungraded DAG uses *Map* directly, without overhead.

```
module Make (F : Forest) =
  Maybe_Graded(Fake_Graded_Map)(Grade_Forest(Fake_Grading)(F))
```

If O'Caml had *polymorphic recursion*, we could think of even more elegant implementations unifying nodes and offspring (cf. the generalized tries in [4]).

GADTs to the rescue?

4.2.4 Unit Tests

```
module Test =
  struct

    let random_int_list imax n =
      let imax_plus = succ imax in
      Array.to_list (Array.init n (fun _ → Random.int imax_plus))

    module OInts =
      struct
        type t = int
        let compare = compare
      end

    module GOInts =
      struct
        type t = int
        let compare = compare
        module G =
          struct
            type t = int
            let compare = compare
          end
        let rank i = i mod 100
      end

    module GM = Graded_Map(GOInts)

    let int_list_to_string l =
      ThoList.to_string string_of_int l

    let int_list2_to_string l =
      ThoList.to_string int_list_to_string l

    let int_pair_to_string (i1, i2) =
      int_list_to_string [i1; i2]

    let uniq l =
      ThoList.uniq (List.sort compare l)

    open OUnit

    let assert_equal_int_pair p1 p2 =
      assert_equal ~printer : int_pair_to_string p1 p2

    let assert_equal_unsorted_int_list l1 l2 =
      assert_equal ~printer : int_list_to_string
        (List.sort compare l1)
        (List.sort compare l2)

    let assert_equal_unsorted_int_list_ignore_duplicates l1 l2 =
      assert_equal ~printer : int_list_to_string (uniq l1) (uniq l2)

    let squares n =
      let data =
        List.map (fun i → (i, i × i)) (random_int_list 10000 n) in
      let map =
        List.fold_left (fun acc (i, s) → GM.add i s acc) GM.empty data in
      (data, map)

    let suite_graded_map =

      "Graded_Map" >:::
        [ "ranks" >::
            (fun () →
              let data, graded_map = squares 100 in
              assert_equal_unsorted_int_list
                (uniq (List.map (fun (i, _) → GOInts.rank i) data))
                (GM.ranks graded_map));

          "min_max_rank" >::
            (fun () →
              match squares 100 with
              | [], _ → failwith "empty␣test␣data"
              | (r0, _) :: data, graded_map →
                  assert_equal_int_pair
                    (List.fold_left
                       (fun (r_min, r_max) (i, _) →
                         let r = GOInts.rank i in
                         (min r r_min, max r r_max))
                       (GOInts.rank r0, GOInts.rank r0) data)
                    (GM.min_max_rank graded_map)) ]
```

We should add more unit tests, time permitting.

```
  let suite =
    "DAG" >:::
      [suite_graded_map]

end
```

—5—
Momenta

5.1 Interface of Momentum

Model the finite combinations

$$p = \sum_{n=1}^{k} c_k \bar p_n, \qquad \text{(with } c_k \in \{0,1\}\text{)} \tag{5.1}$$

of n_{in} incoming and $k - n_{\text{in}}$ outgoing momenta p_n

$$\bar p_n = \begin{cases} -p_n & \text{for } 1 \le n \le n_{\text{in}} \\ p_n & \text{for } n_{\text{in}}+1 \le n \le k \end{cases} \tag{5.2}$$

where momentum is conserved

$$\sum_{n=1}^{k} \bar p_n = 0 \tag{5.3}$$

below, we need the notion of 'rank' and 'dimension':

$$\mathit{dim}(p) = k \tag{5.4a}$$

$$\mathit{rank}(p) = \sum_{n=1}^{k} c_k \tag{5.4b}$$

where 'dimension' is *not* the dimension of the underlying space-time, of course.

```
module type T =
  sig
    type t
```

Constructor: $(k, N) \to p = \sum_{n\in N} \bar p_n$ and $k = \mathit{dim}(p)$ is the *overall* number of independent momenta, while $\mathit{rank}(p) = |N|$ is the number of momenta in p. It would be possible to fix *dim* as a functor argument instead. This might be slightly faster and allow a few more compile time checks, but would be much more tedious to use, since the number of particles will be chosen at runtime.

```
    val of_ints : int → int list → t
```

No two indices may be the same. Implementions of *of_ints* can either raise the exception *Duplicate* or ignore the duplicate, but implementations of *add* are required to raise *Duplicate*.

```
    exception Duplicate of int
```

Raise *Range* iff $n > k$:

```
    exception Range of int
```

Binary oparations require that both momenta have the same dimension. *Mismatch* is raised if this condition is violated.

```
    exception Mismatch of string × t × t
```

Negative is raised if the result of *sub* is undefined.

```
    exception Negative
```

The inverses of the constructor (we have *rank p = List.length (to_ints p)*, but *rank* might be more efficient):

```
    val to_ints : t → int list
```

```
val dim : t → int
val rank : t → int
```

Shortcuts: *singleton* d p = *of_ints* d $[p]$ and *zero* d = *of_ints* d $[\,]$:

```
val singleton : int → int → t
val zero : int → t
```

An arbitrary total order, with the condition $\mathit{rank}(p_1) < \mathit{rank}(p_2) \Rightarrow p_1 < p_2$.

```
val compare : t → t → int
```

Use momentum conservation to construct the negative momentum with positive coefficients:

```
val neg : t → t
```

Return the momentum or its negative, whichever has the lower rank. NB: the present implementation does *not* guarantee that

$$\mathrm{abs}p = \mathrm{abs}q \Longleftrightarrow p = p \vee p = -q \tag{5.5}$$

for momenta with rank = dim/2.

```
val abs : t → t
```

Add and subtract momenta. This can fail, since the coefficients c_k must me either 0 or 1.

```
val add : t → t → t
val sub : t → t → t
```

Once more, but not raising exceptions this time:

```
val try_add : t → t → t option
val try_sub : t → t → t option
```

Not the total order provided by *compare*, but set inclusion of non-zero coefficients instead:

```
val less : t → t → bool
val lesseq : t → t → bool
```

$p_1 + (\pm p_2) + (\pm p_3) = 0$

```
val try_fusion : t → t → t → (bool × bool) option
```

A textual representation for debugging:

```
val to_string : t → string
```

split i n p splits $\bar p_i$ into n momenta $\bar p_i \to \bar p_i + \bar p_{i+1} + \ldots + \bar p_{i+n-1}$ and makes room via $\bar p_{j>i} \to \bar p_{j+n-1}$. This is used for implementating cascade decays, like combining

$$\mathrm{e}^+(p_1)\mathrm{e}^-(p_2) \to \mathrm{W}^-(p_3)\nu_\mathrm{e}(p_4)\mathrm{e}^+(p_5) \tag{5.6a}$$
$$\mathrm{W}^-(p_3) \to \mathrm{d}(p_3')\bar{\mathrm{u}}(p_4') \tag{5.6b}$$

to

$$\mathrm{e}^+(p_1)\mathrm{e}^-(p_2) \to \mathrm{d}(p_3)\bar{\mathrm{u}}(p_4)\nu_\mathrm{e}(p_5)\mathrm{e}^+(p_6) \tag{5.7}$$

in narrow width approximation for the W^-.

```
val split : int → int → t → t
```

5.1.1 Scattering Kinematics

From here on, we assume scattering kinematics $\{1,2\} \to \{3,4,\ldots\}$, i. e. $n_{\mathrm{in}} = 2$.

Since functions like *timelike* can be used for decays as well (in which case they must *always* return true, the representation—and consequently the constructors—should be extended by a flag discriminating between the two cases!

```
module Scattering :
    sig
```

Test if the momentum is an incoming one: $p = \bar p_1 \vee p = \bar p_2$

```
        val incoming : t → bool
```

$p = \bar p_3 \lor p = \bar p_4 \lor \ldots$

```
        val outgoing : t → bool
```

$p^2 \geq 0$. NB: *par abus de langange*, we report the incoming individual momenta as spacelike, instead as timelike. This will be useful for phasespace constructions below.

```
        val timelike : t → bool
```

$p^2 \leq 0$. NB: the simple algebraic criterion can be violated for heavy initial state particles.

```
        val spacelike : t → bool
```

$p = \bar p_1 + \bar p_2$

```
        val s_channel_in : t → bool
```

$p = \bar p_3 + \bar p_4 + \ldots + \bar p_n$

```
        val s_channel_out : t → bool
```

$p = \bar p_1 + \bar p_2 \lor p = \bar p_3 + \bar p_4 + \ldots + \bar p_n$

```
        val s_channel : t → bool
```

$\bar p_1 + \bar p_2 \to \bar p_3 + \bar p_4 + \ldots + \bar p_n$

```
        val flip_s_channel_in : t → t
     end
```

5.1.2 Decay Kinematics

```
  module Decay :
    sig
```

Test if the momentum is an incoming one: $p = \bar p_1$

```
        val incoming : t → bool
```

$p = \bar p_2 \lor p = \bar p_3 \lor \ldots$

```
        val outgoing : t → bool
```

$p^2 \geq 0$. NB: here, we report the incoming individual momenta as timelike.

```
        val timelike : t → bool
```

$p^2 \leq 0$.

```
        val spacelike : t → bool

    end

 end
```

```
module Lists : T
module Bits : T
module Default : T
```

Wolfgang's funny tree codes:

$$(2^n, 2^{n-1}) \to (1, 2, 4, \ldots, 2^{n-2}) \tag{5.8}$$

```
module type Whizard =
 sig
   type t
   val of_momentum : t → int
   val to_momentum : int → int → t
 end
```

```
module ListsW : Whizard with type t = Lists.t
module BitsW : Whizard with type t = Bits.t
module DefaultW : Whizard with type t = Default.t
```

5.2 Implementation of Momentum

```
module type T =
  sig
    type t
    val of_ints : int → int list → t
    exception Duplicate of int
    exception Range of int
    exception Mismatch of string × t × t
    exception Negative
    val to_ints : t → int list
    val dim : t → int
    val rank : t → int
    val singleton : int → int → t
    val zero : int → t
    val compare : t → t → int
    val neg : t → t
    val abs : t → t
    val add : t → t → t
    val sub : t → t → t
    val try_add : t → t → t option
    val try_sub : t → t → t option
    val less : t → t → bool
    val lesseq : t → t → bool
    val try_fusion : t → t → t → (bool × bool) option
    val to_string : t → string
    val split : int → int → t → t
    module Scattering :
        sig
          val incoming : t → bool
          val outgoing : t → bool
          val timelike : t → bool
          val spacelike : t → bool
          val s_channel_in : t → bool
          val s_channel_out : t → bool
          val s_channel : t → bool
          val flip_s_channel_in : t → t
        end
    module Decay :
        sig
          val incoming : t → bool
          val outgoing : t → bool
          val timelike : t → bool
          val spacelike : t → bool
        end
  end
```

5.2.1 Lists of Integers

The first implementation (as part of *Fusion*) was based on sorted lists, because I did not want to preclude the use of more general indices that integers. However, there's probably not much use for this generality (the indices are typically generated automatically and integer are the most natural choice) and it is no longer supported. by the current signature. Thus one can also use the more efficient implementation based on bitvectors below.

```
module Lists =
  struct

    type t = { d : int; r : int; p : int list }

    exception Range of int
    exception Duplicate of int
```

```
let rec check d = function
  | p1 :: p2 :: _ when p2 ≤ p1 → raise (Duplicate p1)
  | p1 :: (p2 :: _ as rest) → check d rest
  | [p] when p < 1 ∨ p > d → raise (Range p)
  | [p] → ()
  | [] → ()

let of_ints d p =
  let p′ = List.sort compare p in
  check d p′;
  { d = d; r = List.length p; p = p′ }

let to_ints p = p.p
let dim p = p.d
let rank p = p.r
let zero d = { d = d; r = 0; p = [] }
let singleton d p = { d = d; r = 1; p = [p] }

let to_string p =
  "[" ^ String.concat "," (List.map string_of_int p.p) ^
  "/" ^ string_of_int p.r ^ "/" ^ string_of_int p.d ^ "]"

exception Mismatch of string × t × t
let mismatch s p1 p2 = raise (Mismatch (s, p1, p2))

let matching f s p1 p2 =
  if p1.d = p2.d then
    f p1 p2
  else
    mismatch s p1 p2

let compare p1 p2 =
  if p1.d = p2.d then begin
    let c = compare p1.r p2.r in
    if c ≠ 0 then
      c
    else
      compare p1.p p2.p
  end else
    mismatch "compare" p1 p2

let rec neg′ d i = function
  | [] →
      if i ≤ d then
        i :: neg′ d (succ i) []
      else
        []
  | i′ :: rest as p →
      if i′ > d then
        failwith "Integer_List.neg:␣internal␣error"
      else if i′ = i then
        neg′ d (succ i) rest
      else
        i :: neg′ d (succ i) p

let neg p = { d = p.d; r = p.d − p.r; p = neg′ p.d 1 p.p }

let abs p =
  if 2 × p.r > p.d then
    neg p
  else
    p

let rec add′ p1 p2 =
  match p1, p2 with
  | [], p → p
  | p, [] → p
  | x1 :: p1', x2 :: p2' →
      if x1 < x2 then
        x1 :: add' p1' p2
      else if x2 < x1 then
        x2 :: add' p1 p2'
      else
        raise (Duplicate x1)

let add p1 p2 =
  if p1.d = p2.d then
    { d = p1.d; r = p1.r + p2.r; p = add' p1.p p2.p }
  else
    mismatch "add" p1 p2

let rec try_add' d r acc p1 p2 =
  match p1, p2 with
  | [], p → Some ({ d = d; r = r; p = List.rev_append acc p })
  | p, [] → Some ({ d = d; r = r; p = List.rev_append acc p })
  | x1 :: p1', x2 :: p2' →
      if x1 < x2 then
        try_add' d r (x1 :: acc) p1' p2
      else if x2 < x1 then
        try_add' d r (x2 :: acc) p1 p2'
      else
        None

let try_add p1 p2 =
  if p1.d = p2.d then
    try_add' p1.d (p1.r + p2.r) [] p1.p p2.p
  else
    mismatch "try_add" p1 p2

exception Negative

let rec sub' p1 p2 =
  match p1, p2 with
  | p, [] → p
  | [], _ → raise Negative
  | x1 :: p1', x2 :: p2' →
      if x1 < x2 then
        x1 :: sub' p1' p2
      else if x1 = x2 then
        sub' p1' p2'
      else
        raise Negative

let rec sub p1 p2 =
  if p1.d = p2.d then begin
    if p1.r ≥ p2.r then
      { d = p1.d; r = p1.r − p2.r; p = sub' p1.p p2.p }
    else
      neg (sub p2 p1)
   end else
    mismatch "sub" p1 p2

let rec try_sub' d r acc p1 p2 =
  match p1, p2 with
  | p, [] → Some ({ d = d; r = r; p = List.rev_append acc p })
  | [], _ → None
  | x1 :: p1', x2 :: p2' →
      if x1 < x2 then
        try_sub' d r (x1 :: acc) p1' p2
      else if x1 = x2 then
        try_sub' d r acc p1' p2'
      else
        None

let try_sub p1 p2 =
  if p1.d = p2.d then begin
    if p1.r ≥ p2.r then
      try_sub' p1.d (p1.r − p2.r) [] p1.p p2.p
    else
      match try_sub' p1.d (p2.r − p1.r) [] p2.p p1.p with
      | None → None
      | Some p → Some (neg p)
  end else
    mismatch "try_sub" p1 p2

let rec less' equal p1 p2 =
  match p1, p2 with
  | [], [] → ¬ equal
  | [], _ → true
  | x1 :: _ , [] → false
  | x1 :: p1', x2 :: p2' when x1 = x2 → less' equal p1' p2'
  | x1 :: p1', x2 :: p2' → less' false p1 p2'

let less p1 p2 =
  if p1.d = p2.d then
    less' true p1.p p2.p
  else
    mismatch "sub" p1 p2

let rec lesseq' p1 p2 =
  match p1, p2 with
  | [], _ → true
  | x1 :: _ , [] → false
  | x1 :: p1', x2 :: p2' when x1 = x2 → lesseq' p1' p2'
  | x1 :: p1', x2 :: p2' → lesseq' p1 p2'

let lesseq p1 p2 =
  if p1.d = p2.d then
    lesseq' p1.p p2.p
  else
    mismatch "lesseq" p1 p2

module Scattering =
  struct

    let incoming p =
      if p.r = 1 then
        match p.p with
        | [1] | [2] → true
        | _ → false
      else
        false

    let outgoing p =
      if p.r = 1 then
        match p.p with
        | [1] | [2] → false
        | _ → true
      else
        false

    let s_channel_in p =
      match p.p with
      | [1; 2] → true
      | _ → false

    let rec s_channel_out' d i = function
```

```
        | [] → $i = succ\ d$
        | $i'$ :: $p$ when $i' = i$ → $s\_channel\_out'\ d\ (succ\ i)\ p$
        | _ → false

      let $s\_channel\_out\ p$ =
        match $p.p$ with
        | 3 :: $p'$ → $s\_channel\_out'\ p.d\ 4\ p'$
        | _ → false

      let $s\_channel\ p = s\_channel\_in\ p \lor s\_channel\_out\ p$

      let $timelike\ p$ =
        match $p.p$ with
        | $p1$ :: $p2$ :: _ → $p1 > 2 \lor (p1 = 1 \land p2 = 2)$
        | $p1$ :: _ → $p1 > 2$
        | [] → false

      let $spacelike\ p = \neg\ (timelike\ p)$

      let $flip\_s\_channel\_in\ p$ =
        if $s\_channel\_in\ p$ then
          $neg\ (of\_ints\ p.d\ [1;2])$
        else
          $p$

    end

  module $Decay$ =
    struct

      let $incoming\ p$ =
        if $p.r = 1$ then
          match $p.p$ with
          | [1] → true
          | _ → false
        else
          false

      let $outgoing\ p$ =
        if $p.r = 1$ then
          match $p.p$ with
          | [1] → false
          | _ → true
        else
          false

      let $timelike\ p$ =
        match $p.p$ with
        | [1] → true
        | $p1$ :: _ → $p1 > 1$
        | [] → false

      let $spacelike\ p = \neg\ (timelike\ p)$

    end

  let $test\_sum\ p\ inv1\ p1\ inv2\ p2$ =
    if $p.d = p1.d$ then begin
      if $p.d = p2.d$ then begin
        match (if $inv1$ then $try\_add$ else $try\_sub$) $p\ p1$ with
        | $None$ → false
        | $Some\ p'$ →
            begin match (if $inv2$ then $try\_add$ else $try\_sub$) $p'\ p2$ with
            | $None$ → false
            | $Some\ p''$ → $p''.r = 0 \lor p''.r = p.d$
            end
      end else
        $mismatch$ "test_sum" $p\ p2$
    end else
      mismatch "test_sum" p p1

  let try_fusion p p1 p2 =
    if test_sum p false p1 false p2 then
      Some (false, false)
    else if test_sum p true p1 false p2 then
      Some (true, false)
    else if test_sum p false p1 true p2 then
      Some (false, true)
    else if test_sum p true p1 true p2 then
      Some (true, true)
    else
      None

  let split i n p =
    let n' = n - 1 in
    let rec split' head = function
      | [] → (p.r, List.rev head)
      | i1 :: ilist →
          if i1 < i then
            split' (i1 :: head) ilist
          else if i1 > i then
            (p.r, List.rev_append head (List.map ((+) n') (i1 :: ilist)))
          else
            (p.r + n',
             List.rev_append head
               ((ThoList.range i1 (i1 + n')) @ (List.map ((+) n') ilist))) in
    let r', p' = split' [] p.p in
    { d = p.d + n'; r = r'; p = p' }

end
```

5.2.2 Bit Fiddlings

Bit vectors are popular in Fortran based implementations [1, 2, 11] and can be more efficient. In particular, when all infomation is packed into a single integer, much of the memory overhead is reduced.

```
module Bits =
  struct

    type t = int
```

Bits 1...21 are used as a bitvector, indicating whether a particular momentum is included. Bits 22...26 represent the numbers of bits set in bits 1...21 and bits 27...31 denote the maximum number of momenta.

```
    let mask n = (1 lsl n) - 1
    let mask2 = mask 2
    let mask5 = mask 5
    let mask21 = mask 21

    let maskd = mask5 lsl 26
    let maskr = mask5 lsl 21
    let maskb = mask21

    let dim0 p = p land maskd
    let rank0 p = p land maskr
    let bits0 p = p land maskb

    let dim p = (dim0 p) lsr 26
    let rank p = (rank0 p) lsr 21
    let bits p = bits0 p

    let drb0 d r b = d lor r lor b
    let drb d r b = d lsl 26 lor r lsl 21 lor b
```

For a 64-bit architecture, the corresponding sizes could be increased to $1 \ldots 51$, $52 \ldots 57$, and $58 \ldots 63$. However, the combinatorical complexity will have killed us long before we can reach these values.

```
exception Range of int
exception Duplicate of int

exception Mismatch of string × t × t
let mismatch s p1 p2 = raise (Mismatch (s, p1, p2))

let of_ints d p =
  let r = List.length p in
  if d ≤ 21 ∧ r ≤ 21 then begin
    List.fold_left (fun b p′ →
      if p′ ≤ d then
        b lor (1 lsl (pred p′))
      else
        raise (Range p′)) (drb d r 0) p
  end else
    raise (Range r)

let zero d = drb d 0 0

let singleton d p = drb d 1 (1 lsl (pred p))

let rec to_ints′ acc p b =
  if b = 0 then
    List.rev acc
  else if (b land 1) = 1 then
    to_ints′ (p :: acc) (succ p) (b lsr 1)
  else
    to_ints′ acc (succ p) (b lsr 1)

let to_ints p = to_ints′ [] 1 (bits p)

let to_string p =
  "[" ^ String.concat "," (List.map string_of_int (to_ints p)) ^
  "/" ^ string_of_int (rank p) ^ "/" ^ string_of_int (dim p) ^ "]"

let compare p1 p2 =
  if dim0 p1 = dim0 p2 then begin
    let c = compare (rank0 p1) (rank0 p2) in
    if c ≠ 0 then
      c
    else
      compare (bits p1) (bits p2)
  end else
    mismatch "compare" p1 p2

let neg p =
  let d = dim p and r = rank p in
  drb d (d − r) ((mask d) land (lnot p))

let abs p =
  if 2 × (rank p) > dim p then
    neg p
  else
    p

let add p1 p2 =
  let d1 = dim0 p1 and d2 = dim0 p2 in
  if d1 = d2 then begin
    let b1 = bits p1 and b2 = bits p2 in
    if b1 land b2 = 0 then
      drb0 d1 (rank0 p1 + rank0 p2) (b1 lor b2)
    else
      raise (Duplicate 0)
  end else
```

```
    mismatch "add" p1 p2

exception Negative

let rec sub p1 p2 =
  let d1 = dim0 p1 and d2 = dim0 p2 in
  if d1 = d2 then begin
    let r1 = rank0 p1 and r2 = rank0 p2 in
    if r1 ≥ r2 then begin
      let b1 = bits p1 and b2 = bits p2 in
      if b1 lor b2 = b1 then
        drb0 d1 (r1 − r2) (b1 lxor b2)
      else
        raise Negative
    end else
      neg (sub p2 p1)
  end else
    mismatch "sub" p1 p2

let try_add p1 p2 =
  let d1 = dim0 p1 and d2 = dim0 p2 in
  if d1 = d2 then begin
    let b1 = bits p1 and b2 = bits p2 in
    if b1 land b2 = 0 then
      Some (drb0 d1 (rank0 p1 + rank0 p2) (b1 lor b2))
    else
      None
  end else
    mismatch "try_add" p1 p2

let rec try_sub p1 p2 =
  let d1 = dim0 p1 and d2 = dim0 p2 in
  if d1 = d2 then begin
    let r1 = rank0 p1 and r2 = rank0 p2 in
    if r1 ≥ r2 then begin
      let b1 = bits p1 and b2 = bits p2 in
      if b1 lor b2 = b1 then
        Some (drb0 d1 (r1 − r2) (b1 lxor b2))
      else
        None
    end else
      begin match try_sub p2 p1 with
      | Some p → Some (neg p)
      | None → None
      end
  end else
    mismatch "sub" p1 p2

let lesseq p1 p2 =
  let d1 = dim0 p1 and d2 = dim0 p2 in
  if d1 = d2 then begin
    let r1 = rank0 p1 and r2 = rank0 p2 in
    if r1 ≤ r2 then begin
      let b1 = bits p1 and b2 = bits p2 in
      b1 lor b2 = b2
    end else
      false
  end else
    mismatch "less" p1 p2

let less p1 p2 = p1 ≠ p2 ∧ lesseq p1 p2

let mask_in1 = 1
let mask_in2 = 2
let mask_in = mask_in1 lor mask_in2
```

```
module Scattering =
  struct

    let incoming p =
      rank p = 1 ∧ (mask_in land p ≠ 0)

    let outgoing p =
      rank p = 1 ∧ (mask_in land p = 0)

    let timelike p =
      (rank p > 0 ∧ (mask_in land p = 0)) ∨ (bits p = mask_in)

    let spacelike p =
      (rank p > 0) ∧ ¬ (timelike p)

    let s_channel_in p =
      bits p = mask_in

    let s_channel_out p =
      rank p > 0 ∧ (mask_in lxor p = 0)

    let s_channel p =
      s_channel_in p ∨ s_channel_out p

    let flip_s_channel_in p =
      if s_channel_in p then
        neg p
      else
        p

  end

module Decay =
  struct

    let incoming p =
      rank p = 1 ∧ (mask_in1 land p = mask_in1)

    let outgoing p =
      rank p = 1 ∧ (mask_in1 land p = 0)

    let timelike p =
      incoming p ∨ (rank p > 0 ∧ mask_in1 land p = 0)

    let spacelike p =
      ¬ (timelike p)

  end

let test_sum p inv1 p1 inv2 p2 =
  let d = dim p in
  if d = dim p1 then begin
    if d = dim p2 then begin
      match (if inv1 then try_add else try_sub) p p1 with
      | None → false
      | Some p′ →
          begin match (if inv2 then try_add else try_sub) p′ p2 with
          | None → false
          | Some p″ →
              let r = rank p″ in
              r = 0 ∨ r = d
          end
    end else
      mismatch "test_sum" p p2
  end else
    mismatch "test_sum" p p1

let try_fusion p p1 p2 =
  if test_sum p false p1 false p2 then
    Some (false, false)
```

```
      else if test_sum p true p1 false p2 then
        Some (true, false)
      else if test_sum p false p1 true p2 then
        Some (false, true)
      else if test_sum p true p1 true p2 then
        Some (true, true)
      else
        None
```

First create a gap of size $n - 1$ and subsequently fill it if and only if the bit i was set.

```
    let split i n p =
      let delta_d = n - 1
      and b = bits p in
      let mask_low = mask (pred i)
      and mask_i = 1 lsl (pred i)
      and mask_high = lnot (mask i) in
      let b_low = mask_low land b
      and b_med, delta_r =
        if mask_i land b ≠ 0 then
          ((mask n) lsl (pred i), delta_d)
        else
          (0, 0)
      and b_high =
        if delta_d > 0 then
          (mask_high land b) lsl delta_d
        else if delta_d = 0 then
          mask_high land b
        else
          (mask_high land b) lsr (−delta_d) in
      drb (dim p + delta_d) (rank p + delta_r) (b_low lor b_med lor b_high)

  end
```

5.2.3 Whizard

```
module type Whizard =
  sig
    type t
    val of_momentum : t → int
    val to_momentum : int → int → t
  end

module BitsW =
  struct
    type t = Bits.t
    open Bits (* NB: this includes the internal functions not in T! *)

    let of_momentum p =
      let d = dim p in
      let bit_in1 = 1 land p
      and bit_in2 = 1 land (p lsr 1)
      and bits_out = ((mask d) land p) lsr 2 in
      bits_out lor (bit_in1 lsl (d − 1)) lor (bit_in2 lsl (d − 2))

    let rec count_non_zero' acc i last b =
      if i > last then
        acc
      else if (1 lsl (pred i)) land b = 0 then
        count_non_zero' acc (succ i) last b
      else
        count_non_zero' (succ acc) (succ i) last b

    let count_non_zero first last b =
      count_non_zero' 0 first last b

    let to_momentum d w =
      let bit_in1 = 1 land (w lsr (d - 1))
      and bit_in2 = 1 land (w lsr (d - 2))
      and bits_out = (mask (d - 2)) land w in
      let b = (bits_out lsl 2) lor bit_in1 lor (bit_in2 lsl 1) in
      drb d (count_non_zero 1 d b) b

  end
```

The following would be a tad more efficient, if coded directly, but there's no point in wasting effort on this.

```
module ListsW =
  struct
    type t = Lists.t
    let of_momentum p =
      BitsW.of_momentum (Bits.of_ints p.Lists.d p.Lists.p)
    let to_momentum d w =
      Lists.of_ints d (Bits.to_ints (BitsW.to_momentum d w))
  end
```

5.2.4 Suggesting a Default Implementation

Lists is better tested, but the more recent *Bits* appears to work as well and is *much* more efficient, resulting in a relative factor of better than 2. This performance ratio is larger than I had expected and we are not likely to reach its limit of 21 independent vectors anyway.

```
module Default = Bits
module DefaultW = BitsW
```

—6— Cascades

6.1 Interface of Cascade_syntax

```
type ('flavor, 'p, 'constant) t =
  | True
  | False
  | On_shell of 'flavor list × 'p
  | On_shell_not of 'flavor list × 'p
  | Off_shell of 'flavor list × 'p
  | Off_shell_not of 'flavor list × 'p
  | Gauss of 'flavor list × 'p
  | Gauss_not of 'flavor list × 'p
  | Any_flavor of 'p
  | And of ('flavor, 'p, 'constant) t list
  | X_Flavor of 'flavor list
  | X_Vertex of 'constant list × 'flavor list list
```

```
val mk_true : unit → ('flavor, 'p, 'constant) t
val mk_false : unit → ('flavor, 'p, 'constant) t
val mk_on_shell : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_on_shell_not : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_off_shell : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_off_shell_not : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_gauss : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_gauss_not : 'flavor list → 'p → ('flavor, 'p, 'constant) t
val mk_any_flavor : 'p → ('flavor, 'p, 'constant) t
val mk_and : ('flavor, 'p, 'constant) t →
  ('flavor, 'p, 'constant) t → ('flavor, 'p, 'constant) t
val mk_x_flavor : 'flavor list → ('flavor, 'p, 'constant) t
val mk_x_vertex : 'constant list → 'flavor list list →
  ('flavor, 'p, 'constant) t
```

```
val to_string : ('flavor → string) → ('p → string) →
  ('constant → string) → ('flavor, 'p, 'constant) t → string
```

```
exception Syntax_Error of string × int × int
```

6.2 Implementation of Cascade_syntax

Concerning the Gaussian propagators, we admit the following: In principle, they would allow for flavor sums like the off-shell lines, but for all practical purposes they are used only for determining the significance of a specified intermediate state. So we select them in the same manner as on-shell states.
False is probably redundant.

```
type ('flavor, 'p, 'constant) t =
  | True
  | False
  | On_shell of 'flavor list × 'p
  | On_shell_not of 'flavor list × 'p
  | Off_shell of 'flavor list × 'p
  | Off_shell_not of 'flavor list × 'p
  | Gauss of 'flavor list × 'p
  | Gauss_not of 'flavor list × 'p
  | Any_flavor of 'p
  | And of ('flavor, 'p, 'constant) t list
  | X_Flavor of 'flavor list
  | X_Vertex of 'constant list × 'flavor list list

let mk_true () = True
let mk_false () = False
let mk_on_shell f p = On_shell (f, p)
let mk_on_shell_not f p = On_shell_not (f, p)
let mk_off_shell f p = Off_shell (f, p)
let mk_off_shell_not f p = Off_shell_not (f, p)
let mk_gauss f p = Gauss (f, p)
let mk_gauss_not f p = Gauss_not (f, p)
let mk_any_flavor p = Any_flavor p

let mk_and c1 c2 =
  match c1, c2 with
  | c, True | True, c → c
  | c, False | False, c → False
  | And cs, And cs' → And (cs @ cs')
  | And cs, c | c, And cs → And (c :: cs)
  | c, c' → And [c; c']

let mk_x_flavor f = X_Flavor f
let mk_x_vertex c fs = X_Vertex (c, fs)

let to_string flavor_to_string momentum_to_string coupling_to_string cascades =
  let flavors_to_string fs =
    String.concat ":" (List.map flavor_to_string fs)
  and couplings_to_string cs =
    String.concat ":" (List.map coupling_to_string cs) in
  let rec to_string' = function
    | True → "true"
    | False → "false"
    | On_shell (fs, p) →
        momentum_to_string p ^ "␣=␣" ^ flavors_to_string fs
    | On_shell_not (fs, p) →
        momentum_to_string p ^ "␣=␣!" ^ flavors_to_string fs
    | Off_shell (fs, p) →
        momentum_to_string p ^ "␣~␣" ^ flavors_to_string fs
    | Off_shell_not (fs, p) →
        momentum_to_string p ^ "␣~␣!" ^ flavors_to_string fs
    | Gauss (fs, p) →
        momentum_to_string p ^ "␣#␣" ^ flavors_to_string fs
    | Gauss_not (fs, p) →
        momentum_to_string p ^ "␣#␣!" ^ flavors_to_string fs
    | Any_flavor p →
        momentum_to_string p ^ "␣~␣?"
    | And cs →
        String.concat "␣&&␣" (List.map (fun c → "(" ^ to_string' c ^ ")") cs)
    | X_Flavor fs →
        "!" ^ String.concat ":" (List.map flavor_to_string fs)
    | X_Vertex (cs, fss) →
        "^" ^ couplings_to_string cs ^
        "[" ^ (String.concat "," (List.map flavors_to_string fss)) ^ "]"
  in
  to_string' cascades

let int_list_to_string p =
  String.concat "+" (List.map string_of_int (List.sort compare p))

exception Syntax_Error of string × int × int
```

6.3 Lexer

```
{
open Cascade_parser
let unquote s =
  String.sub s 1 (String.length s − 2)
}
```

```
let digit = ['0'−'9']
let upper = ['A'−'Z']
let lower = ['a'−'z']
let char = upper | lower
let white = [' ' '\t' '\n']
```

We use a very liberal definition of strings for flavor names.

```
rule token = parse
    white { token lexbuf } (* skip blanks *)
  | '%' [^'\n']* '\n'
                  { token lexbuf } (* skip comments *)
  | digit+ { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | '+' { PLUS }
  | ':' { COLON }
  | '~' { OFFSHELL }
  | '=' { ONSHELL }
  | '#' { GAUSS }
  | '!' { NOT }
  | '&' '&'? { AND }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '^' { HAT }
  | ',' { COMMA }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | char [^ ' ' '\t' '\n' '&' '(' ')' '[' ']' ':' ',' ]*
                  { STRING (Lexing.lexeme lexbuf) }
  | '"' [^ '"']* '"'
                  { STRING (unquote (Lexing.lexeme lexbuf)) }
  | eof { END }
```

6.4 Parser

Header

```
open Cascade_syntax
let parse_error msg =
  raise (Syntax_Error (msg, symbol_start (), symbol_end ()))
```

Token declarations

```
%token < string > STRING
%token < int > INT
%token LPAREN RPAREN LBRACKET RBRACKET
%token AND PLUS COLON COMMA NOT HAT
%token ONSHELL OFFSHELL GAUSS
%token END
%left AND
```

%left $PLUS$ $COLON$ $COMMA$
%left NOT HAT

%start *main*
%type < (*string*, *int list*, *string*) *Cascade_syntax.t* > *main*

Grammar rules

```
main ::=
    END { mk_true () }
  | cascades END { $1 }

cascades ::=
    exclusion { $1 }
  | vertex { $1 }
  | cascade { $1 }
  | LPAREN cascades RPAREN { $2 }
  | cascades AND cascades { mk_and $1 $3 }

exclusion ::=
    NOT string_list { mk_x_flavor $2 }

vertex ::=
    HAT string_list { mk_x_vertex $2 [] }
  | HAT string_list LBRACKET RBRACKET
                                          { mk_x_vertex $2 [] }
  | HAT LBRACKET string_lists RBRACKET
                                          { mk_x_vertex [] $3 }
  | HAT string_list LBRACKET string_lists RBRACKET
                                          { mk_x_vertex $2 $4 }

cascade ::=
    momentum_list { mk_any_flavor $1 }
  | momentum_list ONSHELL string_list
                                          { mk_on_shell $3 $1 }
  | momentum_list ONSHELL NOT string_list
                                          { mk_on_shell_not $4 $1 }
  | momentum_list OFFSHELL string_list
                                          { mk_off_shell $3 $1 }
  | momentum_list OFFSHELL NOT string_list
                                          { mk_off_shell_not $4 $1 }
  | momentum_list GAUSS string_list { mk_gauss $3 $1 }
  | momentum_list GAUSS NOT string_list
                                          { mk_gauss_not $4 $1 }

momentum_list ::=
  | momentum { [$1] }
  | momentum_list PLUS momentum { $3 :: $1 }

momentum ::=
    INT { $1 }

string_list ::=
    STRING { [$1] }
  | string_list COLON STRING { $3 :: $1 }
```

string_lists ::=
    string_list { [$1] }
  | *string_lists COMMA string_list* { $3 :: $1 }

6.5 Interface of Cascade

```
module type T =
  sig
```

```
    type constant
    type flavor
    type p
```

```
    type t
    val of_string_list : int → string list → t
    val to_string : t → string
```

An opaque type that describes the set of all constraints on an amplitude and how to construct it from a cascade description.

```
    type selectors
    val to_selectors : t → selectors
```

Don't throw anything away:

```
    val no_cascades : selectors
```

select_wf s is_timelike f p ps returns true iff either

- the flavor f and momentum p match the selection s or
- *all* combinations of the momenta in ps are compatible, i. e. $\pm\sum p_i \leq q$.

The latter test is only required in theories with quartic or higher vertices, where *ps* will be the list of all incoming momenta in a fusion. *is_timelike* is required to determine, whether particles and anti-particles should be distinct.

```
    val select_wf : selectors → (p → bool) → flavor → p → p list → bool
```

select_p s p ps same as *select_wf s f p ps*, but ignores the flavor f

```
    val select_p : selectors → p → p list → bool
```

on_shell s p

```
    val on_shell : selectors → flavor → p → bool
```

is_gauss s p

```
    val is_gauss : selectors → flavor → p → bool
```

```
    val select_vtx : selectors → constant Coupling.t →
      flavor → flavor list → bool
```

partition s returns a partition of the external particles that can not be reordered without violating the cascade constraints.

```
    val partition : selectors → int list list
```

Diagnostics:

```
    val description : selectors → string option
```

```
  end
```

```
module Make (M : Model.T) (P : Momentum.T) :
    T with type flavor = M.flavor
      and type constant = M.constant
      and type p = P.t
```

6.6 Implementation of Cascade

```
module type T =
  sig

    type constant
    type flavor
    type p

    type t
    val of_string_list : int → string list → t
    val to_string : t → string

    type selectors
    val to_selectors : t → selectors
    val no_cascades : selectors

    val select_wf : selectors → (p → bool) → flavor → p → p list → bool
    val select_p : selectors → p → p list → bool
    val on_shell : selectors → flavor → p → bool
    val is_gauss : selectors → flavor → p → bool

    val select_vtx : selectors → constant Coupling.t →
      flavor → flavor list → bool

    val partition : selectors → int list list
    val description : selectors → string option

  end

module Make (M : Model.T) (P : Momentum.T) :
    (T with type flavor = M.flavor and type constant = M.constant and type p = P.t) =
  struct

    module CS = Cascade_syntax

    type constant = M.constant
    type flavor = M.flavor
    type p = P.t
```

Since we have

$$p \leq q \Longleftrightarrow (-q) \leq (-p) \tag{6.1}$$

also for $\leq$ as set inclusion *lesseq*, only four of the eight combinations are independent

$$\begin{aligned} p \leq q &\quad\Longleftrightarrow\quad (-q) \leq (-p) \\ q \leq p &\quad\Longleftrightarrow\quad (-p) \leq (-q) \\ p \leq (-q) &\quad\Longleftrightarrow\quad q \leq (-p) \\ (-q) \leq p &\quad\Longleftrightarrow\quad (-p) \leq q \end{aligned} \tag{6.2}$$

```
    let one_compatible p q =
      let neg_q = P.neg q in
      P.lesseq p q ∨
      P.lesseq q p ∨
      P.lesseq p neg_q ∨
      P.lesseq neg_q p
```

'tis wasteful ... (at least by a factor of two, because every momentum combination is generated, including the negative ones.

```
    let all_compatible p p_list q =
      let l = List.length p_list in
      if l ≤ 2 then
        one_compatible p q
      else
        let tuple_lengths = ThoList.range 2 (succ l / 2) in
        let tuples = ThoList.flatmap (fun n → Combinatorics.choose n p_list) tuple_lengths in
        let momenta = List.map (List.fold_left P.add (P.zero (P.dim q))) tuples in
        List.for_all (one_compatible q) momenta
```

The following assumes that the *flavor list* is always very short. Otherwise one should use an efficient set implementation.

```
type wf =
  | True
  | False
  | On_shell of flavor list × P.t
  | On_shell_not of flavor list × P.t
  | Off_shell of flavor list × P.t
  | Off_shell_not of flavor list × P.t
  | Gauss of flavor list × P.t
  | Gauss_not of flavor list × P.t
  | Any_flavor of P.t
  | And of wf list

module Constant = Modeltools.Constant (M)

type vtx =
    { couplings : M.constant list;
      fields : flavor list }

type t =
    { wf : wf;
      (* TODO: The following lists should be sets for efficiency. *)
      flavors : flavor list;
      vertices : vtx list }

let default =
  { wf = True;
    flavors = [];
    vertices = [] }

let of_string s =
  Cascade_parser.main Cascade_lexer.token (Lexing.from_string s)
```

If we knew that we're dealing with a scattering, we could apply $P.flip\_s\_channel\_in$ to all momenta, so that $1+2$ accepts the particle and not the antiparticle. Right now, we don't have this information.

```
let only_wf wf = { default with wf = wf }

let cons_and_wf c wfs =
  match c.wf, wfs with
  | True, wfs → wfs
  | False, _ → [False]
  | wf, [] → [wf]
  | wf, wfs → wf :: wfs

let and_cascades_wf c =
  match List.fold_right cons_and_wf c [] with
  | [] → True
  | [wf] → wf
  | wfs → And wfs

let uniq l =
  ThoList.uniq (List.sort compare l)

let import dim cascades =
  let rec import' = function
    | CS.True →
        only_wf True
    | CS.False →
        only_wf False
    | CS.On_shell (f, p) →
```

```
          only_wf
            (On_shell (List.map M.flavor_of_string f, P.of_ints dim p))
      | CS.On_shell_not (f, p) →
          only_wf
            (On_shell_not (List.map M.flavor_of_string f, P.of_ints dim p))
      | CS.Off_shell (fs, p) →
          only_wf
            (Off_shell (List.map M.flavor_of_string fs, P.of_ints dim p))
      | CS.Off_shell_not (fs, p) →
          only_wf
            (Off_shell_not (List.map M.flavor_of_string fs, P.of_ints dim p))
      | CS.Gauss (f, p) →
          only_wf
            (Gauss (List.map M.flavor_of_string f, P.of_ints dim p))
      | CS.Gauss_not (f, p) →
          only_wf
            (Gauss (List.map M.flavor_of_string f, P.of_ints dim p))
      | CS.Any_flavor p →
          only_wf (Any_flavor (P.of_ints dim p))
      | CS.And cs →
          let cs = List.map import' cs in
          { wf = and_cascades_wf cs;
            flavors = uniq (List.concat
                                (List.map (fun c → c.flavors) cs));
            vertices = uniq (List.concat
                                (List.map (fun c → c.vertices) cs)) }
      | CS.X_Flavor fs →
          let fs = List.map M.flavor_of_string fs in
          { default with flavors = uniq (fs @ List.map M.conjugate fs) }
      | CS.X_Vertex (cs, fss) →
          let cs = List.map Constant.of_string cs
          and fss = List.map (List.map M.flavor_of_string) fss in
          let expanded =
            List.map
              (fun fs → { couplings = cs; fields = fs })
              (match fss with
              | [] → [[]] (* Subtle: not an empty list! *)
              | fss → Product.list (fun fs → fs) fss) in
          { default with vertices = expanded }
    in
    import' cascades

  let of_string_list dim strings =
    match List.map of_string strings with
    | [] → default
    | first :: next →
        import dim (List.fold_right CS.mk_and next first)

  let flavors_to_string fs =
    (String.concat ":" (List.map M.flavor_to_string fs))

  let momentum_to_string p =
    String.concat "+" (List.map string_of_int (P.to_ints p))

  let rec wf_to_string = function
    | True →
        "true"
    | False →
        "false"
    | On_shell (fs, p) →
        momentum_to_string p ^ "␣=␣" ^ flavors_to_string fs
    | On_shell_not (fs, p) →
        momentum_to_string p ^ "␣=␣!" ^ flavors_to_string fs
    | Off_shell (fs, p) →
        momentum_to_string p ^ "␣~␣" ^ flavors_to_string fs
    | Off_shell_not (fs, p) →
        momentum_to_string p ^ "␣~␣!" ^ flavors_to_string fs
    | Gauss (fs, p) →
        momentum_to_string p ^ "␣#␣" ^ flavors_to_string fs
    | Gauss_not (fs, p) →
        momentum_to_string p ^ "␣#␣!" ^ flavors_to_string fs
    | Any_flavor p →
        momentum_to_string p ^ "␣~␣?"
    | And cs →
        String.concat "␣&&␣" (List.map (fun c → "(" ^ wf_to_string c ^ ")") cs)

  let vertex_to_string v =
    "^" ^ String.concat ":" (List.map M.constant_symbol v.couplings) ^
    "[" ^ String.concat "," (List.map M.flavor_to_string v.fields) ^ "]"

  let vertices_to_string vs =
    (String.concat "␣&&␣" (List.map vertex_to_string vs))

  let to_string = function
    | { wf = True; flavors = []; vertices = [] } →
        ""
    | { wf = True; flavors = fs; vertices = [] } →
        "!" ^ flavors_to_string fs
    | { wf = True; flavors = []; vertices = vs } →
        vertices_to_string vs
    | { wf = True; flavors = fs; vertices = vs } →
        "!" ^ flavors_to_string fs ^ "␣&&␣" ^ vertices_to_string vs
    | { wf = wf; flavors = []; vertices = [] } →
        wf_to_string wf
    | { wf = wf; flavors = []; vertices = vs } →
        vertices_to_string vs ^ "␣&&␣" ^ wf_to_string wf
    | { wf = wf; flavors = fs; vertices = [] } →
        "!" ^ flavors_to_string fs ^ "␣&&␣" ^ wf_to_string wf
    | { wf = wf; flavors = fs; vertices = vs } →
        "!" ^ flavors_to_string fs ^
        "␣&&␣" ^ vertices_to_string vs ^
        "␣&&␣" ^ wf_to_string wf

  type selectors =
      { select_p : p → p list → bool;
        select_wf : (p → bool) → flavor → p → p list → bool;
        on_shell : flavor → p → bool;
        is_gauss : flavor → p → bool;
        select_vtx : constant Coupling.t → flavor → flavor list → bool;
        partition : int list list;
        description : string option }

  let no_cascades =
    { select_p = (fun _ _ → true);
      select_wf = (fun _ _ _ _ → true);
      on_shell = (fun _ _ → false);
      is_gauss = (fun _ _ → false);
      select_vtx = (fun _ _ _ → true);
      partition = [];
      description = None }

  let select_p s = s.select_p
  let select_wf s = s.select_wf
  let on_shell s = s.on_shell
  let is_gauss s = s.is_gauss
  let select_vtx s = s.select_vtx
  let partition s = s.partition
  let description s = s.description

  let to_select_p cascades p p_in =
    let rec to_select_p' = function
      | True → true
      | False → false
      | On_shell (_, momentum) | On_shell_not (_, momentum)
      | Off_shell (_, momentum) | Off_shell_not (_, momentum)
      | Gauss (_, momentum) | Gauss_not (_, momentum)
      | Any_flavor momentum → all_compatible p p_in momentum
      | And [] → false
      | And cs → List.for_all to_select_p' cs in
    to_select_p' cascades

  let to_select_wf cascades is_timelike f p p_in =
    let f' = M.conjugate f in
    let rec to_select_wf' = function
      | True → true
      | False → false
      | Off_shell (flavors, momentum) →
          if p = momentum then
            List.mem f' flavors ∨ (if is_timelike p then false else List.mem f flavors)
          else if p = P.neg momentum then
            List.mem f flavors ∨ (if is_timelike p then false else List.mem f' flavors)
          else
            one_compatible p momentum ∧ all_compatible p p_in momentum
      | On_shell (flavors, momentum) | Gauss (flavors, momentum) →
           if is_timelike p then begin
             if p = momentum then
               List.mem f' flavors
             else if p = P.neg momentum then
               List.mem f flavors
             else
               one_compatible p momentum ∧ all_compatible p p_in momentum
           end else
             false
      | Off_shell_not (flavors, momentum) →
          if p = momentum then
            ¬ (List.mem f' flavors ∨ (if is_timelike p then false else List.mem f flavors))
          else if p = P.neg momentum then
            ¬ (List.mem f flavors ∨ (if is_timelike p then false else List.mem f' flavors))
          else
            one_compatible p momentum ∧ all_compatible p p_in momentum
      | On_shell_not (flavors, momentum) | Gauss_not (flavors, momentum) →
          if is_timelike p then begin
            if p = momentum then
              ¬ (List.mem f' flavors)
            else if p = P.neg momentum then
              ¬ (List.mem f flavors)
            else
              one_compatible p momentum ∧ all_compatible p p_in momentum
          end else
            false
      | Any_flavor momentum →
          one_compatible p momentum ∧ all_compatible p p_in momentum
      | And [] → false
      | And cs → List.for_all to_select_wf' cs in
    ¬ (List.mem f cascades.flavors) ∧ to_select_wf' cascades.wf
```

In case you're wondering: *to_on_shell f p* and *is_gauss f p* only search for on shell conditions and are to be used in a target, not in *Fusion*!

```
let to_on_shell cascades f p =
    let f′ = M.conjugate f in
    let rec to_on_shell′ = function
      | True | False | Any_flavor _
      | Off_shell (_, _) | Off_shell_not (_, _)
      | Gauss (_, _) | Gauss_not (_, _) → false
      | On_shell (flavors, momentum) →
          (p = momentum ∨ p = P.neg momentum) ∧ (List.mem f flavors ∨ List.mem f′ flavors)
      | On_shell_not (flavors, momentum) →
          (p = momentum ∨ p = P.neg momentum) ∧ ¬ (List.mem f flavors ∨ List.mem f′ flavors)
      | And [] → false
      | And cs → List.for_all to_on_shell′ cs in
    to_on_shell′ cascades

  let to_gauss cascades f p =
    let f′ = M.conjugate f in
    let rec to_gauss′ = function
      | True | False | Any_flavor _
      | Off_shell (_, _) | Off_shell_not (_, _)
      | On_shell (_, _) | On_shell_not (_, _) → false
      | Gauss (flavors, momentum) →
          (p = momentum ∨ p = P.neg momentum) ∧
          (List.mem f flavors ∨ List.mem f′ flavors)
      | Gauss_not (flavors, momentum) →
          (p = momentum ∨ p = P.neg momentum) ∧
          ¬ (List.mem f flavors ∨ List.mem f′ flavors)
      | And [] → false
      | And cs → List.for_all to_gauss′ cs in
    to_gauss′ cascades

  module Fields =
    struct
      type f = M.flavor
      type c = M.constant list
      let compare = compare
      let conjugate = M.conjugate
    end

  module Fusions = Modeltools.Fusions (Fields)

  let dummy3 = Coupling.Scalar_Scalar_Scalar 1
  let dummy4 = Coupling.Scalar4 1
  let dummyn = Coupling.UFO (Algebra.QC.unit, "dummy", [], [], Birdtracks.one)
```

Translate the vertices in a pair of lists: the first is the list of always rejected couplings and the second the remaining vertices suitable as input to *Fusions.of_vertices*.

```
  let translate_vertices vertices =
    List.fold_left
      (fun (cs, (v3, v4, vn) as acc) v →
        match v.fields with
        | [] → (v.couplings @ cs, (v3, v4, vn))
        | [_] | [_; _] → acc
        | [f1; f2; f3] →
            (cs, (((f1, f2, f3), dummy3, v.couplings) :: v3, v4, vn))
        | [f1; f2; f3; f4] →
            (cs, (v3, ((f1, f2, f3, f4), dummy4, v.couplings) :: v4, vn))
        | fs → (cs, (v3, v4, (fs, dummyn, v.couplings) :: vn)))
      ([], ([], [], [])) vertices

  let unpack_constant = function
    | Coupling.V3 (_, _, cs) → cs
    | Coupling.V4 (_, _, cs) → cs
    | Coupling.Vn (_, _, cs) → cs
```

Sometimes, the empty list is a wildcard and matches any coupling:

```
let match_coupling c cs =
  List.mem c cs

let match_coupling_wildcard c = function
  | [] → true
  | cs → match_coupling c cs

let to_select_vtx cascades =
  match cascades.vertices with
  | [] →
      (* No vertex constraints means that we always accept. *)
      (fun c f fs → true)
  | vertices →
      match translate_vertices vertices with
      | [], ([], [], []) →
          (* If cascades.vertices is not empty, we mustn't get here . . . *)
          failwith "Cascade.to_select_vtx:␣unexpected"
      | couplings, ([], [], []) →
          (* No constraints on the fields. Just make sure that the coupling c doesn't appear in the vetoed
couplings. *)
          (fun c f fs →
            let c = unpack_constant c in
            ¬ (match_coupling c couplings))
      | couplings, vertices →
          (* Make sure that Fusions.of_vertices is only evaluated once for efficiency. *)
          let fusions = Fusions.of_vertices vertices in
          (fun c f fs →
            let c = unpack_constant c in
            (* Make sure that none of the vetoed couplings matches. Here an empty couplings list is not
a wildcard. *)
            if match_coupling c couplings then
              false
            else
              (* Also make sure that none of the vetoed vertices matches. Here an empty couplings list
is a wildcard. *)
                ¬ (List.exists
                        (fun (f′, cs′) →
                          let cs′ = unpack_constant cs′ in
                          f = f′ ∧ match_coupling_wildcard c cs′)
                        (Fusions.fuse fusions fs)))
```

Not a working implementation yet, but it isn't used either . . .

```
module IPowSet = PowSet.Make (Int)

let rec coarsest_partition′ = function
    | True | False → IPowSet.empty
    | On_shell (_, momentum) | On_shell_not (_, momentum)
    | Off_shell (_, momentum) | Off_shell_not (_, momentum)
    | Gauss (_, momentum) | Gauss_not (_, momentum)
    | Any_flavor momentum → IPowSet.of_lists [P.to_ints momentum]
    | And [] → IPowSet.empty
    | And cs → IPowSet.basis (IPowSet.union (List.map coarsest_partition′ cs))

let coarsest_partition cascades =
  let p = coarsest_partition′ cascades in
  if IPowSet.is_empty p then
    []
  else
    IPowSet.to_lists p

let part_to_string part =
  "{" ^ String.concat "," (List.map string_of_int part) ^ "}"
  let partition_to_string = function
    | [] → ""
    | parts →
        "␣␣grouping␣{" ^ String.concat "," (List.map part_to_string parts) ^ "}"

  let to_selectors = function
    | { wf = True; flavors = []; vertices = [] } → no_cascades
    | c →
        let partition = coarsest_partition c.wf in
        { select_p = to_select_p c.wf;
          select_wf = to_select_wf c;
          on_shell = to_on_shell c.wf;
          is_gauss = to_gauss c.wf;
          select_vtx = to_select_vtx c;
          partition = partition;
          description = Some (to_string c ^ partition_to_string partition) }

end
```

—7— Arrows and Epsilon Tensors

7.1 Interface of Arrow

The datatypes *Arrow.free* and *Arrow.factor* will be used as building blocks for *Birdtracks.t* below.
For fundamental and adjoint representations, the endpoints of arrows are uniquely specified by a vertex (which will be represented by a number). For representations with more than one outgoing or incoming arrow, we need an additional index. This is abstracted in the *endpoint* type.

```
type endpoint = private
  | I of int
  | M of int × int
```

Endpoints can be the the tip or tail of an arrow or a ghost. Using incompatible types for each forces us to export three identical copies of some functions, but should help to avoid some simple mistakes, in which tips and tails are confused.

```
type tip = private endpoint
type tail = private endpoint
type ghost = private endpoint
```

The position of the endpoint is encoded as an integer, which can be mapped, if necessary.

```
val position_tip : tip → int
val position_tail : tail → int
val position_ghost : ghost → int
val relocate_tip : (int → int) → tip → tip
val relocate_tail : (int → int) → tail → tail
val relocate_ghost : (int → int) → tail → tail
```

An *Arrow.t* is either a genuine arrow or a ghost. The rationale for the polymorphic definition is explained below.

```
type ('tail, 'tip, 'ghost) t =
  | Arrow of 'tail × 'tip
  | Ghost of 'ghost
```

$\epsilon_{i_1 i_2 \cdots i_n}$ and $\bar\epsilon^{i_1 i_2 \cdots i_n}$ are represented by lists $[i_1; i_2; \ldots; i_n]$.

```
type 'tip eps = 'tip list
type 'tail eps_bar = 'tail list
```

We distuish *free* arrows, ϵs and $\bar\epsilon$s that must not contain summation indices from *factor*s that may. Indices are opaque. (*'tail*, *'tip*, *'ghost*) *t* has been defined polymorphic above so that we can use richer *'tail*, *'tip* and *'ghost* in *factor* to identify summation indices. Not that it is *not* enough to identify summation indices by negative integers alone. Due to the presence of double arrows representing gluons, we must distinguish summation indices in the left factor of a product from those in the right factor.

```
type free = (tail, tip, ghost) t
type free_eps = tip eps
type free_eps_bar = tail eps_bar
type factor
type factor_eps
type factor_eps_bar
```

```
val relocate : (int → int) → free → free
val rev : free → free
val rev_eps : free_eps → free_eps_bar
val rev_eps_bar : free_eps_bar → free_eps
```

Useful for testing compatibility when adding terms.

```
val tips : free → tip list
val tips_eps : free_eps → tip list
val tails : free → tail list
val tails_eps_bar : free_eps_bar → tail list
```

For debugging, logging, etc.

```
val free_to_string : free → string
val free_eps_to_string : free_eps → string
val free_eps_bar_to_string : free_eps_bar → string
val factor_to_string : factor → string
val factor_eps_to_string : factor_eps → string
val factor_eps_bar_to_string : factor_eps_bar → string
```

Turn the *endpoint*s satisfying the predicate into a left or right hand side summation index. Left and right refer to the two factors in a product and we must only match arrows with *endpoint*s in both factors, not double lines on either side. Typically, the predicate will be set up to select only the summation indices that appear on both sides.

```
val to_left_factor : (endpoint → bool) → free → factor
val to_left_factor_eps : (endpoint → bool) → free_eps → factor_eps
val to_left_factor_eps_bar : (endpoint → bool) → free_eps_bar → factor_eps_bar
val to_right_factor : (endpoint → bool) → free → factor
val to_right_factor_eps : (endpoint → bool) → free_eps → factor_eps
val to_right_factor_eps_bar : (endpoint → bool) → free_eps_bar → factor_eps_bar
```

The incomplete inverse *of_factor* raises an exception if there are remaining summation indices. *is_free* can be used to check first.

```
val of_factor : factor → free
val of_factor_eps : factor_eps → free_eps
val of_factor_eps_bar : factor_eps_bar → free_eps_bar
val is_free : factor → bool
val is_free_eps : factor_eps → bool
val is_free_eps_bar : factor_eps_bar → bool
```

Return all the endpoints of the arrow that have a *position* encoded as a negative integer. These are treated as summation indices in our applications.

```
val negatives : free → endpoint list
val negatives_eps : free_eps → endpoint list
val negatives_eps_bar : free_eps_bar → endpoint list
```

We will need to test whether an arrow represents a ghost.

```
val is_ghost : free → bool
```

An arrow looping back to itself.

```
val is_tadpole : factor → bool
```

Merging an arrow with another arrow, ϵ or $\bar{\epsilon}$ can give a variety of results:

```
type merge =
  | Match of factor (∗ a tip fits the other's tail: make one arrow out of two ∗)
  | Ghost_Match (∗ two matching ghosts ∗)
  | Loop_Match (∗ both tips fit both tails: drop the arrows ∗)
  | Mismatch (∗ ghost meets arrow: discard ∗)
  | No_Match (∗ nothing to be done ∗)
```

```
val merge_arrow_arrow : factor → factor → merge
```

We can narrow this for ϵ and $\bar{\epsilon}$, where *Loop_Match* and *Ghost_Match* are impossible!

```
type α merge_eps =
```

| *Match_Eps* of α (* a tip fits the other's tail: make one arrow out of two *)
| *Mismatch_Eps* (* ghost meets arrow: discard *)
| *No_Match_Eps* (* nothing to be done *)

val *merge_arrow_eps* : *factor* → *factor_eps* → *factor_eps merge_eps*
val *merge_arrow_eps_bar* : *factor* → *factor_eps_bar* → *factor_eps_bar merge_eps*

In order to merge an ϵ with an $\bar\epsilon$, we use

$$\forall n,N\in\mathbf{N}, 2\le n\le N:\ \epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{j_1j_2\cdots j_n} = \sum_{\sigma\in S_n}(-1)^{\varepsilon(\sigma)}\delta_{i_1}^{\sigma(j_1)}\delta_{i_2}^{\sigma(j_2)}\cdots\delta_{i_n}^{\sigma(j_n)}, \tag{7.1}$$

where $N=\delta_i^i$ is the dimension, to replace the pair by two lists of lists of arrows: the first corresponding to the even permutations, the second to the odd ones. Return *None*, if the rank of ϵ and $\bar\epsilon$ don't match.
See section 7.2.2 on pages 78ff for a justification for using it also in the case $n \neq N$.

val *merge_eps_eps_bar* : *factor_eps* → *factor_eps_bar* → (*factor list list* × *factor list list*) *option*

Break up an arrow *tee a* (*i* => *j*) → [*i* => *a*; *a* => *j*], i. e. insert a gluon. Returns an empty list for a ghost and raises an exception for ϵ and $\bar\epsilon$.

val *tee* : *int* → *free* → *free list*

dir i j arrow returns the direction of the arrow relative to *j* => *i*. Returns 0 for a ghost and raises an exception for ϵ and $\bar\epsilon$.

val *dir* : *int* → *int* → *free* → *int*

It's intuitive to use infix operators to construct the lines.

val *single* : *endpoint* → *endpoint* → *free*
val *double* : *endpoint* → *endpoint* → *free list*
val *ghost* : *endpoint* → *free*

module *Infix* : sig

single i j or *i* => *j* creates a single line from *i* to *j* and *i* ==> *j* is a shorthard for [*i* => *j*].

  val (=>) : *int* → *int* → *free*
  val (==>) : *int* → *int* → *free list*

double i j or *i* <=> *j* creates a double line from *i* to *j* and back.

  val (<=>) : *int* → *int* → *free list*

Single lines with subindices at the tip and/or tail

  val (>=>) : *int* × *int* → *int* → *free*
  val (=>>) : *int* → *int* × *int* → *free*
  val (>=>>) : *int* × *int* → *int* × *int* → *free*

?? *i* creates a ghost at *i*.

  val (??) : *int* → *free*

NB: I wanted to use ~~ instead of ??, but ocamlweb can't handle operators starting with ~ in the index properly.

end

val *epsilon* : *int list* → *free_eps*
val *epsilon_bar* : *int list* → *free_eps_bar*

chain [1; 2; 3] is a shorthand for [1 => 2; 2 => 3] and *cycle* [1; 2; 3] for [1 => 2; 2 => 3; 3 => 1]. Other lists and edge cases are handled in the natural way.

val *chain* : *int list* → *free list*
val *cycle* : *int list* → *free list*

module *Test* : sig val *suite* : *OUnit.test* val *suite_long* : *OUnit.test* end

Pretty printer for the toplevel.

val *pp_free* : *Format.formatter* → *free* → *unit*
val *pp_factor* : *Format.formatter* → *factor* → *unit*

7.2 Implementation of Arrow

7.2.1 Arrows and Epsilons

```
type endpoint =
  | I of int
  | M of int × int

let position_endpoint = function
  | I i → i
  | M (i, _) → i

let relocate_endpoint f = function
  | I i → I (f i)
  | M (i, n) → M (f i, n)

type tip = endpoint
type tail = endpoint
type ghost = endpoint

let position_tip = position_endpoint
let position_tail = position_endpoint
let position_ghost = position_endpoint
let relocate_tip = relocate_endpoint
let relocate_tail = relocate_endpoint
let relocate_ghost = relocate_endpoint
```

Note that in the case of double lines for the adjoint representation the *same endpoint* appears twice: once as a *tip* and once as a *tail*. If we want to multiply two factors by merging arrows with matching *tip* and *tail*, we must make sure that the *tip* is from one factor and the *tail* from the other factor.
The *Free* variant contains positive indices as well as negative indices that don't appear on both sides and will be summed in a later product. *SumL* and *SumR* indices appear on both sides.

```
type α index =
  | Free of α
  | SumL of α
  | SumR of α

let is_free_index = function
  | Free _ → true
  | SumL _ | SumR _ → false

type ('tail, 'tip, 'ghost) t =
  | Arrow of 'tail × 'tip
  | Ghost of 'ghost
type 'tip eps = 'tip list
type 'tail eps_bar = 'tail list

type free = (tail, tip, ghost) t
type free_eps = tip eps
type factor_eps = tip index eps

type factor = (tail index, tip index, ghost index) t
type free_eps_bar = tail eps_bar
type factor_eps_bar = tail index eps_bar

let relocate f = function
  | Arrow (tail, tip) → Arrow (relocate_tail f tail, relocate_tip f tip)
  | Ghost ghost → Ghost (relocate_ghost f ghost)

let rev = function
  | Arrow (tail, tip) → Arrow (tip, tail)
  | Ghost _ as ghost → ghost
let rev_eps tips = tips
let rev_eps_bar tails = tails
let tips = function
  | Arrow (_, tip) → [tip]
  | Ghost _ → []
let tails = function
  | Arrow (tail, _) → [tail]
  | Ghost _ → []
let tips_eps tips = tips
let tails_eps_bar tails = tails

let endpoint_to_string = function
  | I i → string_of_int i
  | M (i, n) → Printf.sprintf "%d.%d" i n

let index_to_string = function
  | Free i → endpoint_to_string i
  | SumL i → endpoint_to_string i ^ "L"
  | SumR i → endpoint_to_string i ^ "R"

let to_string i2s = function
  | Arrow (tail, tip) → Printf.sprintf "%s>%s" (i2s tail) (i2s tip)
  | Ghost ghost → Printf.sprintf "{%s}" (i2s ghost)
let to_string_eps i2s tips = Printf.sprintf ">>>%s" (ThoList.to_string i2s tips)
let to_string_eps_bar i2s tails = Printf.sprintf "<<<%s" (ThoList.to_string i2s tails)

let free_to_string = to_string endpoint_to_string
let free_eps_to_string = to_string_eps endpoint_to_string
let free_eps_bar_to_string = to_string_eps_bar endpoint_to_string

let factor_to_string = to_string index_to_string
let factor_eps_to_string = to_string_eps index_to_string
let factor_eps_bar_to_string = to_string_eps_bar index_to_string

let matching_summation i1 i2 =
  match i1, i2 with
  | SumL i1, SumR i2 | SumR i1, SumL i2 → i1 = i2
  | _ → false

let map f = function
  | Arrow (tail, tip) → Arrow (f tail, f tip)
  | Ghost ghost → Ghost (f ghost)
let map_eps = List.map
let map_eps_bar = List.map

let free_index = function
  | Free i → i
  | SumL i → invalid_arg "Arrow.free_index:␣leftover␣LHS␣summation"
  | SumR i → invalid_arg "Arrow.free_index:␣leftover␣RHS␣summation"

let to_left_index is_sum i =
  if is_sum i then
    SumL i
  else
    Free i

let to_right_index is_sum i =
  if is_sum i then
    SumR i
  else
    Free i

let to_left_factor is_sum = map (to_left_index is_sum)
let to_right_factor is_sum = map (to_right_index is_sum)
let of_factor = map free_index

let to_left_factor_eps is_sum = map_eps (to_left_index is_sum)
let to_right_factor_eps is_sum = map_eps (to_right_index is_sum)
let of_factor_eps = map_eps free_index
let to_left_factor_eps_bar is_sum = map_eps_bar (to_left_index is_sum)
let to_right_factor_eps_bar is_sum = map_eps_bar (to_right_index is_sum)
let of_factor_eps_bar = map_eps_bar free_index

let negatives = function
  | Arrow (tail, tip) →
      if position_tail tail < 0 then
        if position_tip tip < 0 then
          [tail; tip]
        else
          [tail]
      else if position_tip tip < 0 then
        [tip]
      else
        []
  | Ghost ghost →
      if position_ghost ghost < 0 then
        [ghost]
      else
        []
let negatives_eps = List.filter (fun tip → position_tip tip < 0)
let negatives_eps_bar = List.filter (fun tail → position_tail tail < 0)

let is_free = function
  | Arrow (Free _, Free _) | Ghost (Free _) → true
  | Arrow (_, _) | Ghost _ → false
let is_free_eps = List.for_all is_free_index
let is_free_eps_bar = List.for_all is_free_index

let is_ghost = function
  | Ghost _ → true
  | Arrow _ → false

let single tail tip =
  Arrow (tail, tip)

let double a b =
  if a = b then
    [single a b]
  else
    [single a b; single b a]

let ghost g =
  Ghost g

module Infix =
  struct
    let ( => ) i j = single (I i) (I j)
    let ( ==> ) i j = [i => j]
    let ( <=> ) i j = double (I i) (I j)
    let ( >=> ) (i, n) j = single (M (i, n)) (I j)
    let ( =>> ) i (j, m) = single (I i) (M (j, m))
    let ( >=>> ) (i, n) (j, m) = single (M (i, n)) (M (j, m))
    let ( ?? ) i = ghost (I i)
  end

open Infix
```

Split *a_list* at the first element equal to *a* according to *eq*. Return the reversed first part and the rest as a pair and wrap it in *Some*. Return *None* if there is no match.

```
let take_first_match_opt ?(eq = (=)) a a_list =
  let rec take_first_match_opt' rev_head = function
    | [] → None
    | elt :: tail →
        if eq elt a then
```

```
            Some (rev_head, tail)
        else
            take_first_match_opt' (elt :: rev_head) tail in
    take_first_match_opt' [] a_list
```

Split *a_list* and *b_list* at the first element equal according to *eq*. Return the reversed first part and the rest of each as a pair of pairs wrap it in *Some*. Return *None* if there is no match.

This function remains from an earlier version and is no longer used.

```
let take_first_matching_pair_opt ?(eq = (=)) a_list b_list =
  let rec take_first_matching_pair_opt' rev_a_head = function
    | [] → None
    | a :: a_tail →
        begin match take_first_match_opt ~eq a b_list with
        | Some (rev_b_head, b_tail) →
            Some ((rev_a_head, a_tail), (rev_b_head, b_tail))
        | None →
            take_first_matching_pair_opt' (a :: rev_a_head) a_tail
        end in
  take_first_matching_pair_opt' [] a_list
```

Replace the first occurence of an element equal to *a* according to *eq* in *a_list* by *a'* and wrap the new list in *Some*. Return *None* if there is no match.

```
let replace_first_opt ?(eq = (=)) a a' a_list =
  match take_first_match_opt ~eq a a_list with
  | Some (rev_head, tail) → Some (List.rev_append rev_head (a' :: tail))
  | None → None

let tee a = function
  | Arrow (tail, tip) → [Arrow (tail, I a); Arrow (I a, tip)]
  | Ghost _ as g → [g]

let dir i j = function
  | Arrow (tail, tip) →
      let tail = position_tail tail
      and tip = position_tip tip in
      if tip = i ∧ tail = j then
        1
      else if tip = j ∧ tail = i then
        -1
      else
        invalid_arg "Arrow.dir"
  | Ghost _ → 0

type merge =
  | Match of factor
  | Ghost_Match
  | Loop_Match
  | Mismatch
  | No_Match
```

As an optimization, don't attempt to merge if neither of the arrows contains a summation index and return immediately.

```
let merge_arrow_arrow arrow1 arrow2 =
  if is_free arrow1 ∨ is_free arrow2 then
    No_Match
  else
    match arrow1, arrow2 with
    | Ghost g1, Ghost g2 →
        if matching_summation g1 g2 then
          Ghost_Match
        else
```

```
          No_Match
      | Arrow (tail, tip), Ghost g
        | Ghost g, Arrow (tail, tip) →
          if matching_summation g tail ∨ matching_summation g tip then
            Mismatch
          else
            No_Match
      | Arrow (tail, tip), Arrow (tail′, tip′) →
          if matching_summation tip tail′ then
            if matching_summation tip′ tail then
              Loop_Match
            else
              Match (Arrow (tail, tip′))
          else if matching_summation tip′ tail then
            Match (Arrow (tail′, tip))
          else
            No_Match

type α merge_eps =
  | Match_Eps of α
  | Mismatch_Eps
  | No_Match_Eps

let merge_arrow_eps arrow tips =
  if is_free_eps tips ∨ is_free arrow then
    No_Match_Eps
  else
    match arrow with
    | Arrow (tail, tip) →
        begin match replace_first_opt ~eq : matching_summation tail tip tips with
        | None → No_Match_Eps
        | Some tips → Match_Eps tips
        end
    | Ghost g →
        if List.exists (matching_summation g) tips then
          Mismatch_Eps
        else
          No_Match_Eps

let merge_arrow_eps_bar arrow tails =
  if is_free_eps_bar tails ∨ is_free arrow then
    No_Match_Eps
  else
    match arrow with
    | Arrow (tail, tip) →
        begin match replace_first_opt ~eq : matching_summation tip tail tails with
        | None → No_Match_Eps
        | Some tails → Match_Eps tails
        end
    | Ghost g →
        if List.exists (matching_summation g) tails then
          Mismatch_Eps
        else
          No_Match_Eps
```

7.2.2 Evaluation Rules for Epsilon Tensors

In the case of matching dimension $N = \delta_m^m$ and rank n of ϵ and $\bar\epsilon$, the tensor algebra of the δ_i^j, $\epsilon_{i_1 i_2 \cdots i_n}$ and $\bar\epsilon^{j_1 j_2 \cdots j_n}$ is *not* freely generated. Indeed, introducing the *generalized Kronecker* δ symbol

$$\delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} = \sum_{\sigma\in S_n} (-1)^{\varepsilon(\sigma)} \delta_{i_1}^{\sigma(j_1)} \delta_{i_2}^{\sigma(j_2)} \cdots \delta_{i_n}^{\sigma(j_n)} = \sum_{\sigma\in S_n} (-1)^{\varepsilon(\sigma)} \delta^{j_1}_{\sigma(i_1)} \delta^{j_2}_{\sigma(i_2)} \cdots \delta^{j_n}_{\sigma(i_n)} = \begin{vmatrix} \delta_{i_1}^{j_1} & \delta_{i_1}^{j_2} & \cdots & \delta_{i_1}^{j_n} \\ \delta_{i_2}^{j_1} & \delta_{i_2}^{j_2} & \cdots & \delta_{i_2}^{j_n} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{i_n}^{j_1} & \delta_{i_n}^{j_2} & \cdots & \delta_{i_n}^{j_n} \end{vmatrix} , \tag{7.2}$$

there is the relation $\forall n = N \in \mathbf{N}$ with $N \geq 2$:

$$\epsilon_{i_1 i_2 \cdots i_n} \bar\epsilon^{j_1 j_2 \cdots j_n} = \delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} , \tag{7.3}$$

which follows from anti-symmetry and the choice of normalization $\epsilon_{12\cdots n} = 1 = \bar\epsilon^{12\cdots n}$ alone. Contracting k indices in the relation (7.3), we find $\forall k, n, N \in \mathbf{N}$ with $0 \leq k \leq n = N \geq 2$:

$$\epsilon_{m_1 \cdots m_k i_{k+1} \cdots i_n} \bar\epsilon^{m_1 \cdots m_k j_{k+1} \cdots j_n} = k!\, \delta^{j_{k+1} j_{k+2} \cdots j_n}_{i_{k+1} i_{k+2} \cdots i_n} . \tag{7.4}$$

Note that the generalized Kronecker delta (7.2) is well defined for arbitrary rank $n \geq 1$, including $n < N$, and vanishes for $n > N$. It satisfies

$$\delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} \delta^{k_1 k_2 \cdots k_n}_{j_1 j_2 \cdots j_n} = n!\, \delta^{k_1 k_2 \cdots k_n}_{i_1 i_2 \cdots i_n} \tag{7.5a}$$

$$\delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} \epsilon_{j_1 j_2 \cdots j_n} = n!\, \epsilon_{i_1 i_2 \cdots i_n} \tag{7.5b}$$

$$\delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} \bar\epsilon^{i_1 i_2 \cdots i_n} = n!\, \bar\epsilon^{j_1 j_2 \cdots j_n} \tag{7.5c}$$

since every $\sigma \in S_n$ gives the same contribution when contracting totally antisymmetric combinations. Note also that the relations (7.5) are independent of the dimension N and remain valid for rank $n \neq N$, as long as $\epsilon_{i_1 i_2 \cdots i_n}$ and $\bar\epsilon^{j_1 j_2 \cdots j_n}$ are totally antisymmetric.

In our birdtrack based evaluator, the condition $N = n$ is not enforced. Indeed, N is just a variable in Laurent polynomials *Algebra.Laurent.t* and n is the arbitrary length of the lists in *tip Arrow.eps* and *tail Arrow.eps_bar* of colorflows. Therefore, we can use neither (7.3) nor (7.4) directly to test our evaluator.
Nevertheless, for the purpose of testing our evaluator, we can *define* a *formal* evaluation rule for birdtracks in the general case $N \neq n$, that is compatible with anti-symmetry and reduces to (7.3) for $N = n$

$$\epsilon_{i_1 i_2 \cdots i_n} \bar\epsilon^{j_1 j_2 \cdots j_n} \to \delta^{j_1 j_2 \cdots j_n}_{i_1 i_2 \cdots i_n} , \tag{7.6}$$

where we use the arrow $\to$ instead of the equal sign to stress that is a rule and not an equation, in contrast to the special case (7.3) for $n = N$.

```
let merge_eps_eps_bar tips tails =
  if List.length tails ≠ List.length tips then
    None
  else
    Some (List.fold_left
            (fun (even, odd) (eps, tips) →
              if eps > 0 then
                (List.rev_map2 single tails tips :: even, odd)
              else
                (even, List.rev_map2 single tails tips :: odd))
            ([], []) (Combinatorics.permute_signed tips))
```

Contracting one index, we find the equation

$$\delta^{m j_2 \cdots j_n}_{m i_2 \cdots i_n} = \delta_m^m \sum_{\substack{\sigma\in S_n \\ \sigma(m)=m}} (-1)^{\varepsilon(\sigma)} \delta_{i_2}^{\sigma(j_2)} \cdots \delta_{i_n}^{\sigma(j_n)} + \sum_{\substack{\sigma\in S_n \\ \sigma(m)\neq m}} (-1)^{\varepsilon(\sigma)} \delta_m^{\sigma(m)} \delta_{i_2}^{\sigma(j_2)} \cdots \delta_{i_n}^{\sigma(j_n)}$$
$$= N \delta^{j_2 \cdots j_n}_{i_2 \cdots i_n} - (n-1)\, \delta^{j_2 \cdots j_n}_{i_2 \cdots i_n} = (N - n + 1)\, \delta^{j_2 \cdots j_n}_{i_2 \cdots i_n} , \tag{7.7}$$

where the $N = \delta_m^m$ comes from the permutations with $\sigma(m) = m$ that correspond to a loop in the color flow and the $n-1$ from the permutations with $\sigma(m) \in \{i_2, \ldots, i_n\}$ that do not lead to a loop. The minus is due to the fact that there is exactly one transposition $m \leftrightarrow \sigma(m)$. Thus the consistent evalution rule for a contracted ϵ-$\bar\epsilon$-pair is

$$\epsilon_{m i_2 \cdots i_n} \bar\epsilon^{m j_2 \cdots j_n} \to \delta^{m j_2 \cdots j_n}_{m i_2 \cdots i_n} = (N - n + 1)\, \delta^{j_2 \cdots j_n}_{i_2 \cdots i_n} . \tag{7.8}$$

Note that $N-n+1=1$ in the special case $N=n$ when rank and dimension match. Proceeding by induction, we obtain the equation

$$\delta^{m_1\cdots m_k j_{k+1}\cdots j_n}_{m_1\cdots m_k i_{k+1}\cdots i_n} = \frac{(N-n+k)!}{(N-n)!}\,\delta^{j_{k+1}j_{k+2}\cdots j_n}_{i_{k+1}i_{k+2}\cdots i_n} \tag{7.9}$$

and the corresponding evaluation rule

$$\epsilon_{m_1\cdots m_k i_{k+1}\cdots i_n}\bar\epsilon^{m_1\cdots m_k j_{k+1}\cdots j_n} \to \delta^{m_1\cdots m_k j_{k+1}\cdots j_n}_{m_1\cdots m_k i_{k+1}\cdots i_n} = \frac{(N-n+k)!}{(N-n)!}\,\delta^{j_{k+1}j_{k+2}\cdots j_n}_{i_{k+1}i_{k+2}\cdots i_n}\,, \tag{7.10}$$

where

$$\frac{(N-n+k)!}{(N-n)!} = (N-n+1)(N-n+2)\cdots(N-n+k)\,. \tag{7.11}$$

In the case $N=n$, we recover

$$\frac{(N-n+k)!}{(N-n)!} = k! \tag{7.12}$$

as in (7.4), of course.

Ambiguities for $n \neq N$

While (7.6) and (7.10) can be used for a single pair of ϵ and $\bar\epsilon$, it must be stressed that (7.6) is *not* a well defined rule for more general expressions in the case $n \neq N$, because the result depends on the way pairs of ϵ and $\bar\epsilon$ are chosen for the application of the rule.

As a simple example consider the complete pairwise contractions of two ϵ and two $\bar\epsilon$

$$\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{i_1i_2\cdots i_n}\epsilon_{j_1j_2\cdots j_n}\bar\epsilon^{j_1j_2\cdots j_n}\,. \tag{7.13}$$

Using (7.6), this can be evaluated in two ways

$$\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{i_1i_2\cdots i_n}\epsilon_{j_1j_2\cdots j_n}\bar\epsilon^{j_1j_2\cdots j_n} = \left(\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{i_1i_2\cdots i_n}\right)^2 \to \left(\frac{(N-n+n)!}{(N-n)!}\right)^2 = \left(\frac{N!}{(N-n)!}\right)^2 \tag{7.14a}$$

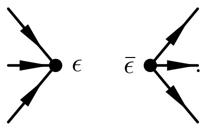
and

$$\begin{aligned}\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{i_1i_2\cdots i_n}\epsilon_{j_1j_2\cdots j_n}\bar\epsilon^{j_1j_2\cdots j_n} &= \left(\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{j_1j_2\cdots j_n}\right)\left(\epsilon_{j_1j_2\cdots j_n}\bar\epsilon^{i_1i_2\cdots i_n}\right)\\ &\to \delta^{j_1j_2\cdots j_n}_{i_1i_2\cdots i_n}\delta^{i_1i_2\cdots i_n}_{j_1j_2\cdots j_n} = n!\,\delta^{j_1j_2\cdots j_n}_{i_1i_2\cdots i_n} = n!\,\frac{(N-n+n)!}{(N-n)!} = \frac{N!n!}{(N-n)!}\,,\end{aligned} \tag{7.14b}$$

which agree only for $N=n$. This observation must be taken into account when interpreting the results of self tests.

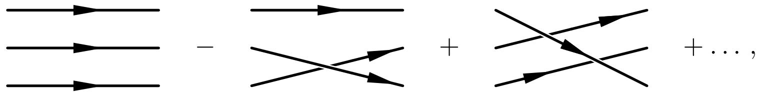
Even if the expressions (7.14a) and (7.14b) agree for $n=N$, one might wonder if they correspond to two different physical interpretations of the color flows. The expression (7.13) appears in the color summed square matrix elements for $2n$ particles that contain color flows of the form

$$\epsilon_{i_1i_2\cdots i_n}\bar\epsilon^{j_1j_2\cdots j_n} = \tag{7.15}$$



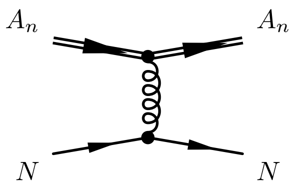
The evaluation (7.14a) corresponds to coupling n particles carrying the flows $\epsilon_{i_1,i_2,\ldots i_n}$ to the n particles carrying the flows $\bar\epsilon^{j_1,j_2,\ldots j_n}$ via an intermediate color singlet state. On the other hand, the evaluation (7.14b) corresponds to substituting this flow by

$$\delta^{j_1j_2\cdots j_n}_{i_1i_2\cdots i_n} = \tag{7.16}$$



which, at first sight, appears to introduce colored intermediate states.

However, this is not really the case, because the colors cancel out for $n = N = N_C$. This can be seen by looking at the scattering of such a state with a particle in the fundamental representation



(7.17)

and calculating the spin summed squared matrix element

$$\sum |M_n|^2 = \text{tr}\left(T_a^{A_n} T_b^{A_n}\right) \text{tr}\left(T_a T_b\right) = \text{tr}\left(T_a^{A_n} T_a^{A_n}\right) = \dim(A_n) C_2(A_n)$$
$$= \frac{N!}{n!(N-n)!} \frac{n(N-n)(N+1)}{N} = \frac{N+1}{(n-1)!} \frac{(N-1)!}{(N-n-1)!} \tag{7.18}$$

where T^{A_n} denotes the generator, $\dim(A_n)$ the dimension and $C_2(A_n)$ the quadratic Casimir (9.18c) in the totally antisymmetric product of n fundamental representations[1]. This expression vanishes for $n \geq N$ and is non-zero for $n < N$. The case $n > N$ is obvious from antisymmetry, but the case $n = N$ depends on the fact that the totally antisymmetric product of N fundamental representations corresponds to a singlet. Therefore, we are free to choose arbitrary pairings of ϵ with $\bar{\epsilon}$ without affecting the our results for summed squared matrix elements.

Nevertheless, there appear to remain ambiguities in amplitudes with more than one ϵ or $\bar{\epsilon}$. For $n = N = 3$, they first appear in amplitudes for 5 particles. These can contain color flows of the form

$$M^k_{i_1 i_2, j_1 j_2} = \epsilon_{i_1 i_2 m_1} \epsilon_{j_1 j_2 m_2} \bar{\epsilon}^{m_1 m_2 k} \tag{7.20}$$

and we have to decide whether to evaluate this as

$$M^k_{i_1 i_2, j_1 j_2} \to M^{(j)\,k}_{i_1 i_2, j_1 j_2} = \epsilon_{i_1 i_2 m_1} (N-2) \left(\delta^k_{j_1} \delta^{m_1}_{j_2} - \delta^{m_1}_{j_1} \delta^k_{j_2}\right) = (N-2) \left(\epsilon_{i_1 i_2 j_2} \delta^k_{j_1} - \epsilon_{i_1 i_2 j_1} \delta^k_{j_2}\right) \tag{7.21a}$$

or

$$M^k_{i_1 i_2, j_1 j_2} \to M^{(i)\,k}_{i_1 i_2, j_1 j_2} = \epsilon_{j_1 j_2 m_2} (N-2) \left(\delta^{m_2}_{i_1} \delta^k_{i_2} - \delta^k_{i_1} \delta^{m_2}_{i_2}\right) = (N-2) \left(\epsilon_{j_1 j_2 i_1} \delta^k_{i_2} - \epsilon_{j_1 j_2 i_2} \delta^k_{i_1}\right)\,, \tag{7.21b}$$

where the superscript denotes which of the ϵ has been contracted with the $\bar{\epsilon}$ using (7.4). These results are manifestly antisymmetric under the exchange of the elements of each of the two pairs of indices separately, but not under the exchange of the pairs.

Fortunately, in the case $n = N$, we can make use of relations of the form

$$\sum_{\sigma \in S_{n+1}} (-1)^{\varepsilon(\sigma)} \epsilon_{\sigma(i_1)\sigma(i_2)\cdots\sigma(i_n)} \delta^j_{\sigma(i_{n+1})} = 0\,, \tag{7.22}$$

that follow from the fact that there is no totally antisymmetric tensor of rank $n > N$ in N dimensions. For example

$$\epsilon_{ijk} \delta^m_l - \epsilon_{lij} \delta^m_k + \epsilon_{kli} \delta^m_j - \epsilon_{jkl} \delta^m_i = 0 \tag{7.23}$$

or

$$\epsilon_{ijk} \delta^m_l - \epsilon_{ijl} \delta^m_k = -\epsilon_{kli} \delta^m_j + \epsilon_{klj} \delta^m_i \tag{7.24}$$

proves that

$$M^{(j)\,k}_{i_1 i_2, j_1 j_2} = -M^{(j)\,k}_{j_1 j_2, i_1 i_2} \tag{7.25}$$

and equivalent relations for $M^{(k)}$ and $M^{(i)}$ in the case $n = N = 3$. Therefore the amplitudes satisfy all symmetry requirements in the physical case, just not manifestly.

[1]We can use (7.18) to test our evaluator and find agreement, e. g. for $n = 2, 3, 4, 5$

$$\sum |M_2|^2 = (N+1)(N-1)(N-2) \tag{7.19a}$$
$$\sum |M_3|^2 = \frac{N+1}{2}(N-1)(N-2)(N-3) \tag{7.19b}$$
$$\sum |M_4|^2 = \frac{N+1}{6}(N-1)(N-2)(N-3)(N-4) \tag{7.19c}$$
$$\sum |M_5|^2 = \frac{N+1}{24}(N-1)(N-2)(N-3)(N-4)(N-5)\,. \tag{7.19d}$$

Note that we could also observe that

$$M^{(i)\,k}_{i_1i_2,j_1j_2} = -M^{(j)\,k}_{j_1j_2,i_1i_2} \tag{7.26}$$

and construct an equivalent amplitude that manifestly satisfies all required antisymmetries

$$M^{k}_{i_1i_2,j_1j_2} = \frac{1}{2}\left(M^{(i)\,k}_{i_1i_2,j_1j_2} + M^{(j)\,k}_{j_1j_2,i_1i_2}\right) = \frac{N-2}{2}\left(\epsilon_{i_1i_2j_2}\delta^k_{j_1} - \epsilon_{i_1i_2j_1}\delta^k_{j_2} + \epsilon_{j_1j_2i_1}\delta^k_{i_2} - \epsilon_{j_1j_2i_2}\delta^k_{i_1}\right)\,. \tag{7.27}$$

However, this approach conflicts with a recursive construction of the amplitudes, since it would require a consideration of the complete amplitude, using more and more complicated variations on (7.22).

Evaluation Strategy

Faced with a non-free tensor algebra, we have to choose an evaluation strategy. If we encounter a pair of ϵ and $\bar\epsilon$ with a joint contracted index, we should use (7.8) immediately. Note this does not yet resolve all ambiguities because there are cases in which an ϵ (or $\bar\epsilon$) can be contracted with more than one $\bar\epsilon$ (or ϵ) and we have to make a choice. However, we will obtain equivalent, if not manifestly equal, results in the case $n = N$.

In the case of disconnected pairs of ϵ and $\bar\epsilon$, we have to decide whether to use (7.6) to produce an amplitude that contains *only* ϵ (or $\bar\epsilon$). A disadvantage of this strategy is that each application of (7.6) produces $n!$ permutations of Kronecker deltas that have to be evaluated. However, keeping all disconnected ϵ and $\bar\epsilon$ will require to try many more color flows for the complete amplitude since there can be both incoming and outgoing lines that are not continued through the diagram. Therefore we decide to *always apply* (7.6) *as soon as possible.*

There remains to determine a prescription for consistently selecting the ϵ-$\bar\epsilon$-pairs to be contracted if there is more than one possibility. In particular, we *must not* give in to the temptation of premature optimization: when evaluating the color flows for a 1POW in a fusion (cf. *Color_Fusion*, pages 122 ff), we know the color flows for all incoming lines. One is therefore tempted to choose a pair with disjoint color flows, since the evaluation for this color flow could be terminated immediately. Unfortunately, this would not be consistent, because a different choice would be made for different color flows. Imagine, for example the fusion of $\bar\epsilon^{123}$ with $\epsilon_{123}\epsilon_{456}$ or $\epsilon_{456}\epsilon_{123}$. In both cases, we will obtain $3!\,\epsilon_{456}$ or 0, depending of our choice. If we were to attempt to optimize the evaluation and make the choice that results in 0, we would not get the correct result.

Instead we have to make the *same* choice for every external color flow. This requires ignoring the external color flow indices. For this to work, we must use an ordered data structure for the unprocessed ϵ and $\bar\epsilon$. In particular, we *must not* use a *Set*, where the ordering of the elements will typically depend on the color flow indices. Instead, we should use lists and apply (7.6) consequently to the heads of these lists. Note that selecting contracted mutually ϵ-$\bar\epsilon$-pairs does not introduce a dependency on the external color flow indices!

```
let is_tadpole = function
  | Arrow (tail, tip) → matching_summation tail tip
  | Ghost _ → false

let epsilon = function
  | [] → invalid_arg "Arrow.epsilon:␣rank␣0"
  | [_] → invalid_arg "Arrow.epsilon:␣rank␣1"
  | tips → List.map (fun tip → I tip) tips

let epsilon_bar = function
  | [] → invalid_arg "Arrow.epsilon_bar:␣rank␣0"
  | [_] → invalid_arg "Arrow.epsilon_bar:␣rank␣1"
  | tails → List.map (fun tail → I tail) tails
```

Composite Arrows.

```
let rec chain = function
  | [] → []
  | [a] → [a => a]
  | [a; b] → [a => b]
  | a :: (b :: _ as rest) → (a => b) :: chain rest

let rec cycle' a = function
  | [] → [a => a]
  | [b] → [b => a]
  | b :: (c :: _ as rest) → (b => c) :: cycle' a rest

let cycle = function
  | [] → []
  | a :: _ as a_list → cycle' a a_list

module Test =
  struct

    open OUnit

    let suite_chain =
      "chain" >:::
        [ "[]" >:: (fun () → assert_equal [] (chain []));
          "[1]" >:: (fun () → assert_equal [1 => 1] (chain [1]));
          "[1;2]" >:: (fun () → assert_equal [1 => 2] (chain [1; 2]));
          "[1;2;3]" >:: (fun () → assert_equal [1 => 2; 2 => 3] (chain [1; 2; 3]));
          "[1;2;3;4]" >:: (fun () → assert_equal [1 => 2; 2 => 3; 3 => 4] (chain [1; 2; 3; 4])) ]

    let suite_cycle =
      "cycle" >:::
        [ "[]" >:: (fun () → assert_equal [] (cycle []));
          "[1]" >:: (fun () → assert_equal [1 => 1] (cycle [1]));
          "[1;2]" >:: (fun () → assert_equal [1 => 2; 2 => 1] (cycle [1; 2]));
          "[1;2;3]" >:: (fun () → assert_equal [1 => 2; 2 => 3; 3 => 1] (cycle [1; 2; 3]));

          "[1;2;3;4]" >:: (fun () → assert_equal [1 => 2; 2 => 3; 3 => 4; 4 => 1] (cycle [1; 2; 3; 4])) ]

    let suite_take =
      "take" >:::
        [ "1␣[]" >:: (fun () → assert_equal None (take_first_match_opt 1 []));
          "1␣[1]" >:: (fun () → assert_equal (Some ([], [])) (take_first_match_opt 1 [1]));
          "1␣[2;3;4]" >:: (fun () → assert_equal None (take_first_match_opt 1 [2;3;4]));
          "1␣[1;2;3]" >:: (fun () → assert_equal (Some ([], [2;3])) (take_first_match_opt 1 [1;2;3]));
          "2␣[1;2;3]" >:: (fun () → assert_equal (Some ([1], [3])) (take_first_match_opt 2 [1;2;3]));
          "3␣[1;2;3]" >:: (fun () → assert_equal (Some ([2;1], [])) (take_first_match_opt 3 [1;2;3])) ]

    let suite_take2 =
      "take2" >:::
        [ "[]␣[]" >::
              (fun () → assert_equal None (take_first_matching_pair_opt [] []));

          "[]␣[1;2;3]" >::
            (fun () → assert_equal None (take_first_matching_pair_opt [] [1;2;3]));

          "[1]␣[2;3;4]" >::
            (fun () → assert_equal None (take_first_matching_pair_opt [1] [2;3;4]));

          "[2;3;4]␣[1]" >::
            (fun () → assert_equal None (take_first_matching_pair_opt [2;3;4] [1]));

          "[1;2;3]␣[4;5;6;7]" >::
            (fun () → assert_equal None (take_first_matching_pair_opt [1;2;3] [4;5;6;7]));

          "[1]␣[1;2;3]" >::
            (fun () →
              assert_equal
                (Some (([],[]), ([],[2;3])))
                (take_first_matching_pair_opt [1] [1;2;3]));

          "[1;2;3]␣[1;20;30]" >::
            (fun () →
              assert_equal
                (Some (([],[2;3]), ([],[20;30])))
                (take_first_matching_pair_opt [1;2;3] [1;20;30]));

          "[1;2;3;4;5;6]␣[10;20;4;30;40]" >::
            (fun () →
              assert_equal
                (Some (([3;2;1],[5;6]), ([20;10],[30;40])))
                (take_first_matching_pair_opt [1;2;3;4;5;6] [10;20;4;30;40])) ]

    let suite_replace =
      "replace" >:::
        [ "1␣10␣[]" >:: (fun () → assert_equal None (replace_first_opt 1 2 []));
          "1␣10␣[1]" >:: (fun () → assert_equal (Some [10]) (replace_first_opt 1 10 [1]));
          "1␣[2;3;4]" >:: (fun () → assert_equal None (replace_first_opt 1 10 [2;3;4]));
          "1␣[1;2;3]" >:: (fun () → assert_equal (Some [10;2;3]) (replace_first_opt 1 10 [1;2;3]));
          "2␣[1;2;3]" >:: (fun () → assert_equal (Some [1;10;3]) (replace_first_opt 2 10 [1;2;3]));
          "3␣[1;2;3]" >:: (fun () → assert_equal (Some [1;2;10]) (replace_first_opt 3 10 [1;2;3])) ]

    let suite =
      "Arrow" >:::
        [suite_chain;
         suite_cycle;
         suite_take;
         suite_take2;
         suite_replace]

    let suite_long =
      "Arrow␣long" >:::
        []

  end

let pp_free fmt f =
  Format.fprintf fmt "%s" (free_to_string f)

let pp_factor fmt f =
  Format.fprintf fmt "%s" (factor_to_string f)
```

—8— Birdtracks

8.1 Interface of Birdtracks

In this module, we implement birdtracks operations on expressions of type t as generally as possible. Module *SU3* (cf. chapter 9), will provide the group specific constructors for type t in the special case SU(N_C) or SU(3).

8.1.1 Types

If there are no ϵs or $\bar\epsilon$s, a term is simply a list of arrows with a coefficient that is a polynomial, allowing negative powers, in N_C. The the type of arrows is not fixed, because *Arrow* has both *free* arrows without summation indices and *factor* arrows that contain summation indices.

type α *aterm* = { *coeff* : *Algebra.Laurent.t*; *arrows* : α *list* }

If there are ϵs, we add them ...

type (α, ε) *eterm* = α *aterm* $\times$ ε *NEList.t*

... and the same for $\bar\epsilon$s.

type (α, β) *bterm* = α *aterm* $\times$ β *NEList.t*

Assuming that ϵ-$\bar\epsilon$-pairs are always reduced as soon as possible, these three alternatives are exhaustive.

type (α, ε, β) *term* =
  | *Arrows* of α *aterm*
  | *Epsilons* of (α, ε) *eterm*
  | *Epsilon_Bars* of (α, β) *bterm*

In the public interface, we deal only with *free* indices, without summation indices.

type *free* = (*Arrow.free*, *Arrow.free_eps*, *Arrow.free_eps_bar*) *term*

An expression is just a sum of terms.

type t = *free list*

8.1.2 Functions

Strip out redundancies.

val *canonicalize* : $t \to t$

Substitute a specific value for N_C. Mainly for debugging.

val *with_nc* : *int* $\to t \to t$

Debugging, logging, etc.

val *to_string* : $t \to$ *string*
val *to_string_raw* : $t \to$ *string*

Extract the number if the birdtrack contains no arrows, ϵs or $\bar\epsilon$s.

val *number* : $t \to$ *Algebra.Laurent.t option*

Test for trivial color flows that correspond to unity.

val *is_unit* : $t \to$ *bool*

Test for vanishing coefficients.

val *is_null* : *t* → *bool*

Purely numeric factors, implemented as Laurent polynomials (cf. *Algebra.Laurent* in N_C with complex rational coefficients and without arrows.

val *const* : *Algebra.Laurent.t* → *t*
val *null* : *t* (∗ 0 ∗)
val *one* : *t* (∗ 1 ∗)
val *two* : *t* (∗ 2 ∗)
val *minus* : *t* (∗ −1 ∗)
val *int* : *int* → *t* (∗ n ∗)
val *fraction* : *int* → *t* (∗ $1/n$ ∗)
val *nc* : *t* (∗ N_C ∗)
val *over_nc* : *t* (∗ $1/N_C$ ∗)
val *imag* : *t* (∗ i ∗)

Shorthand: $\{(c_i, p_i)\}_i \to \sum_i c_i (N_C)^{p_i}$

val *ints* : (*int* × *int*) *list* → *t*

val *scale* : *Algebra.Laurent.c* → *t* → *t*

val *sum* : *t list* → *t*
val *diff* : *t* → *t* → *t*
val *times* : *t* → *t* → *t*
val *multiply* : *t list* → *t*

For convenience, here are infix versions of the above operations.

```
module Infix : sig
  val ( +++ ) : t → t → t
  val ( −−− ) : t → t → t
  val ( *** ) : t → t → t
end
```

We can compute the f_{abc} and d_{abc} invariant tensors from the generators of an arbitrary representation:

$$f_{a_1a_2a_3} = -\mathrm{i}\,\mathrm{tr}\left(T_{a_1}\left[T_{a_2}, T_{a_3}\right]_-\right) = -\mathrm{i}\,\mathrm{tr}\left(T_{a_1}T_{a_2}T_{a_3}\right) + \mathrm{i}\,\mathrm{tr}\left(T_{a_1}T_{a_3}T_{a_2}\right) \tag{8.1a}$$

$$d_{a_1a_2a_3} = \mathrm{tr}\left(T_{a_1}\left[T_{a_2}, T_{a_3}\right]_+\right) = \mathrm{tr}\left(T_{a_1}T_{a_2}T_{a_3}\right) + \mathrm{tr}\left(T_{a_1}T_{a_3}T_{a_2}\right) \tag{8.1b}$$

assuming the normalization $\mathrm{tr}(T_aT_b) = \delta_{ab}$.

NB: this uses the summation indices -1, -2 and -3. Therefore it *must not* appear unevaluated more than once in a product!

val *f_of_rep* : (*int* → *int* → *int* → *t*) → *int* → *int* → *int* → *t*
val *d_of_rep* : (*int* → *int* → *int* → *t*) → *int* → *int* → *int* → *t*

Rename the indices of endpoints in a birdtrack. This is required by our application in *Colorize.It* to match the permutations of lines at a vertex.

val *relocate* : (*int* → *int*) → *t* → *t*

Revert the direction of all lines in a birdtrack.

val *rev* : *t* → *t*

Pretty printer for the toplevel.

val *pp* : *Format.formatter* → *t* → *unit*

Support for unit tests.

val *equal* : *t* → *t* → *unit*
val *assert_zero_vertex* : *t* → *unit*

module *Test* : sig val *suite* : *OUnit.test* val *suite_long* : *OUnit.test* end

8.2 Implementation of Birdtracks

8.2.1 Types

```
module QC = Algebra.QC
module L = Algebra.Laurent
module A = Arrow
open A.Infix
```

There can be one or more ϵ or $\bar{\epsilon}$, but not both at the same time.
I wanted to use a GADT with Peano numerals to track the number of ϵ and $\bar{\epsilon}$ in the type system. However, I would have needed to implement a "multiplication" function of the type *'n1 term → 'n2 term → ('n1 + 'n2) term* that I have not been able to implement using Peano numerals for the type variables *'n1* and *'n2*, due to the lack of an addition operator for Peano numerals in the type system.

Therefore I will use normal lists, sacrificing some type safety.

```
type α aterm = { coeff : L.t; arrows : α list }
type (α, ε) eterm = α aterm × ε NEList.t
type (α, β) bterm = α aterm × β NEList.t
```

```
type (α, ε, β) term =
  | Arrows of α aterm
  | Epsilons of (α, ε) eterm
  | Epsilon_Bars of (α, β) bterm
```

Having already added type annotations for polymorphic recursion, I could use a simple GADT instead of an ADT at the toplevel, trying to maintain some unboxing potential:

```
type (α, ε, β) term = | Arrows : α aterm → (α, ε, β) term | Epsilons : (α, ε) eterm →
(α, ε, β) term | Epsilon_Bars : (α, β) bterm → (α, ε, β) term
```

but it is not obvious that this produces a real performance benefit.

```
type afree = A.free aterm
type efree = (A.free, A.free_eps) eterm
type bfree = (A.free, A.free_eps_bar) bterm
type free = (A.free, A.free_eps, A.free_eps_bar) term
```

```
type afactor = A.factor aterm
type efactor = (A.factor, A.factor_eps) eterm
type bfactor = (A.factor, A.factor_eps_bar) bterm
type factor = (A.factor, A.factor_eps, A.factor_eps_bar) term
```

```
type t = free list
```

8.2.2 Functions

```
let tips_and_tails_of_aterm aterm =
  List.fold_left
    (fun (tips, tails) arrow →
      (List.rev_append (A.tips arrow) tips,
        List.rev_append (A.tails arrow) tails))
    ([], []) aterm.arrows
```

```
let tips_and_tails_raw : free → A.tip list × A.tail list = function
  | Arrows aterm → tips_and_tails_of_aterm aterm
  | Epsilons (aterm, epsilons) →
      let tips, tails = tips_and_tails_of_aterm aterm in
      (List.concat (tips :: NEList.to_list epsilons), tails)
  | Epsilon_Bars (aterm, epsilon_bars) →
      let tips, tails = tips_and_tails_of_aterm aterm in
      (tips, List.concat (tails :: NEList.to_list epsilon_bars))
```

```
let tips_and_tails term =
  let tips, tails = tips_and_tails_raw term in
  (List.sort compare tips, List.sort compare tails)
```

Expressions

```
let const coeff = [ Arrows { coeff; arrows = [] } ]
let ints pairs = const (L.ints pairs)
let null = const L.null
let fraction n = const (L.fraction n)
let one = const (L.int 1)
let two = const (L.int 2)
let minus = const (L.int (−1))
let int n = const (L.int n)
let nc = const (L.nc 1)
let over_nc = const (L.ints [(1, − 1)])
let imag = const (L.imag 1)
```

```
module AMap = Pmap.Tree
```

```
let psort alist = List.sort compare alist
let ne_psort alist = NEList.sort compare alist
```

```
let find_term_opt term map =
  AMap.find_opt compare term map
```

```
let map_aterm fc fa aterm =
  { coeff = fc aterm.coeff; arrows = fa aterm.arrows }
```

```
let map_term fc fa fe fb = function
  | Arrows aterm → Arrows (map_aterm fc fa aterm)
  | Epsilons (aterm, elist) → Epsilons (map_aterm fc fa aterm, fe elist)
  | Epsilon_Bars (aterm, blist) → Epsilon_Bars (map_aterm fc fa aterm, fb blist)
```

```
let map_term_deep fc fa fe fb term =
  map_term fc (List.map fa) (NEList.map fe) (NEList.map fb) term
```

```
let canonicalize_aterm term =
  map_aterm Fun.id psort term
```

We're *not yet* canonicalizing the ϵ and $\bar\epsilon$ themselves. This could be done, if necessary, using *Combinatorics.sort_signed* to keep track of the signs. While we're debugging, it could be beneficial to keep the indices where they are.

```
let canonicalize_term : type a e b. (a, e, b) term → (a, e, b) term =
  fun term →
  map_term Fun.id psort ne_psort ne_psort term
```

```
let split_coeff : type a e b. (a, e, b) term → L.t × (a, e, b) term = function
  | Arrows aterm → (aterm.coeff, Arrows { aterm with coeff = L.int 1 })
  | Epsilons (aterm, epsilons) →
      (aterm.coeff, Epsilons ({ aterm with coeff = L.int 1 }, epsilons))
  | Epsilon_Bars (aterm, epsilon_bars) →
      (aterm.coeff, Epsilon_Bars ({ aterm with coeff = L.int 1 }, epsilon_bars))
```

```
let inject_coeff : type a e b. L.t → (a, e, b) term → (a, e, b) term =
  fun coeff → map_term (fun _ → coeff) Fun.id Fun.id Fun.id
```

Note that the final result must be a homogeneous list with all elements containing the same number of ϵ and $\bar\epsilon$, because otherwise the number of incoming and outgoing color lince would not match.

Nevertheless, we might have to work very hard to avoid too much code duplication.

```
let canonicalize : type a e b. (a, e, b) term list → (a, e, b) term list =
  fun terms →
  let map =
    List.fold_left
      (fun acc term →
        let coeff, term = split_coeff (canonicalize_term term) in
        if L.is_null coeff then
```

```
            acc
          else
            match find_term_opt term acc with
            | None → AMap.add compare term coeff acc
            | Some coeff′ →
               let coeff″ = L.add coeff coeff′ in
               if L.is_null coeff″ then
                 AMap.remove compare term acc
               else
                 AMap.add compare term coeff″ acc)
        AMap.empty terms in
  if AMap.is_empty map then
    []
  else
    AMap.fold (fun term coeff acc → inject_coeff coeff term :: acc) map []

let number v =
  match canonicalize v with
  | [] → Some L.null
  | [Arrows { coeff; arrows = [] }] → Some coeff
  | _ → None

let is_null v =
  match canonicalize v with
  | [] → true
  | _ → false

let is_unit v =
  match canonicalize v with
  | [Arrows { coeff; arrows = [] }] → coeff = L.unit
  | _ → false

let with_nc nc t =
  let substitute c = L.const (L.eval (QC.int nc) c) in
  canonicalize (List.map (map_term substitute Fun.id Fun.id Fun.id) t)

let aterm_to_string f term =
  match term.arrows with
  | [] → Printf.sprintf "(%s)" (L.to_string "N" term.coeff)
  | arrows →
      Printf.sprintf
        "(%s)␣*␣%s"
        (L.to_string "N" term.coeff) (ThoList.to_string f arrows)

let to_string1_aux fa fe fb = function
  | Arrows aterm → aterm_to_string fa aterm
  | Epsilons (aterm, epsilons) →
      aterm_to_string fa aterm ^ "␣*␣" ^ ThoList.to_string fe (NEList.to_list epsilons)
  | Epsilon_Bars (aterm, epsilon_bars) →
      aterm_to_string fa aterm ^ "␣*␣" ^ ThoList.to_string fb (NEList.to_list epsilon_bars)

let to_string1 term =
  to_string1_aux A.free_to_string A.free_eps_to_string A.free_eps_bar_to_string term

let to_string_raw terms =
  ThoList.to_string to_string1 terms

let to_string terms =
  to_string_raw (canonicalize terms)

let pp fmt v =
  Format.fprintf fmt "%s" (to_string v)

let relocate1 f term =
  map_term_deep Fun.id (A.relocate f) (List.map (A.relocate_tip f)) (List.map (A.relocate_tail f)) term

let relocate f = List.map (relocate1 f)
let rev_aterm aterm =
  { aterm with arrows = List.map A.rev aterm.arrows }

let rev1 = function
  | Arrows aterm → Arrows (rev_aterm aterm)
  | Epsilons (aterm, elist) → Epsilon_Bars (rev_aterm aterm, NEList.map A.rev_eps elist)
  | Epsilon_Bars (aterm, blist) → Epsilons (rev_aterm aterm, NEList.map A.rev_eps_bar blist)

let rev = List.map rev1

let of_afactor aterm =
  map_aterm Fun.id (List.map A.of_factor) aterm

let of_factor term =
  map_term_deep Fun.id A.of_factor A.of_factor_eps A.of_factor_eps_bar term

let to_left_factor is_sum term =
  map_term_deep Fun.id
    (A.to_left_factor is_sum)
    (A.to_left_factor_eps is_sum)
    (A.to_left_factor_eps_bar is_sum)
    term

let to_right_factor is_sum term =
  map_term_deep Fun.id
    (A.to_right_factor is_sum)
    (A.to_right_factor_eps is_sum)
    (A.to_right_factor_eps_bar is_sum)
    term
```

We start with the simply recursive evaluation functions, leaving the the more complicated mutually recursive functions for later.
Add one *arrow* to a list of arrows, updating *coeff* if necessary. Accumulate already processed arrows in *seen*. Returns *None* if there is a mismatch (a gluon meeting a ghost) and *Some afactor* containing a coefficient and a list of arrows otherwise.
We assume that the trivial cases of no summation indices and the arrow looping back to itself have already been filtered out.

```
let rec add_arrow_to_arrows_list' coeff seen arrow = function
  | [] → (* visited all arrows: no opportunities for further matches *)
      Some ({ coeff; arrows = arrow :: seen })
  | arrow' :: arrows' →
      begin match A.merge_arrow_arrow arrow arrow' with
      | A.Mismatch →
          None
      | A.Ghost_Match → (* replace matching ghosts by −1/N_C *)
          Some ({ coeff = L.mul (L.over_nc (−1)) coeff;
                  arrows = List.rev_append seen arrows' })
      | A.Loop_Match → (* replace a loop by N_C *)
          Some ({ coeff = L.mul (L.nc 1) coeff;
                  arrows = List.rev_append seen arrows' })
      | A.Match arrow'' → (* two arrows have been merged into one *)
          if A.is_free arrow'' then (* no opportunities for further matches *)
            Some ({ coeff; arrows = arrow'' :: List.rev_append seen arrows' })
          else (* the new arrow'' ist not yet saturated, try again: *)
            add_arrow_to_arrows_list' coeff seen arrow'' arrows'
      | A.No_Match → (* recurse to the remaining arrows *)
          add_arrow_to_arrows_list' coeff (arrow' :: seen) arrow arrows'
      end

let add_arrow_to_arrows_list coeff arrow arrows =
  add_arrow_to_arrows_list' coeff [] arrow arrows
```

Similarly, add one *arrow* to a list of ϵ and accumulate already processed arrows in *seen*. Returns [] if there is no match. Note that there is never the need to update the coefficient and that only the tail of the *arrow* can match.

```
let rec add_arrow_to_epsilon_list' seen arrow = function
  | [] → []
  | epsilon :: epsilons →
      begin match A.merge_arrow_eps arrow epsilon with
      | A.Mismatch_Eps → []
      | A.Match_Eps epsilon' → List.rev_append seen (epsilon' :: epsilons)
      | A.No_Match_Eps → add_arrow_to_epsilon_list' (epsilon :: seen) arrow epsilons
      end
```

```
let add_arrow_to_epsilon_list arrow epsilons =
  add_arrow_to_epsilon_list' [] arrow epsilons
```

Same preocedure for adding one *arrow* to a list of $\bar\epsilon$.

```
let rec add_arrow_to_epsilon_bar_list' seen arrow = function
  | [] → []
  | epsilon_bar :: epsilon_bars →
      begin match A.merge_arrow_eps_bar arrow epsilon_bar with
      | A.Mismatch_Eps → []
      | A.Match_Eps epsilon_bar' → List.rev_append seen (epsilon_bar' :: epsilon_bars)
      | A.No_Match_Eps → add_arrow_to_epsilon_bar_list' (epsilon_bar :: seen) arrow epsilon_bars
      end
```

```
let add_arrow_to_epsilon_bar_list arrow epsilon_bars =
  add_arrow_to_epsilon_bar_list' [] arrow epsilon_bars
```

Avoid a recursion, if there is no summation index in *arrow*. Likewise, if *arrow* loops back to itself, just replace it by a factor of N_C.

```
let add_arrow_to_aterm_trivial : A.factor → afactor → afactor option =
  fun arrow term →
  if A.is_free arrow then
    Some ({ coeff = term.coeff; arrows = arrow :: term.arrows })
  else if A.is_tadpole arrow then
    Some ({ coeff = L.mul (L.nc 1) term.coeff; arrows = term.arrows })
  else
    None
```

Straightforwardly add an arrow or an arrow list to a term containing no ϵ or $\bar\epsilon$, using the functions implemented above.

```
let add_arrow_to_aterm : A.factor → afactor → afactor option =
  fun arrow term →
  match add_arrow_to_aterm_trivial arrow term with
  | None → add_arrow_to_arrows_list term.coeff arrow term.arrows
  | term_opt → term_opt
```

```
let add_arrow_list_to_aterm : A.factor list → afactor → afactor option =
  fun arrows term →
  ThoList.fold_left_opt (Fun.flip add_arrow_to_aterm) term arrows
```

Adding an arrow or an arrow list to a term containing ϵ or $\bar\epsilon$ is not more complicated, we only have to make two attempts.

Caveat: if the arrow matches one of the ϵs and this ϵ has a tip appearing among the remaining tips of this ϵ, the result should be set to zero explicitelty. But such expressions are illegal anyway!

```
let add_arrow_to_eterm : A.factor → efactor → efactor option =
  fun arrow (aterm, epsilons) →
  match add_arrow_to_aterm_trivial arrow aterm with
  | Some aterm → Some (aterm, epsilons)
  | None →
      begin match add_arrow_to_epsilon_list arrow (NEList.to_list epsilons) with
      | [] →
          begin match add_arrow_to_arrows_list aterm.coeff arrow aterm.arrows with
          | None → None
          | Some aterm → Some (aterm, epsilons)
          end
      | epsilon :: epsilons → Some (aterm, NEList.make epsilon epsilons)
      end
```

```
let add_arrow_list_to_eterm : A.factor list → efactor → efactor option =
  fun arrows term →
  ThoList.fold_left_opt (Fun.flip add_arrow_to_eterm) term arrows
```

```
let add_arrow_to_bterm : A.factor → bfactor → bfactor option =
  fun arrow (aterm, epsilon_bars) →
  match add_arrow_to_aterm_trivial arrow aterm with
  | Some aterm → Some (aterm, epsilon_bars)
  | None →
      begin match add_arrow_to_epsilon_bar_list arrow (NEList.to_list epsilon_bars) with
      | [] →
          begin match add_arrow_to_arrows_list aterm.coeff arrow aterm.arrows with
          | None → None
          | Some aterm → Some (aterm, epsilon_bars)
          end
      | epsilon_bar :: epsilon_bars → Some (aterm, NEList.make epsilon_bar epsilon_bars)
      end
```

```
let add_arrow_list_to_bterm : A.factor list → bfactor → bfactor option =
  fun arrows term →
  ThoList.fold_left_opt (Fun.flip add_arrow_to_bterm) term arrows
```

Adding an ϵ to a term containing ϵs is trivial, if there are no summation indices. Otherwise, we add the arrows back in to find matches.

Here's potential for optimization, since the arrows can only match the new ϵ.

```
let add_epsilon_to_eterm : A.factor_eps → efactor → efactor option =
  fun epsilon (aterm, epsilons) →
  if A.is_free_eps epsilon then
    Some (aterm, NEList.cons epsilon epsilons)
  else
    let coeff = { coeff = aterm.coeff; arrows = []} in
    add_arrow_list_to_eterm aterm.arrows (coeff, NEList.cons epsilon epsilons)
```

```
let add_epsilon_list_to_eterm : A.factor_eps list → efactor → efactor option =
  fun epsilons eterm →
  ThoList.fold_left_opt (Fun.flip add_epsilon_to_eterm) eterm epsilons
```

Once more for $\bar{\epsilon}$.

```
let add_epsilon_bar_to_bterm : A.factor_eps_bar → bfactor → bfactor option =
  fun epsilon_bar (aterm, epsilon_bars) →
  if A.is_free_eps_bar epsilon_bar then
    Some (aterm, NEList.cons epsilon_bar epsilon_bars)
  else
    let coeff = { coeff = aterm.coeff; arrows = []} in
    add_arrow_list_to_bterm aterm.arrows (coeff, NEList.cons epsilon_bar epsilon_bars)
```

```
let add_epsilon_bar_list_to_bterm : A.factor_eps_bar list → bfactor → bfactor option =
  fun epsilon_bars bterm →
  ThoList.fold_left_opt (Fun.flip add_epsilon_bar_to_bterm) bterm epsilon_bars
```

Here we simply have to select the correct function.

```
let add_arrow_to_term : A.factor → factor → factor option =
  fun arrow → function
  | Arrows aterm →
      Option.map (fun a → Arrows a) (add_arrow_to_aterm arrow aterm)
  | Epsilons eterm →
      Option.map (fun e → Epsilons e) (add_arrow_to_eterm arrow eterm)
  | Epsilon_Bars bterm →
      Option.map (fun b → Epsilon_Bars b) (add_arrow_to_bterm arrow bterm)
```

```
let add_arrow_list_to_term : A.factor list → factor → factor option =
  fun arrows term →
  ThoList.fold_left_opt (Fun.flip add_arrow_to_term) term arrows
```

```
let scale_aterm : L.t → afactor → afactor =
  fun coeff aterm →
  { coeff = L.mul coeff aterm.coeff; arrows = aterm.arrows}
```

```
let scale_eterm : L.t → efactor → efactor =
  fun coeff (aterm, epsilons) →
  (scale_aterm coeff aterm, epsilons)
```

```
let scale_bterm : L.t → bfactor → bfactor =
  fun coeff (aterm, epsilon_bars) →
  (scale_aterm coeff aterm, epsilon_bars)
```

```
let scale_term : L.t → factor → factor =
  fun coeff → function
  | Arrows aterm → Arrows (scale_aterm coeff aterm)
  | Epsilons eterm → Epsilons (scale_eterm coeff eterm)
  | Epsilon_Bars bterm → Epsilon_Bars (scale_bterm coeff bterm)
```

```
let aterm_times_aterm : afactor → afactor → afactor option =
  fun aterm1 aterm2 →
  Option.map (scale_aterm aterm1.coeff) (add_arrow_list_to_aterm aterm1.arrows aterm2)
```

Almost the same as *aterm_times_term* below, but the arguments are exchanged an the result are *factor*s and not *free*.

```
let term_times_aterm : factor → afactor → factor list =
  fun term aterm →
  match add_arrow_list_to_term aterm.arrows term with
  | None → []
  | Some factor → [scale_term aterm.coeff factor]
```

The return type is *factor list*, because adding a product of ϵ and $\bar\epsilon$ will produce a sum of terms and the result can be a *afactor*, *efactor* or *bfactor* depending on the number of ϵs and $\bar\epsilon$s in the arguments.

Add more tests for multiple ϵ and $\bar\epsilon$! I'm not yet convinced only from playing with the toplevel.

Calling *aterm_times_aterm* in each recursion step and only using the last result ist wasteful. Find a better way!

This would fail if one of *epsilons* or *epsilon_bars* is empty (which does not happen). We could try to replace the ε *list* in type (α, ε) *eterm* by a non empty list type (and similarly for ε *list* in type (α, β) *bterm*.

But is it worth the effort? It probably enough to hide the list in a private ADT that can be deconstructed, but requires a smart constructor that requires at least one element.

```
let rec match_eterm_and_bterm : efactor → bfactor → factor list =
  fun (aterm1, epsilons) (aterm2, epsilon_bars) →
  match NEList.snoc_opt epsilons, NEList.snoc_opt epsilon_bars with
  | (epsilon, epsilons_opt), (epsilon_bar, epsilon_bars_opt) →
      begin match aterm_times_aterm aterm1 aterm2 with
      | None → []
      | Some aterm →
          match A.merge_eps_eps_bar epsilon epsilon_bar with
          | None → []
          | Some (even, odd) →
              let even = List.rev_map (fun arrows → { coeff = L.unit; arrows }) even
              and odd = List.rev_map (fun arrows → { coeff = L.neg L.unit; arrows }) odd in
              let terms =
                match epsilons_opt, epsilon_bars_opt with
                | None, None → [Arrows aterm]
                | Some epsilons, None → [Epsilons (aterm, epsilons)]
                | None, Some epsilon_bars → [Epsilon_Bars (aterm, epsilon_bars)]
                | Some epsilon, Some epsilon_bars →
                    match_eterm_and_bterm (aterm1, epsilon) (aterm2, epsilon_bars) in
              Product.fold2
                (fun term aterm acc →
                  List.rev_append (term_times_aterm term aterm) acc)
                terms (List.rev_append even odd) []
      end
```

NB: we can reject the contributions with unsaturated summation indices from Ghost contributions to T_a only *after* adding all arrows that might saturate an open index.
Note that a negative index might be summed only later in a sequence of binary products and must therefore be treated as free in this product. Therefore, we have to classify the indices as summation indices *not only* based on their sign, but in addition based on whether they appear in both factors. Only then can we reject surviving ghosts.

```
module ESet =
  Set.Make
    (struct
      type t = A.endpoint
      let compare = compare
    end)

let negatives_arrows arrows acc =
  List.fold_right (fun a → List.fold_right ESet.add (A.negatives a)) arrows acc

let negatives_eps epsilons acc =
  NEList.fold_right
    (fun e → List.fold_right ESet.add (A.negatives_eps e))
    epsilons acc

let negatives_eps_bar epsilon_bars acc =
  NEList.fold_right
    (fun b → List.fold_right ESet.add (A.negatives_eps_bar b))
    epsilon_bars acc

let negatives = function
  | Arrows aterm → negatives_arrows aterm.arrows ESet.empty
  | Epsilons (aterm, epsilons) →
      negatives_eps epsilons (negatives_arrows aterm.arrows ESet.empty)
  | Epsilon_Bars (aterm, epsilon_bars) →
      negatives_eps_bar epsilon_bars (negatives_arrows aterm.arrows ESet.empty)

let aterm_times_term : afactor → factor → free list =
  fun aterm term →
  match add_arrow_list_to_term aterm.arrows term with
  | None → []
  | Some factor → [of_factor (scale_term aterm.coeff factor)]

let eterm_times_eterm : efactor → efactor → free list =
  fun (aterm, epsilons) eterm →
  match add_epsilon_list_to_eterm (NEList.to_list epsilons) eterm with
  | None → []
  | Some factor →
      begin match add_arrow_list_to_eterm aterm.arrows factor with
      | None → []
      | Some factor → [of_factor (Epsilons (scale_eterm aterm.coeff factor))]
      end

let bterm_times_bterm : bfactor → bfactor → free list =
  fun (aterm, epsilon_bars) bterm →
  match add_epsilon_bar_list_to_bterm (NEList.to_list epsilon_bars) bterm with
  | None → []
  | Some factor →
```

```
    begin match add_arrow_list_to_bterm aterm.arrows factor with
    | None → []
    | Some factor → [of_factor (Epsilon_Bars (scale_bterm aterm.coeff factor))]
    end

let eterm_times_bterm : efactor → bfactor → free list =
  fun eterm bterm →
  List.map of_factor (match_eterm_and_bterm eterm bterm)

let times1 term1 term2 =
  let summations = ESet.inter (negatives term1) (negatives term2) in
  let is_sum i = ESet.mem i summations in
  match to_left_factor is_sum term1, to_right_factor is_sum term2 with
  | Arrows aterm, factor | factor, Arrows aterm →
     aterm_times_term aterm factor
  | Epsilons eterm1, Epsilons eterm2 →
     eterm_times_eterm eterm1 eterm2
  | Epsilon_Bars bterm1, Epsilon_Bars bterm2 →
     bterm_times_bterm bterm1 bterm2
  | Epsilons eterm, Epsilon_Bars bterm
    | Epsilon_Bars bterm, Epsilons eterm →
     eterm_times_bterm eterm bterm

let sum terms =
  canonicalize (List.concat terms)

let times term term′ =
  canonicalize
    (Product.fold2
       (fun x y → List.rev_append (times1 x y))
       term term′ [])
```

Is that more efficient than the following implementation?

Isn't that the more straightforward implementation?

```
let multiply = function
  | [] → []
  | term :: terms →
     canonicalize (List.fold_left times term terms)

let scale1 : type a e b. L.c → (a, e, b) term → (a, e, b) term =
  fun q term →
  map_term (L.scale q) Fun.id Fun.id Fun.id term

let scale q = List.map (scale1 q)

let diff term1 term2 =
  canonicalize (List.rev_append term1 (scale (QC.int (−1)) term2))

module Infix =
  struct
    let ( +++ ) term term′ = sum [term; term′]
    let ( −−− ) = diff
    let ( ∗∗∗ ) = times
  end

open Infix
```

Compute $\mathrm{tr}(r(T_a)r(T_b)r(T_c))$. NB: this uses the summation indices -1, -2 and -3. Therefore it *must not* appear unevaluated more than once in a product!

```
let trace3 r a b c =
  r a (−1) (−2) ∗∗∗ r b (−2) (−3) ∗∗∗ r c (−3) (−1)

let f_of_rep r a b c =
  minus ∗∗∗ imag ∗∗∗ (trace3 r a b c −−− trace3 r a c b)
```

$d_{abc} = \mathrm{tr}(r(T_a)[r(T_b), r(T_c)]_+)$

```
let d_of_rep r a b c =
  trace3 r a b c +++ trace3 r a c b
```

8.2.3 Unit Tests

```
let vertices_equal v1 v2 =
  is_null (v1 --- v2)
```

```
let assert_zero_vertex v =
  OUnit.assert_equal ~printer : to_string ~cmp : vertices_equal null v
```

As an extra protection agains vacuous tests, we make sure that the LHS does not vanish.

```
let equal v1 v2 =
  OUnit.assert_bool "LHS␣=␣0" (¬ (is_null v1));
  OUnit.assert_equal ~printer : to_string ~cmp : vertices_equal v1 v2
```

```
module Test =
  struct
    open OUnit

    let vertices_equal v1 v2 =
      (canonicalize v1) = (canonicalize v2)

    let eq v1 v2 =
      assert_equal ~printer : to_string_raw ~cmp : vertices_equal v1 v2

    let suite_times1 =
      "times1" >:::
        [ "merge␣two" >::
            (fun () →
              eq
                [Arrows { coeff = L.unit; arrows = 1 ==> 2 }]
                (times1
                    (Arrows { coeff = L.unit; arrows = 1 ==> −1 })
                    (Arrows { coeff = L.unit; arrows = −1 ==> 2 })));

          "merge␣two␣exchanged" >::
            (fun () →
              eq
                [Arrows { coeff = L.unit; arrows = 1 ==> 2 }]
                (times1
                    (Arrows { coeff = L.unit; arrows = −1 ==> 2 })
                    (Arrows { coeff = L.unit; arrows = 1 ==> −1 })));

          "ghost1" >::
            (fun () →
              eq
                [Arrows { coeff = L.over_nc (−1); arrows = 1 ==> 2 }]
                (times1
                    (Arrows { coeff = L.unit; arrows = [−1 => 2; ?? (−3)] })
                    (Arrows { coeff = L.unit; arrows = [ 1 => −1; ?? (−3)] })));

          "ghost2" >::
            (fun () →
              eq
                []
                (times1
                    (Arrows { coeff = L.unit; arrows = [ 1 => −1; ?? (−3)] })
                    (Arrows { coeff = L.unit; arrows = [−1 => 2; −3 => −4; −4 => −3] })));

          "ghost2␣exchanged" >::
            (fun () →
              eq
```

```
                    []
                    (times1
                        (Arrows { coeff = L.unit; arrows = [−1 => 2; −3 => −4; −4 => −3] })
                        (Arrows { coeff = L.unit; arrows = [ 1 => −1; ?? (−3)] }))) ]
   let suite_canonicalize =
      "canonicalize" >:::

         [ ]
   let suite =
      "Birdtracks" >:::
         [suite_times1;
          suite_canonicalize]
   let suite_long =
      "Birdtracks␣long" >:::
         []
end
```

—9—
SU(3)

Using the normalization $\mathrm{tr}(T_a T_b) = \delta_{ab}$, we can check the selfconsistency of the completeness relation

$$T_a^{i_1 j_1} T_a^{i_2 j_2} = \left(\delta^{i_1 j_2} \delta^{i_2 j_1} - \frac{1}{N_C} \delta^{i_1 j_1} \delta^{j_1 j_2} \right) \tag{9.1}$$

as

$$\begin{aligned} T_a^{i_1 j_1} T_a^{i_2 j_2} &= \mathrm{tr}\left(T_{a_1} T_{a_2}\right) T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} = T_{a_1}^{l_1 l_2} T_{a_2}^{l_2 l_1} T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} \\ &= \left(\delta^{l_1 j_1} \delta^{i_1 l_2} - \frac{1}{N_C} \delta^{l_1 l_2} \delta^{i_1 j_1} \right) \left(\delta^{l_2 j_2} \delta^{i_2 l_1} - \frac{1}{N_C} \delta^{l_2 l_1} \delta^{i_2 j_2} \right) = \left(\delta^{i_1 j_2} \delta^{i_2 j_1} - \frac{1}{N_C} \delta^{i_1 i_2} \delta^{j_2 j_1} \right) \end{aligned} \tag{9.2}$$

With

$$\mathrm{i} f_{a_1 a_2 a_3} = \mathrm{tr}\left(T_{a_1} \left[T_{a_2}, T_{a_3}\right]\right) = \mathrm{tr}\left(T_{a_1} T_{a_2} T_{a_3}\right) - \mathrm{tr}\left(T_{a_1} T_{a_3} T_{a_2}\right) \tag{9.3}$$

and

$$\begin{aligned} &\mathrm{tr}\left(T_{a_1} T_{a_2} T_{a_3}\right) T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} T_{a_3}^{i_3 j_3} = T_{a_1}^{l_1 l_2} T_{a_2}^{l_2 l_3} T_{a_3}^{l_3 l_1} T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} T_{a_3}^{i_3 j_3} = \\ &\left(\delta^{l_1 j_1} \delta^{i_1 l_2} - \frac{1}{N_C} \delta^{l_1 l_2} \delta^{i_1 j_1} \right) \left(\delta^{l_2 j_2} \delta^{i_2 l_3} - \frac{1}{N_C} \delta^{l_2 l_3} \delta^{i_2 j_2} \right) \left(\delta^{l_3 j_3} \delta^{i_3 l_1} - \frac{1}{N_C} \delta^{l_3 l_1} \delta^{i_3 j_3} \right) \end{aligned} \tag{9.4}$$

we find the decomposition

$$\mathrm{i} f_{a_1 a_2 a_3} T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} T_{a_3}^{i_3 j_3} = \delta^{i_1 j_2} \delta^{i_2 j_3} \delta^{i_3 j_1} - \delta^{i_1 j_3} \delta^{i_3 j_2} \delta^{i_2 j_1} \,. \tag{9.5}$$

Indeed,

```
symbol nc;
Dimension nc;
vector i1, i2, i3, j1, j2, j3;
index l1, l2, l3;

local [TT] =
        ( j1(l1) * i1(l2) - d_(l1,l2) * i1.j1 / nc )
      * ( j2(l2) * i2(l1) - d_(l2,l1) * i2.j2 / nc );

#procedure TTT(sign)
local [TTT`sign'] =
        ( j1(l1) * i1(l2) - d_(l1,l2) * i1.j1 / nc )
      * ( j2(l2) * i2(l3) - d_(l2,l3) * i2.j2 / nc )
      * ( j3(l3) * i3(l1) - d_(l3,l1) * i3.j3 / nc )
 `sign' ( j1(l1) * i1(l2) - d_(l1,l2) * i1.j1 / nc )
      * ( j3(l2) * i3(l3) - d_(l2,l3) * i3.j3 / nc )
      * ( j2(l3) * i2(l1) - d_(l3,l1) * i2.j2 / nc );
#endprocedure

#call TTT(-)
#call TTT(+)

bracket nc;
print;
.sort
.end
```

gives

```
[TT] =
    + nc^-1 * (  - i1.j1*i2.j2 )
    + i1.j2*i2.j1;

[TTT-] =
    + i1.j2*i2.j3*i3.j1 - i1.j3*i2.j1*i3.j2;

[TTT+] =
    + nc^-2 * (    4*i1.j1*i2.j2*i3.j3 )
    + nc^-1 * (  - 2*i1.j1*i2.j3*i3.j2
                 - 2*i1.j2*i2.j1*i3.j3
                 - 2*i1.j3*i2.j2*i3.j1 )
    + i1.j2*i2.j3*i3.j1 + i1.j3*i2.j1*i3.j2;
```

9.1 *Interface of SU3*

We're computing with a general N_C, but *epsilon* and *epsilonbar* make only sense for $N_C = 3$. Also some of the terminology alludes to $N_C = 3$: triplet, sextet, octet.
We can use all functions from *Birdtracks* that operate on *Birdtracks.t* transparently.

type t = *Birdtracks.t*

9.1.1 Constructors specific to $\mathrm{SU}(N_C)$

Fundamental representation $N = 3$

val *delta3* : *int* → *int* → *t*

"Adjoint" representation, but *without* subtracting ghosts, i. e. $N \otimes \bar{N} = 9$. Therefore, the "8" is a misnomer!

val *delta8* : *int* → *int* → *t*

The trace $\mathrm{tr}(T_a T_b)$ contains additional ghosts

val *delta8_loop* : *int* → *int* → *t*

Gauge boson in the adjoint representation $N \otimes \bar{N} - N \cdot \text{ghost}$

val *gluon* : *int* → *int* → *t*

Symmetric $N \otimes_S N = 6$ and $N \otimes_S N \otimes_S N = 10$.

val *delta6* : *int* → *int* → *t*
val *delta10* : *int* → *int* → *t*

val *t* : *int* → *int* → *int* → *t*
val *f* : *int* → *int* → *int* → *t*
val *d* : *int* → *int* → *int* → *t*

val *epsilon* : *int list* → *t*
val *epsilon_bar* : *int list* → *t*

val *t8* : *int* → *int* → *int* → *t*
val *t6* : *int* → *int* → *int* → *t*
val *t10* : *int* → *int* → *int* → *t*

val *k6* : *int* → *int* → *int* → *t*
val *k6bar* : *int* → *int* → *int* → *t*

val *delta_of_tableau* : *int Young.tableau* → *int* → *int* → *t*
val *t_of_tableau* : *int Young.tableau* → *int* → *int* → *int* → *t*

The Unit tests are in fact the largest part of this module.

module *Test* : sig val *suite* : *OUnit.test* val *suite_long* : *OUnit.test* end

9.2 Implementation of SU3

9.2.1 Import Functions from Birdtracks

```
module A = Arrow
open Arrow.Infix
module L = Algebra.Laurent

type t = Birdtracks.t
open Birdtracks
open Birdtracks.Infix
```

9.2.2 Constructors specific to SU(N_C)

Fundamental and Adjoint Representation

```
let delta3 i j =
  [ Arrows { coeff = L.int 1; arrows = j ==> i } ]

let delta8 a b =
  [ Arrows { coeff = L.int 1; arrows = a <=> b } ]
```

If the δ_{ab} originates from a $\mathrm{tr}(T_aT_b)$, like an effective $gg \to H$ coupling, it makes a difference in the color flow basis and we must write the full expression (6.2) from [17] including the ghosts instead. Note that the sign for the terms with one ghost has not been spelled out in that reference.

```
let delta8_loop a b =
  [ Arrows { coeff = L.int 1; arrows = a <=> b };
    Arrows { coeff = L.int (-1); arrows = [a => a; ?? b] };
    Arrows { coeff = L.int (-1); arrows = [?? a; b => b] };
    Arrows { coeff = L.nc 1; arrows = [?? a; ?? b] } ]
```

The following can be used for computing polarization sums (eventually, this could make the *Flow* module redundant). Note that we have $-N_C$ instead of $-1/N_C$ in the ghost contribution here, because *add_arrow_to_arrows_list′* from the module *Birdtracks* (cf. page 89) will produce a factor of $-1/N_C$ when contracting each one of the two ghost indices. Indeed, with this definition we can maintain all projection properties

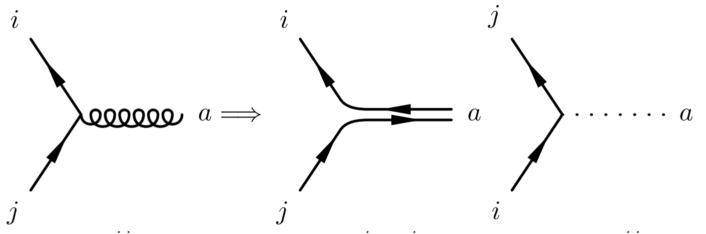
- *gluon* 1 (−3) ∗∗∗ *gluon* (−3) 2 = *gluon* 1 2,
- *delta8* 1 (−3) ∗∗∗ *delta8* (−3) 2 = *delta8* 1 2,
- *ghost* 1 (−3) ∗∗∗ *ghost* (−3) 2 = *ghost* 1 2

and most importantly

- *t* (−1) 1 2 ∗∗∗ *gluon* (−1) (−2) ∗∗∗ *t* (−2) 3 4 = *t* (−1) 1 2 ∗∗∗ *t* (−1) 3 4.

```
let ghost a b =
  [ Arrows { coeff = L.nc (-1); arrows = [?? a; ?? b] } ]

let gluon a b =
  delta8 a b @ ghost a b
```

Note that the arrow is directed from the second to the first index, opposite to our color flow paper [17]. Fortunately, this is just a matter of conventions.

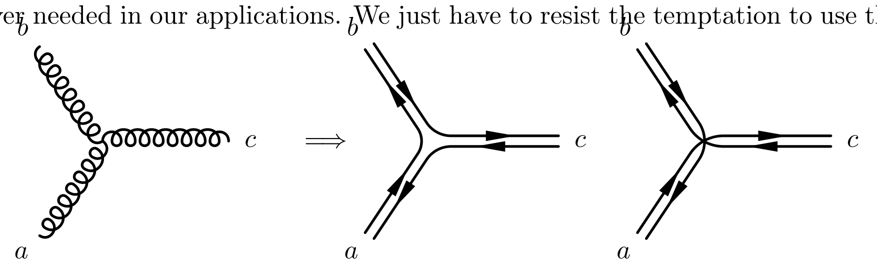


(9.6a)

$$T_a^{ij} \Longrightarrow \delta^{ia}\delta^{aj} \qquad -\,\delta^{ij} \tag{9.6b}$$

```
let t a i j =
  [ Arrows { coeff = L.int 1; arrows = [j => a; a => i] };
    Arrows { coeff = L.int (−1); arrows = [j => i; ?? a] } ]
```

Note that while we expect $\mathrm{tr}(T_a) = T_a^{ii} = 0$, the evaluation of the expression $t\ 1\ (-1)\ (-1)$ will stop at [-1 => 1; 1 => -1] — [-1 => -1; ?? 1] , because the summation index appears in a single term. However, a naive further evaluation would get stuck at [1 => 1] — nc $***$ [?? 1] . Fortunately, traces of single generators are never needed in our applications. We just have to resist the temptation to use them in unit tests.



(9.7)

```
let f a b c =
  [ Arrows { coeff = L.imag ( 1); arrows = A.cycle [a; b; c] };
    Arrows { coeff = L.imag (−1); arrows = A.cycle [a; c; b] } ]
```

The generator in the adjoint representation $T_a^{bc} = -\mathrm{i}f_{abc}$:

```
let t8 a b c =
  minus *** imag *** f a b c
```

This d_{abc} is now compatible with (6.11) in our color flow paper [17]. The signs had been wrong in earlier versions of the code to match the missing sign in the ghost contribution to the generator T_a^{ij} above.

```
let d a b c =
  [ Arrows { coeff = L.int 1; arrows = A.cycle [a; b; c] };
    Arrows { coeff = L.int 1; arrows = A.cycle [a; c; b] };
    Arrows { coeff = L.int (−2); arrows = (a <=> b) @ [?? c] };
    Arrows { coeff = L.int (−2); arrows = (b <=> c) @ [?? a] };
    Arrows { coeff = L.int (−2); arrows = (c <=> a) @ [?? b] };
    Arrows { coeff = L.int 2; arrows = [a => a; ?? b; ?? c] };
    Arrows { coeff = L.int 2; arrows = [?? a; b => b; ?? c] };
    Arrows { coeff = L.int 2; arrows = [?? a; ?? b; c => c] };
    Arrows { coeff = L.nc (−2); arrows = [?? a; ?? b; ?? c] } ]
```

Decomposed Tensor Product Representations

```
let pass_through m n incoming outgoing =
  List.rev_map2 (fun i o → (m, i) >=>> (n, o)) incoming outgoing

let delta_of_permutations n permutations k l =
  let incoming = ThoList.range 0 (pred n)
  and normalization = List.length permutations in
  List.rev_map
    (fun (eps, outgoing) →
       Arrows { coeff = L.fraction (eps × normalization);
                arrows = pass_through l k incoming outgoing } )
    permutations

let totally_symmetric n =
  List.map
    (fun p → (1, p))
    (Combinatorics.permute (ThoList.range 0 (pred n)))

let totally_antisymmetric n =
  (Combinatorics.permute_signed (ThoList.range 0 (pred n)))

let delta_S n k l =
  delta_of_permutations n (totally_symmetric n) k l

let delta_A n k l =
  delta_of_permutations n (totally_antisymmetric n) k l
```

```
let delta6 = delta_S 2
let delta10 = delta_S 3
let delta15 = delta_S 4
```

```
let delta3bar = delta_A 2
```

Mixed symmetries, as in section 9.4 of the birdtracks book.

```
module IM = Partial.Make (struct type t = int let compare = compare end)
module P = Permutation.Default
```

Map the elements of *original* to *permuted* in *all*, with *all* a list of n integers from 0 to $n-1$ in order, and use the resulting list to define a permutation. E. g. *permute_partial* [1; 3] [3; 1] [0; 1; 2; 3; 4] will define a permutation that transposes the second and fourth element in a 5 element list.

```
let permute_partial original permuted all =
  P.of_list (List.map (IM.auto (IM.of_lists original permuted)) all)
```

```
let apply1 (sign, indices) (eps, p) =
  (eps × sign, P.list p indices)
```

```
let apply signed_permutations signed_indices =
  List.rev_map (apply1 signed_indices) signed_permutations
```

```
let apply_list signed_permutations signed_indices =
  ThoList.flatmap (apply signed_permutations) signed_indices
```

```
let symmetrizer_of_permutations n original signed_permutations =
  let incoming = ThoList.range 0 (pred n) in
  List.rev_map
    (fun (eps, permuted) →
      (eps, permute_partial original permuted incoming))
    signed_permutations
```

```
let symmetrizer n indices =
  symmetrizer_of_permutations
    n indices
    (List.rev_map (fun p → (1, p)) (Combinatorics.permute indices))
```

```
let anti_symmetrizer n indices =
  symmetrizer_of_permutations
    n indices
    (Combinatorics.permute_signed indices)
```

```
let symmetrize n elements indices =
  apply_list (symmetrizer n elements) indices
```

```
let anti_symmetrize n elements indices =
  apply_list (anti_symmetrizer n elements) indices
```

```
let id n =
  [(1, ThoList.range 0 (pred n))]
```

We can avoid the recursion here, if we use *Combinatorics.permute_tensor_signed* in *symmetrizer* above.

```
let rec apply_tableau f n tableau indices =
  match tableau with
  | [] | [_] :: _ → indices
  | cells :: rest →
      apply_tableau f n rest (f n cells indices)
```

Here we should at a sanity test for *tableau*: all integers should be consecutive starting from 0 with no duplicates. In additions the rows must not grow in length.

```
let young_tableau_valid_omega y =
      Young.standard_tableau ~offset : 0 y
```

```
let delta_of_tableau tableau i j =
  if young_tableau_valid_omega tableau then
    let n = Young.num_cells_tableau tableau
    and num, den = Young.normalization (Young.diagram_of_tableau tableau)
    and rows = tableau
    and cols = Young.conjugate_tableau tableau in
    let permutations =
      apply_tableau symmetrize n rows (apply_tableau anti_symmetrize n cols (id n)) in
    int num *** fraction den *** delta_of_permutations n permutations i j
  else
    let s = Young.tableau_to_string string_of_int tableau in
    invalid_arg ("SU3.delta_of_tableau:␣" ^ s ^ "␣is␣not␣standard!")
```

```
let incomplete tensor =
  failwith ("SU3:␣" ^ tensor ^ "␣not␣supported␣yet!")
```

```
let experimental tensor =
  Printf.eprintf "SU3:␣%s␣support␣still␣experimental␣and␣untested!\n" tensor
```

```
let distinct integers =
  let rec distinct' seen = function
    | [] → true
    | i :: rest →
       if Sets.Int.mem i seen then
         false
       else
         distinct' (Sets.Int.add i seen) rest in
  distinct' Sets.Int.empty integers
```

All lines start here: they point towards the vertex.

```
let epsilon tips =
  if distinct tips then
    [ Epsilons ({ coeff = L.int 1; arrows = [] }, NEList.singleton (A.epsilon tips)) ]
  else
    []
```

All lines end here: they point away from the vertex.

```
let epsilon_bar tails =
  if distinct tails then
    [ Epsilon_Bars ({ coeff = L.int 1; arrows = [] }, NEList.singleton (A.epsilon_bar tails)) ]
  else
    []
```

In order to get the correct N_C dependence of quadratic Casimir operators, the arrows in the vertex must have the same permutation symmetry as the propagator. This is demonstrated by the unit tests involving Casimir operators on page 113 below. These tests also provide a check of our normalization.

The implementation takes a propagator and uses *Arrow.tee* to replace one arrow by the pair of arrows corresponding to the insertion of a gluon. This is repeated for each arrow. The normalization remains unchanged from the propagator. A minus sign is added for antiparallel arrows, since the conjugate representation is $-T_a^*$.

To this, we add the diagrams with a gluon connected to one arrow. Since these are identical, only one diagram multiplied by the difference of the number of parallel and antiparallel arrows is added.

```
let insert_gluon a k l term =
  let rec insert_gluon' acc left = function
    | [] → acc
    | arrow :: right →
       insert_gluon'
         (Arrows { coeff = Algebra.Laurent.mul (L.int (A.dir k l arrow)) term.coeff;
                   arrows = List.rev_append left ((A.tee a arrow) @ right) } :: acc)
         (arrow :: left)
         right in
  insert_gluon' [] [] term.arrows
```

```
let t_of_delta delta a k l =
  match delta k l with
  | [] → []
  | Arrows { arrows = arrows } :: _ as delta_kl →
      let n =
         List.fold_left
            (fun acc arrow → acc + A.dir k l arrow)
            0 arrows in
      let ghosts =
         List.rev_map
            (fun term →
               match term with
               | Arrows aterm →
                   Arrows { coeff = Algebra.Laurent.mul (L.int (−n)) aterm.coeff;
                            arrows = ?? a :: aterm.arrows }
               | Epsilons _ → failwith "t_of_delta:␣unexpected␣epsilon"
               | Epsilon_Bars _ → failwith "t_of_delta:␣unexpected␣epsilon_bar")
            delta_kl in
      List.fold_left
         (fun acc →
            function
            | Arrows aterm → insert_gluon a k l aterm @ acc
            | Epsilons _ → failwith "t_of_delta:␣unexpected␣epsilon"
            | Epsilon_Bars _ → failwith "t_of_delta:␣unexpected␣epsilon_bar")
         ghosts delta_kl
  | Epsilons _ :: _ → failwith "t_of_delta:␣unexpected␣epsilon"
  | Epsilon_Bars _ :: _ → failwith "t_of_delta:␣unexpected␣epsilon_bar"
```

```
let t_of_delta delta a k l =
  canonicalize (t_of_delta delta a k l)
```

```
let t_S n a k l =
  t_of_delta (delta_S n) a k l
```

```
let t_A n a k l =
  t_of_delta (delta_A n) a k l
```

```
let t6 = t_S 2
let t10 = t_S 3
let t15 = t_S 4
let t3bar = t_A 2
```

Equivalent definition:

```
let t8′ a b c =
  t_of_delta delta8 a b c
```

```
let t_of_tableau tableau a k l =
  t_of_delta (delta_of_tableau tableau) a k l
```

Check the following for a real live UFO file!

In the UFO paper, the Clebsh-Gordan is defined as $K^{(6),ij}{}_m$. Therefore, keeping our convention for the generators $T_a^{(6),j}{}_i$, the must arrows *end* at m.

```
let k6 m i j =
  experimental "k6";
  [ Arrows { coeff = L.int 1; arrows = [i =>> (m, 0); j =>> (m, 1)] };
    Arrows { coeff = L.int 1; arrows = [i =>> (m, 1); j =>> (m, 0)] } ]
```

The arrow are reversed for $\bar{K}^{(6),m}{}_{ij}$ and *start* at m.

```
let k6bar m i j =
  experimental "k6bar";
  [ Arrows { coeff = L.int 1; arrows = [(m, 0) >=> i; (m, 1) >=> j] };
    Arrows { coeff = L.int 1; arrows = [(m, 1) >=> i; (m, 0) >=> j] } ]
```

Playing arround with an example, it appears that we need the opposite direction. Investigate!

```
let k6 m i j =
  experimental "k6";
  [ Arrows { coeff = L.int 1; arrows = [(m, 0) >=> i; (m, 1) >=> j] };
    Arrows { coeff = L.int 1; arrows = [(m, 1) >=> i; (m, 0) >=> j] } ]

let k6bar m i j =
  experimental "k6bar";
  [ Arrows { coeff = L.int 1; arrows = [i =>> (m, 0); j =>> (m, 1)] };
    Arrows { coeff = L.int 1; arrows = [i =>> (m, 1); j =>> (m, 0)] } ]
```

9.2.3 Unit Tests

```
module Test =
  struct
    open OUnit
    module L = Algebra.Laurent

    let exorcise vertex =
      List.filter
        (function
         | Arrows aterm | Epsilons (aterm, _) | Epsilon_Bars (aterm, _) →
            ¬ (List.exists A.is_ghost aterm.arrows))
        vertex

    let exorcised_equal v1 v2 =
      equal (exorcise v1) (exorcise v2)
```

Trivia

```
    let suite_sum =
      "sum" >:::

        [ "atoms" >::
            (fun () →
              equal
                (int 2 *** delta3 1 2)
                (delta3 1 2 +++ delta3 1 2)) ]

    let suite_diff =
      "diff" >:::

        [ "atoms" >::
            (fun () →
              equal
                (delta3 3 4)
                (delta3 1 2 +++ delta3 3 4 --- delta3 1 2)) ]
```

$$\prod_{k=i}^{j} f(k) \tag{9.8}$$

```
    let rec product f i j =
      if i > j then
        null
      else if i = j then
        f i
      else
        f i *** product f (succ i) j
```

In particular

$$\textit{nc\_minus\_n\_plus}\ \mathrm{n\ k} \mapsto N_C - n + k \tag{9.9}$$

and

product (*nc_minus_n_plus n*) *i j* $\mapsto$

$$\prod_{k=i}^{j}(N_C - n + k) = \frac{(N_C-n+j)!}{(N_C-n+i-1)!} = (N_C-n+j)(N_C-n+j-1)\cdots(N_C-n+i) \quad (9.10)$$

```
let nc_minus_n_plus n k =
  const (L.ints [ (1, 1); (−n + k, 0) ])

let contractions rank k =
  product (nc_minus_n_plus rank) 1 k

let suite_times =
  "times" >:::

    [ "reorder␣components␣t1*t2" >:: (* trivial $T_a^{ik}T_a^{kj} = T_a^{kj}T_a^{ik}$ *)
        (fun () →
          let t1 = t (−1) 1 (−2)
          and t2 = t (−1) (−2) 2 in
          equal (t1 *** t2) (t2 *** t1));

      "reorder␣components␣tr(t1*t2)" >:: (* trivial $T_a^{ij}T_a^{ji} = T_a^{ji}T_a^{ij}$ *)
        (fun () →
          let t1 = t 1 (−1) (−2)
          and t2 = t 2 (−2) (−1) in
          equal (t1 *** t2) (t2 *** t1));

      "reorderings" >::
        (fun () →
          let v1 = [Arrows { coeff = L.unit; arrows = [ 1 => −2; −2 => −1; −1 => 1] }]
          and v2 = [Arrows { coeff = L.unit; arrows = [−1 => 2; 2 => −2; −2 => −1] }]
          and v′ = [Arrows { coeff = L.unit; arrows = [ 1 => 1; 2 => 2] }] in
          equal v′ (v1 *** v2));

      "eps*epsbar" >::
        (fun () →
          equal
            (delta3 1 2 *** delta3 3 4 −−− delta3 1 4 *** delta3 3 2)
            (epsilon [1; 3] *** epsilon_bar [2; 4]));

      "eps*epsbar␣-" >::
        (fun () →
          equal
            (delta3 1 4 *** delta3 3 2 −−− delta3 1 2 *** delta3 3 4)
            (epsilon [1; 3] *** epsilon_bar [4; 2]));

      "eps*epsbar␣1" >::
        (fun () →
          equal (* $N_C - 3 + 1 = (N_C - 2)$, for $NC = 3$: 1 *)
            (contractions 3 1 ***
                (delta3 1 2 *** delta3 3 4 −−− delta3 1 4 *** delta3 3 2))
            (epsilon [−1; 1; 3] *** epsilon_bar [−1; 2; 4]));

      "eps*epsbar␣cyclic␣1" >::
        (fun () →
          equal (* $N_C - 3 + 1 = (N_C - 2)$, for $NC = 3$: 1 *)
            (contractions 3 1 ***
                (delta3 1 2 *** delta3 3 4 −−− delta3 1 4 *** delta3 3 2))
            (epsilon [3; −1; 1] *** epsilon_bar [−1; 2; 4]));

      "eps*epsbar␣cyclic␣2" >::
        (fun () →
          equal (* $N_C - 3 + 1 = (N_C - 2)$, for $NC = 3$: 1 *)
            (contractions 3 1 ***
                (delta3 1 2 *** delta3 3 4 −−− delta3 1 4 *** delta3 3 2))
            (epsilon [−1; 1; 3] *** epsilon_bar [4; −1; 2]));
      "eps*epsbar␣2" >::
        (fun () →
          equal (* (N_C − 3 + 2)(N_C − 3 + 1) = (N_C − 1)(N_C − 2), for NC = 3: 2 *)
            (contractions 3 2 *** delta3 1 2)
            (epsilon [−1; −2; 1] *** epsilon_bar [−1; −2; 2]));

      "eps*epsbar␣3" >::
        (fun () →
          equal (* (N_C − 3 + 3)(N_C − 3 + 2)(N_C − 3 + 1) = N_C(N_C − 1)(N_C − 2), for NC = 3: 3! *)
            (contractions 3 3)
            (epsilon [−1; −2; −3] *** epsilon_bar [−1; −2; −3]));

      "eps*epsbar␣big" >::
        (fun () →
          equal (* (N_C − 5 + 3)(N_C − 5 + 2)(N_C − 5 + 1) = (N_C − 2)(N_C − 3)(N_C − 4), for NC = 5: 3! *)
            (contractions 5 3 ***
                (epsilon [4; 5] *** epsilon_bar [6; 7]))
            (epsilon [−1; −2; −3; 4; 5] *** epsilon_bar [−1; −2; −3; 6; 7]));

      "eps*epsbar␣big␣-" >::
        (fun () →
          equal (* (N_C − 5 + 3)(N_C − 5 + 2)(N_C − 5 + 1) = (N_C − 2)(N_C − 3)(N_C − 4), for NC = 5: 3! *)
            (contractions 5 3 ***
                (epsilon [5; 4] *** epsilon_bar [6; 7]))
            (epsilon [−1; 4; −3; −2; 5] *** epsilon_bar [−1; −2; −3; 6; 7])) ]
```

Propagators

Verify the normalization of the propagators by making sure that $D^{ij}D^{jk} = D^{ik}$

```
  let projection_id rep_d =
    equal (rep_d 1 2) (rep_d 1 (−1) *** rep_d (−1) 2)

  let orthogonality d d′ =
    assert_zero_vertex (d 1 (−1) *** d′ (−1) 2)
```

Pass every arrow straight through, without (anti-)symmetrization.

```
  let delta_unsymmetrized n k l =
    delta_of_permutations n [(1, ThoList.range 0 (pred n))] k l

  let completeness n tableaux =
    equal
      (delta_unsymmetrized n 1 2)
      (sum (List.map (fun t → delta_of_tableau t 1 2) tableaux))
```

The following names are of historical origin. From the time, when we didn't have full support for Young tableaux and implemented figure 9.1 from the birdtrack book.

$$\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & \multicolumn{1}{c}{} \\ \cline{1-1} \end{array} \tag{9.11}$$

```
  let delta_SAS i j =
    delta_of_tableau [[0; 1]; [2]] i j
```

$$\begin{array}{|c|c|}\hline 0 & 2 \\ \hline 1 & \multicolumn{1}{c}{} \\ \cline{1-1} \end{array} \tag{9.12}$$

```
  let delta_ASA i j =
    delta_of_tableau [[0; 2]; [1]] i j

  let suite_propagators =
    "propagators" >:::
      [ "D*D=D" >:: (fun () → projection_id delta3);
        "D8*D8=D8" >:: (fun () → projection_id delta8);
        "G*G=G" >:: (fun () → projection_id gluon);
        "D6*D6=D6" >:: (fun () → projection_id delta6);
        "D10*D10=D10" >:: (fun () → projection_id delta10);
```

```
"D15*D15=D15" >:: (fun () → projection_id delta15);
"D3bar*D3bar=D3bar" >:: (fun () → projection_id delta3bar);
"D6*D3bar=0" >:: (fun () → orthogonality delta6 delta3bar);
"D_A3*D_A3=D_A3" >:: (fun () → projection_id (delta_A 3));
"D10*D_A3=0" >:: (fun () → orthogonality delta10 (delta_A 3));
"D_SAS*D_SAS=D_SAS" >:: (fun () → projection_id delta_SAS);
"D_ASA*D_ASA=D_ASA" >:: (fun () → projection_id delta_ASA);
"D_SAS*D_S3=0" >:: (fun () → orthogonality delta_SAS (delta_S 3));
"D_SAS*D_A3=0" >:: (fun () → orthogonality delta_SAS (delta_A 3));
"D_SAS*D_ASA=0" >:: (fun () → orthogonality delta_SAS delta_ASA);
"D_ASA*D_SAS=0" >:: (fun () → orthogonality delta_ASA delta_SAS);
"D_ASA*D_S3=0" >:: (fun () → orthogonality delta_ASA (delta_S 3));
"D_ASA*D_A3=0" >:: (fun () → orthogonality delta_ASA (delta_A 3));
"DU*DU=DU" >:: (fun () → projection_id (delta_unsymmetrized 3));

"S3=[0123]" >::
  (fun () →
    equal (delta_S 4 1 2) (delta_of_tableau [[0; 1; 2; 3]] 1 2));

"A3=[0,1,2,3]" >::
  (fun () →
    equal (delta_A 4 1 2) (delta_of_tableau [[0]; [1]; [2]; [3]] 1 2));

"[0123]*[012,3]=0" >::
  (fun () →
    orthogonality
      (delta_of_tableau [[0; 1; 2; 3]])
      (delta_of_tableau [[0; 1; 2]; [3]]));

"[0123]*[01,23]=0" >::
  (fun () →
    orthogonality
      (delta_of_tableau [[0; 1; 2; 3]])
      (delta_of_tableau [[0; 1]; [2; 3]]));

"[012,3]*[012,3]=[012,3]" >::
  (fun () → projection_id (delta_of_tableau [[0; 1; 2]; [3]]));
```

$$\begin{array}{|c|c|}\hline 0&1\\\hline\end{array}+\begin{array}{|c|}\hline 0\\\hline 1\\\hline\end{array}\tag{9.13}$$

```
"completeness␣2" >:: (fun () → completeness 2 [ [[0; 1]]; [[0]; [1]] ]) ;

"completeness␣2'" >::
  (fun () →
    equal
      (delta_unsymmetrized 2 1 2)
      (delta_S 2 1 2 +++ delta_A 2 1 2));
```

The normalization factors are written for illustration. They are added by *delta_of_tableau* automatically.

$$\begin{array}{|c|c|c|}\hline 0&1&2\\\hline\end{array}+\frac{4}{3}\cdot\begin{array}{|c|c|}\hline 0&1\\\hline \multicolumn{1}{|c|}{2}&\multicolumn{1}{c}{}\\\cline{1-1}\end{array}+\frac{4}{3}\cdot\begin{array}{|c|c|}\hline 0&2\\\hline \multicolumn{1}{|c|}{1}&\multicolumn{1}{c}{}\\\cline{1-1}\end{array}+\begin{array}{|c|}\hline 0\\\hline 1\\\hline 2\\\hline\end{array}\tag{9.14}$$

```
"completeness␣3" >::
  (fun () → completeness 3 [ [[0; 1; 2]]; [[0; 1]; [2]]; [[0; 2]; [1]]; [[0]; [1]; [2]] ]);

"completeness␣3'" >::
  (fun () →
    equal
      (delta_unsymmetrized 3 1 2)
      (delta_S 3 1 2 +++ delta_SAS 1 2 +++ delta_ASA 1 2 +++ delta_A 3 1 2));
```

$$\begin{array}{|c|c|c|c|}\hline 0&1&2&3\\\hline\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|c|}\hline 0&1&2\\\hline \multicolumn{1}{|c|}{3}&\multicolumn{2}{c}{}\\\cline{1-1}\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|c|}\hline 0&1&3\\\hline \multicolumn{1}{|c|}{2}&\multicolumn{2}{c}{}\\\cline{1-1}\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|c|}\hline 0&2&3\\\hline \multicolumn{1}{|c|}{1}&\multicolumn{2}{c}{}\\\cline{1-1}\end{array}+\frac{4}{3}\cdot\begin{array}{|c|c|}\hline 0&1\\\hline 2&3\\\hline\end{array}+\frac{4}{3}\cdot\begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|}\hline 0&1\\\hline \multicolumn{1}{|c|}{2}&\multicolumn{1}{c}{}\\\cline{1-1} \multicolumn{1}{|c|}{3}&\multicolumn{1}{c}{}\\\cline{1-1}\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|}\hline 0&2\\\hline \multicolumn{1}{|c|}{1}&\multicolumn{1}{c}{}\\\cline{1-1} \multicolumn{1}{|c|}{3}&\multicolumn{1}{c}{}\\\cline{1-1}\end{array}+\frac{3}{2}\cdot\begin{array}{|c|c|}\hline 0&3\\\hline \multicolumn{1}{|c|}{1}&\multicolumn{1}{c}{}\\\cline{1-1} \multicolumn{1}{|c|}{2}&\multicolumn{1}{c}{}\\\cline{1-1}\end{array}+\begin{array}{|c|}\hline 0\\\hline 1\\\hline 2\\\hline 3\\\hline\end{array}\tag{9.15}$$

```
      "completeness␣4" >::
        (fun () →
          completeness 4
            [ [[0; 1; 2; 3]];
              [[0; 1; 2]; [3]]; [[0; 1; 3]; [2]]; [[0; 2; 3]; [1]];
              [[0; 1]; [2; 3]]; [[0; 2]; [1; 3]];
              [[0; 1]; [2]; [3]]; [[0; 2]; [1]; [3]]; [[0; 3]; [1]; [2]];
              [[0]; [1]; [2]; [3]] ]) ]
```

Normalization

```
  let suite_normalization =
    "normalization" >:::

    [ "tr(t*t)" >:: (* tr(T_a T_b) = δ_ab + ghosts *)
        (fun () →
          equal
            (delta8_loop 1 2)
            (t 1 (−1) (−2) *** t 2 (−2) (−1)));

      "tr(t*t)␣sans␣ghosts" >:: (* tr(T_a T_b) = δ_ab *)
        (fun () →
          exorcised_equal
            (delta8 1 2)
            (t 1 (−1) (−2) *** t 2 (−2) (−1)));
```

The additional ghostly terms were unexpected, but arises like (6.2) in our color flow paper [17].

```
      "t*t*t" >:: (* T_a T_b T_a = −T_b/N_C + ... *)
        (fun () →
          equal
            (minus *** over_nc *** t 1 2 3
             +++ [Arrows { coeff = L.int 1; arrows = [1 => 1; 3 => 2] };
                  Arrows { coeff = L.nc (−1); arrows = [3 => 2; ?? 1] }])
            (t (−1) 2 (−2) *** t 1 (−2) (−3) *** t (−1) (−3) 3));
```

As expected, these ghostly terms cancel in the summed squares

$$\mathrm{tr}(T_aT_bT_aT_cT_bT_c) = \mathrm{tr}(T_bT_b)/N_C^2 = \delta_{bb}/N_C^2 = (N_C^2-1)/N_C^2 = 1 - 1/N_C^2 \tag{9.16}$$

```
      "sum((t*t*t)^2)" >::
        (fun () →
          equal
            (ints [(1, 0); (−1, −2)])
            (t (−1) (−11) (−12) *** t (−2) (−12) (−13) *** t (−1) (−13) (−14)
             *** t (−3) (−14) (−15) *** t (−2) (−15) (−16) *** t (−3) (−16) (−11)));

      "d*d" >::
        (fun () →
          exorcised_equal
            [ Arrows { coeff = L.ints [(2, 1); (−8, −1)]; arrows = 1 <=> 2 };
              Arrows { coeff = L.ints [(2, 0); ( 4, −2)]; arrows = [1 => 1; 2 => 2] }]
            (d 1 (−1) (−2) *** d 2 (−2) (−1))) ]
```

As proposed in our color flow paper [17], we can get the correct (anti-)symmetrized generators by sandwiching the following unsymmetrized generators between the corresponding (anti-)symmetrized projectors. Therefore, the unsymmetrized generators work as long as they're used in Feynman diagrams, where they are connected by propagators that contain (anti-)symmetrized projectors. They even work in the Lie algebra relations and give the correct normalization there.

They fail however for more general color algebra expressions that can appear in UFO files. In particular, the Casimir operators come out really wrong.

```
  let t_unsymmetrized n k l =
    t_of_delta (delta_unsymmetrized n) k l
```

The following trivial vertices are *not* used anymore, since they don't get the normalization of the Ward identities right. For the quadratic casimir operators, they always produce a result proportional to $C_F = C_2(S_1)$. This can be understood because they correspond to a fundamental representation with spectators.

(Anti-)symmetrizing by sandwiching with projectors almost works, but they must be multiplied by hand by the number of arrows to get the normalization right. They're here just for documenting what doesn't work.

```
let t_trivial n a k l =
  let sterile =
    List.map (fun i → (l, i) >=>> (k, i)) (ThoList.range 1 (pred n)) in
  [ Arrows { coeff = L.int ( 1); arrows = ((l, 0) >=> a) :: (a =>> (k, 0)) :: sterile };
    Arrows { coeff = L.int (−1); arrows = (?? a) :: ((l, 0) >=>> (k, 0)) :: sterile }]

let t6_trivial = t_trivial 2
let t10_trivial = t_trivial 3
let t15_trivial = t_trivial 4

let t_SAS = t_of_delta delta_SAS
let t_ASA = t_of_delta delta_ASA

let symmetrization ?rep_ts rep_tu rep_d =
  let rep_ts =
    match rep_ts with
    | None → rep_tu
    | Some rep_t → rep_t in
  equal
    (rep_ts 1 2 3)
    (gluon 1 (−1) *** rep_d 2 (−2) *** rep_tu (−1) (−2) (−3) *** rep_d (−3) 3)

let suite_symmetrization =
  "symmetrization" >:::

    [ "t6" >:: (fun () → symmetrization t6 delta6);
      "t10" >:: (fun () → symmetrization t10 delta10);
      "t15" >:: (fun () → symmetrization t15 delta15);
      "t3bar" >:: (fun () → symmetrization t3bar delta3bar);
      "t_SAS" >:: (fun () → symmetrization t_SAS delta_SAS);
      "t_ASA" >:: (fun () → symmetrization t_ASA delta_ASA);
      "t6'" >:: (fun () → symmetrization ~rep_ts : t6 (t_unsymmetrized 2) delta6);
      "t10'" >:: (fun () → symmetrization ~rep_ts : t10 (t_unsymmetrized 3) delta10);
      "t15'" >:: (fun () → symmetrization ~rep_ts : t15 (t_unsymmetrized 4) delta15);

      "t6''" >::
        (fun () →
          equal
            (t6 1 2 3)
            (int 2 *** delta6 2 (−1) *** t6_trivial 1 (−1) (−2) *** delta6 (−2) 3));

      "t10''" >::
        (fun () →
          equal
            (t10 1 2 3)
            (int 3 *** delta10 2 (−1) *** t10_trivial 1 (−1) (−2) *** delta10 (−2) 3));

      "t15''" >::
        (fun () →
          equal
            (t15 1 2 3)
            (int 4 *** delta15 2 (−1) *** t15_trivial 1 (−1) (−2) *** delta15 (−2) 3)) ]
```

Traces

Compute (anti-)commutators of generators in the representation r, i. e. $[r(t_a)r(t_b)]_{ij} \mp [r(t_b)r(t_a)]_{ij}$, using $isum < 0$ as summation index in the matrix products.

```
let commutator rep_t i_sum a b i j =
  multiply [rep_t a i i_sum; rep_t b i_sum j]
```

```
    — multiply [rep_t b i i_sum; rep_t a i_sum j]

  let anti_commutator rep_t i_sum a b i j =
    multiply [rep_t a i i_sum; rep_t b i_sum j]
    +++ multiply [rep_t b i i_sum; rep_t a i_sum j]
```

Trace of the product of three generators in the representation r, i. e. $\mathrm{tr}_r(r(t_a)r(t_b)r(t_c))$, using $-1, -2, -3$ as summation indices in the matrix products.

```
  let trace3 rep_t a b c =
    rep_t a (−1) (−2) *** rep_t b (−2) (−3) *** rep_t c (−3) (−1)

  let loop3 a b c =
    [ Arrows { coeff = L.int 1; arrows = A.cycle (List.rev [a; b; c]) };
      Arrows { coeff = L.int (−1); arrows = (a <=> b) @ [?? c] };
      Arrows { coeff = L.int (−1); arrows = (b <=> c) @ [?? a] };
      Arrows { coeff = L.int (−1); arrows = (c <=> a) @ [?? b] };
      Arrows { coeff = L.int 1; arrows = [a => a; ?? b; ?? c] };
      Arrows { coeff = L.int 1; arrows = [?? a; b => b; ?? c] };
      Arrows { coeff = L.int 1; arrows = [?? a; ?? b; c => c] };
      Arrows { coeff = L.nc (−1); arrows = [?? a; ?? b; ?? c] } ]

  let suite_trace =
    "trace" >:::

      [ "tr(ttt)" >::
          (fun () → equal (trace3 t 1 2 3) (loop3 1 2 3));

        "tr(ttt)␣cyclic␣1" >:: (* tr(T_aT_bT_c) = tr(T_bT_cT_a) *)
          (fun () → equal (trace3 t 1 2 3) (trace3 t 2 3 1));

        "tr(ttt)␣cyclic␣2" >:: (* tr(T_aT_bT_c) = tr(T_cT_aT_b) *)
          (fun () → equal (trace3 t 1 2 3) (trace3 t 3 1 2));
```

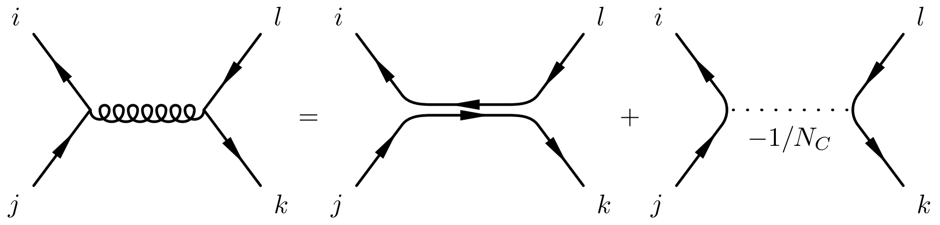
Do we expect this?

```
        "tr(tttt)" >:: (* tr(T_aT_bT_cT_d) = ... *)
          (fun () →
            exorcised_equal
              [ Arrows { coeff = L.int 1; arrows = A.cycle [4; 3; 2; 1] }]
              (t 1 (−1) (−2) *** t 2 (−2) (−3) *** t 3 (−3) (−4) *** t 4 (−4) (−1))) ]

  let suite_ghosts =
    "ghosts" >:::

      [ "H->gg" >::
          (fun () →
            equal
              (delta8_loop 1 2)
              (t 1 (−1) (−2) *** t 2 (−2) (−1)));

        "H->ggg␣f" >::
          (fun () →
            equal
              (imag *** f 1 2 3)
              (trace3 t 1 2 3 −−− trace3 t 1 3 2));

        "H->ggg␣d" >::
          (fun () →
            equal
              (d 1 2 3)
              (trace3 t 1 2 3 +++ trace3 t 1 3 2));

        "H->ggg␣f'" >::
          (fun () →
            equal
              (imag *** f 1 2 3)
```

```
                (t 1 (−3) (−2) *** commutator t (−1) 2 3 (−2) (−3)));

          "H->ggg␣d'" >::
            (fun () →
              equal
                (d 1 2 3)
                (t 1 (−3) (−2) *** anti_commutator t (−1) 2 3 (−2) (−3)));

          "H->ggg␣cyclic'" >::
            (fun () →
              let trace a b c =
                t a (−3) (−2) *** commutator t (−1) b c (−2) (−3) in
              equal (trace 1 2 3) (trace 2 3 1)) ]

    let ff a1 a2 a3 a4 =
      [ Arrows { coeff = L.int (−1); arrows = A.cycle [a1; a2; a3; a4] };
        Arrows { coeff = L.int ( 1); arrows = A.cycle [a2; a1; a3; a4] };
        Arrows { coeff = L.int ( 1); arrows = A.cycle [a1; a2; a4; a3] };
        Arrows { coeff = L.int (−1); arrows = A.cycle [a2; a1; a4; a3] } ]

    let tf j i a b =
      [ Arrows { coeff = L.imag ( 1); arrows = A.chain [i; a; b; j] };
        Arrows { coeff = L.imag (−1); arrows = A.chain [i; b; a; j] } ]

    let suite_ff =
      "f*f" >:::
        [ "1" >:: (fun () → equal (ff 1 2 3 4) (f (−1) 1 2 *** f (−1) 3 4));
          "2" >:: (fun () → equal (ff 1 2 3 4) (f (−1) 1 2 *** f 3 4 (−1)));
          "3" >:: (fun () → equal (ff 1 2 3 4) (f (−1) 1 2 *** f 4 (−1) 3)) ]

    let suite_tf =
      "t*f" >:::
        [ "1" >:: (fun () → equal (tf 1 2 3 4) (t (−1) 1 2 *** f (−1) 3 4)) ]
```

Completeness Relation

Check the completeness relation corresponding to $q\bar{q}$-scattering:



(9.17)

$T_a^{ij} T_a^{kl}$

```
    let tt i j k l =
      t (−1) i j *** t (−1) k l
```

$\delta^{il}\delta^{kj} - \delta^{ij}\delta^{kl}/N_C$

```
    let tt_expected i j k l =
      [ Arrows { coeff = L.int 1; arrows = [l => i; j => k] };
        Arrows { coeff = L.over_nc (−1); arrows = [j => i; l => k] }]

    let suite_tt =
      "t*t" >:::
        [ "1" >:: (* T_a^{ij} T_a^{kl} = δ^{il} δ^{kj} − δ^{ij} δ^{kl}/N_C *)
            (fun () → equal (tt_expected 1 2 3 4) (tt 1 2 3 4)) ]
```

Lie Algebra

Check the commutation relations $[T_a, T_b] = \mathrm{i} f_{abc} T_c$ in various representations.

```
    let lie_algebra_id rep_t =
    let lhs = imag *** f 1 2 (−1) *** t (−1) 3 4
    and rhs = commutator t (−1) 1 2 3 4 in
    equal lhs rhs
```

Check the normalization of the structure consistants $\mathcal{N} f_{abc} = -\mathrm{i}\,\mathrm{tr}(T_a[T_b, T_c])$

```
  let f_of_rep_id norm rep_t =
    let lhs = norm *** f 1 2 3
    and rhs = f_of_rep rep_t 1 2 3 in
    equal lhs rhs
```

Are the normalization factors for the traces of the higher dimensional representations correct?

The traces don't work for the symmetrized generators that we need elsewhere!

```
  let suite_lie =
    "Lie␣algebra␣relations" >:::
      [ "[t,t]=ift" >:: (fun () → lie_algebra_id t);
        "[t8,t8]=ift8" >:: (fun () → lie_algebra_id t8);
        "[t6,t6]=ift6" >:: (fun () → lie_algebra_id t6);
        "[t10,t10]=ift10" >:: (fun () → lie_algebra_id t10);
        "[t15,t15]=ift15" >:: (fun () → lie_algebra_id t15);
        "[t3bar,t3bar]=ift3bar" >:: (fun () → lie_algebra_id t3bar);
        "[tSAS,tSAS]=iftSAS" >:: (fun () → lie_algebra_id t_SAS);
        "[tASA,tASA]=iftASA" >:: (fun () → lie_algebra_id t_ASA);
        "[t6,t6]=ift6'" >:: (fun () → lie_algebra_id (t_unsymmetrized 2));
        "[t10,t10]=ift10'" >:: (fun () → lie_algebra_id (t_unsymmetrized 3));
        "[t15,t15]=ift15'" >:: (fun () → lie_algebra_id (t_unsymmetrized 4));
        "[t6,t6]=ift6''" >:: (fun () → lie_algebra_id t6_trivial);
        "[t10,t10]=ift10''" >:: (fun () → lie_algebra_id t10_trivial);
        "[t15,t15]=ift15''" >:: (fun () → lie_algebra_id t15_trivial);
        "if␣=␣tr(t[t,t])" >:: (fun () → f_of_rep_id one t);
        "2n*if␣=␣tr(t8[t8,t8])" >:: (fun () → f_of_rep_id (two *** nc) t8);
        "n*if␣=␣tr(t6[t6,t6])" >:: (fun () → f_of_rep_id nc t6_trivial);
        "n^2*if␣=␣tr(t10[t10,t10])" >:: (fun () → f_of_rep_id (nc *** nc) t10_trivial);
        "n^3*if␣=␣tr(t15[t15,t15])" >:: (fun () → f_of_rep_id (nc *** nc *** nc) t15_trivial) ]
```

Ward Identities

Testing the color part of basic Ward identities is essentially the same as testing the Lie algebra equations above, but with generators sandwiched between propagators, as in Feynman diagrams, where the relative signs come from the kinematic part of the diagrams after applying the equations of motion..
First the diagram with the three gluon vertex $\mathrm{i} f_{abc} D^{\text{gluon}}_{cd} D^{ik} T^{kl}_d D^{lj}$

```
  let ward_ft rep_t rep_d a b i j =
    imag *** f a b (−11) *** gluon (−11) (−12)
    *** rep_d i (−1) *** rep_t (−12) (−1) (−2) *** rep_d (−2) j
```

then one diagram with two gauge couplings $D^{ik} T^{kl}_c D^{lm} T^{mn}_c D^{nj}$

```
  let ward_tt1 rep_t rep_d a b i j =
    rep_d i (−1) *** rep_t a (−1) (−2) *** rep_d (−2) (−3)
    *** rep_t b (−3) (−4) *** rep_d (−4) j
```

finally the difference of exchanged orders: $D^{ik} T^{kl}_a D^{lm} T^{mn}_b D^{nj} - D^{ik} T^{kl}_b D^{lm} T^{mn}_a D^{nj}$

```
  let ward_tt rep_t rep_d a b i j =
    ward_tt1 rep_t rep_d a b i j −−− ward_tt1 rep_t rep_d b a i j
```

The optional ~*fudge* factor was used for debugging normalizations.

```
  let ward_id ?(fudge = one) rep_t rep_d =
    let lhs = ward_ft rep_t rep_d 1 2 3 4
    and rhs = ward_tt rep_t rep_d 1 2 3 4 in
    equal lhs (fudge *** rhs)

let suite_ward =
  "Ward␣identities" >:::
    [ "fund." >:: (fun () → ward_id t delta3);
      "adj." >:: (fun () → ward_id t8 delta8);
      "S2" >:: (fun () → ward_id t6 delta6);
      "S3" >:: (fun () → ward_id t10 delta10);
      "A2" >:: (fun () → ward_id t3bar delta3bar);
      "A3" >:: (fun () → ward_id (t_A 3) (delta_A 3));
      "SAS" >:: (fun () → ward_id t_SAS delta_SAS);
      "ASA" >:: (fun () → ward_id t_ASA delta_ASA);
      "S2'" >:: (fun () → ward_id ~fudge : two t6_trivial delta6);
      "S3'" >:: (fun () → ward_id ~fudge : (int 3) t10_trivial delta10) ]

let suite_ward_long =
  "Ward␣identities" >:::
    [ "S4" >:: (fun () → ward_id t15 delta15);
      "S4'" >:: (fun () → ward_id ~fudge : (int 4) t15_trivial delta15) ]
```

Jacobi Identities

$T_aT_bT_c$

```
let prod3 rep_t a b c i j =
  rep_t a i (−1) *** rep_t b (−1) (−2) *** rep_t c (−2) j
```

$[T_a, [T_b, T_c]]$

```
let jacobi1 rep_t a b c i j =
  (prod3 rep_t a b c i j --- prod3 rep_t a c b i j)
  -- (prod3 rep_t b c a i j --- prod3 rep_t c b a i j)
```

sum of cyclic permutations of $[T_a, [T_b, T_c]]$

```
let jacobi rep_t =
  sum [jacobi1 rep_t 1 2 3 4 5;
       jacobi1 rep_t 2 3 1 4 5;
       jacobi1 rep_t 3 1 2 4 5]

let jacobi_id rep_t =
  assert_zero_vertex (jacobi rep_t)

let suite_jacobi =
  "Jacobi␣identities" >:::
    [ "fund." >:: (fun () → jacobi_id t);
      "adj." >:: (fun () → jacobi_id f);
      "S2" >:: (fun () → jacobi_id t6);
      "S3" >:: (fun () → jacobi_id t10);
      "A2" >:: (fun () → jacobi_id (t_A 2));
      "A3" >:: (fun () → jacobi_id (t_A 3));
      "SAS" >:: (fun () → jacobi_id t_SAS);
      "ASA" >:: (fun () → jacobi_id t_ASA);
      "S2'" >:: (fun () → jacobi_id t6_trivial);
      "S3'" >:: (fun () → jacobi_id t10_trivial) ]

let suite_jacobi_long =
  "Jacobi␣identities" >:::
    [ "S4" >:: (fun () → jacobi_id t15);
      "S4'" >:: (fun () → jacobi_id t15_trivial) ]
```

Casimir Operators

We can read of the eigenvalues of the Casimir operators for the adjoint, totally symmetric and totally antisymmetric representations of SU(N) from table II of `hep-ph/0611341`

$$C_2(\text{adj}) = 2N \tag{9.18a}$$

$$C_2(S_n) = \frac{n(N-1)(N+n)}{N} \tag{9.18b}$$

$$C_2(A_n) = \frac{n(N-n)(N+1)}{N} \tag{9.18c}$$

adjusted for our normalization. Also from `arxiv:1912.13302`

$$C_3(S_1) = (N^2-1)(N^2-4)/N^2 = \frac{N_C^4 - 5N_C^2 + 4}{N_C^2} \tag{9.19}$$

Building blocks n/N_C and $N_C + n$

```
let n_over_nc n = const (L.ints [ (n, -1) ])
let nc_plus n = const (L.ints [ (1, 1); (n,0) ])
```

$C_2(S_n) = n/N_C(N_C - 1)(N_C + n)$

```
let c2_S n = n_over_nc n *** nc_plus (-1) *** nc_plus n
```

$C_2(A_n) = n/N_C(N_C - n)(N_C + 1)$

```
let c2_A n = n_over_nc n *** nc_plus (-n) *** nc_plus 1

let casimir_tt i j = c2_S 1 *** delta3 i j
let casimir_t6t6 i j = c2_S 2 *** delta6 i j
let casimir_t10t10 i j = c2_S 3 *** delta10 i j
let casimir_t15t15 i j = c2_S 4 *** delta15 i j
let casimir_t3bart3bar i j = c2_A 2 *** delta3bar i j
let casimir_tA3tA3 i j = c2_A 3 *** delta_A 3 i j
```

$C_2(\text{adj}) = 2N_C$

```
let ca = L.ints [(2, 1)]
let casimir_ff a b =
  [ Arrows { coeff = ca; arrows = 1 <=> 2 };
    Arrows { coeff = L.int (-2); arrows = [1 => 1; 2 => 2] }]
```

$C_3(S_1) = N_C^2 - 5 + 4/N_C^2$

```
let c3f = L.ints [(1, 2); (-5, 0); (4, -2)]
let casimir_ttt i j = const c3f *** delta3 i j

let suite_casimir =
  "Casimir␣operators" >:::

    [ "t*t" >::
        (fun () →
          equal
            (casimir_tt 1 2)
            (t (-1) 1 (-2) *** t (-1) (-2) 2));

      "t*t*t" >::
        (fun () →
          equal
            (casimir_ttt 1 2)
            (d (-1) (-2) (-3) ***
               t (-1) 1 (-4) *** t (-2) (-4) (-5) *** t (-3) (-5) 2));

      "f*f" >::
        (fun () →
          equal
            (casimir_ff 1 2)
            (minus *** f (-1) 1 (-2) *** f (-1) (-2) 2));

      "t6*t6" >::
        (fun () →
          equal
            (casimir_t6t6 1 2)
            (t6 (-1) 1 (-2) *** t6 (-1) (-2) 2));
      "t3bar*t3bar" >::
        (fun () →
          equal
            (casimir_t3bart3bar 1 2)
            (t3bar (−1) 1 (−2) *** t3bar (−1) (−2) 2));

      "tA3*tA3" >::
        (fun () →
          equal
            (casimir_tA3tA3 1 2)
            (t_A 3 (−1) 1 (−2) *** t_A 3 (−1) (−2) 2));

      "t_SAS*t_SAS" >::
        (fun () →
          equal
            (const (L.ints [(3,1); (−9,−1)]) *** delta_SAS 1 2)
            (t_SAS (−1) 1 (−2) *** t_SAS (−1) (−2) 2));

      "t_ASA*t_ASA" >::
        (fun () →
          equal
            (const (L.ints [(3,1); (−9,−1)]) *** delta_ASA 1 2)
            (t_ASA (−1) 1 (−2) *** t_ASA (−1) (−2) 2));

      "t10*t10" >::
        (fun () →
          equal
            (casimir_t10t10 1 2)
            (t10 (−1) 1 (−2) *** t10 (−1) (−2) 2)) ]

let suite_casimir_long =
  "Casimir␣operators" >:::

    [ "t15*t15" >::
        (fun () →
          equal
            (casimir_t15t15 1 2)
            (t15 (−1) 1 (−2) *** t15 (−1) (−2) 2)) ]
```

Color Sums

```
let suite_colorsums =
  "(squared)␣color␣sums" >:::

    [ "gluon␣normalization" >::
        (fun () →
          equal
            (delta8 1 2)
            (delta8 1 (−1) *** gluon (−1) (−2) *** delta8 (−2) 2));

      "f*f" >::
        (fun () →
          let sum_ff =
            multiply [ f (−11) (−12) (−13);
                       f (−21) (−22) (−23);
                       gluon (−11) (−21);
                       gluon (−12) (−22);
                       gluon (−13) (−23) ]
          and expected = ints [(2, 3); (−2, 1)] in
          equal expected sum_ff);

      "d*d" >::
        (fun () →
          let sum_dd =
            multiply [ d (−11) (−12) (−13);
```

```
                        d (−21) (−22) (−23);
                        gluon (−11) (−21);
                        gluon (−12) (−22);
                        gluon (−13) (−23) ]
          and expected = ints [(2, 3); (−10, 1); (8, − 1)] in
          equal expected sum_dd);

      "f*d" >::
        (fun () →
          let sum_fd =
            multiply [ f (−11) (−12) (−13);
                        d (−21) (−22) (−23);
                        gluon (−11) (−21);
                        gluon (−12) (−22);
                        gluon (−13) (−23) ] in
          assert_zero_vertex sum_fd);

      "Hgg" >::
        (fun () →
          let sum_hgg =
            multiply [ delta8_loop (−11) (−12);
                        delta8_loop (−21) (−22);
                        gluon (−11) (−21);
                        gluon (−12) (−22) ]
          and expected = ints [(1, 2); (−1, 0)] in
          equal expected sum_hgg) ]

  let suite =
    "SU3" >:::
      [suite_sum;
       suite_diff;
       suite_times;
       suite_normalization;
       suite_symmetrization;
       suite_ghosts;
       suite_propagators;
       suite_trace;
       suite_ff;
       suite_tf;
       suite_tt;
       suite_lie;
       suite_ward;
       suite_jacobi;
       suite_casimir;
       suite_colorsums]

  let suite_long =
    "SU3␣long" >:::
      [suite_ward_long;
       suite_jacobi_long;
       suite_casimir_long]

end
```

—10— Color Propagators

10.1 Interface of Color_Propagator

Possible color flows for a single propagator, as currently supported by WHIZARD.
In a model without ϵ or $\bar\epsilon$ couplings, the color flow can be represented by arrays of identifiers (integers) of color flow lines. One array for incoming lines and another one for outgoing lines. In addition, the propagator can represent a ghost line.

If there are only fundamental, conjugate and adjoint representations with T_a and f_{abc} couplings, there will be at most of incoming and at most one outgoing line. In tensor product representations, there are more than one incoming or outgoing color flow line.

Things become more involved, when there are ϵ or $\bar\epsilon$ couplings. Fortunately, it is not possible to contract two ϵ or two $\bar\epsilon$, while pairs of ϵ and $\bar\epsilon$ can always be replaced by a sum over color flows.
For typechecking, it might be beneficial to make these abstract or private eventually.

```
type cf_in = int
type cf_out = int
```

Note that these do not need to be not mutually recursive, since ϵ can not be nested beneath ϵ (analogously for $\bar\epsilon$) and a $\bar\epsilon$ beneath a ϵ (and vice versa) can be expanded as a sum over permuted color flows.
Also note that the *lists* for *eps* and *eps_bar* have one element less than *s_eps* and *s_eps_bar*. The latter represent fully saturated ϵ and $\bar\epsilon$, while the former have one open index.

```
type eps = cf_out list
type s_eps = cf_out list
type cf_in_or_eps =
  | CF_in of cf_in
  | Epsilon of eps
```

```
type eps_bar = cf_in list
type s_eps_bar = cf_in list
type cf_out_or_eps_bar =
  | CF_out of cf_out
  | Epsilon_Bar of eps_bar
```

These types guarantee that there is never a pair of ϵ and $\bar\epsilon$ that has yet to be contracted.

```
type flow = cf_in PArray.t × cf_out PArray.t
type flow_eps = cf_in_or_eps PArray.t × cf_out PArray.t
type flow_eps_bar = cf_in PArray.t × cf_out_or_eps_bar PArray.t
```

Note that the ghosts might carry fully saturated ϵ and $\bar\epsilon$ originating from deeper in the DAG.

```
type t =
  | Flow of flow
  | Flow_with_Epsilons of flow_eps × s_eps list
  | Flow_with_Epsilon_Bars of flow_eps_bar × s_eps_bar list
  | Ghost
  | Ghost_with_Epsilons of s_eps list
  | Ghost_with_Epsilon_Bars of s_eps_bar list
```

Project onto *Flow*, if possible.

```
val normalize : t → t
```

Simple constructors.

```
val white : t
val of_lists : int list → int list → t
```

Simple predicates.

```
val is_white : t → bool
```

Reverse arrows.

```
val conjugate : t → t
```

Some ordering.

```
val compare : t → t → int
val equal : t → t → bool
```

Allowed as (a part of) an identifier in Fortran and other programming languages.

```
val to_symbol : t → string
```

Pretty printer for the toplevel.

```
val to_string : t → string
val pp : Format.formatter → t → unit
```

10.2 Implementation of *Color_Propagator*

```
type cf_in = int
type cf_out = int

type eps = cf_out list
type s_eps = cf_out list
type cf_in_or_eps =
  | CF_in of cf_in
  | Epsilon of eps

type eps_bar = cf_in list
type s_eps_bar = cf_in list
type cf_out_or_eps_bar =
  | CF_out of cf_out
  | Epsilon_Bar of eps_bar

type flow = cf_in PArray.t × cf_out PArray.t
type flow_eps = cf_in_or_eps PArray.t × cf_out PArray.t
type flow_eps_bar = cf_in PArray.t × cf_out_or_eps_bar PArray.t
type t =
  | Flow of flow
  | Flow_with_Epsilons of flow_eps × s_eps list
  | Flow_with_Epsilon_Bars of flow_eps_bar × s_eps_bar list
  | Ghost
  | Ghost_with_Epsilons of s_eps_bar list
  | Ghost_with_Epsilon_Bars of s_eps_bar list
```

For partial maps of α *Map.t*, an exception is the right choice, since we would have to use α *Map.fold* to reconstruct resulting map completele.

```
exception Fail

let to_cf_in_opt cfi =
  let project = function
    | CF_in cf → cf
    | Epsilon _ → raise Fail in
  try Some (PArray.map project cfi) with Fail → None

let to_cf_out_opt cfo =
  let project = function
    | CF_out cf → cf
    | Epsilon_Bar _ → raise Fail in
  try Some (PArray.map project cfo) with Fail → None
```

```
let normalize = function
  | (Ghost | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _ | Flow _) as flow → flow
  | Flow_with_Epsilons ((cfi, cfo), []) as flow →
      begin match to_cf_in_opt cfi with
      | None → flow
      | Some cfi → Flow (cfi, cfo)
      end
  | Flow_with_Epsilons (_, _ :: _) as flow → flow
  | Flow_with_Epsilon_Bars ((cfi, cfo), []) as flow →
      begin match to_cf_out_opt cfo with
      | None → flow
      | Some cfo → Flow (cfi, cfo)
      end
  | Flow_with_Epsilon_Bars (_, _ :: _) as flow → flow

let white = Flow (PArray.empty, PArray.empty)

let of_lists cfi cfo =
  let cfi = ThoList.mapi (fun n cf → (n, cf)) 0 cfi
  and cfo = ThoList.mapi (fun n cf → (n, cf)) 0 cfo in
  Flow (PArray.of_pairs cfi, PArray.of_pairs cfo)

let is_white = function
  | Flow (incoming, outgoing) → PArray.is_empty incoming ∧ PArray.is_empty outgoing
  | Flow_with_Epsilons (_, _) | Flow_with_Epsilon_Bars (_, _) → false
  | Ghost | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _ → false

let cfi_or_eps_to_cfo_or_eps_bar = function
  | CF_in cf → CF_out cf
  | Epsilon eps → Epsilon_Bar eps

let cfo_or_eps_bar_to_cfi_or_eps = function
  | CF_out cf → CF_in cf
  | Epsilon_Bar eps → Epsilon eps

let conjugate = function
  | Flow (cfi, cfo) → Flow (cfo, cfi)
  | Flow_with_Epsilons ((cfi, cfo), eps) →
      Flow_with_Epsilon_Bars ((cfo, PArray.map cfi_or_eps_to_cfo_or_eps_bar cfi), eps)
  | Flow_with_Epsilon_Bars ((cfi, cfo), eps) →
      Flow_with_Epsilons ((PArray.map cfo_or_eps_bar_to_cfi_or_eps cfo, cfi), eps)
  | Ghost → Ghost
  | Ghost_with_Epsilons eps → Ghost_with_Epsilon_Bars eps
  | Ghost_with_Epsilon_Bars eps → Ghost_with_Epsilons eps

let cf_in_or_eps_to_string = function
  | CF_in i → string_of_int i
  | Epsilon cfos → Printf.sprintf "E(%s)" (ThoList.to_string string_of_int cfos)

let cf_out_or_eps_bar_to_string = function
  | CF_out i → string_of_int i
  | Epsilon_Bar cfis → Printf.sprintf "B(%s)" (ThoList.to_string string_of_int cfis)

let cf_in_out_to_string cfi cfo =
  match PArray.is_empty cfi, PArray.is_empty cfo with
  | true, true → "W"
  | false, true → Printf.sprintf "I(%s)" (PArray.to_string string_of_int cfi)
  | true, false → Printf.sprintf "O(%s)" (PArray.to_string string_of_int cfo)
  | false, false →
      Printf.sprintf "IO(%s,%s)"
        (PArray.to_string string_of_int cfi)
        (PArray.to_string string_of_int cfo)

let to_string = function
  | Ghost → "G"
  | Flow (cfi, cfo) → cf_in_out_to_string cfi cfo
  | Ghost_with_Epsilons epsilons →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Ghost_with_Epsilon_Bars epsilon_bars →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Flow_with_Epsilons ((cfi, cfo), epsilons) →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Flow_with_Epsilon_Bars ((cfi, cfo), epsilon_bars) →
     failwith "Color_Propagator.to_string:␣incomplete"

let digit_option_to_symbol = function
  | None → "-"
  | Some i →
     if i < 0 then
       invalid_arg "Color_Propagator.digit_option_to_symbol:␣negative"
     else
       if i < 10 then
         string_of_int i
       else if i < 36 then
         String.make 1 (Char.chr (Char.code 'A' + i − 10))
       else
         invalid_arg "Color_Propagator.digit_option_to_symbol:␣too␣large"

let cf_in_cf_out_to_symbol cfi cfo =
  match PArray.to_option_list cfi, PArray.to_option_list cfo with
  | [], [] → "w"
  | cfi, [] → "i" ^ String.concat "" (List.map digit_option_to_symbol cfi)
  | [], cfo → "o" ^ String.concat "" (List.map digit_option_to_symbol cfo)
  | cfi, cfo →
     "i" ^ String.concat "" (List.map digit_option_to_symbol cfi) ^
       "_o" ^ String.concat "" (List.map digit_option_to_symbol cfo)

let to_symbol = function
  | Ghost → "g"
  | Flow (cfi, cfo) → cf_in_cf_out_to_symbol cfi cfo
  | Ghost_with_Epsilons epsilons →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Ghost_with_Epsilon_Bars epsilon_bars →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Flow_with_Epsilons ((cfi, cfo), epsilons) →
     failwith "Color_Propagator.to_string:␣incomplete"
  | Flow_with_Epsilon_Bars ((cfi, cfo), epsilon_bars) →
     failwith "Color_Propagator.to_string:␣incomplete"

let pp fmt p =
  Format.fprintf fmt "%s" (to_string p)

let compare_pairs compare_x compare_y (x1, y1) (x2, y2) =
  let c = compare_x x1 x2 in
  if c ≠ 0 then
    c
  else
    compare_y y1 y2

let compare_flows p1 p2 =
  compare_pairs (PArray.compare compare) (PArray.compare compare) p1 p2

let compare_eps e1 e2 =
  compare_pairs (compare_pairs (PArray.compare compare) (PArray.compare compare)) compare e1 e2

let compare p1 p2 =
  match normalize p1, normalize p2 with
  | Flow f1, Flow f2 → compare_flows f1 f2
  | Flow_with_Epsilons (f1, e1), Flow_with_Epsilons (f2, e2) → compare_eps (f1, e1) (f2, e2)
  | Flow_with_Epsilon_Bars (f1, e1), Flow_with_Epsilon_Bars (f2, e2) → compare_eps (f1, e1) (f2, e2)
  | Ghost, Ghost → 0
  | Ghost_with_Epsilons e1, Ghost_with_Epsilons e2 → compare e1 e2
  | Ghost_with_Epsilon_Bars e1, Ghost_with_Epsilon_Bars e2 → compare e1 e2

  | Flow _, (Flow_with_Epsilons _ | Flow_with_Epsilon_Bars _ | Ghost
              | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _)
  | Flow_with_Epsilons _, (Flow_with_Epsilon_Bars _ | Ghost
                            | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _)
  | Flow_with_Epsilon_Bars _ , (Ghost | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _)
  | Ghost, (Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _)
  | Ghost_with_Epsilons _, Ghost_with_Epsilon_Bars _ → −1

  | (Flow_with_Epsilons _ | Flow_with_Epsilon_Bars _ | Ghost
     | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _), Flow _
  | (Flow_with_Epsilon_Bars _ | Ghost
     | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _), Flow_with_Epsilons _
  | (Ghost | Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _), Flow_with_Epsilon_Bars _
  | (Ghost_with_Epsilons _ | Ghost_with_Epsilon_Bars _), Ghost
  | Ghost_with_Epsilon_Bars _, Ghost_with_Epsilons _ → 1

let equal p1 p2 =
  compare p1 p2 = 0
```

Since *PArray.Alist.t* has a unique physical representation, we can fall back on the polymorphic *compare* again.

```
let compare = compare
let equal = (=)
```

—11— Color Fusions

11.1 Interface of Color_Fusion

This module uses a vertex color flow of type *Birdtracks.t* (which aliased to, e. g., *SU3.t*), to fuse a list of *Color_Propagator.t*.
fuse nc vertex children use the color flows in the *vertex* to combine the color flows in the incoming *children* and return the color flows for outgoing particle together with their weights.

```
val fuse : int → Birdtracks.t → Color_Propagator.t list → (Algebra.Laurent.c × Color_Propagator.t) list
```

At the moment, *nc* is substituted for N_C. It this necessary or the desired behavior? Can we use (*Algebra.Laurent.t* × *Color_Propagator.t*) *list* as return type instead, in order to be able to write the symbolic expression to the amplitude? This would necessitate changes in many places, however.

Unit tests.

```
module Test : sig val suite : OUnit.test val suite_long : OUnit.test end
```

11.2 Implementation of Color_Fusion

Here we will use the color flow described by a *Arrow.free list* to determine the possible outgoing color flows for the incoming color flows in a fusion. This translates from vertices described by connections among integers describing factors in the tensor product to color flows with integers describing individual color flow lines. For the treatment of ϵ and $\bar\epsilon$, see the discussion on page 81.

At the moment both the factors in the tensor product and the color flow lines are *int*s. This could be made clearer by abstract types.

This still needs to be extended to ϵ and $\bar\epsilon$, i. e. *Arrow.free_eps* and *Arrow.free_eps_bar*.

```
module A = Arrow
open A.Infix
module CP = Color_Propagator
module L = Algebra.Laurent
module QC = Algebra.QC
```

Take a *Color_Propagator.t list*, ignore the uncolored (*Color_Propagator.W*) ones and construct a map into the colored ones indexed by the offset into the original list. Actually, one could use a *Color_Propagator.t option array* instead, but the elements of *α array* are updated in place, making it harder to keep track.

```
let line_map lines =
  let _, map =
    List.fold_left
      (fun (i, acc) line →
        (succ i,
         if CP.is_white line then
           acc
         else
           PArray.add i line acc))
      (1, PArray.empty)
      lines in
  map
```

clear i lines removes the *Color_Propagator.t* at position *i* from the map *lines*.

```
let clear = PArray.remove
```

Return +1 if the list *l1* is an even permutation of the list *l2*, −1 if *l1* is an odd permutation of *l2* and 0 otherwise.

```
let relative_permutation l1 l2 =
  let eps1, l1 = Combinatorics.sort_signed l1
  and eps2, l2 = Combinatorics.sort_signed l2 in
  if l1 = l2 then
    eps1 × eps2
  else
    0
```

Return the integers in the list *elements* that are not in the list *universe*.

```
let not_in elements universe =
  let universe = Sets.Int.of_list universe in
  let rec collect missing = function
    | [] → missing
    | x :: tail →
        if Sets.Int.mem x universe then
          collect missing tail
        else
          collect (x :: missing) tail in
  collect [] elements
```

open_epsilon is an $\epsilon_{ii_2\cdots i_n}$ (or $\bar\epsilon^{ii_2\cdots i_n}$) with one index i open and *epsilon_bar* a matching $\bar\epsilon^{j_1j_2\cdots j_n}$ (or $\epsilon_{j_1j_2\cdots j_n}$). Replace i by the single $j \in \{j_m\}_{m=1,\ldots,n}$ with $j \notin \{i_m\}_{m=2,\ldots,n}$ and compute

$$\epsilon_{ii_2\cdots i_n}\bar\epsilon^{j_1j_2\cdots j_n} = \delta^{j_1j_2\cdots j_n}_{ii_2\cdots i_n} = \sum_{\sigma\in S_n}(-1)^{\varepsilon(\sigma)}\delta_i^{\sigma(j_1)}\delta_{i_2}^{\sigma(j_2)}\cdots\delta_{i_n}^{\sigma(j_n)}\,. \tag{11.1}$$

Return *None* if the two index sets are not permutations of one another and *Some* (*sign*, *i*) if they are.

```
let open_contract open_epsilon epsilon_bar =
  match not_in epsilon_bar open_epsilon with
  | [] → None
  | [i] →
      let sign = relative_permutation (i :: open_epsilon) epsilon_bar in
      if sign = 0 then
        None
      else
        Some (sign, i)
  | _ → None
```

connect n (*sign*, *flow_n*, *lines*) *arrow* tries to form a new connection in the map *lines* using a single *arrow*. The outgoing line in the fusion is represented by *flow_n* and corresponds to *n* in the *arrow*.
If the arrow is a ghost and is connected to the outgoing line, just add it. If it is connected to an incoming line, remove this propagator, as it is saturated.

```
let connect_ghost_opt n g (sign, flow_n, lines) =
  let g′ = A.position_ghost g in
  if g′ = n then
    Some (sign, CP.Ghost, lines)
  else
    match PArray.get_opt g′ lines with
    | Some CP.Ghost → Some (sign, flow_n, clear g′ lines)
    | Some CP.Ghost_with_Epsilons _ →
        failwith "connect_ghost_opt:␣incomplete"
    | Some CP.Ghost_with_Epsilon_Bars _ →
        failwith "connect_ghost_opt:␣incomplete"
    | _ → None
```

Add the normalized propagator p to the map *lines* at position i, unless it contains no color flows. Remove it in this case.

```
let add_or_remove_if_white i p lines =
  let p = CP.normalize p in
  if CP.is_white p then
    PArray.remove i lines
  else
    PArray.add i p lines
```

If the arrow is a connection and is connected on one side to the outgoing line, find the matching incoming line. If it is connected to two incoming lines, merge them, which amounts to throwing them away.

Here's where the ϵ-$\bar\epsilon$ pairs will be consumed. We should move this to a preprocessing step, so that the repeated application of arrows does not have to take care of it. Or do it in a postprocessing step, which has the advantage that the contractions have been processed and a possible new ϵ or $\bar\epsilon$ is available.

Try to extract an ϵ (or $\bar\epsilon$) from the color flow given as the argument.

```
let take_epsilon cfi =
  let project_opt _ = function
    | CP.CF_in cf → Some cf
    | CP.Epsilon _ → None in
  PArray.take_one project_opt cfi

let take_epsilon_bar cfo =
  let project_opt _ = function
    | CP.CF_out cf → Some cf
    | CP.Epsilon_Bar _ → None in
  PArray.take_one project_opt cfo
```

This is a part of *connect_in_opt* below that requires recursion and therefore needs to be its own function. Keeping track of the overall *sign*, connect the incoming *CP.Flow_with_Epsilons* at index i' at position i in *lines* with the outgoing *CP.Flow_with_Epsilon_Bars* at index n'. Return the updated propagator and *lines* if the color flows match.

```
let rec connect_in_contract_epsilons_opt sign :
          int → CP.flow_eps_bar → CP.eps_bar list →
          int → CP.flow_eps → CP.eps list →
          int → CP.t PArray.t → (int × CP.t × CP.t PArray.t) option =
  fun n' (cfi_n, cfo_n as cf_n) epsilon_bars_n
      i' (cfi_i, cfo_i as cf_i) epsilons_i i lines →
  let open PArray in
  match epsilon_bars_n, epsilons_i with
  | epsilon_bar :: epsilon_bars_n, epsilon :: epsilons_i →
     let relative_sign = relative_permutation epsilon epsilon_bar in
     if relative_sign = 0 then
       None
     else
       connect_in_contract_epsilons_opt (relative_sign × sign)
         n' cf_n epsilon_bars_n i' cf_i epsilons_i i lines
  | epsilon_bar :: _, [] →
     begin match take_epsilon cfi_i with
     | Nothing cfi →
        let flow_n = CP.Flow_with_Epsilon_Bars (cf_n, epsilon_bars_n)
        and pi = CP.Flow (cfi, cfo_i) in
        Some (sign, flow_n, add_or_remove_if_white i pi lines)
     | Single (_, _, cfi_i) →
        failwith "Color_Fusion.connect_in_contract_epsilons_opt:␣incomplete"
     | Multiple (_, _, cfi_i) →
        failwith "Color_Fusion.connect_in_contract_epsilons_opt:␣incomplete"
     end
  | [], epsilon :: _ →
```

```
        begin match take_epsilon_bar cfo_n with
        | Nothing cfo →
            let flow_n = CP.Flow (cfi_n, cfo)
            and pi = CP.Flow_with_Epsilons (cf_i, epsilons_i) in
            Some (sign, flow_n, add_or_remove_if_white i pi lines)
        | Single (_, _, cfo_n) →
            failwith "Color_Fusion.connect_in_contract_epsilons_opt: incomplete"
        | Multiple (_, _, cfo_n) →
            failwith "Color_Fusion.connect_in_contract_epsilons_opt: incomplete"
        end
    | [], [] →
        begin match take_epsilon_bar cfo_n, take_epsilon cfi_i with
        | Nothing cfo, Nothing cfi →
            let flow_n = CP.Flow (cfi_n, cfo)
            and pi = CP.Flow (cfi, cfo_i) in
            Some (sign, flow_n, add_or_remove_if_white i pi lines)
        | _ →
            failwith "Color_Fusion.connect_in_contract_epsilons_opt: incomplete"
        end

let connect_in_opt n' (i, i') (sign, flow_n, lines) =
  let open PArray in
  match get_opt i lines with
  | None → None
  | Some flow_i →
      begin match flow_i with
      | CP.Ghost | CP.Ghost_with_Epsilons _ | CP.Ghost_with_Epsilon_Bars _ → None
      | CP.Flow (cfi_i, cfo_i) →
          begin match get_opt i' cfi_i with
          | None → None
          | Some cfi →
              begin match flow_n with
              | CP.Ghost → None
              | CP.Ghost_with_Epsilons _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Ghost_with_Epsilon_Bars _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Flow (cfi_n, cfo_n) →
                  let flow_n = CP.Flow (add n' cfi cfi_n, cfo_n)
                  and pi = CP.Flow (remove i' cfi_i, cfo_i) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                  let cfi = CP.CF_in cfi in
                  let flow_n = CP.Flow_with_Epsilons ((add n' cfi cfi_n, cfo_n), epsilons_n)
                  and pi = CP.Flow (remove i' cfi_i, cfo_i) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                  let flow_n = CP.Flow_with_Epsilon_Bars ((add n' cfi cfi_n, cfo_n), epsilon_bars_n)
                  and pi = CP.Flow (remove i' cfi_i, cfo_i) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              end
          end
      | CP.Flow_with_Epsilons ((cfi_i, cfo_i), epsilons_i) →
          begin match get_opt i' cfi_i with
          | None → None
          | Some cfi →
              begin match flow_n with
              | CP.Ghost → None
              | CP.Ghost_with_Epsilons _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Ghost_with_Epsilon_Bars _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Flow (cfi_n, cfo_n) →
                  let cfi_n = map (fun cf → CP.CF_in cf) cfi_n in
                  let flow_n = CP.Flow_with_Epsilons ((add n' cfi cfi_n, cfo_n), epsilons_i)
                  and pi = CP.Flow_with_Epsilons ((remove i' cfi_i, cfo_i), []) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                  let flow_n = CP.Flow_with_Epsilons ((add n' cfi cfi_n, cfo_n), epsilons_i @ epsilons_n)
                  and pi = CP.Flow_with_Epsilons ((remove i' cfi_i, cfo_i), []) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                  connect_in_contract_epsilons_opt sign
                    n' (cfi_n, cfo_n) epsilon_bars_n
                    i' (cfi_i, cfo_i) epsilons_i
                    i lines
              end
            end
      | CP.Flow_with_Epsilon_Bars ((cfi_i, cfo_i), epsilon_bars_i) →
            begin match get_opt i' cfi_i with
            | None → None
            | Some cfi →
              begin match flow_n with
              | CP.Ghost → None
              | CP.Ghost_with_Epsilons _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Ghost_with_Epsilon_Bars _ →
                  failwith "connect_in_opt: incomplete"
              | CP.Flow (cfi_n, cfo_n) →
                  let cfo_n = map (fun cf → CP.CF_out cf) cfo_n in
                  let flow_n = CP.Flow_with_Epsilon_Bars ((add n' cfi cfi_n, cfo_n), epsilon_bars_i)
                  and pi = CP.Flow_with_Epsilon_Bars ((remove i' cfi_i, cfo_i), []) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)
              | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                  let flow_n = CP.Flow_with_Epsilon_Bars ((add n' cfi cfi_n, cfo_n), epsilon_bars_i @ epsilon_bars_n)
                  and pi = CP.Flow_with_Epsilon_Bars ((remove i' cfi_i, cfo_i), []) in
                  Some (sign, flow_n, add_or_remove_if_white i pi lines)

              | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                  failwith "Color_Fusion.connect_in_opt: no epsilon contractions yet"
              end
            end
      end

let connect_out_opt n' (o, o') (sign, flow_n, lines) =
  let open PArray in
  match get_opt o lines with
  | None → None
  | Some flow →
      begin match flow with
      | CP.Ghost | CP.Ghost_with_Epsilons _ | CP.Ghost_with_Epsilon_Bars _ → None
      | CP.Flow (cfi_o, cfo_o) →
        begin match get_opt o' cfo_o with
        | None → None
        | Some cfo →
              begin match flow_n with
              | CP.Ghost → None
              | CP.Ghost_with_Epsilons _ →
                  failwith "connect_out_opt: incomplete"
              | CP.Ghost_with_Epsilon_Bars _ →
                  failwith "connect_out_opt: incomplete"
              | CP.Flow (cfi_n, cfo_n) →
                  let flow_n = CP.Flow (cfi_n, add n' cfo cfo_n)
```

```
                and po = CP.Flow (cfi_o, remove o' cfo_o) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                let flow_n = CP.Flow_with_Epsilons ((cfi_n, add n' cfo cfo_n), epsilons_n)
                and po = CP.Flow (cfi_o, remove o' cfo_o) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                let cfo = CP.CF_out cfo in
                let flow_n = CP.Flow_with_Epsilon_Bars ((cfi_n, add n' cfo cfo_n), epsilon_bars_n)
                and po = CP.Flow (cfi_o, remove o' cfo_o) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            end
        end
    | CP.Flow_with_Epsilons ((cfi_o, cfo_o), epsilons_o) →
        begin match get_opt o' cfo_o with
        | None → None
        | Some cfo →
            begin match flow_n with
            | CP.Ghost → None
            | CP.Ghost_with_Epsilons _ →
                failwith "connect_out_opt:␣incomplete"
            | CP.Ghost_with_Epsilon_Bars _ →
                failwith "connect_out_opt:␣incomplete"
            | CP.Flow (cfi_n, cfo_n) →
                let cfi_n = map (fun cf → CP.CF_in cf) cfi_n in
                let flow_n = CP.Flow_with_Epsilons ((cfi_n, add n' cfo cfo_n), epsilons_o)
                and po = CP.Flow_with_Epsilons ((cfi_o, remove o' cfo_o), []) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                let flow_n = CP.Flow_with_Epsilons ((cfi_n, add n' cfo cfo_n), epsilons_o @ epsilons_n)
                and po = CP.Flow_with_Epsilons ((cfi_o, remove o' cfo_o), []) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                failwith "Color_Fusion.connect_out_opt:␣no␣epsilon␣contractions␣yet"
            end
        end
    | CP.Flow_with_Epsilon_Bars ((cfi_o, cfo_o), epsilon_bars_o) →
        begin match get_opt o' cfo_o with
        | None → None
        | Some cfo →
            begin match flow_n with
            | CP.Ghost → None
            | CP.Ghost_with_Epsilons _ →
                failwith "connect_out_opt:␣incomplete"
            | CP.Ghost_with_Epsilon_Bars _ →
                failwith "connect_out_opt:␣incomplete"
            | CP.Flow (cfi_n, cfo_n) →
                let cfo_n = map (fun cf → CP.CF_out cf) cfo_n in
                let flow_n = CP.Flow_with_Epsilon_Bars ((cfi_n, add n' cfo cfo_n), epsilon_bars_o)
                and po = CP.Flow_with_Epsilon_Bars ((cfi_o, remove o' cfo_o), []) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)
            | CP.Flow_with_Epsilon_Bars ((cfi_n, cfo_n), epsilon_bars_n) →
                let flow_n = CP.Flow_with_Epsilon_Bars ((cfi_n, add n' cfo cfo_n), epsilon_bars_o @ epsilon_bars_n)
                and po = CP.Flow_with_Epsilon_Bars ((cfi_o, remove o' cfo_o), []) in
                Some (sign, flow_n, add_or_remove_if_white o po lines)

            | CP.Flow_with_Epsilons ((cfi_n, cfo_n), epsilons_n) →
                failwith "Color_Fusion.connect_out_opt:␣no␣epsilon␣contractions␣yet"
            end
        end
    end
```

```
let connect_in_out_opt (i, i') (o, o') (sign, flow_n, lines) =
  let open PArray in
  match get_opt i lines, get_opt o lines with
  | None, _ | _, None → None
  | Some flow_i, Some flow_o →
     begin match flow_i, flow_o with
     | (CP.Ghost | CP.Ghost_with_Epsilons _ | CP.Ghost_with_Epsilon_Bars _), _
       | _, (CP.Ghost | CP.Ghost_with_Epsilons _ | CP.Ghost_with_Epsilon_Bars _) → None
     | CP.Flow (cfi_i, cfo_i), CP.Flow (cfi_o, cfo_o) →
        begin match get_opt i' cfi_i, get_opt o' cfo_o with
        | Some cfi, Some cfo when cfi = cfo →
           let pi = CP.Flow (remove i' cfi_i, cfo_i)
           and po = CP.Flow (cfi_o, remove o' cfo_o) in
           Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
        | _, _ → None
        end
     | CP.Flow (cfi_i, cfo_i), CP.Flow_with_Epsilons ((cfi_o, cfo_o), epsilons_o) →
        begin match get_opt i' cfi_i, get_opt o' cfo_o with
        | Some cfi, Some cfo when cfi = cfo →
           let pi = CP.Flow (remove i' cfi_i, cfo_i)
           and po = CP.Flow_with_Epsilons ((cfi_o, remove o' cfo_o), epsilons_o) in
           Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
        | _, _ → None
        end
     | CP.Flow_with_Epsilons ((_, _), _), CP.Flow_with_Epsilons ((_, _), _) →
        failwith "Color_Fusion.connect_in_out_opt:␣incomplete"
     | CP.Flow_with_Epsilon_Bars ((cfi_i, cfo_i), epsilon_bars_i), CP.Flow (cfi_o, cfo_o) →
        begin match get_opt i' cfi_i, get_opt o' cfo_o with
        | Some cfi, Some cfo when cfi = cfo →
           let pi = CP.Flow_with_Epsilon_Bars ((remove i' cfi_i, cfo_i), epsilon_bars_i)
           and po = CP.Flow ((cfi_o, remove o' cfo_o)) in
           Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
        | _, _ → None
        end
     | CP.Flow_with_Epsilon_Bars ((_, _), _), CP.Flow_with_Epsilon_Bars ((_, _), _) →
        failwith "Color_Fusion.connect_in_out_opt:␣incomplete"
     | CP.Flow_with_Epsilons ((cfi_i, cfo_i), epsilons_i), CP.Flow (cfi_o, cfo_o) →
        begin match get_opt i' cfi_i, get_opt o' cfo_o with
        | Some (CP.CF_in cfi), Some cfo when cfi = cfo →
           let pi = CP.Flow_with_Epsilons ((remove i' cfi_i, cfo_i), epsilons_i)
           and po = CP.Flow (cfi_o, remove o' cfo_o) in
           Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
        | Some (CP.Epsilon epsilon_i), Some cfo →
           let epsilon_n = cfo :: epsilon_i in
           let flow_n =
             match flow_n with
             | CP.Ghost → CP.Ghost
             | CP.Ghost_with_Epsilons _ →
                failwith "connect_in_out_opt:␣incomplete"
             | CP.Ghost_with_Epsilon_Bars _ →
                failwith "connect_in_out_opt:␣incomplete"
             | CP.Flow (cfo, cfi) →
                let cfi = map (fun cf → CP.CF_in cf) cfi in
                CP.Flow_with_Epsilons ((cfi, cfo), [epsilon_n])
             | CP.Flow_with_Epsilons (flow, epsilons_n) →
                CP.Flow_with_Epsilons (flow, epsilon_n :: epsilons_n)
             | CP.Flow_with_Epsilon_Bars (flow, epsilon_bars_n) →
                failwith "Color_Fusion.connect_in_out_opt:␣no␣epsilon␣contractions␣yet" in
           let pi = CP.Flow_with_Epsilons ((remove i' cfi_i, cfo_i), epsilons_i)
           and po = CP.Flow (cfi_o, remove o' cfo_o) in
           Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
```

```
          | _, _ → None
          end
      | CP.Flow (cfi_i, cfo_i), CP.Flow_with_Epsilon_Bars ((cfi_o, cfo_o), epsilon_bars_o) →
          begin match get_opt i′ cfi_i, get_opt o′ cfo_o with
          | Some cfi, Some (CP.CF_out cfo) when cfi = cfo →
              let pi = CP.Flow (remove i′ cfi_i, cfo_i)
              and po = CP.Flow_with_Epsilon_Bars ((cfi_o, remove o′ cfo_o), epsilon_bars_o) in
              Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
          | Some cfi, Some (CP.Epsilon_Bar epsilon_bar_o) →
              let epsilon_bar_n = cfi :: epsilon_bar_o in
              let flow_n =
                 match flow_n with
                 | CP.Ghost → CP.Ghost
                 | CP.Ghost_with_Epsilons _ →
                     failwith "connect_in_out_opt:␣incomplete"
                 | CP.Ghost_with_Epsilon_Bars _ →
                     failwith "connect_in_out_opt:␣incomplete"
                 | CP.Flow (cfo, cfi) →
                     let cfo = map (fun cf → CP.CF_out cf) cfo in
                     CP.Flow_with_Epsilon_Bars ((cfi, cfo), [epsilon_bar_n])
                 | CP.Flow_with_Epsilon_Bars (flow, epsilon_bars_n) →
                     CP.Flow_with_Epsilon_Bars (flow, epsilon_bar_n :: epsilon_bars_n)
                 | CP.Flow_with_Epsilons (flow, epsilons_n) →
                     failwith "Color_Fusion.connect_in_out_opt:␣no␣epsilon␣contractions␣yet" in
              let pi = CP.Flow (remove i′ cfi_i, cfo_i)
              and po = CP.Flow_with_Epsilon_Bars ((cfi_o, remove o′ cfo_o), epsilon_bars_o) in
              Some (sign, flow_n, add_or_remove_if_white i pi (add_or_remove_if_white o po lines))
          | _, _ → None
          end
      | CP.Flow_with_Epsilons ((_, _), _), CP.Flow_with_Epsilon_Bars ((_, _), _) →
          failwith "Color_Fusion.connect_in_out_opt:␣no␣epsilon␣contractions␣yet"
      | CP.Flow_with_Epsilon_Bars ((_, _), _), CP.Flow_with_Epsilons ((_, _), _) →
          failwith "Color_Fusion.connect_in_out_opt:␣no␣epsilon␣contractions␣yet"
      end
```

11.2.1 Putting Everything Together

```
let decode_endpoint = function
  | A.I n → (n, 0)
  | A.M (n, m) → (n, m)

let decode_tail t = decode_endpoint (t : A.tail :> A.endpoint)
let decode_tip t = decode_endpoint (t : A.tip :> A.endpoint)
let decode_ghost g = decode_endpoint (g : A.ghost :> A.endpoint)

let endpoint_to_string = function
  | A.I n → string_of_int n
  | A.M (n, m) → string_of_int n ^ "." ^ string_of_int m

let tail_to_string t = endpoint_to_string (t : A.tail :> A.endpoint)
let tip_to_string t = endpoint_to_string (t : A.tip :> A.endpoint)
let ghost_to_string g = endpoint_to_string (g : A.ghost :> A.endpoint)

let connect_arrow_opt n i o lines =
  let i, i′ as ii′ = decode_tail i
  and o, o′ as oo′ = decode_tip o in
  if o = n then
    connect_in_opt o′ ii′ lines
  else if i = n then
    connect_out_opt i′ oo′ lines
  else
    connect_in_out_opt ii′ oo′ lines
```

```
let lines_to_string (sign, flow_n, lines) =
  Printf.sprintf
    "%d*%s<%s"
    sign (CP.to_string flow_n)
    (ThoList.to_string
      (fun (i, p) → Printf.sprintf "%s@%d" (CP.to_string p) i)
      (PArray.to_pairs lines))

let connect_arrow_opt_logging n i o lines =
  let result = connect_arrow_opt n i o lines in
  Printf.eprintf
    "␣␣(%s,%s)␣%s␣>>>␣%s\n"
    (tail_to_string i) (tip_to_string o)
    (lines_to_string lines)
    (match result with
     | None → "None"
     | Some lines → lines_to_string lines);
  result
```

Performan a single connection of the *lines* as described by *arrow_or_ghost*. Use n as the index of the outgoing line. Return the updated outgoing and incoming lines.

```
let connect_arrow_or_ghost_opt :
      int → A.free → int × CP.t × CP.t PArray.t → (int × CP.t × CP.t PArray.t) option =
  fun n arrow_or_ghost lines →
  match arrow_or_ghost with
  | A.Ghost g → connect_ghost_opt n g lines
  | A.Arrow (i, o) → connect_arrow_opt n i o lines
```

Return the signed color *flow* iff all color flows in *lines* have been consumed.

```
let all_lines_consumed_opt (sign, flow, lines) =
  if PArray.is_empty lines then
    Some (sign, flow)
  else
    None
```

Try to use the ghosts and arrows in *connections* to combine the color flows in *lines*.

```
let connect_arrows_opt : A.free list → CP.t list → (int × CP.t) option =
  fun connections lines →
  let n = List.length lines + 1 in
  let rec connect' acc = function
    | arrow :: arrows →
       begin match connect_arrow_or_ghost_opt n arrow acc with
       | None → None
       | Some acc → connect' acc arrows
       end
    | [] → Some acc in
  match connect' (1, CP.white, line_map lines) connections with
  | Some acc → all_lines_consumed_opt acc
  | None → None

let extract_lines_opt endpoints lines =
  let rec extract_lines' acc lines = function
    | [] → Some (List.rev acc, lines)
    | A.I i :: rest →
       begin match PArray.get_opt i lines with
       | None → None
       | Some (CP.Flow (_, cfo)) →
          begin match PArray.to_option_list cfo with
          | [Some cf] →
             extract_lines' (cf :: acc) (PArray.remove i lines) rest
          | _ → failwith "extract_lines_opt:␣incomplete"
          end
```

```
        | Some (CP.Flow_with_Epsilons ((_, _), _)) →
            failwith "extract_lines_opt:␣incomplete"
        | Some (CP.Flow_with_Epsilon_Bars ((_, _), _)) →
            failwith "extract_lines_opt:␣incomplete"
        | Some CP.Ghost →
            failwith "extract_lines_opt:␣incomplete"
        | Some (CP.Ghost_with_Epsilons _) →
            failwith "extract_lines_opt:␣incomplete"
        | Some (CP.Ghost_with_Epsilon_Bars _) →
            failwith "extract_lines_opt:␣incomplete"
        end
    | A.M (_, _) :: _ → failwith "extract_lines_opt:␣incomplete" in
  extract_lines' [] endpoints lines

let fuse1 n_c lines arrow =
  let open Birdtracks in
  match arrow with
  | Arrows { coeff; arrows } →
      begin match connect_arrows_opt arrows lines with
      | None → []
      | Some (sign, flow) →
          [(QC.mul (QC.int sign) (L.eval (QC.int n_c) coeff), flow)]
      end
  | Epsilons _ → failwith "Birdtracks.fuse1:␣Epsilons"
  | Epsilon_Bars _ → failwith "Birdtracks.fuse1:␣Epsilon_Bars"

let fuse n_c vertex lines =
  match vertex with
  | [] →
      if List.for_all CP.is_white lines then
        [(QC.unit, CP.white)]
      else
        []
  | vertex →
      ThoList.flatmap (fuse1 n_c lines) vertex

let flow_to_string flow =
  ThoList.to_string
    (fun (c, p) →
      let p = CP.to_string p in
      if QC.is_unit c then
        p
      else
        Printf.sprintf "%s*%s" (QC.to_string c) p)
    flow

let fuse_logging n_c vertex lines =
  let flow_n = fuse n_c vertex lines in
  Printf.eprintf
    "%s␣>>>␣%s\n"
    (ThoList.to_string CP.to_string lines)
    (flow_to_string flow_n);
  flow_n
```

11.2.2 Unit Tests

```
module Test =
  struct
    open OUnit

    let vertices_equal v1 v2 =
      (Birdtracks.canonicalize v1) = (Birdtracks.canonicalize v2)
```

```
let eq v1 v2 =
  assert_equal ~printer : Birdtracks.to_string_raw ~cmp : vertices_equal v1 v2

let suite_open_contract =
  "open_contract" >:::

    [ "[2;3]␣[1;2;4]" >::
        (fun () → assert_equal None (open_contract [2; 3] [1; 2; 4]));

      "[2;3]␣[1;2;3;4]" >::
        (fun () → assert_equal None (open_contract [2; 3] [1; 2; 3; 4]));

      "[2;3]␣[1;2;3]" >::
        (fun () → assert_equal (Some ( 1, 1)) (open_contract [2; 3] [1; 2; 3]));

      "[1;3]␣[1;2;3]" >::
        (fun () → assert_equal (Some (−1, 2)) (open_contract [1; 3] [1; 2; 3])) ]

let signed_flow_option_to_string = function
  | Some (sign, flow) →
      let flow = CP.to_string flow in
      if sign = 1 then
        flow
      else
        Printf.sprintf "%d*%s" sign flow
  | None → "None"

let test_connect_arrows_msg vertex formatter (expected, result) =
  Format.fprintf
    formatter
    "[%s]:␣expected␣%s,␣got␣%s"
    (ThoList.to_string A.free_to_string vertex)
    (signed_flow_option_to_string expected)
    (signed_flow_option_to_string result)

let test_connect_arrows expected lines vertex =
  assert_equal ~printer : signed_flow_option_to_string
    expected (connect_arrows_opt vertex lines)

let test_connect_arrows_permutations expected lines vertex =
  List.iter
    (fun v →
      assert_equal ~pp_diff : (test_connect_arrows_msg v)
        expected (connect_arrows_opt v lines))
    (Combinatorics.permute vertex)

let suite_connect_arrows =
  "connect_arrows" >:::

    [ "delta" >::
        (fun () →
          test_connect_arrows_permutations
            (Some (1, CP.of_lists [1] []))
            [ CP.of_lists [1] []; CP.white]
            ( 1 ==> 3 ));

      "f:␣1->3->2->1" >::
        (fun () →
          test_connect_arrows_permutations
            (Some (1, CP.of_lists [1] [3]))
            [CP.of_lists [1] [2]; CP.of_lists [2] [3]]
            (A.cycle [1; 3; 2]));

      "f:␣1->2->3->1" >::
        (fun () →
          test_connect_arrows_permutations
            (Some (1, CP.of_lists [1] [2]))
            [CP.of_lists [3] [2]; CP.of_lists [1] [3]]
```

```
                (A.cycle [1; 2; 3])) ]

  let test_fuse_msg vertex lines formatter (expected, result) =
    Format.fprintf
      formatter
      "%s // %s => %s failed, got %s instead"
      (Birdtracks.to_string vertex)
      (ThoList.to_string CP.to_string lines)
      (flow_to_string expected)
      (flow_to_string result)

  let compare_fusion (c1, p1) (c2, p2) =
    let c = Algebra.QC.compare c1 c2 in
    if c ≠ 0 then
      c
    else
      CP.compare p1 p2

  let equal_fusion f1 f2 =
    compare_fusion f1 f2 = 0

  let cmp_fusions f1 f2 =
    let f1 = List.sort compare_fusion f1
    and f2 = List.sort compare_fusion f2 in
    try
      List.for_all2 equal_fusion f1 f2
    with
    | Invalid_argument _ → false

  let test_fuse expected vertex lines =
    let nc = 3 in
    assert_equal
      ~cmp : cmp_fusions
      ~pp_diff : (test_fuse_msg vertex lines)
      expected (fuse nc vertex lines)
```

This way, we can write *vertex // lines => expected* in the tests.

```
  let (//) vertex lines = (vertex, lines)
  let (=>) (vertex, lines) expected = test_fuse expected vertex lines
```

Abbreviations

```
  let tf = test_fuse
  let e = QC.unit
  let half = QC.fraction 2
  let w = CP.white
```

Quarks and anti quarks:

```
  let q i = CP.of_lists [i] [ ]
  let aq i = CP.of_lists [ ] [i]
```

Diquarks and anti diquarks:

```
  let dq i j = CP.of_lists [i; j] [ ]
  let adq i j = CP.of_lists [ ] [i; j]
```

Gluons without ghosts

```
  let g i j = CP.of_lists [i] [j]
```

Couplings

```
  let d = SU3.delta3
  let d6 = SU3.delta6
  let t = SU3.t
  let t6 = SU3.t6
  let k6 = SU3.k6
  let k6b = SU3.k6bar
```

```
let suite_binary_qed3 =
  "triplet" >:::
    [ "1␣2␣" >:: (fun () → d 2 1 // [q 1; aq 1] => [(e, w)]);
      "1␣2’" >:: (fun () → d 2 1 // [aq 1; q 1 ] => []);
      "2␣1␣" >:: (fun () → d 1 2 // [aq 1; q 1 ] => [(e, w)]);
      "2␣1’" >:: (fun () → d 1 2 // [q 1; aq 1] => []);
      "1␣3␣" >:: (fun () → d 3 1 // [q 1; w ] => [(e, q 1)]);
      "2␣3␣" >:: (fun () → d 3 2 // [w; q 1 ] => [(e, q 1)]);
      "3␣1␣" >:: (fun () → d 1 3 // [aq 1; w ] => [(e, aq 1)]);
      "3␣2␣" >:: (fun () → d 2 3 // [w; aq 1] => [(e, aq 1)]) ]

let suite_binary_qed6 =
  "sextet" >:::
    [ "1␣2␣␣" >:: (fun () → d6 2 1 // [dq 1 2; adq 1 2] => [(half, w)]);
      "1␣2’␣" >:: (fun () → d6 2 1 // [dq 1 2; adq 2 1] => [(half, w)]);
      "1␣2’’" >:: (fun () → d6 2 1 // [dq 1 2; adq 1 3] => []) ]

let suite_binary_qcd3 =
  "triplet" >:::
    [ "1␣2␣" >:: (fun () → t 3 2 1 // [q 1; aq 2] => [(e, g 1 2)]);
      "1␣2’" >:: (fun () → t 3 2 1 // [aq 1; q 2] => []) ]

let suite_binary_qcd6 =
  "sextet" >:::
    [ "1␣2" >:: (fun () → t6 3 2 1 // [dq 1 2; adq 2 3] => [(half, g 1 3)]) ]

let suite_binary_k6 =
  "k6(bar)" >:::
    [ "321␣␣" >:: (fun () → k6b 3 2 1 // [q 1; q 2 ] => [(e, dq 2 1); (e, dq 1 2)]);
      "321*␣" >:: (fun () → k6 3 2 1 // [aq 1; aq 2] => [(e, adq 2 1); (e, adq 1 2)]);
      "123␣␣" >:: (fun () → k6b 1 2 3 // [adq 1 2; q 1] => [(e, aq 2)]);
      "132␣␣" >:: (fun () → k6b 1 3 2 // [adq 1 2; q 1] => [(e, aq 2)]);
      "123’␣" >:: (fun () → k6b 1 2 3 // [adq 1 2; q 2] => [(e, aq 1)]);
      "132’␣" >:: (fun () → k6b 1 3 2 // [adq 1 2; q 2] => [(e, aq 1)]);
      "213␣␣" >:: (fun () → k6b 2 1 3 // [q 1; adq 1 2] => [(e, aq 2)]);
      "231␣␣" >:: (fun () → k6b 2 3 1 // [q 1; adq 1 2] => [(e, aq 2)]);
      "213’␣" >:: (fun () → k6b 2 1 3 // [q 2; adq 1 2] => [(e, aq 1)]);
      "231’␣" >:: (fun () → k6b 2 3 1 // [q 2; adq 1 2] => [(e, aq 1)]);
      "123␣*" >:: (fun () → k6 1 2 3 // [dq 1 2; aq 1] => [(e, q 2)]);
      "132␣*" >:: (fun () → k6 1 3 2 // [dq 1 2; aq 1] => [(e, q 2)]);
      "123’*" >:: (fun () → k6 1 2 3 // [dq 1 2; aq 2] => [(e, q 1)]);
      "132’*" >:: (fun () → k6 1 3 2 // [dq 1 2; aq 2] => [(e, q 1)]);
      "213␣*" >:: (fun () → k6 2 1 3 // [aq 1; dq 1 2] => [(e, q 2)]);
      "231␣*" >:: (fun () → k6 2 3 1 // [aq 1; dq 1 2] => [(e, q 2)]);
      "213’*" >:: (fun () → k6 2 1 3 // [aq 2; dq 1 2] => [(e, q 1)]);
      "231’*" >:: (fun () → k6 2 3 1 // [aq 2; dq 1 2] => [(e, q 1)]) ]

let suite_binary =
  "binary" >:::
    [ "colorless" >:: (fun () → [] // [w; w] => [(e, w)]);
      "qed" >::: [ suite_binary_qed3; suite_binary_qed6; suite_binary_k6 ];
      "qcd" >::: [ suite_binary_qcd3; suite_binary_qcd6 ] ]

let suite_tertiary =
  "tertiary" >:::
    [ "colorless" >:: (fun () → [] // [w; w; w] => [(e, w)]);
      "qed␣1␣2" >:: (fun () → d 2 1 // [q 1; aq 1; w] => [(e, w)]);
      "qed␣1␣3" >:: (fun () → d 3 1 // [q 1; w; aq 1] => [(e, w)]);
      "qed␣2␣3" >:: (fun () → d 3 2 // [w; q 1; aq 1] => [(e, w)]) ]

let suite_nary =
  "n-ary" >:::
    [ "colorless" >:: (fun () → [] // [w; w; w; w; w] => [(e, w)]) ]

let suite_fuse =
      "fuse" >:::
        [ suite_binary;
          suite_tertiary;
          suite_nary ]

  let suite =
    "Color_Fusion" >:::
      [suite_open_contract;
       suite_connect_arrows;
       suite_fuse]

  let suite_long =
    "Color_Fusion␣long" >:::
      []
end
```

—12— Color

12.1 Interface of Color

```
module type Test =
  sig
    val suite : OUnit.test
    val suite_long : OUnit.test
  end
```

12.1.1 Quantum Numbers

Color is not necessarily the SU(3) of QCD. Conceptually, it can be any *unbroken* symmetry (*broken* symmetries correspond to *Model.flavor*). In order to keep the group theory simple, we confine ourselves to the fundamental and adjoint representation of a single SU(N_C) for the moment. Therefore, particles are either color singlets or live in the defining representation of SU(N_C): $SUN(|N_C|)$, its conjugate $SUN(-|N_C|)$ or in the adjoint representation of SU(N_C): $AdjSUN(N_C)$.

```
type t =
  | Singlet
  | SUN of int
  | AdjSUN of int
  | YT of int Young.tableau
  | YTC of int Young.tableau

val conjugate : t → t
val compare : t → t → int
```

12.1.2 Color Flows

This computes the color flow as used by WHIZARD:

```
module type Flow =
  sig

    type color
    type t = color list × color list
    val rank : t → int

    val of_list : int list → color
    val ghost : unit → color
    val to_lists : t → int list list
    val in_to_lists : t → int list list
    val out_to_lists : t → int list list
    val ghost_flags : t → bool list
    val in_ghost_flags : t → bool list
    val out_ghost_flags : t → bool list
```

A factor is a list of powers

$$\sum_i \left(\frac{num_i}{den_i}\right)^{power_i} \tag{12.1}$$

```
    type power = { num : int; den : int; power : int }
    type factor = power list
```

Compute the product of two color flows.

```
    val factor : t → t → factor
    val zero : factor
```

Take a list of color flows and compute a table of all squares and interferences.

```
    val factor_table : t list → factor array array

    module Test : Test

  end
```

```
module Flow : Flow
```

12.1.3 Vertex Color Flows

```
module Vertex : module type of SU3
```

12.2 Implementation of Color

```
module type Test =
  sig
    val suite : OUnit.test
    val suite_long : OUnit.test
  end
```

12.2.1 Quantum Numbers

```
type t =
  | Singlet
  | SUN of int
  | AdjSUN of int
  | YT of int Young.tableau
  | YTC of int Young.tableau

let conjugate = function
  | Singlet → Singlet
  | SUN n → SUN (−n)
  | AdjSUN n → AdjSUN n
  | YT y → YTC y
  | YTC y → YT y

let compare c1 c2 =
  match c1, c2 with
  | Singlet, Singlet → 0
  | Singlet, _ → −1
  | _, Singlet → 1
  | SUN n, SUN n′ → compare n n′
  | SUN _, AdjSUN _ → −1
  | AdjSUN _, SUN _ → 1
  | AdjSUN n, AdjSUN n′ → compare n n′
  | YT y, YT y′ → compare y y′
  | YT _, YTC _ → −1
  | YTC _, YT _ → 1
  | YTC y, YTC y′ → compare y y′
  | _, (YT _ | YTC _) → −1
  | (YT _ | YTC _) , _ → 1
```

12.2.2 Color Flows

```
module type Flow =
  sig
    type color
    type t = color list × color list
    val rank : t → int
    val of_list : int list → color
    val ghost : unit → color
    val to_lists : t → int list list
    val in_to_lists : t → int list list
    val out_to_lists : t → int list list
    val ghost_flags : t → bool list
    val in_ghost_flags : t → bool list
    val out_ghost_flags : t → bool list
    type power = { num : int; den : int; power : int }
    type factor = power list
    val factor : t → t → factor
    val zero : factor
    val factor_table : t list → factor array array
    module Test : Test
  end

module Flow : Flow =
  struct
```

All *int*s are non-zero!

```
    type color =
      | Flow of Color_Propagator.flow
      | Ghost

    let to_cp = function
      | Flow cf → Color_Propagator.Flow cf
      | Ghost → Color_Propagator.Ghost

    let color_to_string c =
      Color_Propagator.to_string (to_cp c)
```

Incoming and outgoing, since we need to cross the incoming states.

```
    type t = color list × color list

    let rank cflow =
      2
```

Constructors

```
    let ghost () =
      Ghost

    let of_list = function
      | [0; 0] → Flow (PArray.empty, PArray.empty)
      | [c; 0] → Flow (PArray.of_pairs [(1, c)], PArray.empty)
      | [0; c] → Flow (PArray.empty, PArray.of_pairs [(1, − c)])
      | [c1; c2] → Flow (PArray.of_pairs [(1, c1)], PArray.of_pairs [(1, − c2)])
      | _ → invalid_arg "Color.Flow.of_list:␣num_lines␣!=␣2"

    let to_list = function
      | Ghost → [0; 0]
      | Flow (cfi, cfo) →
         begin match PArray.to_pairs cfi, PArray.to_pairs cfo with
         | [], [] → [0; 0]
         | [(1, c)], [] → [c; 0]
         | [], [(1, c)] → [0; − c]
```

```
        | [(1, c1)], [(1, c2)] → [c1; − c2]
        | _, _ → failwith "Color.Flow.to_list:␣incomplete"
        end

  let to_lists (cfin, cfout) =
    (List.map to_list cfin) @ (List.map to_list cfout)

  let in_to_lists (cfin, _) =
    List.map to_list cfin

  let out_to_lists (_, cfout) =
    List.map to_list cfout

  let ghost_flag = function
    | Flow _ → false
    | Ghost → true

  let ghost_flags (cfin, cfout) =
    (List.map ghost_flag cfin) @ (List.map ghost_flag cfout)

  let in_ghost_flags (cfin, _) =
    List.map ghost_flag cfin

  let out_ghost_flags (_, cfout) =
    List.map ghost_flag cfout
```

Evaluation

```
  type power = { num : int; den : int; power : int }
  type factor = power list
  let zero = []

  let factor_to_string = function
    | [] → "0"
    | factor →
        String.concat "+"
          (List.map
              (fun p →
                 Printf.sprintf
                   "%d%s%s"
                   p.num
                   (if p.den ≠ 1 then "/" ^ string_of_int p.den else "")
                   (match p.power with
                    | 0 → ""
                    | 1 → "*N"
                    | n → "*N^" ^ string_of_int n))
              factor)

  let conjugate = function
    | Flow (cfi, cfo) → Flow (cfo, cfi)
    | Ghost → Ghost

  let cross_in (cin, cout) =
    cin @ (List.map conjugate cout)

  let cross_out (cin, cout) =
    (List.map conjugate cin) @ cout
```

Handling $\mathrm{tr}(F_{\mu\nu}F^{\mu\nu})$ *couplings, a.k.a. Hgg*

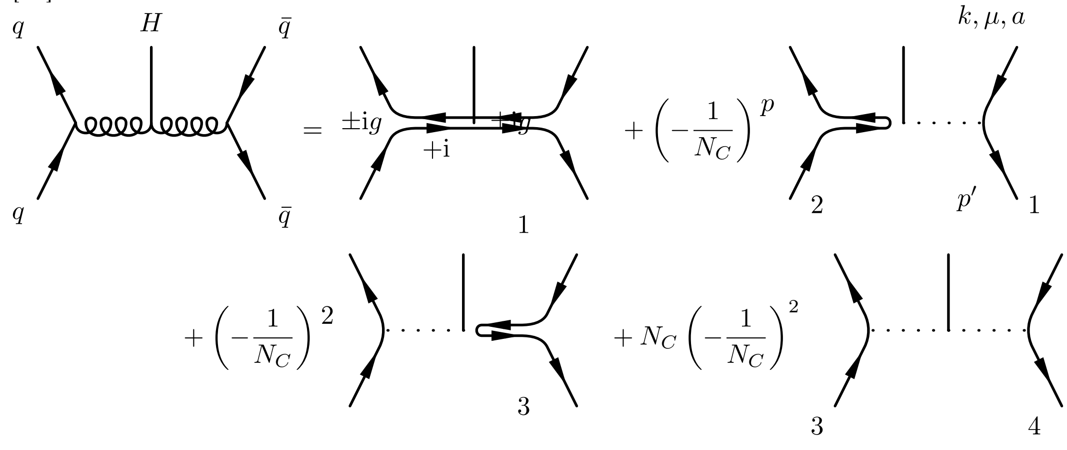
If the model contains couplings of the form $\mathrm{tr}(F_{\mu\nu}F^{\mu\nu})$, e. g. the effective Hgg couplings, the color flow rules and the evaluation of color weights require special care. These couplings are problematic in our recursive construction, since fusing a colorless state with a U(1) ghost produces a trace gluon in addition to a U(1) ghost. But for this fresh trace gluon, no canonical color flow index is available!

A possible solution could be the introduction of "wild card" color flow that are replaced be concrete color flows only at the matching of the brakets. This is worth investigating, but can be postponed in favor of the well tested pragmatic approach.

There are three different cases to consider:

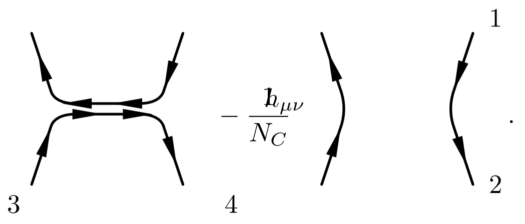
1. First consider the case that neither gluon is directly connected by a string of such couplings to the external states. In this case, the gluons must be connected to matter, since the gluon self couplings contain no ghost terms. Fortunately, if suffices to ajust the ghost-ghost coupling to account for the missing ghost-trace couplings.

   The prototypical example is Higgs production in $q\bar{q}$ scattering via the effective Hgg coupling expanded as in [17]:



(12.2a)

   the sum of which corresponds to the same simple color flows as gluon exchange



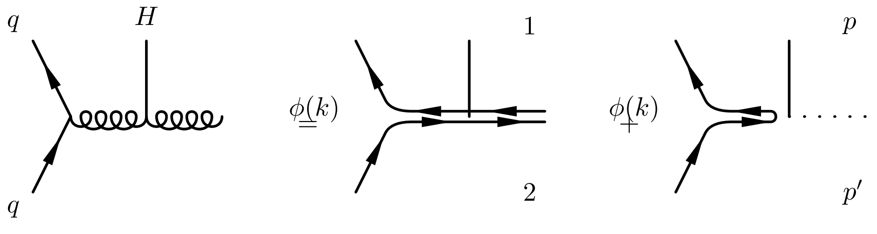
(12.2b)

   Squaring and summing these produces the correct result

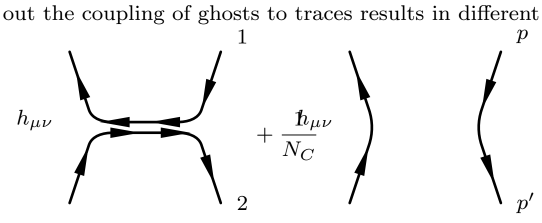
$$N_C^2 + N_C\left(-\frac{1}{N_C}\right) + N_C\left(-\frac{1}{N_C}\right) + N_C^2\left(-\frac{1}{N_C}\right)^2 = N_C^2 - 1\,. \tag{12.2c}$$

   This result can be reproduced without coupling of trace gluons to ghosts by simply replacing the ghost-ghost coupling N_C by $-N_C$ in order to cancel the minus sign from the additional ghost propagator[1].

2. In the second case of one gluon connected to matter and the other to an external state, no special treatment is required. The prototypical example is $q\bar{q}\to Hg$

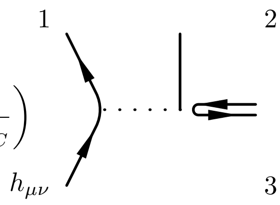


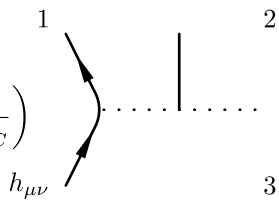
[1]For comparison, naively leaving out the coupling of ghosts to traces results in different color flows



Squaring and summing these would produce the incorrect result

$$N_C^2 + N_C\frac{1}{N_C} + N_C\frac{1}{N_C} + N_C^2\left(\frac{1}{N_C}\right)^2 = N_C^2 + 3\,.$$

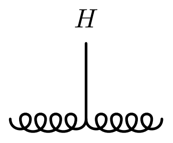
$+\left(-\frac{1}{N_C}\right)$


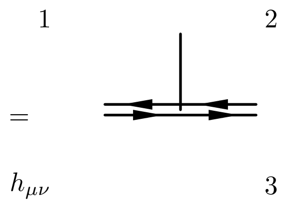
$+N_C\left(-\frac{1}{N_C}\right)$


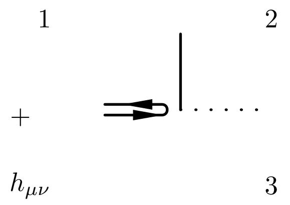
(12.3)

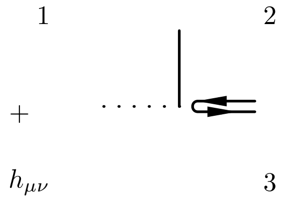
The correct result for the summed square is again $N_C^2 - 1$, where the two color flow diagrams with an external ghost cancel. In the simplified rules, the $\mathrm{U}(N_C)$ gluons contribute N_C^2 and the ghost -1.

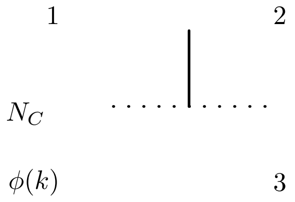
3. In the third and final case of both gluons connected to external states, we have to apply a fudge factor replacing N_C^2 by $N_C^2 - 2$ for each cycle of color disconnected gluons. The calculation is straightforward, since there is no interference of external ghosts and $\mathrm{U}(N_C)$ gluons in the sum of squares.



$=$


$+$


$+$


$+N_C$


(12.4)

The latter contributes a factor of N_C^2 (two loops) and the former a factor of $(-N_C)^2(-1/N_C)^2 = 1$ (one $-N_C$ fom each vertex and one $-1/N_C$ from each line across the cut). Therefore the sum would be N_C^2+1 in contrast to the correct result $N_C^2 - 1$. The correct result is then obtained by multiplying the gluon term N_C^2 by $1 - 2/N_C^2$

$$N_C^2 + 1 \to N_C^2\left(1 - \frac{2}{N_C^2}\right) + 1 = N_C^2 - 2 + 1 = N_C^2 - 1\,. \tag{12.5}$$

The factor $(1 - 2/N_C^2)^n$ in the formula

$$N_C^l \left(-\frac{1}{N_C}\right)^k \left(\frac{N_C^2-2}{N_C^2}\right)^n, \tag{12.6}$$

where l is the number of closed color cycles (*cycles* below), k is the number of external ghosts (*ghosts*) and n is the number of gluon cycles (*gluon_cycles*). is the fudge factor taking care of the couplings of U(1) ghosts to trace gluons.
endpoints_of_colors colors creates maps from the position of the external colors in *colors* to the tips and tails connected by color flow lines. Also produce a set of the positions of external ghosts.

```
module IMap = Map.Make(Int)
module ISet = Set.Make(Int)

type endpoints =
  { tails : int IMap.t;
    tips : int IMap.t;
    ghosts : ISet.t }

type color_kind =
  | CK_Flow of int × int
  | CK_Ghost

let color_kind = function
  | Flow (cfi, cfo) → CK_Flow (List.length (PArray.to_pairs cfi), List.length (PArray.to_pairs cfo))
  | Ghost → CK_Ghost

let equal_color_kind1 c1 c2 =
  color_kind c1 = color_kind c2
```

```
let equal_color_kind f1 f2 =
  List.for_all2 equal_color_kind1 f1 f2

let empty_endpoints =
  { tails = IMap.empty;
    tips = IMap.empty;
    ghosts = ISet.empty }

let add_endpoint endpoints n = function
  | Ghost → { endpoints with ghosts = ISet.add n endpoints.ghosts }
  | Flow (cfi, cfo) →
      begin match PArray.to_pairs cfi, PArray.to_pairs cfo with
      | [], [] → endpoints
      | [(1, c)], [] → { endpoints with tips = IMap.add (abs c) n endpoints.tips }
      | [], [(1, c)] → { endpoints with tails = IMap.add (abs c) n endpoints.tails }
      | [(1, c1)], [(1, c2)] →
          { endpoints with
            tips = IMap.add (abs c1) n endpoints.tips;
            tails = IMap.add (abs c2) n endpoints.tails }
      | _, _ → failwith "Color.Flow.add_endpoint:␣incomplete"
      end

let endpoints_of_colors colors =
  let _, endpoints =
    List.fold_left
      (fun (n, endpoints) endpoint → (succ n, add_endpoint endpoints n endpoint))
      (1, empty_endpoints) colors in
  endpoints
```

Merge the maps of tips and tails to find the pair of connected external colors.

```
let links_of_endpoints endpoints =
  IMap.merge
    (fun _ tail tip →
      match tail, tip with
      | None, None → None
      | Some tail, Some tip → Some (tail, tip)
      | Some tail, None → invalid_arg ("no␣tip␣for␣tail␣" ^ string_of_int tail)
      | None, Some tip → invalid_arg ("no␣tail␣for␣tip␣" ^ string_of_int tip))
    endpoints.tails endpoints.tips
```

Create an *Arrow.free list* that can be used by *Birdtracks*.

```
let arrows_of_links links =
  IMap.fold (fun _ (tail, tip) acc → Arrow.Infix.( tail => tip ) :: acc) links []

module LSet = Set.Make (struct type t = int × int let compare = Stdlib.compare end)
```

Find the set bidirectional links by computing the intersection of the set of links with the set of reversed links. We must keep both directions for *Birdtracks.multiply* to succeed.

```
let double_links links =
  let links, rev_links =
    IMap.fold
      (fun _ (tail, tip) (links, rev_links) →
        (LSet.add (tail, tip) links, LSet.add (tip, tail) rev_links))
      links (LSet.empty, LSet.empty) in
  LSet.inter links rev_links

let birdtracks_of_arrows arrows =
  Birdtracks.( relocate (−) [ Arrows { coeff = Algebra.Laurent.unit; arrows } ] )

type flow =
  { flows : Birdtracks.t;
    gluons : Birdtracks.t }

let birdtracks colors =
  let endpoints = endpoints_of_colors colors in
```

```
    let links = links_of_endpoints endpoints in
    let gluons = double_links links in
    let flow =
      ISet.fold
        (fun ghost acc → Arrow.Infix.( ?? ghost) :: acc)
        endpoints.ghosts (arrows_of_links links)
    and gluons =
      LSet.fold (fun (tail, tip) acc → Arrow.Infix.( tail => tip ) :: acc) gluons [] in
    { flows = birdtracks_of_arrows flow;
      gluons = birdtracks_of_arrows gluons }
```

$1 - 2/N_C^2$

```
  let fudge_factor =
    Algebra.Laurent.ints [(1, 0); (−2, −2)]

  let factor_birdtracks f1 f2 =
    let open Birdtracks in
    match number (Infix.( f1.flows *** rev f2.flows )) with
    | None → failwith "factor_new"
    | Some result →
       if Algebra.Laurent.is_null result then
         result
       else
         let gluons = Infix.( f1.gluons *** rev f2.gluons ) in
         match number gluons with
         | None → result
         | Some gluons →
            begin match Algebra.Laurent.log gluons with
            | None → failwith "factor_birdtracks␣log"
            | Some (coeff, 0) → result
            | Some (coeff, n) →
               if ¬ (Algebra.QC.is_unit coeff) then
                 failwith "factor_birdtracks␣log␣is_unit";
               if n mod 2 ≠ 0 then
                 failwith "factor_birdtracks␣log␣is␣odd";
               Algebra.Laurent.mul result (Algebra.Laurent.pow fudge_factor (n/2))
            end

  let factor f1 f2 =
    let f1 = cross_out f1
    and f2 = cross_out f2 in
    if equal_color_kind f1 f2 then
      factor_birdtracks (birdtracks f1) (birdtracks f2)
    else
      Algebra.Laurent.null

  let factor_of_laurent l =
    List.map
      (fun (c, power) →
        let num, den = Algebra.Q.to_ratio (Algebra.QC.re c) in
        { num; den; power} )
      (Algebra.Laurent.to_list l)

  let factor_birdtracks f1 f2 =
    factor_of_laurent (factor_birdtracks f1 f2)

  let factor f1 f2 =
    factor_of_laurent (factor f1 f2)

  let factor_table cf_list =
    let cf_array = Array.of_list (List.map cross_out cf_list) in
    let birdtracks_array = Array.map birdtracks cf_array in
    let ncf = Array.length cf_array in
    let cf_table = Array.make_matrix ncf ncf zero in
    for i = 0 to pred ncf do
      for j = 0 to i do
        if equal_color_kind cf_array.(i) cf_array.(j) then
          begin
            cf_table.(i).(j) ← factor_birdtracks birdtracks_array.(i) birdtracks_array.(j);
            cf_table.(j).(i) ← cf_table.(i).(j)
          end
      done
    done;
    cf_table

  module Test : Test =
    struct

      open OUnit
```

Here and elsewhere, we have to resist the temptation to define these tests as functions with an additional argument () in the hope to avoid having to package them into an explicit thunk fun () → *eq v1 v2* in order to delay evaluation. It turns out that the runtime would then sometimes evaluate the argument *v1* or *v2* even *before* the test is run. For pure functions, there is no difference, but the compiler appears to treat explicit thunks specially.

I haven't yet managed to construct a small demonstrator to find out in which circumstances the premature evaluation happens.

```
      let suite =
        "Color.Flow" >:::
          [ ]

      let suite_long =
        "Color.Flow␣long" >:::
          [ ]

    end
 end
```

12.2.3 SU(N_C)

```
module Vertex = SU3
```

—13— Colorization

13.1 Interface of Colorize

13.1.1 . . .

```
module It (M : Model.T) :
    Model.Colorized with type flavor_sans_color = M.flavor
    and type constant = M.constant
    and type coupling_order = M.coupling_order

module Gauge (M : Model.Gauge) :
    Model.Colorized_Gauge with type flavor_sans_color = M.flavor
    and type constant = M.constant
    and type coupling_order = M.coupling_order
```

13.2 Implementation of Colorize

13.2.1 Auxiliary functions

Exceptions

```
let incomplete s =
  failwith ("Colorize." ^ s ^ " not done yet!")

let invalid s =
  invalid_arg ("Colorize." ^ s ^ " must not be evaluated!")

let impossible s =
  invalid_arg ("Colorize." ^ s ^ " can't happen! (but just did ...)")

let mismatch s =
  invalid_arg ("Colorize." ^ s ^ " mismatch of representations!")

let su0 s =
  invalid_arg ("Colorize." ^ s ^ ": found SU(0)!")

let colored_vertex s =
  invalid_arg ("Colorize." ^ s ^ ": colored vertex!")

let non_legacy_color s cp =
  invalid_arg ("Colorize." ^ s ^ ": non legacy color in legacy code: " ^
                Color_Propagator.to_string cp)

let baryonic_vertex s =
  invalid_arg ("Colorize." ^ s ^
               ": baryonic (i.e. eps_ijk) vertices not supported yet!")

let color_flow_ambiguous s =
  invalid_arg ("Colorize." ^ s ^ ": ambiguous color flow!")

let color_flow_of_string s =
  let c = int_of_string s in
  if c < 1 then
    invalid_arg ("Colorize." ^ s ^ ":␣color␣flow␣#␣<␣1!")
  else
    c

let young_tableaux s =
  failwith ("Colorize." ^ s ^ "␣classic␣colorizer␣can't␣support␣Young␣tableaux!")
```

Multiplying Vertices by a Constant Factor

```
module Q = Algebra.Q
module QC = Algebra.QC

let of_int n =
  QC.make (Q.make n 1) Q.null

let integer z =
  if Q.is_null (QC.im z) then
    let x = QC.re z in
    try
      Some (Q.to_integer x)
    with
    | _ → None
  else
    None

let mult_vertex3 x v =
  let open Coupling in
  match v with
  | FBF (c, fb, coup, f) →
      FBF ((x × c), fb, coup, f)
  | PBP (c, fb, coup, f) →
      PBP ((x × c), fb, coup, f)
  | BBB (c, fb, coup, f) →
      BBB ((x × c), fb, coup, f)
  | GBG (c, fb, coup, f) →
      GBG ((x × c), fb, coup, f)
  | Gauge_Gauge_Gauge c →
      Gauge_Gauge_Gauge (x × c)
  | I_Gauge_Gauge_Gauge c →
      I_Gauge_Gauge_Gauge (x × c)
  | Aux_Gauge_Gauge c →
      Aux_Gauge_Gauge (x × c)
  | Scalar_Vector_Vector c →
      Scalar_Vector_Vector (x × c)
  | Aux_Vector_Vector c →
      Aux_Vector_Vector (x × c)
  | Aux_Scalar_Vector c →
      Aux_Scalar_Vector (x × c)
  | Scalar_Scalar_Scalar c →
      Scalar_Scalar_Scalar (x × c)
  | Aux_Scalar_Scalar c →
      Aux_Scalar_Scalar (x × c)
  | Vector_Scalar_Scalar c →
      Vector_Scalar_Scalar (x × c)
  | Graviton_Scalar_Scalar c →
      Graviton_Scalar_Scalar (x × c)
  | Graviton_Vector_Vector c →
      Graviton_Vector_Vector (x × c)
  | Graviton_Spinor_Spinor c →
      Graviton_Spinor_Spinor (x × c)
  | Dim4_Vector_Vector_Vector_T c →
```

```
    Dim4_Vector_Vector_Vector_T (x × c)
  | Dim4_Vector_Vector_Vector_L c →
    Dim4_Vector_Vector_Vector_L (x × c)
  | Dim4_Vector_Vector_Vector_T5 c →
    Dim4_Vector_Vector_Vector_T5 (x × c)
  | Dim4_Vector_Vector_Vector_L5 c →
    Dim4_Vector_Vector_Vector_L5 (x × c)
  | Dim6_Gauge_Gauge_Gauge c →
    Dim6_Gauge_Gauge_Gauge (x × c)
  | Dim6_Gauge_Gauge_Gauge_5 c →
    Dim6_Gauge_Gauge_Gauge_5 (x × c)
  | Aux_DScalar_DScalar c →
    Aux_DScalar_DScalar (x × c)
  | Aux_Vector_DScalar c →
    Aux_Vector_DScalar (x × c)
  | Dim5_Scalar_Gauge2 c →
    Dim5_Scalar_Gauge2 (x × c)
  | Dim5_Scalar_Gauge2_Skew c →
    Dim5_Scalar_Gauge2_Skew (x × c)
  | Dim5_Scalar_Vector_Vector_T c →
    Dim5_Scalar_Vector_Vector_T (x × c)
  | Dim5_Scalar_Vector_Vector_U c →
    Dim5_Scalar_Vector_Vector_U (x × c)
  | Dim5_Scalar_Vector_Vector_TU c →
    Dim5_Scalar_Vector_Vector_TU (x × c)
  | Dim5_Scalar_Scalar2 c →
    Dim5_Scalar_Scalar2 (x × c)
  | Scalar_Vector_Vector_t c →
    Scalar_Vector_Vector_t (x × c)
  | Dim6_Vector_Vector_Vector_T c →
    Dim6_Vector_Vector_Vector_T (x × c)
  | Tensor_2_Vector_Vector c →
    Tensor_2_Vector_Vector (x × c)
  | Tensor_2_Vector_Vector_cf c →
    Tensor_2_Vector_Vector_cf (x × c)
  | Tensor_2_Scalar_Scalar c →
    Tensor_2_Scalar_Scalar (x × c)
  | Tensor_2_Scalar_Scalar_cf c →
    Tensor_2_Scalar_Scalar_cf (x × c)
  | Tensor_2_Vector_Vector_1 c →
    Tensor_2_Vector_Vector_1 (x × c)
  | Tensor_2_Vector_Vector_t c →
    Tensor_2_Vector_Vector_t (x × c)
  | Dim5_Tensor_2_Vector_Vector_1 c →
    Dim5_Tensor_2_Vector_Vector_1 (x × c)
  | Dim5_Tensor_2_Vector_Vector_2 c →
    Dim5_Tensor_2_Vector_Vector_2 (x × c)
  | TensorVector_Vector_Vector c →
    TensorVector_Vector_Vector (x × c)
  | TensorVector_Vector_Vector_cf c →
    TensorVector_Vector_Vector_cf (x × c)
  | TensorVector_Scalar_Scalar c →
    TensorVector_Scalar_Scalar (x × c)
  | TensorVector_Scalar_Scalar_cf c →
    TensorVector_Scalar_Scalar_cf (x × c)
  | TensorScalar_Vector_Vector c →
    TensorScalar_Vector_Vector (x × c)
  | TensorScalar_Vector_Vector_cf c →
    TensorScalar_Vector_Vector_cf (x × c)
  | TensorScalar_Scalar_Scalar c →
    TensorScalar_Scalar_Scalar (x × c)
  | TensorScalar_Scalar_Scalar_cf c →
      TensorScalar_Scalar_Scalar_cf (x × c)
  | Dim7_Tensor_2_Vector_Vector_T c →
      Dim7_Tensor_2_Vector_Vector_T (x × c)
  | Dim6_Scalar_Vector_Vector_D c →
      Dim6_Scalar_Vector_Vector_D (x × c)
  | Dim6_Scalar_Vector_Vector_DP c →
      Dim6_Scalar_Vector_Vector_DP (x × c)
  | Dim6_HAZ_D c →
      Dim6_HAZ_D (x × c)
  | Dim6_HAZ_DP c →
      Dim6_HAZ_DP (x × c)
  | Gauge_Gauge_Gauge_i c →
      Gauge_Gauge_Gauge_i (x × c)
  | Dim6_GGG c →
      Dim6_GGG (x × c)
  | Dim6_AWW_DP c →
      Dim6_AWW_DP (x × c)
  | Dim6_AWW_DW c →
      Dim6_AWW_DW (x × c)
  | Dim6_Gauge_Gauge_Gauge_i c →
      Dim6_Gauge_Gauge_Gauge_i (x × c)
  | Dim6_HHH c →
      Dim6_HHH (x × c)
  | Dim6_WWZ_DPWDW c →
      Dim6_WWZ_DPWDW (x × c)
  | Dim6_WWZ_DW c →
      Dim6_WWZ_DW (x × c)
  | Dim6_WWZ_D c →
      Dim6_WWZ_D (x × c)

let cmult_vertex3 z v =
  match integer z with
  | None → invalid_arg "cmult_vertex3"
  | Some x → mult_vertex3 x v

let mult_vertex4 x v =
  let open Coupling in
  match v with
  | Scalar4 c →
      Scalar4 (x × c)
  | Scalar2_Vector2 c →
      Scalar2_Vector2 (x × c)
  | Vector4 ic4_list →
      Vector4 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | DScalar4 ic4_list →
      DScalar4 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | DScalar2_Vector2 ic4_list →
      DScalar2_Vector2 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | GBBG (c, fb, b2, f) →
      GBBG ((x × c), fb, b2, f)
  | Vector4_K_Matrix_tho (c, ic4_list) →
      Vector4_K_Matrix_tho ((x × c), ic4_list)
  | Vector4_K_Matrix_jr (c, ch2_list) →
      Vector4_K_Matrix_jr ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_t0 (c, ch2_list) →
      Vector4_K_Matrix_cf_t0 ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_t1 (c, ch2_list) →
      Vector4_K_Matrix_cf_t1 ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_t2 (c, ch2_list) →
      Vector4_K_Matrix_cf_t2 ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_t_rsi (c, ch2_list) →
```

```
      Vector4_K_Matrix_cf_t_rsi ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_m0 (c, ch2_list) →
      Vector4_K_Matrix_cf_m0 ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_m1 (c, ch2_list) →
      Vector4_K_Matrix_cf_m1 ((x × c), ch2_list)
  | Vector4_K_Matrix_cf_m7 (c, ch2_list) →
      Vector4_K_Matrix_cf_m7 ((x × c), ch2_list)
  | DScalar2_Vector2_K_Matrix_ms (c, ch2_list) →
      DScalar2_Vector2_K_Matrix_ms ((x × c), ch2_list)
  | DScalar2_Vector2_m_0_K_Matrix_cf (c, ch2_list) →
      DScalar2_Vector2_m_0_K_Matrix_cf ((x × c), ch2_list)
  | DScalar2_Vector2_m_1_K_Matrix_cf (c, ch2_list) →
      DScalar2_Vector2_m_1_K_Matrix_cf ((x × c), ch2_list)
  | DScalar2_Vector2_m_7_K_Matrix_cf (c, ch2_list) →
      DScalar2_Vector2_m_7_K_Matrix_cf ((x × c), ch2_list)
  | DScalar4_K_Matrix_ms (c, ch2_list) →
      DScalar4_K_Matrix_ms ((x × c), ch2_list)
  | Dim8_Scalar2_Vector2_1 c →
      Dim8_Scalar2_Vector2_1 (x × c)
  | Dim8_Scalar2_Vector2_2 c →
      Dim8_Scalar2_Vector2_1 (x × c)
  | Dim8_Scalar2_Vector2_m_0 c →
      Dim8_Scalar2_Vector2_m_0 (x × c)
  | Dim8_Scalar2_Vector2_m_1 c →
      Dim8_Scalar2_Vector2_m_1 (x × c)
  | Dim8_Scalar2_Vector2_m_7 c →
      Dim8_Scalar2_Vector2_m_7 (x × c)
  | Dim8_Scalar4 c →
      Dim8_Scalar4 (x × c)
  | Dim8_Vector4_t_0 ic4_list →
      Dim8_Vector4_t_0 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim8_Vector4_t_1 ic4_list →
      Dim8_Vector4_t_1 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim8_Vector4_t_2 ic4_list →
      Dim8_Vector4_t_2 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim8_Vector4_m_0 ic4_list →
      Dim8_Vector4_m_0 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim8_Vector4_m_1 ic4_list →
      Dim8_Vector4_m_1 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim8_Vector4_m_7 ic4_list →
      Dim8_Vector4_m_7 (List.map (fun (c, icl) → (x × c, icl)) ic4_list)
  | Dim6_H4_P2 c →
      Dim6_H4_P2 (x × c)
  | Dim6_AHWW_DPB c →
      Dim6_AHWW_DPB (x × c)
  | Dim6_AHWW_DPW c →
      Dim6_AHWW_DPW (x × c)
  | Dim6_AHWW_DW c →
      Dim6_AHWW_DW (x × c)
  | Dim6_Vector4_DW c →
      Dim6_Vector4_DW (x × c)
  | Dim6_Vector4_W c →
      Dim6_Vector4_W (x × c)
  | Dim6_Scalar2_Vector2_PB c →
      Dim6_Scalar2_Vector2_PB (x × c)
  | Dim6_Scalar2_Vector2_D c →
      Dim6_Scalar2_Vector2_D (x × c)
  | Dim6_Scalar2_Vector2_DP c →
      Dim6_Scalar2_Vector2_DP (x × c)
  | Dim6_HHZZ_T c →
      Dim6_HHZZ_T (x × c)
  | Dim6_HWWZ_DW c →
      Dim6_HWWZ_DW (x × c)
  | Dim6_HWWZ_DPB c →
      Dim6_HWWZ_DPB (x × c)
  | Dim6_HWWZ_DDPW c →
      Dim6_HWWZ_DDPW (x × c)
  | Dim6_HWWZ_DPW c →
      Dim6_HWWZ_DPW (x × c)
  | Dim6_AHHZ_D c →
      Dim6_AHHZ_D (x × c)
  | Dim6_AHHZ_DP c →
      Dim6_AHHZ_DP (x × c)
  | Dim6_AHHZ_PB c →
      Dim6_AHHZ_PB (x × c)
```

```
let cmult_vertex4 z v =
  match integer z with
  | None → invalid_arg "cmult_vertex4"
  | Some x → mult_vertex4 x v
```

```
let mult_vertexn x = function
  | _ → incomplete "mult_vertexn"
```

```
let cmult_vertexn z v =
  let open Coupling in
  match v with
  | UFO (c, v, s, fl, col) →
      UFO (QC.mul z c, v, s, fl, col)
```

```
let mult_vertex x v =
  let open Coupling in
  match v with
  | V3 (v, fuse, c) → V3 (mult_vertex3 x v, fuse, c)
  | V4 (v, fuse, c) → V4 (mult_vertex4 x v, fuse, c)
  | Vn (v, fuse, c) → Vn (mult_vertexn x v, fuse, c)
```

```
let cmult_vertex z v =
  let open Coupling in
  match v with
  | V3 (v, fuse, c) → V3 (cmult_vertex3 z v, fuse, c)
  | V4 (v, fuse, c) → V4 (cmult_vertex4 z v, fuse, c)
  | Vn (v, fuse, c) → Vn (cmult_vertexn z v, fuse, c)
```

13.2.2 Flavors Adorned with Colorflows

```
module Flavor (M : Model.T) =
  struct
```

```
    type cf_in = int
    type cf_out = int
```

The legacy types *CF_in*, etc, are not orthogonal to *Color_Propagator.t*, unfortunately, but we will have to life with this for a while.

```
    module CP = Color_Propagator
```

```
    type t =
      | White of M.flavor
      | CF_in of M.flavor × cf_in
      | CF_out of M.flavor × cf_out
      | CF_io of M.flavor × cf_in × cf_out
      | CF_aux of M.flavor
      | CF of M.flavor × CP.t
    let flavor_sans_color = function
      | White f → f
      | CF_in (f, _) → f
      | CF_out (f, _) → f
      | CF_io (f, _, _) → f
      | CF_aux f → f
      | CF (f, _) → f

    let pullback f arg1 =
      f (flavor_sans_color arg1)
```

Since the alternatives in the sum type t are not orthogonal, we have make sure that we don't produce false negatives. In addition, non trivial color flows of type *Color_Propagator.t* need a special equality.

Converting everything to *CF* $(f,\ cp)$ first is the most concise, but not the most efficient approach. However, it's probably not worth the effort to cook up an optimized comparison before we retire the other alternatives in t.

```
    let to_cp = function
      | White f → (f, CP.white)
      | CF_in (f, cfi) → (f, CP.of_lists [cfi] [])
      | CF_out (f, cfo) → (f, CP.of_lists [] [cfo])
      | CF_io (f, cfi, cfo) → (f, CP.of_lists [cfi] [cfo])
      | CF_aux f → (f, CP.Ghost)
      | CF (f, cp) → (f, cp)

    let equal f1 f2 =
      let f1, cp1 = to_cp f1
      and f2, cp2 = to_cp f2 in
      f1 = f2 ∧ CP.equal cp1 cp2

  end
```

13.2.3 The Legacy Implementation

We have to keep this legacy implementation around, because it infers the color flows from the SU(3) representations of a particle in vertices with three and four legs (except for four triplets, where the connections are ambiguous). The new implementation is already used for UFO models exclusively, since they don't use *Coupling.V2* and *Coupling.V3* at all.

```
module Legacy_Implementation (M : Model.T) =
  struct

    module C = Color

    module Colored_Flavor = Flavor(M)
    open Colored_Flavor

    open Coupling

    let nc = M.nc
```

Auxiliary functions

Below, we will need to permute Lorentz structures. The following permutes the three possible contractions of four vectors. We permute the first three indices, as they correspond to the particles entering the fusion.

```
    type permutation4 =
      | P123 | P231 | P312
      | P213 | P321 | P132

    let permute_contract4 = function
      | P123 →
          begin function
            | C_12_34 → C_12_34
            | C_13_42 → C_13_42
            | C_14_23 → C_14_23
          end
      | P231 →
          begin function
            | C_12_34 → C_14_23
            | C_13_42 → C_12_34
            | C_14_23 → C_13_42
          end
      | P312 →
          begin function
            | C_12_34 → C_13_42
            | C_13_42 → C_14_23
            | C_14_23 → C_12_34
          end
      | P213 →
          begin function
            | C_12_34 → C_12_34
            | C_13_42 → C_14_23
            | C_14_23 → C_13_42
          end
      | P321 →
          begin function
            | C_12_34 → C_14_23
            | C_13_42 → C_13_42
            | C_14_23 → C_12_34
          end
      | P132 →
          begin function
            | C_12_34 → C_13_42
            | C_13_42 → C_12_34
            | C_14_23 → C_14_23
          end

    let permute_contract4_list perm ic4_list =
      List.map (fun (i, c4) → (i, permute_contract4 perm c4)) ic4_list

    let permute_vertex4' perm = function
      | Scalar4 c →
          Scalar4 c
      | Vector4 ic4_list →
          Vector4 (permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_jr (c, ic4_list) →
          Vector4_K_Matrix_jr (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_t0 (c, ic4_list) →
          Vector4_K_Matrix_cf_t0 (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_t1 (c, ic4_list) →
          Vector4_K_Matrix_cf_t1 (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_t2 (c, ic4_list) →
          Vector4_K_Matrix_cf_t2 (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_t_rsi (c, ic4_list) →
          Vector4_K_Matrix_cf_t_rsi (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_m0 (c, ic4_list) →
          Vector4_K_Matrix_cf_m0 (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_m1 (c, ic4_list) →
          Vector4_K_Matrix_cf_m1 (c, permute_contract4_list perm ic4_list)
      | Vector4_K_Matrix_cf_m7 (c, ic4_list) →
          Vector4_K_Matrix_cf_m7 (c, permute_contract4_list perm ic4_list)
      | DScalar2_Vector2_K_Matrix_ms (c, ic4_list) →
          DScalar2_Vector2_K_Matrix_ms (c, permute_contract4_list perm ic4_list)
      | DScalar2_Vector2_m_0_K_Matrix_cf (c, ic4_list) →
          DScalar2_Vector2_m_0_K_Matrix_cf (c, permute_contract4_list perm ic4_list)
      | DScalar2_Vector2_m_1_K_Matrix_cf (c, ic4_list) →
```

```
    DScalar2_Vector2_m_1_K_Matrix_cf (c, permute_contract4_list perm ic4_list)
  | DScalar2_Vector2_m_7_K_Matrix_cf (c, ic4_list) →
    DScalar2_Vector2_m_7_K_Matrix_cf (c, permute_contract4_list perm ic4_list)
  | DScalar4_K_Matrix_ms (c, ic4_list) →
    DScalar4_K_Matrix_ms (c, permute_contract4_list perm ic4_list)
  | Scalar2_Vector2 c →
    incomplete "permute_vertex4'␣Scalar2_Vector2"
  | DScalar4 ic4_list →
    incomplete "permute_vertex4'␣DScalar4"
  | DScalar2_Vector2 ic4_list →
    incomplete "permute_vertex4'␣DScalar2_Vector2"
  | GBBG (c, fb, b2, f) →
    incomplete "permute_vertex4'␣GBBG"
  | Vector4_K_Matrix_tho (c, ch2_list) →
    incomplete "permute_vertex4'␣Vector4_K_Matrix_tho"
  | Dim8_Scalar2_Vector2_1 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar2_Vector2_1"
  | Dim8_Scalar2_Vector2_2 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar2_Vector2_2"
  | Dim8_Scalar2_Vector2_m_0 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar2_Vector2_m_0"
  | Dim8_Scalar2_Vector2_m_1 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar2_Vector2_m_1"
  | Dim8_Scalar2_Vector2_m_7 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar2_Vector2_m_7"
  | Dim8_Scalar4 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Scalar4"
  | Dim8_Vector4_t_0 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_t_0"
  | Dim8_Vector4_t_1 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_t_1"
  | Dim8_Vector4_t_2 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_t_2"
  | Dim8_Vector4_m_0 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_m_0"
  | Dim8_Vector4_m_1 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_m_1"
  | Dim8_Vector4_m_7 ic4_list →
    incomplete "permute_vertex4'␣Dim8_Vector4_m_7"
  | Dim6_H4_P2 ic4_list →
    incomplete "permute_vertex4'␣Dim6_H4_P2"
  | Dim6_AHWW_DPB ic4_list →
    incomplete "permute_vertex4'␣Dim6_AHWW_DPB"
  | Dim6_AHWW_DPW ic4_list →
    incomplete "permute_vertex4'␣Dim6_AHWW_DPW"
  | Dim6_AHWW_DW ic4_list →
    incomplete "permute_vertex4'␣Dim6_AHWW_DW"
  | Dim6_Vector4_DW ic4_list →
    incomplete "permute_vertex4'␣Dim6_Vector4_DW"
  | Dim6_Vector4_W ic4_list →
    incomplete "permute_vertex4'␣Dim6_Vector4_W"
  | Dim6_Scalar2_Vector2_D ic4_list →
    incomplete "permute_vertex4'␣Dim6_Scalar2_Vector2_D"
  | Dim6_Scalar2_Vector2_DP ic4_list →
    incomplete "permute_vertex4'␣Dim6_Scalar2_Vector2_DP"
  | Dim6_Scalar2_Vector2_PB ic4_list →
    incomplete "permute_vertex4'␣Dim6_Scalar2_Vector2_PB"
  | Dim6_HHZZ_T ic4_list →
    incomplete "permute_vertex4'␣Dim6_HHZZ_T"
  | Dim6_HWWZ_DW ic4_list →
    incomplete "permute_vertex4'␣Dim6_HWWZ_DW"
  | Dim6_HWWZ_DPB ic4_list →
      incomplete "permute_vertex4'␣Dim6_HWWZ_DPB"
  | Dim6_HWWZ_DDPW ic4_list →
      incomplete "permute_vertex4'␣Dim6_HWWZ_DDPW"
  | Dim6_HWWZ_DPW ic4_list →
      incomplete "permute_vertex4'␣Dim6_HWWZ_DPW"
  | Dim6_AHHZ_D ic4_list →
      incomplete "permute_vertex4'␣Dim6_AHHZ_D"
  | Dim6_AHHZ_DP ic4_list →
      incomplete "permute_vertex4'␣Dim6_AHHZ_DP"
  | Dim6_AHHZ_PB ic4_list →
      incomplete "permute_vertex4'␣Dim6_AHHZ_PB"

let permute_vertex4 perm = function
  | V3 (v, fuse, c) → V3 (v, fuse, c)
  | V4 (v, fuse, c) → V4 (permute_vertex4' perm v, fuse, c)
  | Vn (v, fuse, c) → Vn (v, fuse, c)
```

Cubic Vertices

The following pattern matches could eventually become quite long. The O'Caml compiler will (hopefully) optimize them aggressively (http://pauillac.inria.fr/~maranget/papers/opat/).

```
let colorize_fusion2 f1 f2 (f, v) =
  match M.color f with

  | C.Singlet →
      begin match f1, f2 with

      | White _, White _ →
          [White f, v]

      | CF_in (_, c1), CF_out (_, c2')
      | CF_out (_, c1), CF_in (_, c2') →
          if c1 = c2' then
            [White f, v]
          else
            []

      | CF_io (f1, c1, c1'), CF_io (f2, c2, c2') →
          if c1 = c2' ∧ c2 = c1' then
            [White f, v]
          else
            []

      | CF_aux f1, CF_aux f2 →
          [White f, mult_vertex (− (nc ())) v]

      | CF_aux _, CF_io _ | CF_io _, CF_aux _ →
          []

      | (CF_in _ | CF_out _ | CF_io _ | CF_aux _), White _
      | White _, (CF_in _ | CF_out _ | CF_io _ | CF_aux _)
      | (CF_io _ | CF_aux _), (CF_in _ | CF_out _)
      | (CF_in _ | CF_out _), (CF_io _ | CF_aux _)
      | CF_in _, CF_in _ | CF_out _, CF_out _ →
          colored_vertex "colorize_fusion2"

      | CF (_, c), _ | _, CF (_, c) → non_legacy_color "colorize_fusion2" c
      end

  | C.SUN nc1 →
      begin match f1, f2 with

      | CF_in (_, c1), (White _ | CF_aux _)
      | (White _ | CF_aux _), CF_in (_, c1) →
```

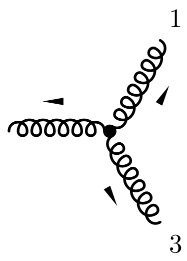
```
            if nc1 > 0 then
              [CF_in (f, c1), v]
            else
              colored_vertex "colorize_fusion2"
        | CF_out (_, c1'), (White _ | CF_aux _)
        | (White _ | CF_aux _), CF_out (_, c1') →
            if nc1 < 0 then
              [CF_out (f, c1'), v]
            else
              colored_vertex "colorize_fusion2"
        | CF_in (_, c1), CF_io (_, c2, c2')
        | CF_io (_, c2, c2'), CF_in (_, c1) →
            if nc1 > 0 then begin
              if c1 = c2' then
                [CF_in (f, c2), v]
              else
                []
            end else
              colored_vertex "colorize_fusion2"
        | CF_out (_, c1'), CF_io (_, c2, c2')
        | CF_io (_, c2, c2'), CF_out (_, c1') →
            if nc1 < 0 then begin
              if c1' = c2 then
                [CF_out (f, c2'), v]
              else
                []
            end else
              colored_vertex "colorize_fusion2"
        | CF_in _, CF_in _ →
            if nc1 > 0 then
              baryonic_vertex "colorize_fusion2"
            else
              colored_vertex "colorize_fusion2"
        | CF_out _, CF_out _ →
            if nc1 < 0 then
              baryonic_vertex "colorize_fusion2"
            else
              colored_vertex "colorize_fusion2"
        | CF_in _, CF_out _ | CF_out _, CF_in _
        | (White _ | CF_io _ | CF_aux _),
            (White _ | CF_io _ | CF_aux _) →
          colored_vertex "colorize_fusion2"
        | CF (_, c), _ | _, CF (_, c) → non_legacy_color "colorize_fusion2" c
      end
    | C.AdjSUN _ →
      begin match f1, f2 with
        | White _, CF_io (_, c1, c2') | CF_io (_, c1, c2'), White _ →
            [CF_io (f, c1, c2'), v]
```

Note that for $\mathrm{tr}(F_{mu\nu}F^{\mu\nu})$ couplings, like the effective Hgg coupling, we can't inplement the rules derived in [17]. fusing *White* with *CF_aux* would have to produce a *CF_io*, but there is canonical source for a fresh color flow index! If the gluons are not connected via an inbroken string of such couplings to an external line, we can use the considerations in (12.2) to replace the factor N_C by $-N_C$. In order to account for the gluons that are connected via an inbroken string of such couplings to an external line, we apply a correction factor $1 - 2/N_C^2$ for each gluon loop in the very end.

```
        | White _, CF_aux _ | CF_aux _, White _ →
            [CF_aux f, mult_vertex (− (nc ())) v]
```

```
| CF_in (_, c1), CF_out (_, c2')
| CF_out (_, c2'), CF_in (_, c1) →
    if c1 ≠ c2' then
      [CF_io (f, c1, c2'), v]
    else
      [CF_aux f, v]
```

In the adjoint representation



$$= g f_{a_1 a_2 a_3} C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) \tag{13.1a}$$

with

$$C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) = \\ (g^{\mu_1\mu_2}(k_1^{\mu_3}-k_2^{\mu_3}) + g^{\mu_2\mu_3}(k_2^{\mu_1}-k_3^{\mu_1}) + g^{\mu_3\mu_1}(k_3^{\mu_2}-k_1^{\mu_2})) \tag{13.1b}$$

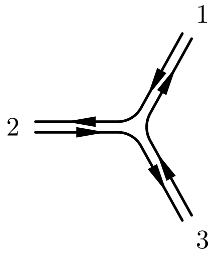
while in the color flow basis find from

$$\mathrm{i} f_{a_1 a_2 a_3} = \mathrm{tr}\,(T_{a_1}\,[T_{a_2},T_{a_3}]) = \mathrm{tr}\,(T_{a_1}T_{a_2}T_{a_3}) - \mathrm{tr}\,(T_{a_1}T_{a_3}T_{a_2}) \tag{13.2}$$

the decomposition

$$\mathrm{i} f_{a_1 a_2 a_3} T_{a_1}^{i_1 j_1} T_{a_2}^{i_2 j_2} T_{a_3}^{i_3 j_3} = \delta^{i_1 j_2}\delta^{i_2 j_3}\delta^{i_3 j_1} - \delta^{i_1 j_3}\delta^{i_3 j_2}\delta^{i_2 j_1}\,. \tag{13.3}$$

The resulting Feynman rule is



$$= \mathrm{i}g\left(\delta^{i_1 j_3}\delta^{i_2 j_1}\delta^{i_3 j_2} - \delta^{i_1 j_2}\delta^{i_2 j_3}\delta^{i_3 j_1}\right) C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) \tag{13.4}$$

We have to generalize this for cases of three particles in the adjoint that are not all gluons (gluinos, scalar octets):

- scalar-scalar-scalar
- scalar-scalar-vector
- scalar-vector-vector
- scalar-fermion-fermion
- vector-fermion-fermion

We could use a better understanding of the signs for the gaugino-gaugino-gaugeboson couplings!!!

```
| CF_io (f1, c1, c1'), CF_io (f2, c2, c2') →
    let phase =
      begin match v with
      | V3 (Gauge_Gauge_Gauge _, _, _)
      | V3 (I_Gauge_Gauge_Gauge _, _, _)
      | V3 (Aux_Gauge_Gauge _, _, _) → of_int 1
      | V3 (FBF (_, _, _, _), fuse2, _) →
          begin match fuse2 with
          | F12 → of_int 1 (* works, needs underpinning *)
          | F21 → of_int (−1) (* dto. *)
          | F31 → of_int 1 (* dto. *)
          | F32 → of_int (−1) (* transposition of F12 *)
          | F23 → of_int 1 (* transposition of F21 *)
```

```
                  | F13 → of_int (−1) (* transposition of F12 *)
                  end
                | V3 _ → incomplete "colorize_fusion2␣(V3␣_)"
                | V4 _ → impossible "colorize_fusion2␣(V4␣_)"
                | Vn _ → impossible "colorize_fusion2␣(Vn␣_)"
                end in
              if c1′ = c2 then
                [CF_io (f, c1, c2′), cmult_vertex (QC.neg phase) v]
              else if c2′ = c1 then
                [CF_io (f, c2, c1′), cmult_vertex ( phase) v]
              else
                []

          | CF_aux _ , CF_io _
          | CF_io _ , CF_aux _
          | CF_aux _ , CF_aux _ →
              []

          | White _, White _
          | (White _ | CF_io _ | CF_aux _), (CF_in _ | CF_out _)
          | (CF_in _ | CF_out _), (White _ | CF_io _ | CF_aux _)
          | CF_in _, CF_in _ | CF_out _, CF_out _ →
              colored_vertex "colorize_fusion2"

          | CF (_, c), _ | _, CF (_, c) → non_legacy_color "colorize_fusion2" c
          end

      | C.YT _ | C.YTC _ → young_tableaux "colorize_fusion2"
```

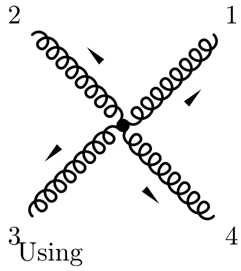
Quartic Vertices

```
  let colorize_fusion3 f1 f2 f3 (f, v) =
    match M.color f with

    | C.Singlet →
        begin match f1, f2, f3 with

        | White _, White _, White _ →
            [White f, v]

        | (White _ | CF_aux _), CF_in (_, c1), CF_out (_, c2′)
        | (White _ | CF_aux _), CF_out (_, c1), CF_in (_, c2′)
        | CF_in (_, c1), (White _ | CF_aux _), CF_out (_, c2′)
        | CF_out (_, c1), (White _ | CF_aux _), CF_in (_, c2′)
        | CF_in (_, c1), CF_out (_, c2′), (White _ | CF_aux _)
        | CF_out (_, c1), CF_in (_, c2′), (White _ | CF_aux _) →
            if c1 = c2′ then
              [White f, v]
            else
              []

        | White _, CF_io (_, c1, c1′), CF_io (_, c2, c2′)
        | CF_io (_, c1, c1′), White _, CF_io (_, c2, c2′)
        | CF_io (_, c1, c1′), CF_io (_, c2, c2′), White _ →
            if c1 = c2′ ∧ c2 = c1′ then
              [White f, v]
            else
              []

        | White _, CF_aux _, CF_aux _
        | CF_aux _, White _, CF_aux _
        | CF_aux _, CF_aux _, White _ →
            [White f, mult_vertex (− (nc ())) v]

        | White _, CF_io _, CF_aux _
```

```
| White _, CF_aux _, CF_io _
| CF_io _, White _, CF_aux _
| CF_aux _, White _, CF_io _
| CF_io _, CF_aux _, White _
| CF_aux _, CF_io _, White _ →
    []
| CF_io (_, c1, c1'), CF_in (_, c2), CF_out (_, c3')
| CF_io (_, c1, c1'), CF_out (_, c3'), CF_in (_, c2)
| CF_in (_, c2), CF_io (_, c1, c1'), CF_out (_, c3')
| CF_out (_, c3'), CF_io (_, c1, c1'), CF_in (_, c2)
| CF_in (_, c2), CF_out (_, c3'), CF_io (_, c1, c1')
| CF_out (_, c3'), CF_in (_, c2), CF_io (_, c1, c1') →
    if c1 = c3' ∧ c1' = c2 then
      [White f, v]
    else
      []
| CF_io (_, c1, c1'), CF_io (_, c2, c2'), CF_io (_, c3, c3') →
    if c1' = c2 ∧ c2' = c3 ∧ c3' = c1 then
      [White f, mult_vertex (−1) v]
    else if c1' = c3 ∧ c2' = c1 ∧ c3' = c2 then
      [White f, mult_vertex ( 1) v]
    else
      []
| CF_io _, CF_io _, CF_aux _
| CF_io _, CF_aux _, CF_io _
| CF_aux _, CF_io _, CF_io _
| CF_io _, CF_aux _, CF_aux _
| CF_aux _, CF_io _, CF_aux _
| CF_aux _, CF_aux _, CF_io _
| CF_aux _, CF_aux _, CF_aux _ →
    []
| CF_in _, CF_in _, CF_in _
| CF_out _, CF_out _, CF_out _ →
    baryonic_vertex "colorize_fusion3"
| CF_in _, CF_in _, CF_out _
| CF_in _, CF_out _, CF_in _
| CF_out _, CF_in _, CF_in _
| CF_in _, CF_out _, CF_out _
| CF_out _, CF_in _, CF_out _
| CF_out _, CF_out _, CF_in _
| White _, White _, (CF_io _ | CF_aux _)
| White _, (CF_io _ | CF_aux _), White _
| (CF_io _ | CF_aux _), White _, White _
| (White _ | CF_io _ | CF_aux _), CF_in _, CF_in _
| CF_in _, (White _ | CF_io _ | CF_aux _), CF_in _
| CF_in _, CF_in _, (White _ | CF_io _ | CF_aux _)
| (White _ | CF_io _ | CF_aux _), CF_out _, CF_out _
| CF_out _, (White _ | CF_io _ | CF_aux _), CF_out _
| CF_out _, CF_out _, (White _ | CF_io _ | CF_aux _)
| (CF_in _ | CF_out _),
    (White _ | CF_io _ | CF_aux _),
    (White _ | CF_io _ | CF_aux _)
| (White _ | CF_io _ | CF_aux _),
    (CF_in _ | CF_out _),
    (White _ | CF_io _ | CF_aux _)
| (White _ | CF_io _ | CF_aux _),
    (White _ | CF_io _ | CF_aux _),
          (CF_in _ | CF_out _) →
            colored_vertex "colorize_fusion3"
        | CF (_, c), _, _ | _, CF (_, c), _ | _, _, CF (_, c) →
          non_legacy_color "colorize_fusion3" c
        end
    | C.SUN nc1 →
        begin match f1, f2, f3 with
        | CF_in (_, c1), CF_io (_, c2, c2'), CF_io (_, c3, c3')
        | CF_io (_, c2, c2'), CF_in (_, c1), CF_io (_, c3, c3')
        | CF_io (_, c2, c2'), CF_io (_, c3, c3'), CF_in (_, c1) →
            if nc1 > 0 then
              if c1 = c2' ∧ c2 = c3' then
                [CF_in (f, c3), v]
              else if c1 = c3' ∧ c3 = c2' then
                [CF_in (f, c2), v]
              else
                []
            else
              colored_vertex "colorize_fusion3"
        | CF_out (_, c1'), CF_io (_, c2, c2'), CF_io (_, c3, c3')
        | CF_io (_, c2, c2'), CF_out (_, c1'), CF_io (_, c3, c3')
        | CF_io (_, c2, c2'), CF_io (_, c3, c3'), CF_out (_, c1') →
            if nc1 < 0 then
              if c1' = c2 ∧ c2' = c3 then
                [CF_out (f, c3'), v]
              else if c1' = c3 ∧ c3' = c2 then
                [CF_out (f, c2'), v]
              else
                []
            else
              colored_vertex "colorize_fusion3"
        | CF_aux _, CF_in (_, c1), CF_io (_, c2, c2')
        | CF_aux _, CF_io (_, c2, c2'), CF_in (_, c1)
        | CF_in (_, c1), CF_aux _, CF_io (_, c2, c2')
        | CF_io (_, c2, c2'), CF_aux _, CF_in (_, c1)
        | CF_in (_, c1), CF_io (_, c2, c2'), CF_aux _
        | CF_io (_, c2, c2'), CF_in (_, c1), CF_aux _ →
            if nc1 > 0 then
              if c1 = c2' then
                [CF_in (f, c2), mult_vertex ( 2) v]
              else
                []
            else
              colored_vertex "colorize_fusion3"
        | CF_aux _, CF_out (_, c1'), CF_io (_, c2, c2')
        | CF_aux _, CF_io (_, c2, c2'), CF_out (_, c1')
        | CF_out (_, c1'), CF_aux _, CF_io (_, c2, c2')
        | CF_io (_, c2, c2'), CF_aux _, CF_out (_, c1')
        | CF_out (_, c1'), CF_io (_, c2, c2'), CF_aux _
        | CF_io (_, c2, c2'), CF_out (_, c1'), CF_aux _ →
            if nc1 < 0 then
              if c1' = c2 then
                [CF_out (f, c2'), mult_vertex ( 2) v]
              else
                []
            else
              colored_vertex "colorize_fusion3"
        | White _, CF_in (_, c1), CF_io (_, c2, c2')
```

```
| White _, CF_io (_, c2, c2'), CF_in (_, c1)
| CF_in (_, c1), White _, CF_io (_, c2, c2')
| CF_io (_, c2, c2'), White _, CF_in (_, c1)
| CF_in (_, c1), CF_io (_, c2, c2'), White _
| CF_io (_, c2, c2'), CF_in (_, c1), White _ →
    if nc1 > 0 then
      if c1 = c2' then
        [CF_in (f, c2), v]
      else
        []
    else
      colored_vertex "colorize_fusion3"

| White _, CF_out (_, c1'), CF_io (_, c2, c2')
| White _, CF_io (_, c2, c2'), CF_out (_, c1')
| CF_out (_, c1'), White _, CF_io (_, c2, c2')
| CF_io (_, c2, c2'), White _, CF_out (_, c1')
| CF_out (_, c1'), CF_io (_, c2, c2'), White _
| CF_io (_, c2, c2'), CF_out (_, c1'), White _ →
    if nc1 < 0 then
      if c2 = c1' then
        [CF_out (f, c2'), v]
      else
        []
    else
      colored_vertex "colorize_fusion3"

| CF_in (_, c1), CF_aux _, CF_aux _
| CF_aux _, CF_in (_, c1), CF_aux _
| CF_aux _, CF_aux _, CF_in (_, c1) →
    if nc1 > 0 then
      [CF_in (f, c1), mult_vertex ( 2) v]
    else
      colored_vertex "colorize_fusion3"

| CF_in (_, c1), CF_aux _, White _
| CF_in (_, c1), White _, CF_aux _
| CF_in (_, c1), White _, White _
| CF_aux _, CF_in (_, c1), White _
| White _, CF_in (_, c1), CF_aux _
| White _, CF_in (_, c1), White _
| CF_aux _, White _, CF_in (_, c1)
| White _, CF_aux _, CF_in (_, c1)
| White _, White _, CF_in (_, c1) →
    if nc1 > 0 then
      [CF_in (f, c1), v]
    else
      colored_vertex "colorize_fusion3"

| CF_out (_, c1'), CF_aux _, CF_aux _
| CF_aux _, CF_out (_, c1'), CF_aux _
| CF_aux _, CF_aux _, CF_out (_, c1') →
    if nc1 < 0 then
      [CF_out (f, c1'), mult_vertex ( 2) v]
    else
      colored_vertex "colorize_fusion3"

| CF_out (_, c1'), CF_aux _, White _
| CF_out (_, c1'), White _, CF_aux _
| CF_out (_, c1'), White _, White _
| CF_aux _, CF_out (_, c1'), White _
| White _, CF_out (_, c1'), CF_aux _
| White _, CF_out (_, c1'), White _
| CF_aux _, White _, CF_out (_, c1')
```

```
      | White _, CF_aux _, CF_out (_, c1')
      | White _, White _, CF_out (_, c1') →
          if nc1 < 0 then
            [CF_out (f, c1'), v]
          else
            colored_vertex "colorize_fusion3"

      | CF_in _, CF_in _, CF_out _
      | CF_in _, CF_out _, CF_in _
      | CF_out _, CF_in _, CF_in _ →
          if nc1 > 0 then
            color_flow_ambiguous "colorize_fusion3"
          else
            colored_vertex "colorize_fusion3"

      | CF_in _, CF_out _, CF_out _
      | CF_out _, CF_in _, CF_out _
      | CF_out _, CF_out _, CF_in _ →
          if nc1 < 0 then
            color_flow_ambiguous "colorize_fusion3"
          else
            colored_vertex "colorize_fusion3"

      | CF_in _, CF_in _, CF_in _
      | CF_out _, CF_out _, CF_out _

      | (White _ | CF_io _ | CF_aux _),
        (White _ | CF_io _ | CF_aux _),
        (White _ | CF_io _ | CF_aux _)

      | (CF_in _ | CF_out _),
          (CF_in _ | CF_out _),
          (White _ | CF_io _ | CF_aux _)
      | (CF_in _ | CF_out _),
          (White _ | CF_io _ | CF_aux _),
          (CF_in _ | CF_out _)
      | (White _ | CF_io _ | CF_aux _),
          (CF_in _ | CF_out _),
          (CF_in _ | CF_out _) →
          colored_vertex "colorize_fusion3"

      | CF (_, c), _, _ | _, CF (_, c), _ | _, _, CF (_, c) →
          non_legacy_color "colorize_fusion3" c
      end

  | C.AdjSUN nc →
      begin match f1, f2, f3 with

      | CF_in (_, c1), CF_out (_, c1'), White _
      | CF_out (_, c1'), CF_in (_, c1), White _
      | CF_in (_, c1), White _, CF_out (_, c1')
      | CF_out (_, c1'), White _, CF_in (_, c1)
      | White _, CF_in (_, c1), CF_out (_, c1')
      | White _, CF_out (_, c1'), CF_in (_, c1) →
          if c1 ≠ c1' then
            [CF_io (f, c1, c1'), v]
          else
            [CF_aux f, v]

      | CF_in (_, c1), CF_out (_, c1'), CF_aux _
      | CF_out (_, c1'), CF_in (_, c1), CF_aux _
      | CF_in (_, c1), CF_aux _, CF_out (_, c1')
      | CF_out (_, c1'), CF_aux _, CF_in (_, c1)
      | CF_aux _, CF_in (_, c1), CF_out (_, c1')
      | CF_aux _, CF_out (_, c1'), CF_in (_, c1) →
          if c1 ≠ c1' then
            [$CF\_io\ (f,\ c1,\ c1'),\ mult\_vertex\ (2)\ v$]
          else
            [$CF\_aux\ f,\ mult\_vertex\ (2)\ v$]
      | $CF\_in\ (\_,\ c1),\ CF\_out\ (\_,\ c1'),\ CF\_io\ (\_,\ c2,\ c2')$
      | $CF\_out\ (\_,\ c1'),\ CF\_in\ (\_,\ c1),\ CF\_io\ (\_,\ c2,\ c2')$
      | $CF\_in\ (\_,\ c1),\ CF\_io\ (\_,\ c2,\ c2'),\ CF\_out\ (\_,\ c1')$
      | $CF\_out\ (\_,\ c1'),\ CF\_io\ (\_,\ c2,\ c2'),\ CF\_in\ (\_,\ c1)$
      | $CF\_io\ (\_,\ c2,\ c2'),\ CF\_in\ (\_,\ c1),\ CF\_out\ (\_,\ c1')$
      | $CF\_io\ (\_,\ c2,\ c2'),\ CF\_out\ (\_,\ c1'),\ CF\_in\ (\_,\ c1)\ \rightarrow$
          if $c1 = c2' \wedge c2 = c1'$ then
            [$CF\_aux\ f,\ mult\_vertex\ (2)\ v$]
          else if $c1 = c2'$ then
            [$CF\_io\ (f,\ c2,\ c1'),\ v$]
          else if $c2 = c1'$ then
            [$CF\_io\ (f,\ c1,\ c2'),\ v$]
          else
            []
```

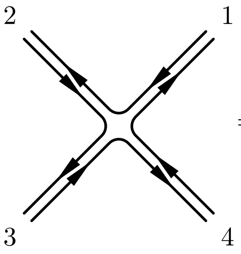


$$= \begin{array}{l} -\mathrm{i}g^2 f_{a_1a_2b}f_{a_3a_4b}(g_{\mu_1\mu_3}g_{\mu_4\mu_2} - g_{\mu_1\mu_4}g_{\mu_2\mu_3}) \\ -\mathrm{i}g^2 f_{a_1a_3b}f_{a_4a_2b}(g_{\mu_1\mu_4}g_{\mu_2\mu_3} - g_{\mu_1\mu_2}g_{\mu_3\mu_4}) \\ -\mathrm{i}g^2 f_{a_1a_4b}f_{a_2a_3b}(g_{\mu_1\mu_2}g_{\mu_3\mu_4} - g_{\mu_1\mu_3}g_{\mu_4\mu_2}) \end{array}$$

Using

$$\mathcal{P}_4 = \{\{1,2,3,4\},\{1,3,4,2\},\{1,4,2,3\},\{1,2,4,3\},\{1,4,3,2\},\{1,3,2,4\}\} \tag{13.6}$$

as the set of permutations of $\{1,2,3,4\}$ with the cyclic permutations factored out, we have:



$$= \begin{array}{l} \mathrm{i}g^2 \displaystyle\sum_{\{\alpha_k\}_{k=1,2,3,4}\in\mathcal{P}_4} \delta^{i_{\alpha_1}j_{\alpha_2}}\delta^{i_{\alpha_2}j_{\alpha_3}}\delta^{i_{\alpha_3}j_{\alpha_4}}\delta^{i_{\alpha_4}j_{\alpha_1}} \\ \left(2g_{\mu_{\alpha_1}\mu_{\alpha_3}}g_{\mu_{\alpha_4}\mu_{\alpha_2}} - g_{\mu_{\alpha_1}\mu_{\alpha_4}}g_{\mu_{\alpha_2}\mu_{\alpha_3}} - g_{\mu_{\alpha_1}\mu_{\alpha_2}}g_{\mu_{\alpha_3}\mu_{\alpha_4}}\right) \end{array} \tag{13.7}$$

The different color connections correspond to permutations of the particles entering the fusion and have to be matched by a corresponding permutation of the Lorentz structure:

We have to generalize this for cases of four particles in the adjoint that are not all gluons:

- scalar-scalar-scalar-scalar
- scalar-scalar-vector-vector

and even ones including fermions (gluinos) if higher dimensional operators are involved.

```
  | $CF\_io\ (\_,\ c1,\ c1'),\ CF\_io\ (\_,\ c2,\ c2'),\ CF\_io\ (\_,\ c3,\ c3')\ \rightarrow$
      if $c1' = c2 \wedge c2' = c3$ then
        [$CF\_io\ (f,\ c1,\ c3'),\ permute\_vertex4\ P123\ v$]
      else if $c1' = c3 \wedge c3' = c2$ then
        [$CF\_io\ (f,\ c1,\ c2'),\ permute\_vertex4\ P132\ v$]
      else if $c2' = c3 \wedge c3' = c1$ then
        [$CF\_io\ (f,\ c2,\ c1'),\ permute\_vertex4\ P231\ v$]
      else if $c2' = c1 \wedge c1' = c3$ then
        [$CF\_io\ (f,\ c2,\ c3'),\ permute\_vertex4\ P213\ v$]
      else if $c3' = c1 \wedge c1' = c2$ then
        [$CF\_io\ (f,\ c3,\ c2'),\ permute\_vertex4\ P312\ v$]
      else if $c3' = c2 \wedge c2' = c1$ then
        [$CF\_io\ (f,\ c3,\ c1'),\ permute\_vertex4\ P321\ v$]
      else
        []
```

```
| CF_io _, CF_io _, CF_aux _
| CF_io _, CF_aux _, CF_io _
| CF_aux _, CF_io _, CF_io _
| CF_io _, CF_aux _, CF_aux _
| CF_aux _, CF_aux _, CF_io _
| CF_aux _, CF_io _, CF_aux _
| CF_aux _, CF_aux _, CF_aux _ →
    []

| CF_io (_, c1, c1'), CF_io (_, c2, c2'), White _
| CF_io (_, c1, c1'), White _, CF_io (_, c2, c2')
| White _, CF_io (_, c1, c1'), CF_io (_, c2, c2') →
    if c1' = c2 then
      [CF_io (f, c1, c2'), mult_vertex (−1) v]
    else if c2' = c1 then
      [CF_io (f, c2, c1'), mult_vertex ( 1) v]
    else
      []

| CF_io (_, c1, c1'), CF_aux _, White _
| CF_aux _, CF_io (_, c1, c1'), White _
| CF_io (_, c1, c1'), White _, CF_aux _
| CF_aux _, White _, CF_io (_, c1, c1')
| White _, CF_io (_, c1, c1'), CF_aux _
| White _, CF_aux _, CF_io (_, c1, c1') →
    []

| CF_aux _, CF_aux _, White _
| CF_aux _, White _, CF_aux _
| White _, CF_aux _, CF_aux _ →
    []

| White _, White _, CF_io (_, c1, c1')
| White _, CF_io (_, c1, c1'), White _
| CF_io (_, c1, c1'), White _, White _ →
    [CF_io (f, c1, c1'), v]

| White _, White _, CF_aux _
| White _, CF_aux _, White _
| CF_aux _, White _, White _ →
    []

| White _, White _, White _

| (White _ | CF_io _ | CF_aux _),
    (White _ | CF_io _ | CF_aux _),
    (CF_in _ | CF_out _)
| (White _ | CF_io _ | CF_aux _),
    (CF_in _ | CF_out _),
    (White _ | CF_io _ | CF_aux _)
| (CF_in _ | CF_out _),
    (White _ | CF_io _ | CF_aux _),
    (White _ | CF_io _ | CF_aux _)

| CF_in _, CF_in _, (White _ | CF_io _ | CF_aux _)
| CF_in _, (White _ | CF_io _ | CF_aux _), CF_in _
| (White _ | CF_io _ | CF_aux _), CF_in _, CF_in _

| CF_out _, CF_out _, (White _ | CF_io _ | CF_aux _)
| CF_out _, (White _ | CF_io _ | CF_aux _), CF_out _
| (White _ | CF_io _ | CF_aux _), CF_out _, CF_out _

| (CF_in _ | CF_out _),
    (CF_in _ | CF_out _),
    (CF_in _ | CF_out _) →
    colored_vertex "colorize_fusion3"
```

```
          | CF (_, c), _, _ | _, CF (_, c), _ | _, _, CF (_, c) →
              non_legacy_color "colorize_fusion3" c
          end
      | C.YT _ | C.YTC _ → young_tableaux "colorize_fusion3"
```

Quintic and Higher Vertices

```
    let is_white = function
      | White _ → true
      | _ → false

    let colorize_fusionn flist (f, v) =
      let incomplete_match () =
        incomplete
          ("colorize_fusionn␣{␣" ^
              String.concat ",␣" (List.map (pullback M.flavor_to_string) flist) ^
            "␣}␣->␣" ^ M.flavor_to_string f) in
      match M.color f with
      | C.Singlet →
          if List.for_all is_white flist then
            [White f, v]
          else
            incomplete_match ()
      | C.SUN _ →
          if List.for_all is_white flist then
            colored_vertex "colorize_fusionn"
          else
            incomplete_match ()
      | C.AdjSUN _ →
          if List.for_all is_white flist then
            colored_vertex "colorize_fusionn"
          else
            incomplete_match ()
      | C.YT _ | C.YTC _ → young_tableaux "colorize_fusionn"

  end
```

13.2.4 Colorizing a Monochrome Model

```
module It (M : Model.T) =
  struct

    open Coupling

    module C = Color
    module CA = Arrow
    module CV = Color.Vertex

    module Colored_Flavor = Flavor(M)

    type flavor = Colored_Flavor.t
    type flavor_sans_color = M.flavor
    let flavor_sans_color = Colored_Flavor.flavor_sans_color

    type gauge = M.gauge
    type constant = M.constant
    let options = M.options
    let caveats = M.caveats

    type coupling_order = M.coupling_order
    let all_coupling_orders = M.all_coupling_orders
    let coupling_orders = M.coupling_orders
    let coupling_order_to_string = M.coupling_order_to_string
```

```
open Colored_Flavor

let flavor_equal = Colored_Flavor.equal

let color = pullback M.color
let nc = M.nc
let pdg = pullback M.pdg
let lorentz = pullback M.lorentz

module Ch = M.Ch
let charges = pullback M.charges
```

For the propagator we cannot use pullback because we have to add the case of the color singlet propagator by hand.

```
let cf_aux_propagator = function
  | Prop_Scalar → Prop_Col_Scalar (* Spin 0 octets. *)
  | Prop_Majorana → Prop_Col_Majorana (* Spin 1/2 octets. *)
  | Prop_Feynman → Prop_Col_Feynman (* Spin 1 states, massless. *)
  | Prop_Unitarity → Prop_Col_Unitarity (* Spin 1 states, massive. *)
  | Aux_Scalar → Aux_Col_Scalar (* constant colored scalar propagator *)
  | Aux_Vector → Aux_Col_Vector (* constant colored vector propagator *)
  | Aux_Tensor_1 → Aux_Col_Tensor_1 (* constant colored tensor propagator *)
  | Prop_Col_Scalar | Prop_Col_Feynman
  | Prop_Col_Majorana | Prop_Col_Unitarity
  | Aux_Col_Scalar | Aux_Col_Vector | Aux_Col_Tensor_1
    → failwith ("Colorize.It().colorize_propagator:␣already␣colored␣particle!")
  | _ → failwith ("Colorize.It().colorize_propagator:␣impossible!")

let propagator = function
  | CF_aux f → cf_aux_propagator (M.propagator f)
  | White f → M.propagator f
  | CF_in (f, _) → M.propagator f
  | CF_out (f, _) → M.propagator f
  | CF_io (f, _, _) → M.propagator f
  | CF (f, c) →
     begin match c with
     | CP.Flow _ | CP.Flow_with_Epsilons _ | CP.Flow_with_Epsilon_Bars _ →
        M.propagator f
     | CP.Ghost | CP.Ghost_with_Epsilons _ | CP.Ghost_with_Epsilon_Bars _ →
        cf_aux_propagator (M.propagator f)
     end

let width = pullback M.width

let goldstone = function
  | White f →
      begin match M.goldstone f with
      | None → None
      | Some (f′, g) → Some (White f′, g)
      end
  | CF_in (f, c) →
      begin match M.goldstone f with
      | None → None
      | Some (f′, g) → Some (CF_in (f′, c), g)
      end
  | CF_out (f, c) →
      begin match M.goldstone f with
      | None → None
      | Some (f′, g) → Some (CF_out (f′, c), g)
      end
  | CF_io (f, c1, c2) →
      begin match M.goldstone f with
      | None → None
      | Some (f′, g) → Some (CF_io (f′, c1, c2), g)
```

```
        end
  | CF_aux f →
        begin match M.goldstone f with
        | None → None
        | Some (f′, g) → Some (CF_aux f′, g)
        end
  | CF (f, c) →
        begin match M.goldstone f with
        | None → None
        | Some (f′, g) → Some (CF (f′, c), g)
        end

let conjugate = function
  | White f → White (M.conjugate f)
  | CF_in (f, c) → CF_out (M.conjugate f, c)
  | CF_out (f, c) → CF_in (M.conjugate f, c)
  | CF_io (f, c1, c2) → CF_io (M.conjugate f, c2, c1)
  | CF_aux f → CF_aux (M.conjugate f)
  | CF (f, c) → CF (M.conjugate f, CP.conjugate c)

let conjugate_sans_color = M.conjugate

let fermion = pullback M.fermion

let max_degree = M.max_degree

let flavors () =
  invalid "flavors"

let external_flavors () =
  invalid "external_flavors"

let parameters = M.parameters

let split_color_string s =
  try
    let i1 = String.index s '/' in
    let i2 = String.index_from s (succ i1) '/' in
    let sf = String.sub s 0 i1
    and sc1 = String.sub s (succ i1) (i2 − i1 − 1)
    and sc2 = String.sub s (succ i2) (String.length s − i2 − 1) in
    (sf, sc1, sc2)
  with
  | Not_found → (s, "", "")

let flavor_of_string s =
  try
    let sf, sc1, sc2 = split_color_string s in
    let f = M.flavor_of_string sf in
    match M.color f with
    | C.Singlet → White f
    | C.SUN nc →
        if nc > 0 then
          CF_in (f, color_flow_of_string sc1)
        else
          CF_out (f, color_flow_of_string sc2)
    | C.AdjSUN _ →
        begin match sc1, sc2 with
        | "", "" → CF_aux f
        | _, _ → CF_io (f, color_flow_of_string sc1, color_flow_of_string sc2)
        end
    | C.YT _ | C.YTC _ →
        incomplete "flavor_of_string:␣Young␣tableaux"
  with
  | Failure s →
      if s = "int_of_string" then
```

```
          invalid_arg "Colorize().flavor_of_string: expecting integer"
        else
          failwith ("Colorize().flavor_of_string: unexpected Failure(" ^ s ^ ")")

  let flavor_to_string = function
    | White f ->
        M.flavor_to_string f
    | CF_in (f, c) ->
        M.flavor_to_string f ^ "/" ^ string_of_int c ^ "/"
    | CF_out (f, c) ->
        M.flavor_to_string f ^ "//" ^ string_of_int c
    | CF_io (f, c1, c2) ->
        M.flavor_to_string f ^ "/" ^ string_of_int c1 ^ "/" ^ string_of_int c2
    | CF_aux f ->
       M.flavor_to_string f ^ "//"
    | CF (f, cp) ->
       M.flavor_to_string f ^ "/" ^ CP.to_string cp
```

CP.to_string need to be replaced!

```
  let flavor_to_TeX = function
    | White f ->
        M.flavor_to_TeX f
    | CF_in (f, c) ->
        "{" ^ M.flavor_to_TeX f ^ "}_{\\mathstrut " ^ string_of_int c ^ "}"
    | CF_out (f, c) ->
        "{" ^ M.flavor_to_TeX f ^ "}_{\\mathstrut\\overline{" ^
        string_of_int c ^ "}}"
    | CF_io (f, c1, c2) ->
        "{" ^ M.flavor_to_TeX f ^ "}_{\\mathstrut " ^
        string_of_int c1 ^ "\\overline{" ^ string_of_int c2 ^ "}}"
    | CF_aux f ->
        "{" ^ M.flavor_to_TeX f ^ "}_{\\mathstrut 0}"
    | CF (f, cp) ->
        "{" ^ M.flavor_to_TeX f ^ "}_{\\mathstrut " ^ CP.to_string cp ^ "}"

  let flavor_symbol = function
    | White f ->
       "f" ^ M.flavor_symbol f
    | CF_in (f, c) ->
       "f" ^ M.flavor_symbol f ^ "_i" ^ string_of_int c
    | CF_out (f, c) ->
       "f" ^ M.flavor_symbol f ^ "_o" ^ string_of_int c
    | CF_io (f, c1, c2) ->
       "f" ^ M.flavor_symbol f ^ "_i" ^ string_of_int c1 ^ "o" ^ string_of_int c2
    | CF_aux f ->
       "f" ^ M.flavor_symbol f ^ "_g"
    | CF (f, cp) ->
       "f" ^ M.flavor_symbol f ^ "_" ^ CP.to_symbol cp

  let gauge_symbol = M.gauge_symbol
```

Masses and widths must not depend on the colors anyway!

```
  let mass_symbol = pullback M.mass_symbol
  let width_symbol = pullback M.width_symbol

  let constant_symbol = M.constant_symbol
```

Vertices

vertices are *only* used by functor applications and for indexing a cache of precomputed fusion rules, which is not used for colorized models.

```
let vertices () =
  invalid "vertices"

module Legacy = Legacy_Implementation (M)

let colorize_fusion2 f1 f2 (f, v) =
  match v with
  | V3 _ → Legacy.colorize_fusion2 f1 f2 (f, v)
  | _ → []

let colorize_fusion3 f1 f2 f3 (f, v) =
  match v with
  | V4 _ → Legacy.colorize_fusion3 f1 f2 f3 (f, v)
  | _ → []
```

In order to match the *correct* positions of the fields in the vertices, we have to undo the permutation effected by the fusion according to *Coupling.fusen*.

```
module PosMap =
  Partial.Make (struct type t = int let compare = compare end)
```

Note that due to the *inverse*, the list l' can be interpreted here as a map reshuffling the indices. E. g., *inverse* (*Permutation.Default.list* [2;0;1]) applied to [1;2;3] gives [3;1;2].

```
let partial_map_redoing_permutation l l' =
  let module P = Permutation.Default in
  let p = P.inverse (P.of_list (List.map pred l')) in
  PosMap.of_lists l (P.list p l)
```

Note that, the list l' can not be interpreted as a map reshuffling the indices, but gives the new order of the argument. E. g., *Permutation.Default.list* [2;0;1] applied to [1;2;3] gives [2;3;1].

```
let partial_map_undoing_permutation l l' =
  let module P = Permutation.Default in
  let p = P.of_list (List.map pred l') in
  PosMap.of_lists l (P.list p l)

let color_sans_flavor = function
  | White _ → CP.white
  | CF_in (_, cfi) → CP.of_lists [cfi] []
  | CF_out (_, cfo) → CP.of_lists [] [cfo]
  | CF_io (_, cfi, cfo) → CP.of_lists [cfi] [cfo]
  | CF_aux _ → CP.Ghost
  | CF (f, cp) → cp
```

Should we continue to translate the flows back and forth?

```
let color_with_flavor f = function
  | CP.Flow (cfis, cfos) as cp →
     begin match PArray.to_option_list cfis, PArray.to_option_list cfos with
     | [], [] → White f
     | [Some cfi], [] → CF_in (f, cfi)
     | [], [Some cfo] → CF_out (f, cfo)
     | [Some cfi], [Some cfo] → CF_io (f, cfi, cfo)
     | _, _ → CF (f, cp)
     end
  | CP.Flow_with_Epsilons (_, _) →
     failwith "Colorize.color_with_flavor:␣unexpected␣epsilon"
  | CP.Flow_with_Epsilon_Bars (_, _) →
     failwith "Colorize.color_with_flavor:␣unexpected␣epsilon␣bar"
  | CP.Ghost → CF_aux f
  | CP.Ghost_with_Epsilons _ →
     failwith "Colorize.color_with_flavor:␣unexpected␣epsilon"
  | CP.Ghost_with_Epsilon_Bars _ →
     failwith "Colorize.color_with_flavor:␣unexpected␣epsilon␣bar"
```

```
let colorize vertex_list flavors f v =
  List.map
    (fun (coef, cf) → (color_with_flavor f cf, cmult_vertex coef v))
    (Color_Fusion.fuse (nc ()) vertex_list (List.map color_sans_flavor flavors))

let partial_map_undoing_fusen fusen =
  partial_map_undoing_permutation
    (ThoList.range 1 (List.length fusen))
    fusen

let undo_permutation_of_fusen fusen =
  PosMap.apply_with_fallback
    (fun _ → invalid_arg "permutation_of_fusen")
    (partial_map_undoing_fusen fusen)

let colorize_fusionn_ufo flist f c v spins flines color fuse xtra =
  let v = Vn (UFO (c, v, spins, flines, Birdtracks.one), fuse, xtra) in
  let p = undo_permutation_of_fusen fuse in
  colorize (Birdtracks.relocate p color) flist f v

let colorize_fusionn flist (f, v) =
  match v with
  | Vn (UFO (c, v, spins, flines, color), fuse, xtra) →
      colorize_fusionn_ufo flist f c v spins flines color fuse xtra
  | _ → []

let fuse_list flist =
  ThoList.flatmap
    (colorize_fusionn flist)
    (M.fuse (List.map flavor_sans_color flist))

let fuse2 f1 f2 =
  List.rev_append
    (fuse_list [f1; f2])
    (ThoList.flatmap
        (colorize_fusion2 f1 f2)
        (M.fuse2
            (flavor_sans_color f1)
            (flavor_sans_color f2)))

let fuse3 f1 f2 f3 =
  List.rev_append
    (fuse_list [f1; f2; f3])
    (ThoList.flatmap
        (colorize_fusion3 f1 f2 f3)
        (M.fuse3
            (flavor_sans_color f1)
            (flavor_sans_color f2)
            (flavor_sans_color f3)))

let fuse = function
  | [] | [_] → invalid_arg "Colorize.It().fuse"
  | [f1; f2] → fuse2 f1 f2
  | [f1; f2; f3] → fuse3 f1 f2 f3
  | flist → fuse_list flist

let max_degree = M.max_degree
```

Adding Color to External Particles

Count the color strings in *f_list*: one incoming each quark and gluon, one outgoing for each antiquark and gluon. Keep track of the number of gluons separately.
Count the number of color lines for a given combination of flavors, assuming that the incoming lines have been crossed. Returns a pair $(n_{\text{in}}, n_{\text{out}})$, corresponding to the number of incoming and outgoing lines respectively. Note that the two lines of gluons are included in n_{in} and n_{out}.

```
let count_color_strings f_list =
  let rec count_color_strings' n_in n_out = function
    | f :: rest →
        begin match M.color f with
        | C.Singlet → count_color_strings' n_in n_out rest
        | C.SUN nc →
            if nc > 0 then
              count_color_strings' (succ n_in) n_out rest
            else if nc < 0 then
              count_color_strings' n_in (succ n_out) rest
            else
              su0 "count_color_strings"
        | C.AdjSUN _ →
            count_color_strings' (succ n_in) (succ n_out) rest
        | C.YT y →
            count_color_strings' (Young.num_cells_tableau y + n_in) n_out rest
        | C.YTC y →
            count_color_strings' n_in (Young.num_cells_tableau y + n_out) rest
        end
    | [] → (n_in, n_out)
  in
  count_color_strings' 0 0 f_list
```

Return a list of all permutations of outgoing color lines.

Currently, this assumes that there are an equal number of incoming and outgoing lines. This has to change, since we want to support ϵ- and $\bar\epsilon$-couplings that act as sources and sinks of lines.

For efficiency, we could check whether the model contains ϵ- or $\bar\epsilon$-couplings and produce only conserved color lines if not.

We can do even better if we add an optional parameter that contains the number of ϵ- and $\bar\epsilon$-couplings appearing in the amplitude. This can be computed from the still uncolorized *DAG.t* by the calling function.

If there are an equal number of incoming and outgoing color strings, generate all permutations, e. g. for $n = 2$ we get $([1,2],[1,2]);([1,2],[2,1])$.

```
let external_color_flows f_list =
  let n_in, n_out = count_color_strings f_list in
  if n_in ≠ n_out then
    []
  else
    let color_strings = ThoList.range 1 n_in in
    List.rev_map
      (fun permutation → (color_strings, permutation))
      (Combinatorics.permute color_strings)
```

If there are only adjoints *and* there are no couplings of adjoints to singlets, we can ignore the U(1)-ghosts.

```
let pure_adjoints f_list =
  List.for_all (fun f → match M.color f with C.AdjSUN _ → true | _ → false) f_list

let two_adjoints_couple_to_singlets () =
  let vertices3, vertices4, verticesn = M.vertices () in
  List.exists (fun ((f1, f2, f3), _, _) →
    match M.color f1, M.color f2, M.color f3 with
    | C.AdjSUN _, C.AdjSUN _, C.Singlet
    | C.AdjSUN _, C.Singlet, C.AdjSUN _
    | C.Singlet, C.AdjSUN _, C.AdjSUN _ → true
    | _ → false) vertices3 ∨
  List.exists (fun ((f1, f2, f3, f4), _, _) →
    match M.color f1, M.color f2, M.color f3, M.color f4 with
    | C.AdjSUN _, C.AdjSUN _, C.Singlet, C.Singlet
    | C.AdjSUN _, C.Singlet, C.AdjSUN _, C.Singlet
    | C.Singlet, C.AdjSUN _, C.AdjSUN _, C.Singlet
    | C.AdjSUN _, C.Singlet, C.Singlet, C.AdjSUN _
    | C.Singlet, C.AdjSUN _, C.Singlet, C.AdjSUN _
    | C.Singlet, C.Singlet, C.AdjSUN _, C.AdjSUN _ → true
    | _ → false) vertices4 ∨
  List.exists (fun (flist, _, g) → true) verticesn
```

colorize_crossed_amplitude_opt ghosts flavors (*cfi*, *cfo*) attempts to join the *flavors* with the external color flow (*cfi*, *cfo*). Includes U(1) ghosts iff *ghosts* is true (i. e. iff there are *only* external gluons). Note that, despite the name, this only maps the external states and not yet the *DAG.t* describing the scattering amplitude. This will happen in *Fusion* (chapter 15).

```
let external_ghosts f_list =
  if pure_adjoints f_list then
    two_adjoints_couple_to_singlets ()
  else
    true

let snoc = function
  | [] → invalid_arg "Colorize.It().snoc:␣not␣enough␣color␣flow␣lines"
  | a :: alist → (a, alist)

let snoc_n n alist =
  try
    ThoList.splitn n alist
  with
  | Invalid_argument _ →
     invalid_arg "Colorize.It().snoc_n:␣not␣enough␣color␣flow␣lines"

let rec cca_opt ghosts acc f_list (ecf_in, ecf_out) =
  match f_list, ecf_in, ecf_out with
  | [], [], [] → Some (List.rev acc)
  | [], _, _ →
      invalid_arg "Colorize.It().colorize_crossed_amplitude_opt:␣leftover␣color␣flow␣lines"
  | f :: rest, _, _ →
      begin match M.color f with
      | C.Singlet → cca_opt ghosts (White f :: acc) rest (ecf_in, ecf_out)
      | C.SUN nc →
          if nc > 0 then
            let cfi, ecf_in = snoc ecf_in in
            cca_opt ghosts (CF_in (f, cfi) :: acc) rest (ecf_in, ecf_out)
          else if nc < 0 then
            let cfo, ecf_out = snoc ecf_out in
            cca_opt ghosts (CF_out (f, cfo) :: acc) rest (ecf_in, ecf_out)
          else
            su0 "colorize_flavor"
      | C.AdjSUN _ →
          let cfi, ecf_in = snoc ecf_in
          and cfo, ecf_out = snoc ecf_out in
          if cfi = cfo then begin
            if ghosts then
              cca_opt ghosts (CF_aux f :: acc) rest (ecf_in, ecf_out)
            else
              None
          end else
            cca_opt ghosts (CF_io (f, cfi, cfo) :: acc) rest (ecf_in, ecf_out)
      | C.YT y →
         let cfi, ecf_in = snoc_n (Young.num_cells_tableau y) ecf_in in
         cca_opt ghosts (CF (f, CP.of_lists cfi []) :: acc) rest (ecf_in, ecf_out)
      | C.YTC y →
         let cfo, ecf_out = snoc_n (Young.num_cells_tableau y) ecf_out in
         cca_opt ghosts (CF (f, CP.of_lists [] cfo) :: acc) rest (ecf_in, ecf_out)
      end

let colorize_crossed_amplitude_opt ghosts f_list (ecf_in, ecf_out) =
  cca_opt ghosts [ ] f_list (ecf_in, ecf_out)

let colorize_crossed_amplitude f_list =
  let ghosts = external_ghosts f_list in
  List.fold_left
    (fun ca_list ecf →
      match colorize_crossed_amplitude_opt ghosts f_list ecf with
      | None → ca_list
      | Some ca → ca :: ca_list)
    [ ] (external_color_flows f_list)

let colorize_crossed_amplitude_logging f_list =
  let amplitudes = colorize_crossed_amplitude f_list in
  List.iter (fun a → Printf.eprintf "%s\n" (ThoList.to_string flavor_to_string a)) amplitudes;
  amplitudes

let cross_uncolored p_in p_out =
  (List.map M.conjugate p_in) @ p_out

let uncross_colored n_in p_lists_colorized =
  let p_in_out_colorized = List.map (ThoList.splitn n_in) p_lists_colorized in
  List.map
    (fun (p_in_colored, p_out_colored) →
      (List.map conjugate p_in_colored, p_out_colored))
    p_in_out_colorized

let amplitude p_in p_out =
  uncross_colored
    (List.length p_in)
    (colorize_crossed_amplitude (cross_uncolored p_in p_out))
```

The −-sign in the second component is redundant, but a Whizard convention.

The case *CF* (*f*, *cp*) needs to be handled properly!

```
    let indices = function
      | White _ → Color.Flow.of_list [0; 0]
      | CF_in (_, c) → Color.Flow.of_list [c; 0]
      | CF_out (_, c) → Color.Flow.of_list [0; − c]
      | CF_io (_, c1, c2) → Color.Flow.of_list [c1; − c2]
      | CF_aux _ → Color.Flow.ghost ()
      | CF (f, cp) →
          Printf.eprintf
            "Colorize.indices:␣color␣flow␣‘%s’␣not␣handled␣yet\n"
            (CP.to_string cp);
          Color.Flow.of_list [−42; − 42]

    let flow p_in p_out =
      (List.map indices p_in, List.map indices p_out)

  end
```

13.2.5 Colorizing a Monochrome Gauge Model

```
module Gauge (M : Model.Gauge) =
  struct

    module CM = It(M)

    type flavor = CM.flavor
    type flavor_sans_color = CM.flavor_sans_color
    type gauge = CM.gauge
    type constant = CM.constant
    type coupling_order = CM.coupling_order
    module Ch = CM.Ch
    let all_coupling_orders = CM.all_coupling_orders
    let coupling_orders = CM.coupling_orders
    let coupling_order_to_string = CM.coupling_order_to_string
    let charges = CM.charges
    let flavor_sans_color = CM.flavor_sans_color
    let flavor_equal = CM.flavor_equal
    let color = CM.color
    let pdg = CM.pdg
    let lorentz = CM.lorentz
    let propagator = CM.propagator
    let width = CM.width
    let conjugate = CM.conjugate
    let conjugate_sans_color = CM.conjugate_sans_color
    let fermion = CM.fermion
    let max_degree = CM.max_degree
    let vertices = CM.vertices
    let fuse2 = CM.fuse2
    let fuse3 = CM.fuse3
    let fuse = CM.fuse
    let flavors = CM.flavors
    let nc = CM.nc
    let external_flavors = CM.external_flavors
    let goldstone = CM.goldstone
    let parameters = CM.parameters
    let flavor_of_string = CM.flavor_of_string
    let flavor_to_string = CM.flavor_to_string
    let flavor_to_TeX = CM.flavor_to_TeX
    let flavor_symbol = CM.flavor_symbol
    let gauge_symbol = CM.gauge_symbol
    let mass_symbol = CM.mass_symbol
    let width_symbol = CM.width_symbol
    let constant_symbol = CM.constant_symbol
    let options = CM.options
    let caveats = CM.caveats

    let incomplete s =
      failwith ("Colorize.Gauge()." ^ s ^ "␣not␣done␣yet!")

    type matter_field = M.matter_field
    type gauge_boson = M.gauge_boson
    type other = M.other

    type field =
      | Matter of matter_field
      | Gauge of gauge_boson
      | Other of other

    let field f =
      incomplete "field"

    let matter_field f =
      incomplete "matter_field"

    let gauge_boson f =
      incomplete "gauge_boson"

    let other f =
      incomplete "other"

    let amplitude = CM.amplitude

    let flow = CM.flow

  end
```

—14— Count Coupling Constants

14.1 Interface of Orders

14.1.1 Conditions

The function *of_strings* parses a small domain specific language. The list of strings can be understood as multiple command line options or as lines in a file:

- except for newlines, white space is *not* significant.
- newlines are only significant as terminator for comments that start with a `"#"`.
- *coupling_order*s are represented as unquoted strings, taken from the codomain of the model's *coupling_order_to_string* function. Strings outside of the codomain trigger a non-terminal error message and are ignored.
- sets of *coupling_order*s are written as comma separated lists, enclosed in matching braces, e. .g. `"{QED,QCD}"`.
- the braces are optional for single element sets, i. e. `"QED"` and `"{QED}"` are equivalent.
- the empty set is represented by `"{}"`.
- `"~"` denotes the set complement with respect to the model's *all_coupling_orders* (). In particular, `"~{}"` denotes *all_coupling_orders* () and `"~{QED,QCD}"` all coupling orders except `"QED"` and `"QCD"`.
- set difference is denoted by \, i. e. `"{QED,QCD}␣\␣QCD"` is just `"QED"` and `"~{}␣\␣QED"` is a complicated way to write `"~QED"`. NB: as long as there are no variables for sets, the set difference is probably only useful as syntactic sugar for very few cases. Typical applications can be expressed as set complements. Set union and intersection would be trivial, but appear to be even less useful.
- ranges of orders come as
  - slices `"{2..3}"` and
  - intervals `"[2..3]"`.

  In the case of slices, code for amplitudes at all orders in the range is generated, while in the case of intervals, code for the sum of these is generated. If there is only one order in the range, the notations `"{3..3}"` or `"{3}"` and `"[3..3]"` or `"[3]"` produce equivalent physics, of course, but the interface code for the generated amplitudes are slightly different of course. In the case of a slice `"{3..3}"`, the order 3 will be exposed, while it will not be visible in the case of an interval `"[3..3]"`. The abbreviation by a single integer, `"3"`, behaves exactly as the slice `"{3..3}"` or `"{3}"`. If the systematic expansion is performed in the squared matrix element, slices are more useful than intervals.
- ranges can be limited on one side or on both sides: in the former case, `"[..3]"` is equivalent to `"[0..3]"`, while `"[0..]"` is equivalent to no limit at all.
- ranges for sets of coupling constants are set with an equal sign, as in `"{QED,QCD}␣=␣{2..4}"`. Note that the range `"0"` need not be spelled out: `"~{QCD}"` is equivalent to `"~{QCD}␣=␣0"` and switches off all couplings with a positive QCD coupling order.
- specifications can be combined by a logical AND `"&&"` or logical OR `"||"` both operators associate to the left and parentheses `"("` and `")"` can be used for grouping (the support for logical OR is limited, but might be extended in the future to fill a gap in *Cascade*).

- combining conditions by a semicolon `";"` or as separate strings corresponds to a logical AND. For example, the following
  - *of_strings* [`"QED␣=␣{..4};␣QCD␣=␣{..2}"`]
  - *of_strings* [`"QED␣=␣{..4}␣&&␣QCD␣=␣{..2}"`]
  - *of_strings* [`"QED␣=␣{..4}"`; `"QCD␣=␣{..2}"`]

  are equivalent ways to select upto and including second order in QCD and fourth order in QED
- a logical AND translates to set intersection for coupling orders, e. g. `"QCD␣=␣{2,4};␣QCD␣=␣{3,5}"` is equivalent to `"QCD␣=␣{3,4}"`. In the case of mixed types, the result will be a slice, if at least one of the sets is a slice.
- a natural consequence is that an empty intersection corresponds to switching off the coupling order completely e. g. `"QCD␣=␣2;␣QCD␣=␣4"` is equivalent to `"QCD"` or `"QCD=0"`
- for convenience, there is one exception to this rule: in a logical AND, if one set is `"{0}"`, it is ignored and the result is the other set, e. g. `"~{};␣QCD␣=␣3"` is equivalent to the more verbose `"~{QCD};␣QCD␣=␣3"`.
- since logical AND associates to the left, the above rules imply that `"QCD␣=␣2;␣QCD␣=␣4;␣QCD␣=␣6"` is equivalent to `"QCD␣=␣0;␣QCD␣=␣6"` and finally to `"QCD␣=␣6"`.

The powers of all the coupling orders that are neither set to zero nor summed over will be encoded into the variable names for the shell wave functions. If there are to many of these, we will run into the target language's limits on variable names. In models like typical SMEFT implementations, that define many different coupling orders, one can not ask for `"~{QED,QCD}␣=␣[..1]"` in order get all first order new physics contributions. The list of all new physics coupling orders is just too long. Instead one needs to select a specfic coupling or a small set like in `"~{QED,QCD};␣NP␣=␣[..1]"`

```
module type Conditions =
  sig
```

This is the same as *coupling_order* from *Model.T*.

```
    type coupling_order
```

Orders is just an abbreviation to make the interface more readable.

```
    type orders = (coupling_order × int) list
```

This type collects the conditions on the orders of coupling constants and will be used by the functions below to select coupling constants, fusions and brakets.

```
    type t
```

Keep all orders and sum them.

```
    val trivial : t
```

Parse a list of strings as described above.

```
    val of_strings : string list → t
```

Return a human readable textual representation that can be inserted into the output source code for documentation.

```
    val to_strings : t → string list
```

The following three predicates test whether coupling orders

- have been switched off completely (*constant*)
- still can be added to (*fusion*)
- satisfy the overall condition (*braket*).

constant condition (*M.coupling_orders c*) checks that none of the *coupling_order*s of the coupling constant *c* is non-zero and switched off in *condition* at the same time. If not, the corresponding fusion or braket can be discarded immediately.

NB: this can be used very early, before colorization or even during the model definition to avoid constructing pieces that will eventually be discarded anyways.

val *constant* : *t* → *orders* → *bool*

Check that none of the *coupling_order*s exceeds the limits. They can be below the lower bounds, since additional fusions might add more powers.

val *fusion* : *t* → *orders* → *bool*

Check that all of the *coupling_order*s are inside the limits. Return only the *coupling_order*s corresponding to slices. This performs the sum over intervals implicitely.

val *braket* : *t* → *orders* → *orders option*

The list of coupling orders that is neither set to zero nor summed over without constraints.

val *exclusive_fusion* : *t* → *coupling_order list*

The list of coupling orders with fixed powers.

val *exclusive_braket* : *t* → *coupling_order list*

Compute the coupling order conditions on the scattering amplitude that allow to compute the squared amplitude to the given order. Note that intervals must be converted to slices, to be able to compute the interferences. For example

$$|\mathcal{M}_{\text{SM}} + \lambda\mathcal{M}_{\text{BSM}}|^2 = \mathcal{M}^*_{\text{SM}}\mathcal{M}_{\text{SM}} + \lambda\mathcal{M}^*_{\text{SM}}\mathcal{M}_{\text{BSM}} + \lambda\mathcal{M}^*_{\text{BSM}}\mathcal{M}_{\text{SM}} + \mathcal{O}(\lambda^2) \tag{14.1}$$

For the general case, we arrange n coupling orders $\{c_k\}_{k=1,\ldots,n}$ in a sequence

$$c = (c_1, c_2, \ldots, c_n)\,, \tag{14.2}$$

so that we can introduce a multi index notation for the powers

$$i = (i_1, i_2, \ldots, i_n) \tag{14.3}$$

and write

$$c^i = \prod_{k=1}^{n} c_k^{i_k}\,. \tag{14.4}$$

The matrix element is then

$$\mathcal{M}_\chi = \sum_i \chi(i) c^i \mathcal{M}_i\,, \tag{14.5}$$

where the function $\chi : \mathbf{N}_0^n \to \{0,1\}$ encodes the conditions on the coupling orders. For the squared matrix element with the condition $\chi_2 : \mathbf{N}_0^n \to \{0,1\}$ we must find all $\mathcal{M}_i$ that contribute to the sum

$$|\mathcal{M}|^2_{\chi_2} = \sum_{i,j} \chi_2(i+j) c^{i+j} \mathcal{M}^*_i \mathcal{M}_j\,. \tag{14.6}$$

This means, that we need to find a function χ such that

$$\forall i,j \in \mathrm{N}_0^n : \chi_2(i+j) = 1 \Rightarrow \chi(i) = \chi(j) = 1\,. \tag{14.7}$$

There are infinitely many of such χ, of course, and we want the function that is non-zero for the smallest possible subset of N_0^n.
If χ_2 is non-zero for only one $\hat{\imath}$, it is straightforward to construct a corresponding set $I = \{i\}$ for which χ doesn't vanish as a cartesian product

$$I = \times_{k=1}^{n}\{0, 1, \ldots \hat{\imath}_k\}\,. \tag{14.8}$$

If there is a larger set of i for which $\chi_2(i) = 1$, we can form the union by selecting the maximum order for each coupling order independently. This can be implemented easily by replacing each slice and interval by the slice running from 0 to the upper limit.
Infortunately, this will in general *not* be the smallest such set for a given amplitude, because not all coupling order combinations can contribute. Therefore, only *after* constructing the sliced amplitude, we can find all matching pairs.

In addition, we should provide the Fortran code with the combinations of coupling orders to be multiplied an summed.

```
    val square_root : t → t
```

Return a compact textual representation that can be parsed again by *of_strings*. This is useful for testing and debugging.

```
    val to_string : t → string
    val pp : Format.formatter → t → unit
  end
```

A projection of *Model.T* containing only coupling constants and coupling orders. This is useful for testing without having to link real models.

```
module type Model_CO =
  sig
    type constant
    type coupling_order
    val all_coupling_orders : unit → coupling_order list
    val coupling_order_to_string : coupling_order → string
    val coupling_orders : constant → (coupling_order × int) list
  end

module Conditions (M : Model_CO (* ⊂ Model.T *)) : Conditions
       with type coupling_order = M.coupling_order
```

14.1.2 Slicing

The idea is to slice a *DAG.t* representing an amplitude into pieces that correspond to given orders in a set of coupling constants. This allows to assign a fixed order to all brakets and to write the corresponding amplitude.

The mapping from one amplitude to many amplitudes is analogous to colorization and can be implemented as such.

There is a certain co-product vibe to this, but I don't know if it is useful to investigate the analogy further. First get a working prototype.

It is not obvious whether it is more efficient to

1. slice first, colorize later
2. colorize first, slice later

In the first case, we have to slice a smaller *DAG.t*, but subsequently colorize a more complicated *DAG.*. In the second case, we have to colorize a smaller *DAG.t*, but subsequently slice a more complicated *DAG.*. Probably, this varies from amplitude to amplitude and doesn't matter. For the moment we choose route of slicing the colorized *DAG.t*, because we don't have to touch the *Colorize.It*() functor.

```
module Slice (CM : Model.Colorized) : Model.Sliced_by_Orders
       with type flavor_all_orders = CM.flavor
         and type flavor_sans_color = CM.flavor_sans_color
         and type constant = CM.constant
         and type coupling_order = CM.coupling_order
         and type orders = (CM.coupling_order × int) list
```

14.1.3 Tests

```
module Test : sig val suite : OUnit.test end
```

14.2 Implementation of Orders

14.2.1 Conditions

```
module type Conditions =
  sig
    type coupling_order
    type orders = (coupling_order × int) list
    type t
    val trivial : t
    val of_strings : string list → t
    val to_strings : t → string list
    val constant : t → orders → bool
    val fusion : t → orders → bool
    val braket : t → orders → orders option
    val exclusive_fusion : t → coupling_order list
    val exclusive_braket : t → coupling_order list
    val square_root : t → t

    val to_string : t → string
    val pp : Format.formatter → t → unit
  end
```

A projection of *Model.T* containing only coupling constants and coupling orders. This is useful for testing without having to link real models.

```
module type Model_CO =
  sig
    type constant
    type coupling_order
    val all_coupling_orders : unit → coupling_order list
    val coupling_order_to_string : coupling_order → string
    val coupling_orders : constant → (coupling_order × int) list
  end

module Conditions (M : Model_CO (* ⊂ Model.T *)) : Conditions
       with type coupling_order = M.coupling_order =
  struct

    type coupling_order = M.coupling_order
    type orders = (coupling_order × int) list

    module CO = struct type t = coupling_order let compare = Stdlib.compare end
    module COSet = Set.Make(CO)
    module COMap = Map.Make(CO)
    module COSMap = Partial.Make(String)
```

Add a *unit* argument to support *Model.Mutable*:

```
    let co_set () =
      COSet.of_list (M.all_coupling_orders ())

    let co_map () =
      COSMap.of_list (List.map (fun co → (M.coupling_order_to_string co, co)) (M.all_coupling_orders ()))

    let co_set_of_strings pmap co_list =
      List.fold_left
        (fun acc s →
          match COSMap.apply_opt pmap s with
          | None →
              Printf.eprintf "omega:␣ignoring␣unknown␣coupling_order␣‘%s’!\n" s;
              acc
          | Some co →
              COSet.add co acc)
      COSet.empty co_list

    let complement = COSet.diff
```

All the integers are non negative. We don't need a *LE* constructor, because $i \le n$ is equivalent to $0 \le i \le n$ in this case. This saves us redundant match cases below.

```
    type range =
      | GE of int
      | IN of int × int
      | EQ of int
```

```
type mode = Slice | Sum
```

The lists of type *orders* must be very short to allow encoding of the counted coupling orders in Fortran variable names! That's why we keep the potentially much larger set of couplings that are set to zero separate.

One could think of supporting a union of non overlapping ranges, but this adds a lot of complexity for little practical value.

The correct semantics for *OR*-ing conditions on *different* coupling orders can not be implemented with the following data type. One would need a set or list of (*range* × *mode*) *COMap.t* for *orders*. It is not clear if this is worth the effort.

fusion is the union of *braket* and *only_fusion*. One of the three is therefore redundant, but we maintain all three for convenience. Similarly, *exclusive_braket* and *exclusive_fusion* are simply the result of applying *List.map fst* to *braket* and *fusion*. They are here just for convenience.

```
type t =
  { braket : (coupling_order × range) list;
    fusion : (coupling_order × range) list;
    only_fusion : (coupling_order × range) list;
    exclusive_braket : coupling_order list;
    exclusive_fusion : coupling_order list;
    is_null : COSet.t }

let trivial =
  { braket = [];
    fusion = [];
    only_fusion = [];
    exclusive_braket = [];
    exclusive_fusion = [];
    is_null = COSet.empty }

type t_intermediate =
  { orders_map : (range × mode) COMap.t;
    null_set : COSet.t }

let range_to_string l r = function
  | IN (i, j) → Printf.sprintf "%c%d..%d%c" l i j r
  | GE i → Printf.sprintf "%c%d..%c" l i r
  | EQ i → Printf.sprintf "%d" i

let interval_to_string = range_to_string '[' ']'
let slice_to_string = range_to_string '{' '}'

let co_and_interval_to_string (co, r) =
  M.coupling_order_to_string co ^ "␣=␣" ^ interval_to_string r

let co_and_slice_to_string (co, r) =
  M.coupling_order_to_string co ^ "␣=␣" ^ slice_to_string r

let to_string c =
  let is_null =
    match COSet.elements c.is_null with
    | [] → []
    | [co] → [M.coupling_order_to_string co ^ "␣=␣0"]
    | is_null → ["{" ^ String.concat ",␣" (List.map M.coupling_order_to_string is_null) ^ "}␣=␣0"]
  and intervals = List.map co_and_interval_to_string c.only_fusion
  and slices = List.map co_and_slice_to_string c.braket in
  String.concat ";␣" (is_null @ intervals @ slices)

let to_string_raw c =
  let is_null = String.concat ",␣" (List.map M.coupling_order_to_string (COSet.elements c.is_null))
  and braket = List.map co_and_slice_to_string c.braket
  and fusion = List.map co_and_interval_to_string c.fusion
  and only_fusion = List.map co_and_interval_to_string c.only_fusion in
  Printf.sprintf
    "is_null␣=␣{%s};␣braket␣=␣(%s);␣fusion␣=␣(%s);␣only_fusion␣=␣(%s)"
    is_null (String.concat ",␣" braket) (String.concat ",␣" fusion) (String.concat ",␣" only_fusion)
```

```
let to_strings c =
  let intervals = List.map co_and_interval_to_string c.only_fusion
  and slices = List.map co_and_slice_to_string c.braket in
  match COSet.elements c.is_null with
  | [] → List.concat [intervals; slices]
  | is_null →
      List.concat
        [intervals;
         slices;
         List.map
           (fun co_list →
              "disabled:␣" ^ String.concat ",␣" (List.map M.coupling_order_to_string co_list))
           (ThoList.chopn 5 is_null)]

let accept_all =
  { orders_map = COMap.empty;
    null_set = COSet.empty }

module S = Orders_syntax

let rec compile_set all_co pmap = function
  | S.Set co_list → co_set_of_strings pmap co_list
  | S.Diff (set, set′) → COSet.diff (compile_set all_co pmap set) (compile_set all_co pmap set′)
  | S.Complement (S.Complement set) → compile_set all_co pmap set
  | S.Complement set → complement all_co (compile_set all_co pmap set)

let compile_range = function
  | S.Range (i, j) →
      if i = j then
        EQ i
      else if i < j then
        IN (i, j)
      else
        EQ 0
  | S.Min i →
      GE (max i 0)
  | S.Max j →
      if j > 0 then
        IN (0, j)
      else
        EQ 0

let make_interval_or_slice mode all_co pmap co_set range =
  let co_set = compile_set all_co pmap co_set in
  let orders_map =
    COSet.fold (fun co map → COMap.add co (compile_range range, mode) map) co_set COMap.empty in
  { accept_all with orders_map }

let compile_atom all_co pmap = function
  | S.Null co_set | S.Exact (co_set, 0)
  | S.Interval (co_set, (S.Max 0 | S.Range (_, 0)))
  | S.Slices (co_set, (S.Max 0 | S.Range (_, 0))) →
      { accept_all with null_set = compile_set all_co pmap co_set }
  | S.Exact (co_set, n) →
      let co_set = compile_set all_co pmap co_set in
      let orders_map = COSet.fold (fun co map → COMap.add co (EQ n, Slice) map) co_set COMap.empty in
      { accept_all with orders_map }
  | S.Interval (co_set, range) →
      make_interval_or_slice Sum all_co pmap co_set range
  | S.Slices (co_set, range) →
      make_interval_or_slice Slice all_co pmap co_set range

let in_or_eq i j =
  if i = j then
    Some (EQ i)
  else if i ≤ j then
    Some (IN (i, j))
  else
    None

let and_range_opt r1 r2 =
  match r1, r2 with
  | GE i1, GE i2 →
      Some (GE (max i1 i2))
  | EQ i1, EQ i2 →
      if i1 = i2 then Some (EQ i1) else None
  | IN (i1, j1), IN (i2, j2) →
      in_or_eq (max i1 i2) (min j1 j2)
  | IN (i, j), GE k | GE k, IN (i, j) →
      in_or_eq (max i k) j
  | GE i, EQ j | EQ j, GE i →
      if i ≤ j then Some (EQ i) else None
  | IN (i, j), EQ k | EQ k, IN (i, j) →
      if i ≤ k ∧ k ≤ j then Some (EQ k) else None

let prefer_slice m1 m2 =
  match m1, m2 with
  | Sum, Sum → Sum
  | Slice, Sum | Sum, Slice | Slice, Slice → Slice

let and_range co (r1, m1) (r2, m2) =
  match and_range_opt r1 r2 with
  | None → None
  | Some r → Some (r, prefer_slice m1 m2)

let and_pair c1 c2 =
  { null_set = COSet.union c1.null_set c2.null_set;
    orders_map = COMap.union and_range c1.orders_map c2.orders_map }

let gap co =
  let co = M.coupling_order_to_string co in
  invalid_arg (Printf.sprintf "or_range:␣%s:␣ranges␣with␣gaps␣not␣supported!" co)

let or_range_opt co r1 r2 =
  match r1, r2 with
  | GE i1, GE i2 →
      Some (GE (max 0 (min i1 i2)))
  | EQ i1, EQ i2 →
      if i1 = i2 then
        Some (EQ i1)
      else if i1 = pred i2 then
        Some (IN (i1, i2))
      else if i1 = succ i2 then
        Some (IN (i2, i1))
      else
        gap co
  | IN (i1, j1), IN (i2, j2) →
      if i2 ≤ succ j1 then
        Some (IN (i1, j2))
      else if i1 ≤ succ j2 then
        Some (IN (i2, j1))
      else
        gap co
  | IN (i, j), GE k | GE k, IN (i, j) →
      if k ≤ succ j then Some (GE i) else gap co
  | GE i, EQ j | EQ j, GE i →
      if j ≥ pred j then Some (GE j) else gap co
  | IN (i, j), EQ k | EQ k, IN (i, j) →
      if i ≤ k ∧ k ≤ j then
```

```
          Some (IN (i, j))
        else if k = pred i then
          Some (IN (k, j))
        else if k = succ j then
          Some (IN (i, k))
        else
          gap co

let or_range co (r1, m1) (r2, m2) =
  match or_range_opt co r1 r2 with
  | None → None
  | Some r → Some (r, prefer_slice m1 m2)
```

This will be used with *COMap.merge* and fails if the coupling order *co* appears as key in only one of the maps.

```
let merge_or_range co r1 r2 =
  match r1, r2 with
  | None, None → None
  | Some r1, Some r2 → or_range co r1 r2
  | None, Some _ | Some _, None →
      let co = M.coupling_order_to_string co in
      invalid_arg (Printf.sprintf "or_range:␣%s:␣OR␣of␣different␣coupling_orders␣not␣supported!" co)

let or_pair c1 c2 =
  { null_set = COSet.inter c1.null_set c2.null_set;
    orders_map = COMap.merge merge_or_range c1.orders_map c2.orders_map }

let cleanup_condition c =
  let null_set =
    COMap.fold
      (fun co (r, _) set →
        match r with
        | EQ 0 | IN (_, 0) → COSet.add co set
        | _ → COSet.remove co set)
      c.orders_map c.null_set in
  let orders_map = COMap.filter (fun co _ → ¬ (COSet.mem co null_set)) c.orders_map in
  { null_set; orders_map }

let combine_conditions combine_pairs = function
  | [] → accept_all
  | c0 :: clist → cleanup_condition (List.fold_left combine_pairs c0 clist)

let compile expr =
  let all_co = co_set ()
  and pmap = co_map () in
  let rec compile' = function
    | S.Atom atom → compile_atom all_co pmap atom
    | S.And clist → combine_conditions and_pair (List.map compile' clist)
    | S.Or clist → combine_conditions or_pair (List.map compile' clist) in
  let c = cleanup_condition (compile' expr) in
  let braket_rev, fusion_rev, only_fusion_rev =
    COMap.fold
      (fun co (range, mode) (braket, fusion, only_fusion) →
        let co_range = (co, range) in
        match mode with
        | Slice → (co_range :: braket, co_range :: fusion, only_fusion)
        | Sum → (braket, co_range :: fusion, co_range :: only_fusion))
    c.orders_map ([], [], []) in
  { braket = List.rev braket_rev;
    fusion = List.rev fusion_rev;
    only_fusion = List.rev only_fusion_rev;
    exclusive_braket = List.rev_map fst braket_rev;
    exclusive_fusion = List.rev_map fst fusion_rev;
    is_null = c.null_set}
```

An empty list of ranges is interpreted as no constraint. This is used for brakets.

```
let in_range n = function
  | GE i → n ≥ i
  | IN (i, j) → n ≥ i ∧ n ≤ j
  | EQ i → n = i
```

In fusions, the coupling orders may still be below the final range.

```
let beneath_range n = function
  | IN (_, i) | EQ i → n ≤ i
  | GE _ → true
```

Test whether to include a vertex at all.

```
let test_condition range_tester is_null condition co_list =
  let rec test_condition' acc = function
    | [], [] → (* we're done *)
        Some (List.rev acc)
    | (co, r) :: rest, [] → (* conditions on some orders remain, add them with power 0 *)
        if range_tester 0 r then
          test_condition' ((co, 0) :: acc) (rest, [])
        else
          None
    | [], (co', n') :: rest' → (* no further conditions, check that the remaining couplings are allowed *)
        if n' > 0 ∧ COSet.mem co' is_null then
          None
        else
          test_condition' acc ([], rest')
    | ((co, r) :: rest as orders), ((co', n') :: rest' as orders') →
        if n' > 0 ∧ COSet.mem co' is_null then (* bail if the coupling is forbidden *)
          None
        else if co = co' then (* condition and coupling line up *)
          begin
            if range_tester n' r then
              test_condition' ((co', n') :: acc) (rest, rest')
            else
              None
          end
        else if co < co' then (* condition missing from the couplings *)
          begin
            if range_tester 0 r then
              test_condition' ((co, 0) :: acc) (rest, orders')
            else
              None
          end
        else (* coupling not in the conditions, skip it *)
          test_condition' acc (orders, rest') in
  test_condition' [] (condition, co_list)
```

Check that a the sum of coupling orders in a fusion does not exceed the limits.

```
let fusion condition co_list =
  match test_condition beneath_range condition.is_null condition.fusion co_list with
  | None → false
  | Some _ → true
```

Check both the intervals in *only_fusion* and the slices in *braket*, but return only the matches of the latter:

```
let braket condition co_list =
  match test_condition in_range condition.is_null condition.only_fusion co_list with
  | None → None
  | Some _ → test_condition in_range condition.is_null condition.braket co_list

let constant condition co_list =
  ¬ (List.exists (fun (co, n) → n > 0 ∧ COSet.mem co condition.is_null) co_list)
```

```
    let exclusive_fusion c = c.exclusive_fusion
    let exclusive_braket c = c.exclusive_braket
```

Turn all intervals into slices, since we need to sum products. Include *all* lower orders.

```
    let square_root_range = function
      | GE _ → GE 0
      | IN (_, j) | EQ j → IN (0, j)

    let square_root_ranges ranges =
      List.map (fun (co, range) → (co, square_root_range range)) ranges

    let square_root c =
      let fusion =
        square_root_ranges
          (List.sort
             (fun (co1, _) (co2, _) → Stdlib.compare co1 co2)
             (List.rev_append c.only_fusion c.braket))
      and exclusive_fusion =
        List.sort Stdlib.compare (List.rev_append c.exclusive_fusion c.exclusive_braket) in
      { fusion;
        braket = fusion;
        only_fusion = [];
        exclusive_fusion;
        exclusive_braket = exclusive_fusion;
        is_null = c.is_null }

    let parse_string s =
      Orders_parser.main Orders_lexer.token (Lexing.from_string s)

    let parse_strings slist =
      parse_string (String.concat "; " slist)

    let of_strings slist =
      compile (parse_strings slist)

    let pp fmt c =
      Format.fprintf fmt "%s" (to_string_raw c)

  end
```

14.2.2 Decorate Flavors with Coupling Constant Orders

```
module type Coupling_Orders =
  sig
    type coupling_order
```

The list is ordered wrt. *order* and there must be no duplicate entry. Note that we're using lists instead of *Map.S.t*, because we want to be able to use the polymorphic *compare* as long as possible. The lists are assumed to be short and we don't care about tail recursion.

Eventually, we want to make this type abstract!

```
    type orders = (coupling_order × int) list
```

Simple constructors.

```
    val null : orders
```

Sort the list and test it for duplicates.

```
    val of_list : (coupling_order × int) list → orders
    val to_list : orders → (coupling_order × int) list
```

Add the matching powers of the coupling orders. The coupling orders in both operands *must* be identical and the *must* appear in the same order. If the coupling orders would be known at compile time, we could implement this in a type safe way as tuples, but the coupling orders can be selected on the command line and in UFO models not even the set of possible coupling orders is known at compile time.

```
    val add : orders → orders → orders
```

Increment the powers of the coupling orders in the second operand by the powers of matching coupling orders in the first operand. Ignore the other coupling orders in the first operand. The coupling orders in the operands *must* be ordered according to the same ordering relation.

```
    val incr : orders → orders → orders
```

square_root condition orders_list returns a triple (*used*, *squares*, *interferences*) where *used* is a list of are all combinations of powers of coupling orders that appear at least once in *squares* or *interferences*. *squares* are the terms that satisfy *condition* when multiplied with themselves and the pairs in *interferences* satisfy *condition* when multiplied.

```
    val square_root : (orders → bool) →
                          orders list × orders list × (orders × orders) list
```

Debugging:

```
    val to_string : orders → string

  end

module Coupling_Orders (M : sig type coupling_order val coupling_order_to_string : coupling_order →
 string end) : Coupling_Orders
      with type coupling_order = M.coupling_order =
  struct

    type coupling_order = M.coupling_order
    type orders = (coupling_order × int) list

    let to_string ol =
      "{" ^ ThoList.to_string (fun (co, n) → M.coupling_order_to_string co ^ ":" ^ string_of_int n) ol ^ "}"

    let null = []

    let rec duplicates = function
      | [] | [_] → false
      | (o1, _) :: ((o2, _) :: _ as tail) →
          if o1 = o2 then
            true
          else
            duplicates tail

    let of_list o =
      let o = List.sort (fun (o1, _) (o2, _) → Stdlib.compare o1 o2) o in
      if duplicates o then
        invalid_arg "Orders.Flavor.of_list:␣duplicates"
      else
        o

    let to_list o = o
```

Here's a dedicated version, but …

```
    let rec add ol1 ol2 =
      match ol1, ol2 with
      | [], [] → []
      | [], tail | tail, [] → invalid_arg "Orders.Coupling_Orders.add:␣length␣mismatch"
      | (o1, n1) :: tail1, (o2, n2) :: tail2 →
          if o1 = o2 then
            (o1, n1 + n2) :: add tail1 tail2
          else
            invalid_arg
              (Printf.sprintf "Orders.Coupling_Orders.add:␣mismatch␣'%s'␣<>␣'%s'"
                 (M.coupling_order_to_string o1) (M.coupling_order_to_string o2))
```

Here's a tail recursive version. Once we can use a modern compiler with the tail-mod-cons optimization, we can go back to the first version.

```
    let add ol1 ol2 =
    let rec add' acc ol1 ol2 =
      match ol1, ol2 with
      | [], [] → List.rev acc
      | [], tail | tail, [] → invalid_arg "Orders.Coupling_Orders.add:␣length␣mismatch"
      | (o1, n1) :: tail1, (o2, n2) :: tail2 →
         if o1 = o2 then
           add' ((o1, n1 + n2) :: acc) tail1 tail2
         else
           invalid_arg
             (Printf.sprintf "Orders.Coupling_Orders.add:␣mismatch␣'%s'␣<>␣'%s'"
                (M.coupling_order_to_string o1) (M.coupling_order_to_string o2)) in
  add' [] ol1 ol2
```

This is very similar to *add*, but coupling orders that appear only in the first, but not the second argument are ignored.

```
let rec incr ol1 ol2 =
  match ol1, ol2 with
  | _, [] → (* we're done with the second argument, ignore the rest of the first *)
     []
  | [], tail → (* we're done with the first argument, keep the rest of the second *)
     tail
  | (o1, n1) :: tail1, (o2, n2 as on2) :: tail2 →
     if o1 = o2 then (* coupling orders match, add the powers *)
       (o1, n1 + n2) :: incr tail1 tail2
     else if o1 < o2 then (* o1 does not appear in the second argument, ignore it *)
       incr tail1 ol2
     else (* o2 does not appear in the first argument, keep it unchanged *)
       on2 :: incr ol1 tail2
```

Here's again a tail recursive version.

```
let incr ol1 ol2 =
  let rec incr' acc ol1 ol2 =
    match ol1, ol2 with
    | _, [] → (* we're done with the second argument, ignore the rest of the first *)
       List.rev acc
    | [], tail → (* we're done with the first argument, keep the rest of the second *)
       List.rev_append acc tail
    | (o1, n1) :: tail1, (o2, n2 as on2) :: tail2 →
       if o1 = o2 then (* coupling orders match, add the powers *)
         incr' ((o1, n1 + n2) :: acc) tail1 tail2
       else if o1 < o2 then (* o1 does not appear in the second argument, ignore it *)
         incr' acc tail1 ol2
       else (* o2 does not appear in the first argument, keep it unchanged *)
         incr' (on2 :: acc) ol1 tail2 in
  incr' [] ol1 ol2

let _add ol1 ol2 =
  let ol = add ol1 ol2 in
  Printf.eprintf "add␣%s␣%s␣->␣%s\n" (to_string ol1) (to_string ol2) (to_string ol);
  ol

let _incr ol1 ol2 =
  let ol = incr ol1 ol2 in
  Printf.eprintf "incr␣%s␣%s␣->␣%s\n" (to_string ol1) (to_string ol2) (to_string ol);
  ol
```

Resist the temptation to implement this as *List.fold_left add null olist*, because then *add* would need to accept orders of different lengths.

```
let sum = function
  | [] → null
  | o :: rest → List.fold_left add o rest
```

We use the polymorphic compare, because we don't need a particular ordering to test of equality in a *Set*.

```
    module OSet = Set.Make(struct type t = orders let compare = Stdlib.compare end)
```

Return the list of all pairs of elements of a list, where the first element appears before the second in the list. E. g. *ordered_pairs* [1; 2; 3] = [(1, 2); (1, 3); (2, 3)]
For longer lists for which the result will be passed to *List.fold*, an implementation of the corresponding *fold* would be more efficient, but the lists will always be short.

```
    let rec ordered_pairs = function
      | [] → []
      | a1 :: a2_list → List.map (fun a2 → (a1, a2)) a2_list @ ordered_pairs a2_list

    let square_root condition orders =
      let used = OSet.empty in
      let squares, used =
        List.fold_right
          (fun o (squares, used as acc) →
            if condition (add o o) then
              (o :: squares, OSet.add o used)
            else
              acc)
        orders ([], used) in
      let interferences, used =
        List.fold_right
          (fun (o1, o2 as o12) (interferences, used as acc) →
            if condition (add o1 o2) then
              (o12 :: interferences, OSet.add o1 (OSet.add o2 used))
            else
              acc)
          (ordered_pairs orders) ([], used) in
      (OSet.elements used, squares, interferences)

  end
```

Conceptually, there is no need to demand a *Colorized* model as a functor argument. Nevertheless, we should first implement a working example for the common use case, before embarking on a generalization that is mostly of academic interest.

```
module Flavor (M : Model.Colorized) =
  struct

    module CO = Coupling_Orders(M)

    type orders = CO.orders

    let add_orders = CO.add
    let incr_orders = CO.incr
    let null = CO.null
    let orders_of_list = CO.of_list

    type t = { all_orders : M.flavor; orders : orders }
    let all_orders f = f.all_orders
    let pullback f a = f (all_orders a)
    let make all_orders orders = { all_orders; orders }
    let trivial f = make f null
```

Resist the temptation to implement this as *List.fold_right* (fun *f* → *add_orders f.orders*) *f_list null*, because then *add_orders* would need to accept orders of different lengths.

```
    let fuse_orders = function
      | [] → null
      | f :: rest → List.fold_right (fun f → add_orders f.orders) rest f.orders

    let orders_to_string = CO.to_string

    let digit_to_symbol i =
      if i < 0 then
        invalid_arg "Orders.Flavor.digit_to_symbol:␣negative"
      else
        if i < 10 then
          string_of_int i
        else if i < 36 then
          String.make 1 (Char.chr (Char.code 'A' + i - 10))
        else
          invalid_arg "Orders.Flavor.digit_to_symbol:␣too␣large"

    let orders_symbol orders =
      match CO.to_list orders with
      | [] → ""
      | orders →
          if List.for_all (fun (_, n) → n = 0) orders then
            ""
          else
            "_c" ^ String.concat "" (List.map (fun (_, n) → digit_to_symbol n) orders)

    let to_string f =
      M.flavor_to_string f.all_orders ^ orders_to_string f.orders

    let to_symbol f =
      M.flavor_symbol f.all_orders ^ orders_symbol f.orders

  end
```

14.2.3 Slice Amplitudes According to Coupling Constant Orders

```
let incomplete s =
  failwith ("Orders.Slice()." ^ s ^ "␣not␣done␣yet!")

module Slice (CM : Model.Colorized) =
  struct

    module OCF = Flavor(CM)

    type flavor = OCF.t
    type flavor_sans_color = CM.flavor_sans_color
    type flavor_all_orders = CM.flavor
    type gauge = CM.gauge
    type constant = CM.constant
    type coupling_order = CM.coupling_order
    type orders = OCF.orders
    module Ch = CM.Ch
    let charges = OCF.pullback CM.charges
    let flavor_sans_color = OCF.pullback CM.flavor_sans_color
    let flavor_all_orders = OCF.all_orders
    let trivial = OCF.trivial
    let orders f = f.OCF.orders
    let add_orders = OCF.add_orders
    let incr_orders = OCF.incr_orders
    let orders_to_string = OCF.orders_to_string
    let orders_symbol = OCF.orders_symbol
    let flavor_equal f1 f2 =
      CM.flavor_equal (flavor_all_orders f1) (flavor_all_orders f2) ∧ f1.orders = f2.orders
    let color = OCF.pullback CM.color
    let pdg = OCF.pullback CM.pdg
    let lorentz = OCF.pullback CM.lorentz
    let propagator = OCF.pullback CM.propagator
    let width = OCF.pullback CM.width
    let conjugate f = { f with OCF.all_orders = CM.conjugate f.OCF.all_orders }
    let conjugate_sans_color = CM.conjugate_sans_color
    let conjugate_all_orders = CM.conjugate
    let fermion = OCF.pullback CM.fermion
    let max_degree = CM.max_degree
    let max_degree = CM.max_degree

    let vertices () =
      incomplete "vertices"

    let coupling = function
      | Coupling.V3 (_, _, c) | Coupling.V4 (_, _, c) | Coupling.Vn (_, _, c) → c

    let incr_coupling_orders orders (f, c) =
      let coupling_orders = CM.coupling_orders (coupling c) in
      let orders = OCF.incr_orders (OCF.orders_of_list coupling_orders) orders in
      (OCF.make f orders, c)

    let fuse2 f1 f2 =
      let orders = OCF.fuse_orders [f1; f2] in
      List.map (incr_coupling_orders orders) (CM.fuse2 (flavor_all_orders f1) (flavor_all_orders f2))

    let fuse3 f1 f2 f3 =
      let orders = OCF.fuse_orders [f1; f2; f3] in
      List.map (incr_coupling_orders orders) (CM.fuse3 (flavor_all_orders f1) (flavor_all_orders f2) (flavor_all_orders f3

    let fuse flavors =
      let orders = OCF.fuse_orders flavors in
      List.map (incr_coupling_orders orders) (CM.fuse (List.map flavor_all_orders flavors))

    let flavors () =
      List.map OCF.trivial (CM.flavors ())

    let all_coupling_orders = CM.all_coupling_orders
    let coupling_order_to_string = CM.coupling_order_to_string
    let coupling_orders = CM.coupling_orders

    let nc = CM.nc

    let external_flavors () =
      List.map
        (fun (group, flavors) →
          (group, List.map OCF.trivial flavors))
        (CM.external_flavors ())

    let goldstone f =
      match CM.goldstone (OCF.all_orders f) with
      | None → None
      | Some (f, c) → Some (OCF.trivial f, c)

    let parameters = CM.parameters
    let flavor_of_string s = OCF.trivial (CM.flavor_of_string s)
    let flavor_to_string = OCF.to_string
    let flavor_to_TeX = OCF.pullback CM.flavor_to_TeX
    let flavor_symbol = OCF.to_symbol
    let gauge_symbol = CM.gauge_symbol
    let mass_symbol = OCF.pullback CM.mass_symbol
    let width_symbol = OCF.pullback CM.width_symbol
    let constant_symbol = CM.constant_symbol
    let options = CM.options
    let caveats = CM.caveats

    let amplitude orders fin fout =
      (List.map (fun f → OCF.make f orders) fin,
       List.map (fun f → OCF.make f orders) fout)

    let flow fin fout =
      CM.flow (List.map flavor_all_orders fin) (List.map flavor_all_orders fout)

  end
```

14.2.4 Unit Tests

```
module Test =
  struct

    module O = Coupling_Orders (struct type coupling_order = int let coupling_order_to_string = string_of_int end)

    open OUnit

    let suite_add =

      "add" >:::
        [ "[(1,1);␣(2,4)]␣+␣[(1,2);␣(2,3)]" >::
            (fun () → assert_equal [(1,3); (2,7)] (O.add [(1,1); (2,4)] [(1,2); (2,3)])) ]

    let suite_incr =

      "incr" >:::
        [ "[(1,1);␣(3,4)]␣+␣[(2,2);␣(3,3)]" >::
            (fun () → assert_equal [(2,2); (3,7)] (O.incr [(1,1); (3,4)] [(2,2); (3,3)])) ]

    module M (* : Model_CO *) =
      struct
        type constant = E | G | G2 | L
        type coupling_order = EW | QCD | BSM
        let all_coupling_orders () = [EW; QCD; BSM]
        let coupling_order_to_string = function
          | EW → "EW"
          | QCD → "QCD"
          | BSM → "BSM"
        let coupling_orders = function
          | E → [(EW, 1)]
          | G → [(QCD, 1)]
          | G2 → [(QCD, 2)]
          | L → [(BSM, 1)]
      end

    module C = Conditions (M)

    let pup expected slist =
      assert_equal ~printer : (fun s → "\"" ^ s ^ "\"")
        expected (C.to_string (C.of_strings slist))

    let suite_parser =
      "parsing" >:::
        [ "EW=1" >:: (fun () → pup "EW␣=␣1" ["EW=1"]);
          "~EW" >:: (fun () → pup "{QCD,␣BSM}␣=␣0" ["~EW"]);
          "!BSM,QCD" >:: (fun () → pup "BSM␣=␣0;␣QCD␣=␣{1..2}" ["BSM;␣QCD={1..2}"]);
          "!BSM,QCD'" >:: (fun () → pup "BSM␣=␣0;␣QCD␣=␣{1..2}" ["BSM={0};␣QCD={1..2}"]);
          "EW/QCD" >:: (fun () → pup "EW␣=␣2;␣QCD␣=␣1" ["EW=2;␣QCD=1"]);
          "EW/QCD" >:: (fun () → pup "EW␣=␣1;␣QCD␣=␣1" ["EW=1;␣QCD=1"]);
          "EW/QCD'" >:: (fun () → pup "EW␣=␣1;␣QCD␣=␣1" ["{EW,QCD}=1"]);
          "EW=1,2,3" >:: (fun () → pup "EW␣=␣3" ["EW=1;EW=2;EW=3"]) ]

    let cos_option_to_string = function
      | None → "*"
      | Some co_list →
          ThoList.to_string (fun (co, n) → M.coupling_order_to_string co ^ "=" ^ string_of_int n) co_list

    let sort orders =
      List.sort (fun (co1, _) (co2, _) → compare co1 co2) orders

    let map_opt f = function
      | None → None
      | Some a → Some (f a)

    let assert_braket expected conditions orders =
      let conditions = C.of_strings conditions in
      assert_equal ~printer : cos_option_to_string
        (map_opt sort expected)
        (map_opt sort (C.braket conditions (sort orders)))

    let assert_fusion expected conditions orders =
      let conditions = C.of_strings conditions in
      assert_equal ~printer : string_of_bool expected (C.fusion conditions (sort orders))

    let suite_fusion =
      let open M in
      "fusion" >:::
        [ "BSM;EW=2;QCD=1:␣QCD=1" >::
            (fun () → assert_fusion true ["BSM;EW=2;QCD=1"] [(QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1" >::
            (fun () → assert_fusion true ["BSM;EW=2;QCD=1"] [(EW, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1;QCD=1" >::
            (fun () → assert_fusion true ["BSM;EW=2;QCD=1"] [(EW, 1); (QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=2;QCD=1" >::
            (fun () → assert_fusion true ["BSM;EW=2;QCD=1"] [(EW, 2); (QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1;QCD=2" >::
            (fun () → assert_fusion false ["BSM;EW=2;QCD=1"] [(EW, 1); (QCD, 2)]);

          "BSM;EW=2;QCD=1:␣BSM=1" >::
            (fun () → assert_fusion false ["BSM;EW=2;QCD=1"] [(BSM, 1)]);

          "BSM;EW=2;QCD=1:␣BSM=0" >::
            (fun () → assert_fusion true ["BSM;EW=2;QCD=1"] [(BSM, 0)]) ]

    let suite_braket =
      let open M in
      "braket" >:::
        [ "BSM;EW=2;QCD=1:␣QCD=1" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(EW, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1;QCD=1" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(EW, 1); (QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=2;QCD=1" >::
            (fun () → assert_braket (Some [(EW, 2); (QCD, 1)]) ["BSM;EW=2;QCD=1"] [(EW, 2); (QCD, 1)]);

          "BSM;EW=2;QCD=1:␣EW=1;QCD=2" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(EW, 1); (QCD, 2)]);

          "BSM;EW=2;QCD=1:␣BSM=1" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(BSM, 1)]);

          "BSM;EW=2;QCD=1:␣BSM=0" >::
            (fun () → assert_braket None ["BSM;EW=2;QCD=1"] [(BSM, 0)]);

          "EW={0..}:␣BSM=0" >::
            (fun () → assert_braket (Some [(EW, 0)]) ["EW={0..}"] [(BSM, 0)]);

          "EW={0..}:␣EW=1" >::
            (fun () → assert_braket (Some [(EW, 1)]) ["EW={0..}"] [(EW, 1)]);

          "EW={0..}:␣BSM=1;EW=1" >::
            (fun () → assert_braket (Some [(EW, 1)]) ["EW={0..}"] [(BSM, 1); (EW, 1)]) ]
```

We should add more unit tests, time permitting.

```
  let suite =
    "Orders" >:::
      [ suite_add;
```

```
          suite_incr;
          suite_parser;
          (* suite_fusion; *)
          suite_braket ]
  end
```

14.3 Interface of Orders_syntax

We represent coupling orders simply as *string*s so that lexing and parsing are independent of the model. Checking for validity is done later in the functor *Orders.Conditions* that depends on the model.

```
type co = string

type co_set =
  | Set of co list
  | Diff of co_set × co_set
  | Complement of co_set

type range =
  | Range of int × int
  | Min of int
  | Max of int
```

We distinguish intervals and slices:

- for the slice `QCD = {2..4}` all amplitudes with 2, 3 and 4 QCD couplings are generated separately and
- for the interval `QCD = [2..4]`, these are summed up.

Obviously, for one coupling order, there is no difference between interval and slice.

```
type atom =
  | Interval of co_set × range
  | Slices of co_set × range
  | Exact of co_set × int
  | Null of co_set

type t =
  | Atom of atom
  | And of t list
  | Or of t list

exception Syntax_Error of string × int × int
```

14.4 Implementation of Orders_syntax

```
type co = string

type co_set =
  | Set of co list
  | Diff of co_set × co_set
  | Complement of co_set

type range =
  | Range of int × int
  | Min of int
  | Max of int

type atom =
  | Interval of co_set × range
  | Slices of co_set × range
  | Exact of co_set × int
  | Null of co_set

type t =
  | Atom of atom
  | And of t list
  | Or of t list
exception Syntax_Error of string × int × int
```

14.5 Lexer

```
{
open Orders_parser
let unquote s =
  String.sub s 1 (String.length s − 2)
}
let digit = ['0'−'9']
let upper = ['A'−'Z']
let lower = ['a'−'z']
let char = upper | lower
let word = char | digit | '_'
let white = [' ' '\t' '\n']
```

We use a very liberal definition of strings for flavor names.

```
rule token = parse
    white { token lexbuf } (∗ skip blanks ∗)
  | '#' [^'\n']⋆ '\n'
                    { token lexbuf } (∗ skip comments ∗)
  | digit+ { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | '=' { EQ }
  | '~' { TILDE }
  | '\\' '\\'? { BACKSLASH }
  | '{' { LBRACE }
  | '}' { RBRACE }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | ';' { SEMI }
  | '&' '&'? { AND }
  | '|' '|'? { OR }
  | '.' '.' { RANGE }
  | ',' { COMMA }
  | char word⋆ { ID (Lexing.lexeme lexbuf) }
  | eof { END }
```

14.6 Parser

Header

```
open Orders_syntax
let parse_error msg =
  raise (Syntax_Error (msg, symbol_start (), symbol_end ()))
```

Token declarations

```
%token < string > ID
%token < int > INT
```

```
%token OR AND EQ BACKSLASH TILDE RANGE COMMA
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMI
%token END

%left OR
%left AND
%left BACKSLASH
%nonassoc TILDE

%start main
%type < Orders_syntax.t > main
```

Grammar rules

```
main ::=
     END { And [] }
   | condition END { $1 }
   | conjunction END { And $1 }
   | alternative END { Or $1 }


condition ::=
     atom { Atom $1 }
   | LPAREN conjunction RPAREN { And $2 }
   | LPAREN alternative RPAREN { Or $2 }


conjunction ::=
     condition { [$1] }
   | condition AND conjunction { $1 :: $3 }
   | condition SEMI conjunction { $1 :: $3 }


alternative ::=
     condition { [$1] }
   | condition OR alternative { $1 :: $3 }


atom ::=
     set EQ LBRACE range RBRACE { Slices ($1, $4) }
   | set EQ LBRACKET range RBRACKET { Interval ($1, $4) }
   | set EQ INT { Exact ($1, $3) }
   | set { Null $1 }


set ::=
     LBRACE RBRACE { Set [] }
   | ID { Set [$1] }
   | LBRACE orders RBRACE { Set $2 }
   | TILDE set { Complement $2 }
   | set BACKSLASH set { Diff ($1, $3) }


orders ::=
     ID { [$1] }
   | ID COMMA orders { $1 :: $3 }


range ::=
     RANGE INT { Max $2 }
   | INT RANGE { Min $1 }
   | INT RANGE INT { Range ($1, $3) }
   | INT { Range ($1, $1) }
```

—15—
Fusions

15.1 Interface of Fusion

15.1.1 Signature of Fusion.T

```
module type T =
  sig
    val options : Options.t
```

JRR's implementation of Majoranas needs a special case.

```
    val vintage : bool
```

Wavefunctions are an abstract data type, containing a momentum p and additional quantum numbers, collected in *flavor*.

```
    type wf
```

Return the wave function with the the same momentum and a charge conjugated *flavor*.

```
    val conjugate : wf → wf
```

Obviously, *flavor* is not restricted to the physical notion of flavor, but can carry spin, color, etc. See the implementation of *Model.T* for the physics.

```
    type flavor
    val flavor : wf → flavor
```

If *flavor* contains powers of coupling orders, it is sometimes useful for organizing the output and for diagnostics to be able to strip it away.

```
    type flavor_all_orders
    val flavor_all_orders : wf → flavor_all_orders
```

If *flavor* contains SU(3) color, it is sometimes useful for organizing the output and for diagnostics to be able to strip it away.

```
    type flavor_sans_color
    val flavor_sans_color : wf → flavor_sans_color
```

Momenta are represented by an abstract datatype (defined in *Momentum*) that is optimized for performance. They can be accessed either abstractly or as lists of indices of the external momenta. These indices are assigned sequentially by *amplitude* below.

```
    type p
    val momentum : wf → p
    val momentum_list : wf → int list
```

Coupling constants

```
    type constant
```

and right hand sides of assignments. The latter are formed from a sign from Fermi statistics, a coupling (constand and Lorentz structure) and wave functions of the children.

```
    type coupling
    type rhs
```

There is no deep reason for defining a polymorphic type α *children*, since we will only ever use *wf children*.

```
type α children
```

Keep track of statistics.

```
val sign : rhs → int
```

Extract the coupling (constant and structure) fusing the children.

```
val coupling : rhs → constant Coupling.t
```

In renormalized perturbation theory, couplings come in different orders of the loop expansion. Be prepared: val *order* : *rhs* → *int*

The concrete return type *wf list* is here only for the benefit of *Target* and could become *wf children* in a more refined interface ...

```
val children : rhs → wf list
```

Fusions come in two types: fusions of wave functions to off-shell wave functions:

$$\phi'(p+q) = \phi_1(p)\phi_2(q)$$

```
type fusion
val lhs : fusion → wf
val rhs : fusion → rhs list
```

and products at the keystones:

$$\langle\phi'(-p-q)|\phi_1(p)\phi_2(q)\rangle$$

```
type braket
val bra : braket → wf
val ket : braket → rhs list
```

amplitude goldstones incoming outgoing calculates the amplitude for scattering of *incoming* to *outgoing*. If *goldstones* is true, also non-propagating off-shell Goldstone amplitudes are included to allow the checking of Slavnov-Taylor identities. *selectors* is an instance of *Cascade.T.selectors* and used to select certain parts of an amplitude, see section 6.

```
type amplitude
type amplitude_sans_color
type selectors
type slicings
val amplitudes : bool → selectors → slicings option →
  flavor_sans_color list → flavor_sans_color list → amplitude list
val amplitudes_all_orders : bool → selectors →
  flavor_sans_color list → flavor_sans_color list → amplitude list
val amplitude_sans_color : bool → selectors →
  flavor_sans_color list → flavor_sans_color list → amplitude_sans_color
```

How a given wave function depends on other wave functions and couplings. This is used for finding subexpressions common among different color flow amplitudes.

```
val dependencies : amplitude → wf → (wf, coupling) Tree2.t
```

We should be precise regarding the semantics of the following functions, since modules implementating *Target* must not make any mistakes interpreting the return values. Instead of calculating the amplitude

$$\langle f_3, p_3, f_4, p_4, \ldots|T|f_1, p_1, f_2, p_2\rangle \tag{15.1a}$$

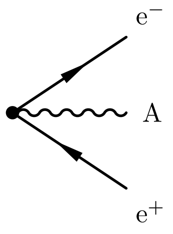
directly, O'Mega calculates the—equivalent, but more symmetrical—crossed amplitude

$$\langle \bar f_1, -p_1, \bar f_2, -p_2, f_3, p_3, f_4, p_4, \ldots|T|0\rangle \tag{15.1b}$$

For the benefit of the people implementing *Model*s, however, all flavors are represented internally by the charge conjugates

$$A(f_1, -p_1, f_2, -p_2, \bar f_3, p_3, \bar f_4, p_4, \ldots) \tag{15.1c}$$

Indeed, the vertex and corresponding term in the lagrangian



$$: \bar\psi \not{A} \psi \tag{15.2}$$

suggests to denote the *outgoing* particle by the flavor of the *anti*particle and the *outgoing* *anti*particle by the flavor of the particle, since this choice allows to represent the vertex by a triple

$$\bar\psi \not{A} \psi : (\mathrm{e}^+, A, \mathrm{e}^-) \tag{15.3}$$

which is more intuitive than the alternative $(\mathrm{e}^-, A, \mathrm{e}^+)$. Also, when thinking in terms of building wavefunctions from the outside in, the outgoing *antiparticle* is represented by a *particle* propagator and vice versa[1]. Note that *incoming* and *outgoing* are the physical flavors as in (15.1a) or in the argument of *amplitudes*, but with the color flow quantum numbers added.

```
val incoming : amplitude → flavor list
val outgoing : amplitude → flavor list
```

In contrast, *externals* are flavors and momenta as in (15.1c)

```
val externals : amplitude → wf list
```

Return all off-shell wave functions so that *Target* can allocate variables for them.

```
val variables : amplitude → wf list
```

Return all *fusions* in an order so that all right hand sides have been computed before they are used.

```
val fusions : amplitude → fusion list
```

Return all *brakets*.

```
type α slices
val brakets : amplitude → braket list slices
```

Test if an off-shell wave function has been forced on-shell or is smeared as as gaussian.

```
val on_shell : amplitude → wf → bool
val is_gauss : amplitude → wf → bool
```

Describe the constraints in the *selectors* argument to *amplitudes*.

```
val constraints : amplitude → string option
```

Human readable description of the requested slicings of type *Orders.Conditions.t*

```
val slicings : amplitude → string list
```

Compute the symmetry factor $\prod_i n_i!$ for identical outgoing particles.

```
val symmetry : amplitude → int
```

Quickly test whether an amplitude vanishes.

```
val allowed : amplitude → bool
```

Diagnostics

Compute a list of all charge conservation violating vertices in the *Model*.

```
val check_charges : unit → flavor_sans_color list list
```

Count the fusions and propagators that are computed and compare to the number of Feynman diagrams appearing in the amplitude.

```
val count_fusions : amplitude → int
val count_propagators : amplitude → int
```

[1] Even if this choice will appear slightly counter-intuitive on the *Target* side, one must keep in mind that much more people are expected to prepare *Models*.

```
    val count_diagrams : amplitude → int
```

Expand the *DAG* beneath an off-shell wave function into the corresponding forest. *Use with caution for complicated processes!*

```
    val forest : wf → amplitude → ((wf × coupling option, wf) Tree.t) list
```

A list of all combinations of off-shell wave functions in the Feynman diagrams described by the *DAG*. This could be used for phase space mappings, but lies dormant at the moment.

At the moment, the result contains empty lists and many redundancies. This should be cleaned up!

```
    val poles : amplitude → wf list list
```

A list of all s-channel poles in the *DAG*. Helpful for phase space mappings and for fudging widths.

```
    val s_channel : amplitude → wf list
```

Prepare `.dot` files as input fot `graphviz` to draw graphical representations of the tower of of-shell wavefunctions and the dag corresponding to the amplitude.

```
    val tower_to_dot : out_channel → amplitude → unit
    val amplitude_to_dot : out_channel → amplitude → unit
```

WHIZARD

Phase space descriptions for `WHIZARD`. Once as written and once with the incoming particles exchanged. This way we can write a tree starting from the first and one from the second incoming particle.

```
    val phase_space_channels : out_channel → amplitude_sans_color → unit
    val phase_space_channels_flipped : out_channel → amplitude_sans_color → unit
  end
```

15.1.2 Various Functors generating Fusion.T

There is more than one way to make fusions, differing in the unterlying topology of diagrams.

```
module type Maker =
    functor (P : Momentum.T) → functor (M : Model.T) →
      T with type p = P.t
      and type flavor = Orders.Slice(Colorize.It(M)).flavor
      and type flavor_all_orders = Colorize.It(M).flavor
      and type flavor_sans_color = M.flavor
      and type constant = M.constant
      and type selectors = Cascade.Make(M)(P).selectors
      and type slicings = Orders.Conditions(Colorize.It(M)).t
      and type α slices = (Orders.Slice(Colorize.It(M)).orders × α) list
```

Straightforward Dirac fermions vs. slightly more complicated Majorana fermions:

```
exception Majorana

module Binary : Maker
module Binary_Majorana : Maker

module Mixed23 : Maker
module Mixed23_Majorana : Maker

module Nary : functor (B : Tuple.Bound) → Maker
module Nary_Majorana : functor (B : Tuple.Bound) → Maker
```

We can also proceed á la [2]. Empirically, this will use slightly ($O(10\%)$) fewer fusions than the symmetric factorization. Our implementation uses significantly ($O(50\%)$) fewer fusions than reported by [2]. Our pruning of the DAG might be responsible for this.

```
module Helac_Binary : Maker
module Helac_Binary_Majorana : Maker
module Helac_Mixed23 : Maker
module Helac_Mixed23_Majorana : Maker
module Helac : functor (B : Tuple.Bound) → Maker
module Helac_Majorana : functor (B : Tuple.Bound) → Maker
```

15.1.3 Multiple Amplitudes

```
module type Multi =
  sig
    exception Mismatch
    val options : Options.t

    type flavor
    type process = flavor list × flavor list
    type amplitude
    type fusion
    type wf
    type selectors
    type slicings
    type coupling_order
    type amplitudes
```

Construct all possible color flow amplitudes for a given process.

```
    val amplitudes : bool → int option →
      selectors → slicings option → process list → amplitudes
    val empty : amplitudes
```

The list of all combinations of incoming and outgoing particles with a nonvanishing scattering amplitude.

```
    val flavors : amplitudes → process list
```

The list of all combinations of incoming and outgoing particles that don't lead to any color flow with non vanishing scattering amplitude.

```
    val vanishing_flavors : amplitudes → process list
```

The list of all color flows with a nonvanishing scattering amplitude.

```
    val color_flows : amplitudes → Color.Flow.t list
```

The coupling orders that are not summed over and their powers.

```
    val coupling_orders : amplitudes → (coupling_order list × int list list) option
```

The list of all valid helicity combinations.

```
    val helicities : amplitudes → (int list × int list) list
```

The list of all amplitudes.

```
    val processes : amplitudes → amplitude list
```

(*process_table a*).(f).(c) returns the amplitude for the fth allowed flavor combination and the cth allowed color flow as an *amplitude option*.

```
    val process_table : amplitudes → amplitude option array array
```

(*process_table a*).(*co*).(f).(c) returns the amplitude for the oth set of coupling orders, the fth allowed flavor combination and the cth allowed color flow as an *amplitude option*.

```
    val process_table_new : amplitudes → amplitude option array array array
```

The list of all non redundant fusions together with the amplitudes they came from.

```
    val fusions : amplitudes → (fusion × amplitude) list
```

If there's more than external flavor state, the wavefunctions are *not* uniquely specified by *flavor* and *Momentum.t*. This function can be used to determine how many variables must be allocated.

```
    val multiplicity : amplitudes → wf → int
```

This function can be used to disambiguate wavefunctions with the same combination of *flavor* and *Momentum.t*.

```
    val dictionary : amplitudes → amplitude → wf → int
```

(*color_factors a*).(*c1*).(*c2*) power of N_C for the given product of color flows.

```
    val color_factors : amplitudes → Color.Flow.factor array array
```

A description of optional diagram selectors.

```
    val constraints : amplitudes → string option
```

Human readable description of the requested slicings of type *Orders.Conditions.t*.

```
    val slicings : amplitudes → string list
  end
```

```
module type Multi_Maker = functor (Fusion_Maker : Maker) →
  functor (P : Momentum.T) →
    functor (M : Model.T) →
      Multi with type flavor = M.flavor
      and type amplitude = Fusion_Maker(P)(M).amplitude
      and type fusion = Fusion_Maker(P)(M).fusion
      and type wf = Fusion_Maker(P)(M).wf
      and type selectors = Fusion_Maker(P)(M).selectors
      and type slicings = Orders.Conditions(Colorize.It(M)).t
      and type coupling_order = Orders.Slice(Colorize.It(M)).coupling_order
module Multi : Multi_Maker
```

15.2 Implementation of Fusion

```
module IMap = Map.Make(Int)
module type T =
  sig
    val options : Options.t
    val vintage : bool
    type wf
    val conjugate : wf → wf
    type flavor
    type flavor_all_orders
    type flavor_sans_color
    val flavor : wf → flavor
    val flavor_all_orders : wf → flavor_all_orders
    val flavor_sans_color : wf → flavor_sans_color
    type p
    val momentum : wf → p
    val momentum_list : wf → int list
    type constant
    type coupling
    type rhs
    type α children
    val sign : rhs → int
    val coupling : rhs → constant Coupling.t
    val children : rhs → wf list
    type fusion
    val lhs : fusion → wf
    val rhs : fusion → rhs list
    type braket
    val bra : braket → wf
    val ket : braket → rhs list
    type amplitude
    type amplitude_sans_color
    type selectors
    type slicings
    val amplitudes : bool → selectors → slicings option →
      flavor_sans_color list → flavor_sans_color list → amplitude list
    val amplitudes_all_orders : bool → selectors →
      flavor_sans_color list → flavor_sans_color list → amplitude list
    val amplitude_sans_color : bool → selectors →
      flavor_sans_color list → flavor_sans_color list → amplitude_sans_color
    val dependencies : amplitude → wf → (wf, coupling) Tree2.t
    val incoming : amplitude → flavor list
    val outgoing : amplitude → flavor list
    val externals : amplitude → wf list
    val variables : amplitude → wf list
    val fusions : amplitude → fusion list
    type α slices
    val brakets : amplitude → braket list slices
    val on_shell : amplitude → wf → bool
    val is_gauss : amplitude → wf → bool
    val constraints : amplitude → string option
    val slicings : amplitude → string list
    val symmetry : amplitude → int
    val allowed : amplitude → bool
    val check_charges : unit → flavor_sans_color list list
    val count_fusions : amplitude → int
    val count_propagators : amplitude → int
    val count_diagrams : amplitude → int
    val forest : wf → amplitude → ((wf × coupling option, wf) Tree.t) list
    val poles : amplitude → wf list list
    val s_channel : amplitude → wf list
    val tower_to_dot : out_channel → amplitude → unit
    val amplitude_to_dot : out_channel → amplitude → unit
    val phase_space_channels : out_channel → amplitude_sans_color → unit
    val phase_space_channels_flipped : out_channel → amplitude_sans_color → unit
  end
```

```
module type Maker =
    functor (P : Momentum.T) → functor (M : Model.T) →
      T with type p = P.t
      and type flavor = Orders.Slice(Colorize.It(M)).flavor
      and type flavor_all_orders = Colorize.It(M).flavor
      and type flavor_sans_color = M.flavor
      and type constant = M.constant
      and type selectors = Cascade.Make(M)(P).selectors
      and type slicings = Orders.Conditions(Colorize.It(M)).t
      and type α slices = (Orders.Slice(Colorize.It(M)).orders × α) list
```

15.2.1 Fermi Statistics

```
module type Stat =
  sig
```

This will be *Model.T.flavor*.

```
    type flavor
```

A record of the fermion lines in the 1POW.

```
    type stat
```

Vertices with an odd number of fermion fields.

```
    exception Impossible
```

External lines.

```
    val stat : flavor → int → stat
```

stat_fuse (*Some flines*) *slist f* combines the fermion lines in the elements of *slist* according to the connections listed in *flines*. On the other hand, *stat_fuse None slist f* corresponds to the legacy mode with *at most* two fermions. The resulting flavor f of the 1POW can be ignored for models with only Dirac fermions, except for debugging, since the direction of the arrows is unambiguous. However, in the case of Majorana fermions and/or fermion number violating interactions, the flavor f must be used.

```
  val stat_fuse :
    Coupling.fermion_lines option → stat list → flavor → stat
```

Analogous to *stat_fuse*, but for the finalizing keystone instead of the 1POW.

```
  val stat_keystone :
    Coupling.fermion_lines option → stat list → flavor → stat
```

Compute the sign corresponding to the fermion lines in a 1POW or keystone.

```
  val stat_sign : stat → int
```

Debugging and consistency checks ...

```
  val stat_to_string : stat → string
  val equal : stat → stat → bool
  val saturated : stat → bool

end

module type Stat_Maker = functor (M : Model.T) →
  Stat with type flavor = M.flavor
```

15.2.2 Dirac Fermions

```
let dirac_log silent logging = logging
let dirac_log silent logging = silent

exception Majorana

module Stat_Dirac (M : Model.T) : (Stat with type flavor = M.flavor) =
  struct
    type flavor = M.flavor
```

$$\gamma_\mu\psi(1)\,G^{\mu\nu}\,\bar\psi(2)\gamma_\nu\psi(3) - \gamma_\mu\psi(3)\,G^{\mu\nu}\,\bar\psi(2)\gamma_\nu\psi(1) \tag{15.4}$$

The endpoints are *int option* instead of plain *int*, so that we can use *None* for open ends in *stat_sign* below.

We could do one level of unboxing as a performance hack by using 0 or -1 for open ends. Then we just need to enforce that all line numbers are strictly positive.

```
    type line = int option × int option

    let line_to_string = function
      | Some i, Some j → Printf.sprintf "%d>%d" i j
      | Some i, None → Printf.sprintf "%d>*" i
      | None, Some j → Printf.sprintf "*>%d" j
      | None, None → "*>*"

    type stat =
      | Fermion of int × line list
      | AntiFermion of int × line list
      | Boson of line list

    let lines_to_string lines =
      ThoList.to_string line_to_string lines

    let stat_to_string = function
      | Boson lines → Printf.sprintf "Boson␣%s" (lines_to_string lines)
      | Fermion (p, lines) →
          Printf.sprintf "Fermion␣(%d,␣%s)" p (lines_to_string lines)
      | AntiFermion (p, lines) →
          Printf.sprintf "AntiFermion␣(%d,␣%s)" p (lines_to_string lines)

    let equal s1 s2 =
      match s1, s2 with
      | Boson l1, Boson l2 →
          List.sort compare l1 = List.sort compare l2
      | Fermion (p1, l1), Fermion (p2, l2)
```

```
  | AntiFermion (p1, l1), AntiFermion (p2, l2) →
     p1 = p2 ∧ List.sort compare l1 = List.sort compare l2
  | _ → false

let saturated = function
  | Boson _ → true
  | _ → false

let stat f p =
  match M.fermion f with
  | 0 → Boson []
  | 1 → Fermion (p, [])
  | −1 → AntiFermion (p, [])
  | 2 → raise Majorana
  | _ → invalid_arg "Fusion.Stat_Dirac:␣invalid␣fermion␣number"

exception Impossible

let stat_fuse_pair_legacy f s1 s2 =
  match s1, s2 with
  | Boson l1, Boson l2 → Boson (l1 @ l2)
  | Boson l1, Fermion (p, l2) → Fermion (p, l1 @ l2)
  | Boson l1, AntiFermion (p, l2) → AntiFermion (p, l1 @ l2)
  | Fermion (p, l1), Boson l2 → Fermion (p, l1 @ l2)
  | AntiFermion (p, l1), Boson l2 → AntiFermion (p, l1 @ l2)
  | AntiFermion (pbar, l1), Fermion (p, l2) →
      Boson ((Some pbar, Some p) :: l1 @ l2)
  | Fermion (p, l1), AntiFermion (pbar, l2) →
      Boson ((Some pbar, Some p) :: l1 @ l2)
  | Fermion _, Fermion _ | AntiFermion _, AntiFermion _ →
      raise Impossible

let stat_fuse_legacy s1 s23__n f =
  List.fold_right (stat_fuse_pair_legacy f) s23__n s1

let stat_fuse_legacy_logging s1 s23__n f =
  let s = stat_fuse_legacy s1 s23__n f in
  Printf.eprintf
    "stat_fuse_legacy:␣%s␣<-␣%s␣->␣%s\n"
    (M.flavor_to_string f)
    (ThoList.to_string stat_to_string (s1 :: s23__n))
    (stat_to_string s);
  s

let stat_fuse_legacy =
  dirac_log stat_fuse_legacy stat_fuse_legacy_logging

type partial =
  { stat : stat (* the stat accumulated so far *);
    fermions : int IMap.t (* a map from the indices in the vertex to open fermion lines *);
    antifermions : int IMap.t (* a map from the indices in the vertex to open antifermion lines *);
    n : int (* the number of incoming propagators *) }

let partial_to_string p =
  Printf.sprintf
    "{␣fermions=%s,␣antifermions=%s,␣state=%s,␣#=%d␣}"
    (ThoList.to_string
       (fun (i, f) → Printf.sprintf "%d@%d" f i)
       (IMap.bindings p.fermions))
    (ThoList.to_string
       (fun (i, f) → Printf.sprintf "%d@%d" f i)
       (IMap.bindings p.antifermions))
    (stat_to_string p.stat)
    p.n

let add_lines l = function
```

```
    | Boson l' → Boson (List.rev_append l l')
    | Fermion (n, l') → Fermion (n, List.rev_append l l')
    | AntiFermion (n, l') → AntiFermion (n, List.rev_append l l')

let partial_of_slist slist =
  List.fold_left
    (fun acc s →
      let n = succ acc.n in
      match s with
      | Boson l →
          { acc with
            stat = add_lines l acc.stat;
            n }
      | Fermion (p, l) →
          { acc with
            fermions = IMap.add n p acc.fermions;
            stat = add_lines l acc.stat;
            n }
      | AntiFermion (p, l) →
          { acc with
            antifermions = IMap.add n p acc.antifermions;
            stat = add_lines l acc.stat;
            n } )
    { stat = Boson [];
      fermions = IMap.empty;
      antifermions = IMap.empty;
      n = 0 }
    slist

let match_fermion_line p (i, j) =
  if i ≤ p.n ∧ j ≤ p.n then
    match IMap.find_opt i p.fermions, IMap.find_opt j p.antifermions with
    | (Some _ as f), (Some _ as fbar) →
        { p with
          stat = add_lines [fbar, f] p.stat;
          fermions = IMap.remove i p.fermions;
          antifermions = IMap.remove j p.antifermions }
    | _ →
        invalid_arg "match_fermion_line:␣mismatched␣boson"
  else if i ≤ p.n then
    match IMap.find_opt i p.fermions, p.stat with
    | Some f, Boson l →
        { p with
          stat = Fermion (f, l);
          fermions = IMap.remove i p.fermions }
    | _ →
        invalid_arg "match_fermion_line:␣mismatched␣fermion"
  else if j ≤ p.n then
    match IMap.find_opt j p.antifermions, p.stat with
    | Some fbar, Boson l →
        { p with
          stat = AntiFermion (fbar, l);
          antifermions = IMap.remove j p.antifermions }
    | _ →
        invalid_arg "match_fermion_line:␣mismatched␣antifermion"
  else
    failwith "match_fermion_line:␣impossible"

let match_fermion_line_logging p (i, j) =
  Printf.eprintf
    "match_fermion_line␣%s␣(%d,␣%d)"
    (partial_to_string p) i j;
  let p' = match_fermion_line p (i, j) in
  Printf.eprintf "␣>>␣%s\n" (partial_to_string p');
  p'

let match_fermion_line =
  dirac_log match_fermion_line match_fermion_line_logging

let match_fermion_lines flines s1 s23__n =
  let p = partial_of_slist (s1 :: s23__n) in
  List.fold_left match_fermion_line p flines

let stat_fuse_new flines s1 s23__n f =
  (match_fermion_lines flines s1 s23__n).stat

let stat_fuse_new_checking flines s1 s23__n f =
  let stat = stat_fuse_new flines s1 s23__n f in
  if List.length flines < 2 then
    begin
      let legacy = stat_fuse_legacy s1 s23__n f in
      if ¬ (equal stat legacy) then
        failwith
          (Printf.sprintf
             "Fusion.Stat_Dirac.stat_fuse_new:␣%s␣<>␣%s!"
             (stat_to_string stat)
             (stat_to_string legacy))
    end;
  stat

let stat_fuse_new_logging flines s1 s23__n f =
  Printf.eprintf
    "stat_fuse_new:␣connecting␣fermion␣lines␣%s␣in␣%s␣<-␣%s\n"
    (UFO_Lorentz.fermion_lines_to_string flines)
    (M.flavor_to_string f)
    (ThoList.to_string stat_to_string (s1 :: s23__n));
  stat_fuse_new_checking flines s1 s23__n f

let stat_fuse_new =
  dirac_log stat_fuse_new stat_fuse_new_logging

let stat_fuse flines_opt slist f =
  match slist with
  | [] → invalid_arg "Fusion.Stat_Dirac.stat_fuse:␣empty"
  | s1 :: s23__n →
     begin match flines_opt with
     | Some flines → stat_fuse_new flines s1 s23__n f
     | None → stat_fuse_legacy s1 s23__n f
     end

let stat_fuse_logging flines_opt slist f =
  Printf.eprintf
    "stat_fuse:␣%s␣<-␣%s\n"
    (M.flavor_to_string f)
    (ThoList.to_string stat_to_string slist);
  stat_fuse flines_opt slist f

let stat_fuse =
  dirac_log stat_fuse stat_fuse_logging

let stat_keystone_legacy s1 s23__n f =
  let s2 = List.hd s23__n
  and s34__n = List.tl s23__n in
  stat_fuse_legacy s1 [stat_fuse_legacy s2 s34__n (M.conjugate f)] f

let stat_keystone_legacy_logging s1 s23__n f =
  let s = stat_keystone_legacy s1 s23__n f in
  Printf.eprintf
    "stat_keystone_legacy:␣%s␣(%s)␣%s␣->␣%s\n"
    (stat_to_string s1)
```

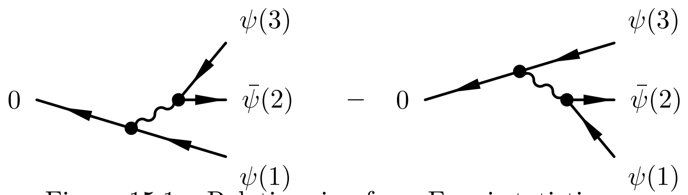


Figure 15.1: Relative sign from Fermi statistics.

```
      (M.flavor_to_string f)
      (ThoList.to_string stat_to_string s23__n)
      (stat_to_string s);
    s

let stat_keystone_legacy =
  dirac_log stat_keystone_legacy stat_keystone_legacy_logging

let stat_keystone flines_opt slist f =
  match slist with
  | [] → invalid_arg "Fusion.Stat_Dirac.stat_keystone: empty"
  | [s] → invalid_arg "Fusion.Stat_Dirac.stat_keystone: singleton"
  | s1 :: (s2 :: s34__n as s23__n) →
      begin match flines_opt with
      | None → stat_keystone_legacy s1 s23__n f
      | Some flines →
          (* The fermion line indices in flines must match the lines on one side of the keystone. *)
          let stat =
            stat_fuse_legacy s1 [stat_fuse_new flines s2 s34__n f] f in
          if saturated stat then
            stat
          else
            failwith
              (Printf.sprintf
                 "Fusion.Stat_Dirac.stat_keystone: incomplete %s!"
                 (stat_to_string stat))
      end

let stat_keystone_logging flines_opt slist f =
  let s = stat_keystone flines_opt slist f in
  Printf.eprintf
    "stat_keystone:         %s (%s) %s -> %s\n"
    (stat_to_string (List.hd slist))
    (M.flavor_to_string f)
    (ThoList.to_string stat_to_string (List.tl slist))
    (stat_to_string s);
  s

let stat_keystone =
  dirac_log stat_keystone stat_keystone_logging
```

$$\epsilon\left(\{(0,1),(2,3)\}\right) = -\epsilon\left(\{(0,3),(2,1)\}\right) \tag{15.5}$$

```
let permutation lines =
  let fout, fin = List.split lines in
  let eps_in, _ = Combinatorics.sort_signed fin
  and eps_out, _ = Combinatorics.sort_signed fout in
  (eps_in × eps_out)
```

This comparing of permutations of fermion lines is a bit tedious and takes a macroscopic fraction of time. However, it's less than 20 %, so we don't focus on improving on it yet.

```
let stat_sign = function
  | Boson lines → permutation lines
```

```
    | Fermion (p, lines) → permutation ((None, Some p) :: lines)
    | AntiFermion (pbar, lines) → permutation ((Some pbar, None) :: lines)
end
```

15.2.3 Amplitudes: Monochrome, Colored and Sliced

Computing the colored amplitudes from the uncolored amplitudes by adding color flows is the same algorithm as computing the uncolored amplitudes from the topology by adding flavors. The algorithm for adding powers of coupling constants is again almost identical, with only a small twist (see the type α *slices* below). Therefore we define a common module that we can instantiate thrice: once without color, once with and once with powers coupling constants on top.
In the future, we might want to have *Coupling* among the functor arguments. However, for the moment, *Coupling* is assumed to be comprehensive.

```
module type Amplitude =
  sig
```

An off-shell wavefunction is uniquely characterized by a *flavor* (which will contain the physical flavor and might contain color flows and coupling order powers) and a momentum

```
    type flavor
    type p
    type wf = { flavor : flavor; momentum : p }
```

Conjugate the flavor, keeping the momentum.

```
    val conjugate : wf → wf
```

Extract flavor and momentum from a wave function. *momentum_list* is a convenience function that composes *momentum* and *Momentum.to_ints*.

```
    val flavor : wf → flavor
    val momentum : wf → p
    val momentum_list : wf → int list
```

An ordering that guarantees that wavefunctions will be ordered according to *increasing Momentum().rank* of their momenta. For tree level amplitudes, this can be used to get the correct order of evaluation.

```
    val order_wf : wf → wf → int
```

external_wfs rank constructs a list of wavefunctions from pairs of *flavor*s and indices of external momenta, using *rank* in the representation of momenta.

```
    val external_wfs : int → (flavor × int) list → wf list
```

The couplings are model dependent, of course and we also must keep track of a sign for Fermi statistics. The value of *sign* must be either $+1$ or -1.

```
    type constant
    type coupling = { sign : int; coupling : constant Coupling.t }
```

The incoming wavefunctions (a. k. a. *children*) in a fusion can be represented by a *list* or a *Tuple* and we .

```
    type α children
    type rhs = coupling × wf children
    val sign : rhs → int
    val coupling : rhs → constant Coupling.t
    val children : rhs → wf list
```

In a *fusion*, we can have more than one term contribute on the right hand side.

```
    type fusion = wf × rhs list
    val lhs : fusion → wf
    val rhs : fusion → rhs list
```

In a *braket*, we can have more than one term contribute on the *ket*, if we factor common *bra*s.

```
    type braket = wf × rhs list
    val bra : braket → wf
    val ket : braket → rhs list
```

The small twist alluded to above is that in the case of counting powers coupling constants there will be different sets of *braket*s that correspond to different powers of coupling constants.

Therefore, we wrap the *braket list* as *braket list Slicer.t* that can be implented in a functor argument either trivially as *braket list* in the module *Unsliced* or as a (*orders* × *braket list*) *list*, as in the module *By_Orders* below.

Note that slicing a list of whole amplitudes instead of the *braket list* would lead to unnecessary duplication of *fusion*s.

```
type α slices
val unsliced : α → α slices
```

That's the big bad DAG that implents the recursive construction of off-shell wave functions.

```
module D : DAG.T with type node = wf and type edge = coupling and type children = wf children
```

Return the list of all unique wavefunctions appearing in list of *braket*s on the left and right hand sides.

```
val wavefunctions : braket list → wf list
```

That's the type that holds the result of our computations.

```
type t =
  { fusions : fusion list;
    brakets : braket list slices;
    on_shell : (wf → bool);
    is_gauss : (wf → bool);
    constraints : string option;
    slicings : string list;
    incoming : flavor list;
    outgoing : flavor list;
    externals : wf list;
    symmetry : int;
    dependencies : (wf → (wf, coupling) Tree2.t);
    fusion_tower : D.t;
    fusion_dag : D.t }
```

The following accessor functions are redundant, since the type *t* is not abstract, but they are convenient, nevertheless.
The *flavor*s of the incoming and outgoing particles.

```
val incoming : t → flavor list
val outgoing : t → flavor list
```

The on-shell wave functions for the external particles in the crossed amplitude with all particles incoming. The outgoing flavors have been replaced by their charge conjugates. The *Target* must declare variables for them and initialize these from the momenta.

```
val externals : t → wf list
```

All off-shell wave functions. The *Target* must declare variables for them.

```
val variables : t → wf list
```

All fusions. The *Target* uses them to recursively compute the off-shell wavefunctions.

```
val fusions : t → fusion list
```

All slices of brakets. The *Target* evaluates each braket and adds the results for each slice to obtain the corresponding scattering amplitude.

```
val brakets : t → braket list slices
```

Test if the user requested to replace the propagator for the off-shell wavefunction by an on-shell condition or a gaussian.

```
val on_shell : t → wf → bool
val is_gauss : t → wf → bool
```

Human readable description of the constraints of type *Cascades*().*selectors* that have been applied to the amplitude.

```
val constraints : t → string option
```

Human readable description of the requested slicings of type *Orders.Conditions.t*

```
    val slicings : t → string list
```

Size of the permutation symmetry group for identical outgoing patricles.

```
    val symmetry : t → int
```

The DAG that will be transformed by colorization and slicing.

```
    val fusion_dag : t → D.t
```

This is used for diagnostics.

```
    val dependencies : t → wf → (wf, coupling) Tree2.t
  end
```

Investigate if we can optimize also the unsliced amplitudes by keeping only one *DAG.t* and slice the brakets.

```
module type Slicer =
  sig
    type α t
    val all : α → α t
  end

module Unsliced =
  struct
    type α t = α
    let all a = a
  end

module Amplitude (PT : Tuple.Poly) (P : Momentum.T) (M : Model.T) (S : Slicer) : Amplitude
        with type p = P.t
         and type flavor = M.flavor
         and type constant = M.constant
         and type α children = α PT.t
         and type α slices = α S.t =
  struct

    type flavor = M.flavor
    type p = P.t

    type wf = { flavor : flavor; momentum : p }

    let flavor wf = wf.flavor
    let conjugate wf = { wf with flavor = M.conjugate wf.flavor }
    let momentum wf = wf.momentum
    let momentum_list wf = P.to_ints wf.momentum

    let external_wfs rank particles =
      List.map
        (fun (f, p) →
          { flavor = f;
            momentum = P.singleton rank p })
        particles
```

Order wavefunctions so that the external come first, then the pairs, etc. Also put possible Goldstone bosons *before* their gauge bosons.

```
    let lorentz_ordering f =
      match M.lorentz f with
      | Coupling.Scalar → 0
      | Coupling.Spinor → 1
      | Coupling.ConjSpinor → 2
      | Coupling.Majorana → 3
      | Coupling.Vector → 4
      | Coupling.Massive_Vector → 5
      | Coupling.Tensor_2 → 6
```

```
  | Coupling.Tensor_1 → 7
  | Coupling.Vectorspinor → 8
  | Coupling.BRS Coupling.Scalar → 9
  | Coupling.BRS Coupling.Spinor → 10
  | Coupling.BRS Coupling.ConjSpinor → 11
  | Coupling.BRS Coupling.Majorana → 12
  | Coupling.BRS Coupling.Vector → 13
  | Coupling.BRS Coupling.Massive_Vector → 14
  | Coupling.BRS Coupling.Tensor_2 → 15
  | Coupling.BRS Coupling.Tensor_1 → 16
  | Coupling.BRS Coupling.Vectorspinor → 17
  | Coupling.BRS _ → invalid_arg "Fusion.lorentz_ordering:␣not␣needed"
  | Coupling.Maj_Ghost → 18

let order_flavor f1 f2 =
  let c = compare (lorentz_ordering f1) (lorentz_ordering f2) in
  if c ≠ 0 then
    c
  else
    compare f1 f2
```

Note that *Momentum*().*compare* guarantees that wavefunctions will be ordered according to *increasing Momentum*().*rank* of their momenta.

```
let order_wf wf1 wf2 =
  let c = P.compare wf1.momentum wf2.momentum in
  if c ≠ 0 then
    c
  else
    order_flavor wf1.flavor wf2.flavor
```

This *must* be a pair matching the *edge* × *node children* pairs of *DAG.Forest*!

```
type α children = α PT.t
type constant = M.constant
type coupling = { sign : int; coupling : constant Coupling.t }
type rhs = coupling × wf children
let sign (c, _) = c.sign
let coupling (c, _) = c.coupling
let children (_, wfs) = PT.to_list wfs

type fusion = wf × rhs list
let lhs (l, _) = l
let rhs (_, r) = r

type braket = wf × rhs list
let bra (b, _) = b
let ket (_, k) = k

module WF = struct type t = wf let compare = order_wf end
module CPL = struct type t = coupling let compare = compare end
module D = DAG.Make(DAG.Forest(PT)(WF)(CPL))

module WFSet = Set.Make(WF)

let wavefunctions brakets =
  WFSet.elements
    (List.fold_left
       (fun set (wf1, wf23) →
          WFSet.add wf1 (List.fold_left
                           (fun set′ (_, wfs) →
                              PT.fold_right WFSet.add wfs set′)
                           set wf23))
       WFSet.empty brakets)

type α slices = α S.t
```

```
  let unsliced a = S.all a

  type t =
    { fusions : fusion list;
      brakets : braket list slices;
      on_shell : (wf → bool);
      is_gauss : (wf → bool);
      constraints : string option;
      slicings : string list;
      incoming : flavor list;
      outgoing : flavor list;
      externals : wf list;
      symmetry : int;
      dependencies : (wf → (wf, coupling) Tree2.t);
      fusion_tower : D.t;
      fusion_dag : D.t }

  let incoming a = a.incoming
  let outgoing a = a.outgoing
  let externals a = a.externals
  let fusions a = a.fusions
  let brakets a = a.brakets
  let symmetry a = a.symmetry
  let on_shell a = a.on_shell
  let is_gauss a = a.is_gauss
  let constraints a = a.constraints
  let slicings a = a.slicings
  let variables a = List.map lhs a.fusions
  let dependencies a = a.dependencies
  let fusion_dag a = a.fusion_dag

end
```

15.2.4 The Fusion.Make Functor

```
module Make (PT : Tuple.Poly)
    (Stat : Stat_Maker) (T : Topology.T with type α children = α PT.t)
    (P : Momentum.T) (M : Model.T) =
  struct

  let vintage = false

  let options = Options.create
      [ ]

  module S = Stat(M)

  type stat = S.stat
  let stat = S.stat
  let stat_sign = S.stat_sign
```

This will do *something* for 4-, 6-, ... fermion vertices, but not necessarily the right thing ...

This is copied from *Colorize* and should be factored!

In the long run, it will probably be beneficial to apply the permutations in *Modeltools.add_vertexn*!

```
  module PosMap =
    Partial.Make (struct type t = int let compare = compare end)

  let partial_map_undoing_permutation l l′ =
    let module P = Permutation.Default in
    let p = P.of_list (List.map pred l′) in
    PosMap.of_lists l (P.list p l)

  let partial_map_undoing_fuse fuse =
    partial_map_undoing_permutation
      (ThoList.range 1 (List.length fuse))
      fuse

  let undo_permutation_of_fuse fuse =
    PosMap.apply_with_fallback
      (fun _ → invalid_arg "permutation_of_fuse")
      (partial_map_undoing_fuse fuse)

  let fermion_lines = function
    | Coupling.V3 _ | Coupling.V4 _ → None
    | Coupling.Vn (Coupling.UFO (_, _, _, fl, _), fuse, _) →
        Some (UFO_Lorentz.map_fermion_lines (undo_permutation_of_fuse fuse) fl)

  type constant = M.constant
```

Wave Functions

```
  module A = Amplitude(PT)(P)(M)(Unsliced)
```

Operator insertions can be fused only if they are external.

```
  let is_source wf =
    match M.propagator wf.A.flavor with
    | Only_Insertion → P.rank wf.A.momentum = 1
    | _ → true
```

is_goldstone_of g v is true if and only if g is the Goldstone boson corresponding to the gauge particle v.

```
  let is_goldstone_of g v =
    match M.goldstone v with
    | None → false
    | Some (g′, _) → g = g′
```

In the end, *PT.to_list* should become redudant!

```
  let fuse_rhs rhs = M.fuse (PT.to_list rhs)
```

Vertices

Compute the set of all vertices in the model from the allowed fusions and the set of all flavors:

One could think of using *M.vertices* instead of *M.fuse2*, *M.fuse3* and *M.fuse* ...

```
  module VSet = Map.Make(struct type t = A.flavor let compare = compare end)

  let add_vertices f rhs m =
    VSet.add f (try rhs :: VSet.find f m with Not_found → [rhs]) m

  let collect_vertices rhs =
    List.fold_right (fun (f1, c) → add_vertices (M.conjugate f1) (c, rhs))
      (fuse_rhs rhs)
```

The set of all vertices with common left fields factored.
I used to think that constant initializers are a good idea to allow compile time optimizations. The down side turned out to be that the constant initializers will be evaluated *every time* the functor is applied. *Relying on the fact that the functor will be called only once is not a good idea!*

```
  type vertices = (A.flavor × (constant Coupling.t × A.flavor PT.t) list) list
```

This is *very* inefficient for *max_degree* > 6. Find a better approach that avoids precomputing the huge lookup table!

I should revive the above Idea to use *M.vertices* instead directly, instead of rebuilding it from *M.fuse2*, *M.fuse3* and *M.fuse*!

```
let vertices_nocache max_degree flavors : vertices =
  VSet.fold (fun f rhs v → (f, rhs) :: v)
    (PT.power_fold
        ~truncate : (pred max_degree)
        collect_vertices flavors VSet.empty) [ ]
```

Performance hack:

```
type vertex_table =
        ((A.flavor × A.flavor × A.flavor) × constant Coupling.vertex3 × constant) list
      × ((A.flavor × A.flavor × A.flavor × A.flavor)
           × constant Coupling.vertex4 × constant) list
      × (A.flavor list × constant Coupling.vertexn × constant) list

let vertices = vertices_nocache

let vertices′ max_degree flavors =
  Printf.eprintf ">>>␣vertices␣%d␣..." max_degree;
  flush stderr;
  let v = vertices max_degree flavors in
  Printf.eprintf "␣done.\n";
  flush stderr;
  v

let filter_vertices select_vtx vertices =
  List.fold_left
    (fun acc (f, cfs) →
      let f′ = M.conjugate f in
      let cfs =
        List.filter
          (fun (c, fs) → select_vtx c f′ (PT.to_list fs))
          cfs
      in
      match cfs with
      | [ ] → acc
      | cfs → (f, cfs) :: acc)
    [ ] vertices
```

K-Matrix Filtering

Vertices that are not crossing invariant need special treatment so that they're only generated for the correct combinations of momenta.

NB: the *crossing* checks here are a bit redundant, because *CM.fuse* below will bring the killed vertices back to life and will have to filter once more. Nevertheless, we keep them here, for the unlikely case that anybody ever wants to use uncolored amplitudes directly.

NB: the analogous problem does not occur for *select_wf*, because this applies to momenta instead of vertices.

This approach worked before the colorize, but has become *futile*, because *CM.fuse* will bring the killed vertices back to life. We need to implement the same checks there again!!!

Using *PT.Mismatched_arity* is not really good style ...

Tho's approach doesn't work since he does not catch charge conjugated processes or crossed processes. Another very strange thing is that O'Mega seems always to run in the q2 q3 timelike case, but not in the other two. (Property of how the DAG is built?). For the *ZZZZ* vertex I add the same vertex again, but interchange 1 and 3 in the *crossing* vertex

```
let timelike_sut momenta =
  let timelike p q = P.Scattering.timelike (P.add p q) in
  match PT.to_list momenta with
  | [q1; q2; q3] → (timelike q1 q2, timelike q2 q3, timelike q1 q3)
  | _ → raise PT.Mismatched_arity

let kmatrix_cuts c momenta =
  let open Coupling in
  match c with
  | V4 (Vector4_K_Matrix_tho (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_jr (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_t0 (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_t1 (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_t2 (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_t_rsi (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_m0 (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_m1 (disc, _), fusion, _)
  | V4 (Vector4_K_Matrix_cf_m7 (disc, _), fusion, _) →
      let s12, s23, s13 = timelike_sut momenta in
      begin match disc, s12, s23, s13, fusion with
      | 0, true, false, false, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
      | 0, false, true, false, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
      | 0, false, false, true, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) →
          true
      | 1, true, false, false, (F341 | F431 | F342 | F432)
      | 1, false, true, false, (F134 | F143 | F234 | F243)
      | 1, false, false, true, (F314 | F413 | F324 | F423) →
          true
      | 2, true, false, false, (F123 | F213 | F124 | F214)
      | 2, false, true, false, (F312 | F321 | F412 | F421)
      | 2, false, false, true, (F132 | F231 | F142 | F241) →
          true
      | 3, true, false, false, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234)
      | 3, false, true, false, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423)
      | 3, false, false, true, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) →
          true
      | _ → false
      end

  | V4 (DScalar2_Vector2_K_Matrix_ms (disc, _), fusion, _)
  | V4 (DScalar2_Vector2_m_0_K_Matrix_cf (disc, _), fusion, _)
  | V4 (DScalar2_Vector2_m_1_K_Matrix_cf (disc, _), fusion, _)
  | V4 (DScalar2_Vector2_m_7_K_Matrix_cf (disc, _), fusion, _) →
      let s12, s23, s13 = timelike_sut momenta in
      begin match disc, s12, s23, s13, fusion with
      | 0, true, false, false, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
      | 0, false, true, false, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
      | 0, false, false, true, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) →
          true
      | 1, true, false, false, (F341 | F432 | F123 | F214)
      | 1, false, true, false, (F134 | F243 | F312 | F421)
      | 1, false, false, true, (F314 | F423 | F132 | F241) →
          true
      | 2, true, false, false, (F431 | F342 | F213 | F124)
      | 2, false, true, false, (F143 | F234 | F321 | F412)
      | 2, false, false, true, (F413 | F324 | F231 | F142) →
          true
      | 3, true, false, false, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234)
      | 3, false, true, false, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423)
      | 3, false, false, true, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) →
          true
      | 4, true, false, false, (F142 | F413 | F231 | F324)
      | 4, false, true, false, (F214 | F341 | F123 | F432)
      | 4, false, false, true, (F124 | F431 | F213 | F342) →
          true
      | 5, true, false, false, (F143 | F412 | F321 | F234)
```

```
        | 5, false, true, false, (F314 | F241 | F132 | F423)
        | 5, false, false, true, (F134 | F421 | F312 | F243) →
            true
        | 6, true, false, false, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423)
        | 6, false, true, false, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342)
        | 6, false, false, true, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) →
            true
        | 7, true, false, false, (F134 | F312 | F421 | F243)
        | 7, false, true, false, (F413 | F231 | F142 | F324)
        | 7, false, false, true, (F143 | F321 | F412 | F432) →
            true
        | 8, true, false, false, (F132 | F314 | F241 | F423)
        | 8, false, true, false, (F213 | F431 | F124 | F342)
        | 8, false, false, true, (F123 | F341 | F214 | F234) →
            true
        | _ → false
        end
```

Are the missing cases 1 and 2 for *disc* an oversight here?

```
    | V4 (DScalar4_K_Matrix_ms (disc, _), fusion, _) →
        let s12, s23, s13 = timelike_sut momenta in
        begin match disc, s12, s23, s13, fusion with
        | 0, true, false, false, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
        | 0, false, true, false, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
        | 0, false, false, true, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) →
            true
        | 3, true, false, false, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234)
        | 3, false, true, false, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423)
        | 3, false, false, true, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) →
            true
        | 4, true, false, false, (F142 | F413 | F231 | F324)
        | 4, false, true, false, (F214 | F341 | F123 | F432)
        | 4, false, false, true, (F124 | F431 | F213 | F342) →
            true
        | 5, true, false, false, (F143 | F412 | F321 | F234)
        | 5, false, true, false, (F314 | F241 | F132 | F423)
        | 5, false, false, true, (F134 | F421 | F312 | F243) →
            true
        | 6, true, false, false, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423)
        | 6, false, true, false, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342)
        | 6, false, false, true, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) →
            true
        | 7, true, false, false, (F134 | F312 | F421 | F243)
        | 7, false, true, false, (F413 | F231 | F142 | F324)
        | 7, false, false, true, (F143 | F321 | F412 | F432) →
            true
        | 8, true, false, false, (F132 | F314 | F241 | F423)
        | 8, false, true, false, (F213 | F431 | F124 | F342)
        | 8, false, false, true, (F123 | F341 | F214 | F234) →
            true
        | _ → false
        end
    | _ → true
```

Match a set of flavors to a set of momenta. Form the direct product for the lists of momenta two and three with the list of couplings and flavors two and three.

```
  let flavor_keystone select_p dim (f1, f23) (p1, p23) =
    ({ A.flavor = f1;
       A.momentum = P.of_ints dim p1 },
     Product.fold2 (fun (c, f) p acc →
       try
         let p′ = PT.map (P.of_ints dim) p in
         if select_p (P.of_ints dim p1) (PT.to_list p′) ∧ kmatrix_cuts c p′ then
           (c, PT.map2 (fun f″ p″ → { A.flavor = f″;
                                       A.momentum = p″ }) f p′) :: acc
         else
           acc
       with
       | PT.Mismatched_arity → acc) f23 p23 [])
```

Produce all possible combinations of vertices (flavor keystones) and momenta by forming the direct product. The semantically equivalent *Product.list2 (flavor_keystone select_wf n) vertices keystones* with *subsequent* filtering would be a *very bad* idea, because a potentially huge intermediate list is built for large models. E. g. for the MSSM this would lead to non-termination by thrashing for $2 \to 4$ processes on most PCs.

```
let flavor_keystones filter select_p dim vertices keystones =
  Product.fold2 (fun v k acc →
    filter (flavor_keystone select_p dim v k) acc) vertices keystones []
```

Flatten the nested lists of vertices into a list of attached lines.

```
let flatten_keystones t =
  ThoList.flatmap (fun (p1, p23) →
    p1 :: (ThoList.flatmap (fun (_, rhs) → PT.to_list rhs) p23)) t
```

Subtrees

Fuse a tuple of wavefunctions, keeping track of Fermi statistics. Record only the the sign *relative* to the children. (The type annotation is only for documentation.)

```
let fuse select_wf select_vtx wfss : (A.wf × stat × A.rhs) list =
  if PT.for_all (fun (wf, _) → is_source wf) wfss then
    try
      let wfs, ss = PT.split wfss in
      let flavors = PT.map A.flavor wfs
      and momenta = PT.map A.momentum wfs in
      let p = PT.fold_left_internal P.add momenta in
      List.fold_left
        (fun acc (f, c) →
          if select_wf f p (PT.to_list momenta)
            ∧ select_vtx c f (PT.to_list flavors)
            ∧ kmatrix_cuts c momenta then
            let s = S.stat_fuse (fermion_lines c) (PT.to_list ss) f in
            let flip = PT.fold_left (fun acc s′ → acc × stat_sign s′) (stat_sign s) ss in
            ({ A.flavor = f;
               A.momentum = p }, s,
             ({ A.sign = flip;
                A.coupling = c }, wfs)) :: acc
          else
            acc)
        [] (fuse_rhs flavors)
    with
    | P.Duplicate _ | S.Impossible → []
  else
    []
```

Eventually, the pairs of *tower* and *dag* in *fusion_tower′* below could and should be replaced by a graded *DAG*. This will look like, but currently *tower* containts statistics information that is missing from *dag*:

```
Type node = flavor * p is not compatible with type wf * stat
```

This should be easy to fix. However, replacing type *t = wf* with type *t = wf × stat* is *not* a good idea because the variable *stat* makes it impossible to test for the existance of a particular *wf* in a *DAG*.

In summary, it seems that $(wf \times stat)$ *list array* $\times A.D.t$ should be replaced by $(wf \rightarrow stat) \times A.D.t$.

```
module GF =
  struct
    module Nodes =
      struct
        type t = A.wf
        module G = struct type t = int let compare = compare end
        let compare = A.order_wf
        let rank wf = P.rank wf.A.momentum
      end
    module Edges = struct type t = A.coupling let compare = compare end
    module F = DAG.Forest(PT)(Nodes)(Edges)
    type node = Nodes.t
    type edge = F.edge
    type children = F.children
    type t = F.t
    let compare = F.compare
    let for_all = F.for_all
    let fold = F.fold
  end

module D' = DAG.Graded(GF)

let tower_of_dag dag =
  let _, max_rank = D'.min_max_rank dag in
  Array.init max_rank (fun n → D'.ranked n dag)
```

The function *fusion_tower'* recursively builds the tower of all fusions from bottom up to a chosen level. The argument *tower* is an array of lists, where the i-th sublist (counting from 0) represents all off shell wave functions depending on $i+1$ momenta and their Fermistatistics.

$$\begin{aligned}\Big[&\{\phi_1(p_1),\phi_2(p_2),\phi_3(p_3),\ldots\},\\ &\{\phi_{12}(p_1+p_2),\phi'_{12}(p_1+p_2),\ldots,\phi_{13}(p_1+p_3),\ldots,\phi_{23}(p_2+p_3),\ldots\},\\ &\ldots\\ &\{\phi_{1\cdots n}(p_1+\cdots+p_n),\phi'_{1\cdots n}(p_1+\cdots+p_n),\ldots\}\Big]\end{aligned} \tag{15.6}$$

The argument *dag* is a DAG representing all the fusions calculated so far. NB: The outer array in *tower* is always very short, so we could also have accessed a list with *List.nth*. Appending of new members at the end brings no loss of performance. NB: the array is supposed to be immutable.
The towers must be sorted so that the combinatorical functions can make consistent selections.

Intuitively, this seems to be correct. However, one could have expected that no element appears twice and that this ordering is not necessary ...

```
let grow select_wf select_vtx tower =
  let rank = succ (Array.length tower) in
  List.sort Stdlib.compare
    (PT.graded_sym_power_fold rank
       (fun wfs acc → fuse select_wf select_vtx wfs @ acc) tower [])

let add_offspring dag (wf, _, rhs) =
  A.D.add_offspring wf rhs dag

let filter_offspring fusions =
  List.map (fun (wf, s, _) → (wf, s)) fusions

let rec fusion_tower' n_max select_wf select_vtx tower dag : (A.wf × stat) list array × A.D.t =
  if Array.length tower ≥ n_max then
    (tower, dag)
  else
    let tower' = grow select_wf select_vtx tower in
    fusion_tower' n_max select_wf select_vtx
```

```
        (Array.append tower [|filter_offspring tower'|])
        (List.fold_left add_offspring dag tower')
```

Discard the tower and return a map from wave functions to Fermistatistics together with the DAG.

```
let make_external_dag wfs =
  List.fold_left (fun m (wf, _) → A.D.add_node wf m) A.D.empty wfs

let mixed_fold_left f acc lists =
  Array.fold_left (List.fold_left f) acc lists

module WF = struct type t = A.wf let compare = A.order_wf end
module FWMap = Map.Make(WF)

let fusion_tower height select_wf select_vtx wfs : (A.wf → stat) × A.D.t =
  let tower, dag =
    fusion_tower' height select_wf select_vtx [|wfs|] (make_external_dag wfs) in
  let stats = mixed_fold_left
      (fun m (wf, s) → FWMap.add wf s m) FWMap.empty tower in
  ((fun wf → FWMap.find wf stats), dag)
```

Calculate the minimal tower of fusions that suffices for calculating the amplitude.

```
let minimal_fusion_tower n select_wf select_vtx wfs : (A.wf → stat) × A.D.t =
  fusion_tower (T.max_subtree n) select_wf select_vtx wfs
```

Calculate the complete tower of fusions. It is much larger than required, but it allows a complete set of gauge checks.

```
let complete_fusion_tower select_wf select_vtx wfs : (A.wf → stat) × A.D.t =
  fusion_tower (List.length wfs − 1) select_wf select_vtx wfs
```

There is a natural product of two DAGs using *fuse*. Can this be used in a replacement for *fusion_tower*? The hard part is to avoid double counting, of course. A straight forward solution could do a diagonal sum (in order to reject flipped offspring representing the same fusion) and rely on the uniqueness in *DAG* otherwise. However, this will (probably) slow down the procedure significanty, because most fusions (including Fermi signs!) will be calculated before being rejected by *DAG*().*add_offspring*.

Add to *dag* all Goldstone bosons defined in *tower* that correspond to gauge bosons in *dag*. This is only required for checking Slavnov-Taylor identities in unitarity gauge. Currently, it is not used, because we use the complete tower for gauge checking.

```
let harvest_goldstones tower dag =
  A.D.fold_nodes (fun wf dag' →
    match M.goldstone wf.A.flavor with
    | Some (g, _) →
        let wf' = { wf with A.flavor = g } in
        if A.D.is_node wf' tower then begin
          A.D.harvest tower wf' dag'
        end else begin
          dag'
        end
    | None → dag') dag dag
```

Calculate the sign from Fermi statistics that is not already included in the children.

```
let strip_fermion_lines = function
  | (Coupling.V3 _ | Coupling.V4 _ as v) → v
  | Coupling.Vn (Coupling.UFO (c, l, s, fl, col), f, x) →
      Coupling.Vn (Coupling.UFO (c, l, s, [], col), f, x)

let num_fermion_lines_v3 = function
  | Coupling.FBF _ | Coupling.PBP _ | Coupling.BBB _ | Coupling.GBG _ → 1
  | _ → 0

let num_fermion_lines = function
  | Coupling.Vn (Coupling.UFO (c, l, s, fl, col), f, x) → List.length fl
  | Coupling.V3 (v3, _, _) → num_fermion_lines_v3 v3
```

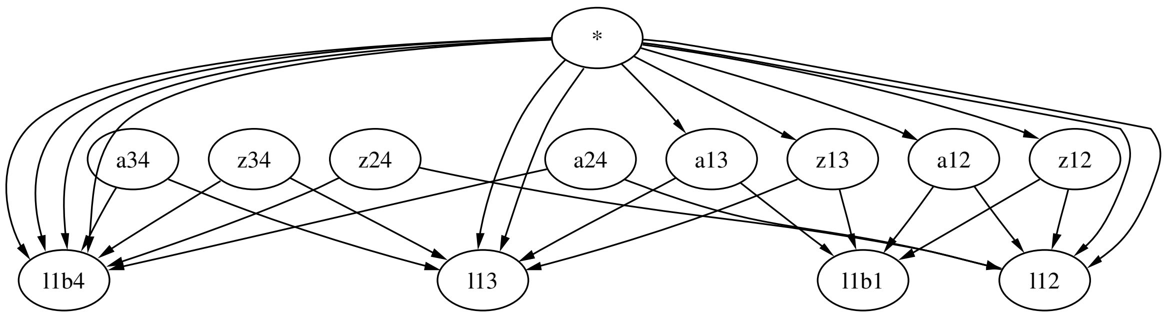
```
    | Coupling.V4 _ → 0

  let stat_keystone v stats wf1 wfs =
    let wf1' = stats wf1
    and wfs' = PT.map stats wfs in
    let f = A.flavor wf1 in
    let slist = wf1' :: PT.to_list wfs' in
    let stat = S.stat_keystone (fermion_lines v) slist f in
    (* We can compare with the legacy implementation only if there are no fermion line ambiguities possible,
i. e. for at most one line. *)
    if num_fermion_lines v < 2 then
      begin
        let legacy = S.stat_keystone None slist f in
        if ¬ (S.equal stat legacy) then
          failwith
            (Printf.sprintf
               "Fusion.stat_keystone:␣%s␣<>␣%s!"
               (S.stat_to_string legacy)
               (S.stat_to_string stat));
        if ¬ (S.saturated legacy) then
          failwith
            (Printf.sprintf
               "Fusion.stat_keystone:␣legacy␣incomplete:␣%s!"
               (S.stat_to_string legacy))
      end;
    if ¬ (S.saturated stat) then
      failwith
        (Printf.sprintf
           "Fusion.stat_keystone:␣incomplete:␣%s!"
           (S.stat_to_string stat));
    stat_sign stat
      × PT.fold_left (fun acc wf → acc × stat_sign wf) (stat_sign wf1') wfs'

  let stat_keystone_logging v stats wf1 wfs =
    let sign = stat_keystone v stats wf1 wfs in
    Printf.eprintf
      "Fusion.stat_keystone:␣%s␣*␣%s␣->␣%d\n"
      (M.flavor_to_string (A.flavor wf1))
      (ThoList.to_string
         (fun wf → M.flavor_to_string (A.flavor wf))
         (PT.to_list wfs))
      sign;
    sign
```

Test all members of a list of wave functions are defined by the DAG simultaneously:

```
  let test_rhs dag (_, wfs) =
    PT.for_all (fun wf → is_source wf ∧ A.D.is_node wf dag) wfs
```

Add the keystone (*wf1*, *pairs*) to *acc* only if it is present in *dag* and calculate the statistical factor depending on *stats en passant*:

```
  let filter_keystone stats dag (wf1, pairs) acc =
    if is_source wf1 ∧ A.D.is_node wf1 dag then
      match List.filter (test_rhs dag) pairs with
      | [] → acc
      | pairs' → (wf1, List.map (fun (c, wfs) →
          ({ A.sign = stat_keystone c stats wf1 wfs;
             A.coupling = c },
           wfs)) pairs') :: acc
    else
      acc
```



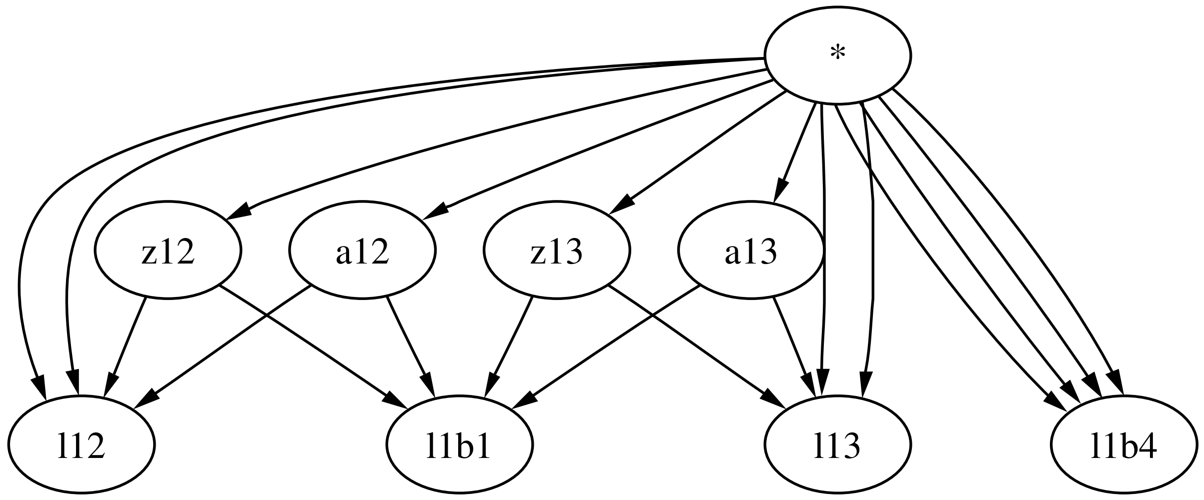


Figure 15.2: The DAGs for Bhabha scattering before and after weeding out unused nodes. The blatant asymmetry of these DAGs is caused by our prescription for removing doubling counting for an even number of external lines.

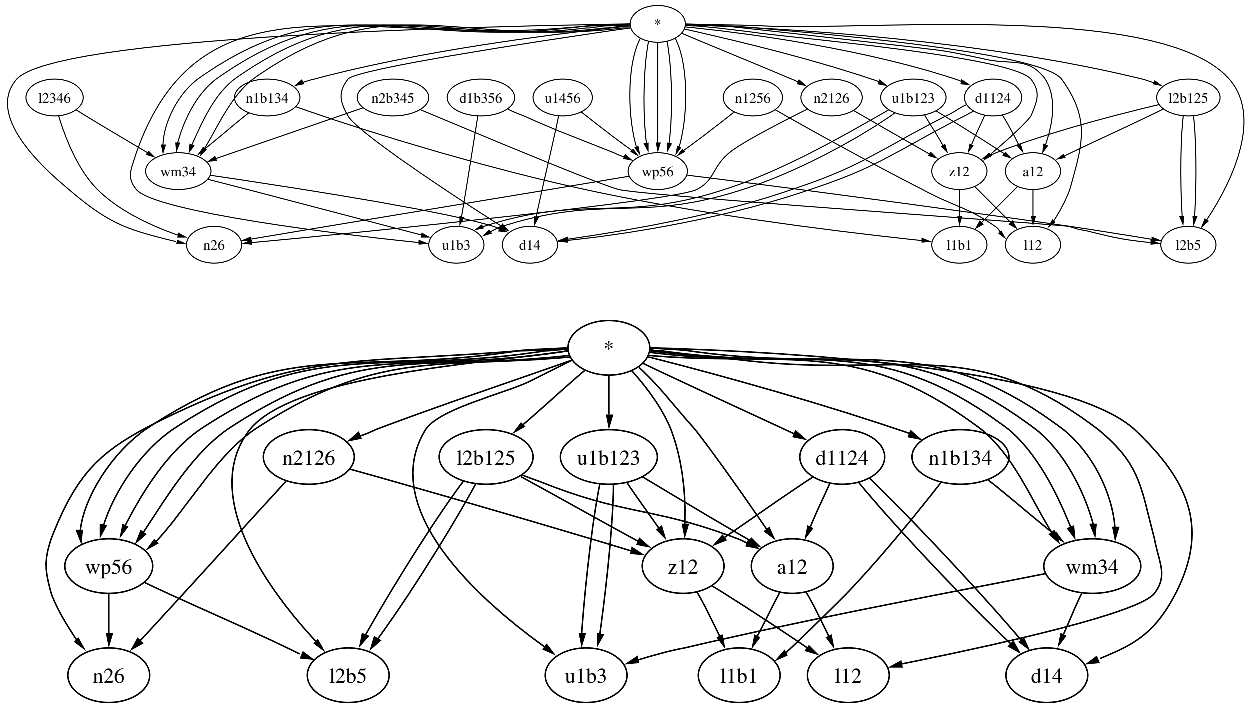


Figure 15.3: The DAGs for $e^+e^- \to u\bar{d}\mu^-\bar{\nu}_\mu$ before and after weeding out unused nodes.

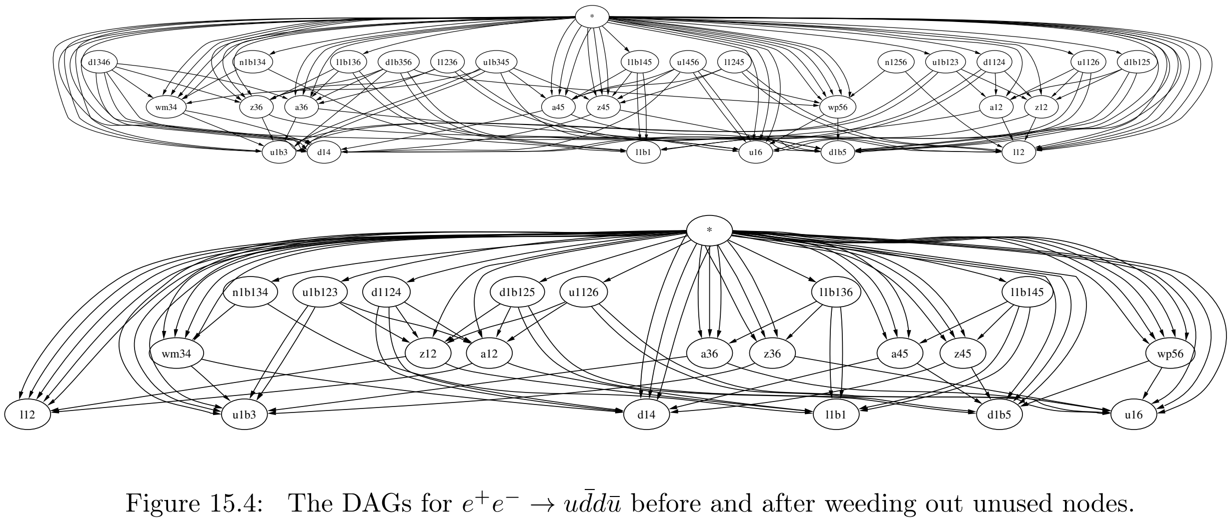


Figure 15.4: The DAGs for $e^+e^- \to u\bar{d}d\bar{u}$ before and after weeding out unused nodes.

Amplitudes

```
module C = Cascade.Make(M)(P)
type selectors = C.selectors
type slicings = Orders.Conditions(Colorize.It(M)).t

let external_wfs n particles =
  List.map (fun (f, p) →
    ({ A.flavor = f;
        A.momentum = P.singleton n p },
     stat f p)) particles
```

Main Function

```
module WFMap = Map.Make(WF)
```

This is the main function that constructs the amplitude for sets of incoming and outgoing particles and returns the results in conveniently packaged pieces.

```
let amplitude goldstones selectors fin fout =
```

Set up external lines and match flavors with numbered momenta.

```
  let f = fin @ List.map M.conjugate fout in
  let nin, nout = List.length fin, List.length fout in
  let n = nin + nout in
  let externals = List.combine f (ThoList.range 1 n) in
  let wfs = external_wfs n externals in
  let select_p = C.select_p selectors in
  let select_wf =
    match fin with
    | [_] → C.select_wf selectors P.Decay.timelike
    | _ → C.select_wf selectors P.Scattering.timelike in
  let select_vtx = C.select_vtx selectors in
```

Build the full fusion tower (including nodes that are never needed in the amplitude).

```
  let stats, tower =
    if goldstones then
      complete_fusion_tower select_wf select_vtx wfs
    else
      minimal_fusion_tower n select_wf select_vtx wfs in
```

Find all vertices for which *all* off shell wavefunctions are defined by the tower.

```
  let brakets =
    flavor_keystones (filter_keystone stats tower) select_p n
      (filter_vertices select_vtx
          (vertices (min n (M.max_degree ())) (M.flavors ())))
      (T.keystones (ThoList.range 1 n)) in
```

Remove the part of the DAG that is never needed in the amplitude.

```
  let dag =
    if goldstones then
      tower
    else
      A.D.harvest_list tower (A.wavefunctions brakets) in
```

Remove the leaf nodes of the DAG, corresponding to external lines.

```
  let fusions =
    List.filter (function (_, []) → false | _ → true) (A.D.lists dag) in
```

Calculate the symmetry factor for identical particles in the final state.

```
  let symmetry =
    Combinatorics.symmetry fout in
  let dependencies_map =
    A.D.fold (fun wf _ → WFMap.add wf (A.D.dependencies dag wf)) dag WFMap.empty in
```

Finally: package the results:

```
    { A.fusions = fusions;
      A.brakets = brakets;
      A.on_shell = (fun wf → C.on_shell selectors (A.flavor wf) wf.A.momentum);
      A.is_gauss = (fun wf → C.is_gauss selectors (A.flavor wf) wf.A.momentum);
      A.constraints = C.description selectors;
      A.slicings = [];
      A.incoming = fin;
      A.outgoing = fout;
      A.externals = List.map fst wfs;
      A.symmetry = symmetry;
      A.dependencies = (fun wf → WFMap.find wf dependencies_map);
      A.fusion_tower = tower;
      A.fusion_dag = dag }
```

Color

```
  module CM = Colorize.It(M)
  module CA = Amplitude(PT)(P)(CM)(Unsliced)

  let colorize_wf flavor wf =
    { CA.flavor = flavor;
      CA.momentum = wf.A.momentum }

  let uncolorize_wf wf =
    { A.flavor = CM.flavor_sans_color wf.CA.flavor;
      A.momentum = wf.CA.momentum }
```

At the end of the day, I shall want to have some sort of *fibered DAG* as abstract data type, with a projection of colored nodes to their uncolored counterparts.

```
  module CWFBundle = Bundle.Make
      (struct
        type elt = CA.wf
        let compare_elt = compare
        type base = A.wf
        let compare_base = compare
        let pi = uncolorize_wf
      end)
```

For now, we can live with simple aggregation:

```
  type fibered_dag = { dag : CA.D.t; bundle : CWFBundle.t }
```

O'Caml is perfectly able to infer the types of the following functions by itself, but it helps our understanding to spell them out explicitely and to introduce type abbreviations.
The function f : *wf_colorizer* takes a leaf wavefunction from the uncolored *DAG* and a *fibered_dag* and returns a colored node together with an updated bundle.

```
  type wf_colorizer = A.wf → fibered_dag → CA.wf × CWFBundle.t
```

colorize_sterile_nodes applies this function and adds the colored wavefunction to the colored *DAG*. Below, closures build from *colorize_sterile_nodes* will be passed to *A.D.fold_nodes* to lay the foundation for the colorized *DAG*.

```
  let colorize_sterile_nodes : A.D.t → wf_colorizer → A.wf → fibered_dag → fibered_dag =
    fun dag f wf fibered_dag →
    if A.D.is_sterile wf dag then
      let wf′, wf_bundle′ = f wf fibered_dag in
      { dag = CA.D.add_node wf′ fibered_dag.dag;
        bundle = wf_bundle′ }
    else
    fibered_dag
```

The function f : *node_colorizer* takes a fusion from the uncolored *DAG* and a *fibered_dag* and returns a list of colored fusions etc. together with an updated bundle.

```
type colored_fusion = CA.D.node × (CA.D.edge × CA.D.children)
type node_colorizer =
  A.D.node → A.D.edge × A.D.children → fibered_dag → colored_fusion list × CWFBundle.t
```

The colored fusions are added to the colored *DAG*. Below, closures build from *colorize_nodes* will be passed to *A.D.fold* to complete the construction of the colorized *DAG*.

```
let colorize_nodes : node_colorizer → A.wf → A.rhs → fibered_dag → fibered_dag =
  fun f wf rhs fibered_dag →
  let wf_rhs_list′, wf_bundle′ = f wf rhs fibered_dag in
  let dag′ =
    List.fold_right
      (fun (wf′, rhs′) → CA.D.add_offspring wf′ rhs′)
      wf_rhs_list′ fibered_dag.dag in
  { dag = dag′;
    bundle = wf_bundle′ }
```

Build a colorized *DAG* as a *fibered_dag* from an uncolored *DAG* growing the *wf_bundle*. In our applications, the initial *wf_bundle* will contain the colorized external wavefunctions.

```
let colorize_dag : node_colorizer → wf_colorizer → A.D.t → CWFBundle.t → fibered_dag =
  fun f_node f_ext dag wf_bundle →
  A.D.fold (colorize_nodes f_node) dag
    (A.D.fold_nodes (colorize_sterile_nodes dag f_ext) dag
       { dag = CA.D.empty; bundle = wf_bundle })
```

This is only a consistency check, verifying that the fiber of the *fibered_dag* that projects to *wf* contains one and only one element.

```
let colorize_external : wf_colorizer =
  fun wf fibered_dag →
  match CWFBundle.inv_pi fibered_dag.bundle wf with
  | [c_wf] → (c_wf, fibered_dag.bundle)
  | [] → failwith "colorize_external:␣not␣found"
  | _ → failwith "colorize_external:␣not␣unique"
```

Take the wavefunctions in the *rhs* and compute all colored fusions according to the colored Feynman rules. Keep only the flavors that match *wf* without colors and apply the *kmatrix_cuts* filter if necessary. While this ist color independent, it must be done again, because *CM.fuse* will reintroduce all couplings that might have been filtered out before.

```
let fuse_c_wf : A.wf → CA.wf CA.children → (CM.flavor × CM.constant Coupling.t) list =
  fun wf rhs →
  let momenta = PT.map (fun wf → wf.CA.momentum) rhs in
  List.filter
    (fun (f, c) →
       CM.flavor_sans_color f = wf.A.flavor ∧ kmatrix_cuts c momenta)
    (CM.fuse (List.map (fun wf → wf.CA.flavor) (PT.to_list rhs)))

let fuse_c_wf_logging wf rhs =
  let fusion = fuse_c_wf wf rhs in
  Printf.eprintf
    "fuse_c_wf␣%s(%s)␣%s␣=>␣%s\n"
    (M.flavor_to_string wf.A.flavor)
    (ThoList.to_string string_of_int (P.to_ints wf.A.momentum))
    (ThoList.to_string
       (fun wf →
          Printf.sprintf "%s(%s)"
            (CM.flavor_to_string wf.CA.flavor)
            (ThoList.to_string string_of_int (P.to_ints wf.CA.momentum)))
       (PT.to_list rhs))
    (ThoList.to_string (fun (f, _) → CM.flavor_to_string f) fusion);
  fusion

let colorize_coupling c coupling =
  { CA.sign = coupling.A.sign;
    CA.coupling = c }
```

Look up all colored versions of the *children* in the *fibered_dag*.

```
let find_colored fibered_dag wf =
  CWFBundle.inv_pi fibered_dag.bundle wf
```

All combinations of colored versions of the *children*.

```
let colored_children_list fibered_dag children =
  PT.product (PT.map (find_colored fibered_dag) children)
```

colorize_fusion wf rhs fibered_dag uses all colored versions of the wave functions on the *rhs* in the *fibered_dag* and returns all fusions (according to *fuse_c_wf*) with matching flavor together with the updated *fibered_dag*, including the new colored wave functions.

```
let match_flavor f' (f, _) =
  CM.flavor_sans_color f = f'

let colorize_fusion : node_colorizer =
  fun wf (coupling, children) fibered_dag →
  let fuse colored_children = fuse_c_wf wf colored_children
  and colorize colored_children (f, c) =
    (colorize_wf f wf, (colorize_coupling c coupling, colored_children)) in
  let fusions =
    ThoList.flatmap
      (fun colored_children →
        List.map (colorize colored_children) (fuse colored_children))
      (colored_children_list fibered_dag children) in
  let bundle =
    List.fold_left
      (fun acc (c_wf, _) → CWFBundle.add acc c_wf)
      fibered_dag.bundle fusions in
  (fusions, bundle)
```

Since each *PArray.Alist.t* has a unique representation, we can write *CM.conjugate bra.CA.flavor* = *f* instead of *CM.flavor_equal* (*CM.conjugate bra.CA.flavor*) *f* again.
Note that we must only keep the bras and kets with matching colors.

TODO: avoid building intermediate lists that must be factorized again using the approach for coupling orders slicing below.

```
let colorize_braket1 fibered_dag wf (coupling, children) =
  Product.fold2
    (fun bra ket acc →
      let bra_bar = uncolorize_wf (CA.conjugate bra) in
      List.fold_left
        (fun brakets (f, c) →
          if CM.conjugate bra.CA.flavor = f then
            (bra, (colorize_coupling c coupling, ket)) :: brakets
          else
            brakets)
        acc (fuse_c_wf bra_bar ket))
    (find_colored fibered_dag wf) (PT.product (PT.map (find_colored fibered_dag) children)) []

module CWF = struct type t = CA.wf let compare = CA.order_wf end
module CRHS = struct type t = CA.rhs let compare = compare end
module CWFSet = Set.Make(CWF)
module CWFMap = Map.Make(CWF)
module CRHSMap = ThoMap.Buckets(CWF)(CRHS)
```

CRHSMap.factorize takes a list of (*bra*, *ket*) pairs and groups the *ket*s according to *bra*. This is very similar to *ThoList.factorize* on page 683, but the latter keeps duplicate copies, while we keep only one, with equality determined by *CA.order_wf*.

```
let colorize_braket fibered_dag (wf, rhs_list) =
  CRHSMap.factorize_batches (List.map (colorize_braket1 fibered_dag wf) rhs_list)
```

colorize_amplitude a fin fout takes an amplitude *a* for uncolored particles and colored incoming particles *fin* and outgoing particles *fout* and returns the corresponding colored amplitude.

```
let colorize_amplitude a fin fout =
  let f = fin @ List.map CM.conjugate fout in
  let nin, nout = List.length fin, List.length fout in
  let n = nin + nout in
  let externals = List.combine f (ThoList.range 1 n) in
  let external_wfs = CA.external_wfs n externals in
  let wf_bundle = CWFBundle.of_list external_wfs in
  let fibered_dag = colorize_dag colorize_fusion colorize_external a.A.fusion_dag wf_bundle in
  let brakets = ThoList.flatmap (colorize_braket fibered_dag) a.A.brakets in
  let dag = CA.D.harvest_list fibered_dag.dag (CA.wavefunctions brakets) in
  let fusions = List.filter (function (_, []) → false | _ → true) (CA.D.lists dag) in
  let dependencies_map =
    CA.D.fold (fun wf _ → CWFMap.add wf (CA.D.dependencies dag wf)) dag CWFMap.empty in
  { CA.fusions = fusions;
    CA.brakets = brakets;
    CA.constraints = a.A.constraints;
    CA.slicings = a.A.slicings;
    CA.incoming = fin;
    CA.outgoing = fout;
    CA.externals = external_wfs;
    CA.fusion_dag = dag;
    CA.fusion_tower = dag;
    CA.symmetry = a.A.symmetry;
    CA.on_shell = (fun wf → a.A.on_shell (uncolorize_wf wf));
    CA.is_gauss = (fun wf → a.A.is_gauss (uncolorize_wf wf));
    CA.dependencies = (fun wf → CWFMap.find wf dependencies_map) }

let colorize_amplitudes a =
  List.fold_left
    (fun amps (fin, fout) →
      let amp = colorize_amplitude a fin fout in
      match amp.CA.brakets with
      | [] → amps
      | _ → amp :: amps)
    [] (CM.amplitude a.A.incoming a.A.outgoing)

let amplitudes_unsliced goldstones selectors fin fout =
  colorize_amplitudes (amplitude goldstones selectors fin fout)

let amplitude_sans_color goldstones selectors fin fout =
  amplitude goldstones selectors fin fout
```

Coupling Order Slicing

The following is structurally rather similar to the application of *Colorize.It*() above. Unfortunately, there are enough differences that will make a unification rather complicated.
Unfortunately, the O'Caml type checker insists on *Orders.Conditions*(*Colorize.It*(*M*)) here and everywhere. The more concise and superficially equivalent *Orders.Conditions*(*CM*) will lead to type errors down the road, when the *Fusion.Make* functor is applied. The problem appears to be that *CM* is not available in the type constraints for the functors.
The prefix *SC* to these and the following modules should be read as "sliced-colorized" or "colorized and sliced":

```
module COC = Orders.Conditions(Colorize.It(M))
module SCM = Orders.Slice(Colorize.It(M))

module By_Orders =
  struct
    type orders = SCM.orders
    type α t = (orders × α) list
    let all a = [([], a)]
  end

module SCA = Amplitude(PT)(P)(SCM)(By_Orders)
type α slices = α SCA.slices
type amplitude = SCA.t

let slice_wf flavor wf =
  { SCA.flavor = flavor;
    SCA.momentum = wf.CA.momentum }

let unslice_wf wf =
  { CA.flavor = SCM.flavor_all_orders wf.SCA.flavor;
    CA.momentum = wf.SCA.momentum }

module SCWF = struct type t = SCA.wf let compare = SCA.order_wf end
module SCWFSet = Set.Make(SCWF)

module SCWFBundle = Bundle.Make
    (struct
      type elt = SCA.wf
      let compare_elt = compare
      type base = CA.wf
      let compare_base = compare
      let pi = unslice_wf
    end)

let allowed amplitude =
  match amplitude.SCA.brakets with
  | [] → false
  | _ → true

type flavor = SCA.flavor
type flavor_all_orders = CA.flavor
type flavor_sans_color = A.flavor
type p = A.p
type wf = SCA.wf
let conjugate = SCA.conjugate
let flavor = SCA.flavor
let flavor_sans_color wf = CM.flavor_sans_color (SCM.flavor_all_orders (SCA.flavor wf))
let momentum = SCA.momentum
let momentum_list = SCA.momentum_list

type coupling = SCA.coupling

let sign = SCA.sign
let coupling = SCA.coupling

type α children = α SCA.children
type rhs = SCA.rhs
let children = SCA.children

type fusion = SCA.fusion
let lhs = SCA.lhs
let rhs = SCA.rhs

type braket = SCA.braket
let bra = SCA.bra
let ket = SCA.ket

type amplitude_sans_color = A.t
```

Accessor Functions

```
let incoming = SCA.incoming
let outgoing = SCA.outgoing
let externals = SCA.externals
let fusions = SCA.fusions
let brakets = SCA.brakets
let symmetry = SCA.symmetry
let on_shell = SCA.on_shell
let is_gauss = SCA.is_gauss
let constraints = SCA.constraints
let slicings = SCA.slicings
let variables a = List.map lhs (fusions a)
let dependencies = SCA.dependencies

let flavor_all_orders wf = SCM.flavor_all_orders (SCA.flavor wf)

type sliced_fibered_dag =
  { sliced_dag : SCA.D.t; sliced_bundle : SCWFBundle.t }

type wf_slicer = CA.wf → sliced_fibered_dag → SCA.wf × SCWFBundle.t

let slice_sterile_nodes : CA.D.t → wf_slicer → CA.D.node → sliced_fibered_dag → sliced_fibered_dag =
  fun dag f wf fibered_dag →
  if CA.D.is_sterile wf dag then
    let wf', wf_bundle' = f wf fibered_dag in
    { sliced_dag = SCA.D.add_node wf' fibered_dag.sliced_dag;
      sliced_bundle = wf_bundle' }
  else
    fibered_dag

type sliced_fusion = SCA.wf × SCA.rhs
type node_slicer = CA.wf → CA.rhs → sliced_fibered_dag → sliced_fusion list × SCWFBundle.t

let slice_nodes : node_slicer → CA.wf → CA.rhs → sliced_fibered_dag → sliced_fibered_dag =
  fun f wf rhs fibered_dag →
  let wf_rhs_list', wf_bundle' = f wf rhs fibered_dag in
  let dag' =
    List.fold_right
      (fun (wf', rhs') → SCA.D.add_offspring wf' rhs')
      wf_rhs_list' fibered_dag.sliced_dag in
  { sliced_dag = dag';
    sliced_bundle = wf_bundle' }

let slice_dag : node_slicer → wf_slicer → CA.D.t → SCWFBundle.t → sliced_fibered_dag =
  fun f_node f_ext dag wf_bundle →
  CA.D.fold (slice_nodes f_node) dag
    (CA.D.fold_nodes (slice_sterile_nodes dag f_ext) dag
       { sliced_dag = SCA.D.empty; sliced_bundle = wf_bundle })

let slice_external : wf_slicer =
  fun wf fibered_dag →
  match SCWFBundle.inv_pi fibered_dag.sliced_bundle wf with
  | [c_wf] → (c_wf, fibered_dag.sliced_bundle)
  | [] → failwith "slice_external:␣not␣found"
  | _ → failwith "slice_external:␣not␣unique"

let coupling_orders = function
  | Coupling.V3 (_, _, c) | Coupling.V4 (_, _, c) | Coupling.Vn (_, _, c) →
      CM.coupling_orders c

let coupling_orders_to_string co =
  "{" ^
    String.concat ","
      (List.map (fun (o, n) → CM.coupling_order_to_string o ^ ":" ^ string_of_int n) co) ^ "}"
```

Ideally, one would want to test for the allowed coupling constants with *COC.constant* early inside of *SCM.fuse*. However, this requires a more general signature than *fuse* in *Model.T*. Let's see if this is worth the effort.

```
let fuse_s_wf : COC.t → CA.wf → SCA.wf SCA.children → (SCM.flavor × SCM.constant Coupling.t) list =
  fun slicings wf rhs →
  let momenta = PT.map (fun wf → wf.SCA.momentum) rhs in
  List.filter
    (fun (f, c) →
      SCM.flavor_all_orders f = wf.CA.flavor
      ∧ COC.constant slicings (coupling_orders c)
      ∧ COC.fusion slicings (SCM.orders f)
      ∧ kmatrix_cuts c momenta)
    (SCM.fuse (List.map (fun wf → wf.SCA.flavor) (PT.to_list rhs)))

let slice_coupling c coupling =
  { SCA.sign = coupling.CA.sign;
    SCA.coupling = c }
```

Look up all versions of the *children* in the *fibered_dag*.

```
let find_sliced fibered_dag wf =
  SCWFBundle.inv_pi fibered_dag.sliced_bundle wf
```

All combinations of the *children* with different coupling orders.

```
let sliced_children_list fibered_dag children =
  PT.product (PT.map (find_sliced fibered_dag) children)

let slice_fusion : COC.t → node_slicer =
  fun slicings wf (coupling, children) fibered_dag →
  let fuse sliced_children = fuse_s_wf slicings wf sliced_children
  and slice sliced_children (f, c) =
    (slice_wf f wf, (slice_coupling c coupling, sliced_children)) in
  let fusions =
    ThoList.flatmap
      (fun sliced_children →
        List.map (slice sliced_children) (fuse sliced_children))
      (sliced_children_list fibered_dag children) in
  let bundle =
    List.fold_left
      (fun acc (s_wf, _) → SCWFBundle.add acc s_wf)
      fibered_dag.sliced_bundle fusions in
  (fusions, bundle)
```

When producing all combinations of coupling orders, bras and kets, we need to group them by common coupling orders and by common bras. This is most straightforwardly (and asymptotically efficiently) done by constructing a map from coupling orders to maps from bras to sets of kets.
For this we need to order the sets of coupling orders, bras (wave functions) and kets (right hand sides)

```
module CO = struct type t = SCM.orders let compare = compare end
module SCBra = struct type t = SCA.wf let compare = SCA.order_wf end
module SCKet = struct type t = SCA.rhs let compare = compare end
```

in order to define maps from coupling orders and from bras

```
module COMap = Map.Make(CO)
module SCBraMap = Map.Make(SCBra)
```

as well a buckets for kets, indexed by bras:

```
module SCKetBuckets = ThoMap.Buckets(SCBra)(SCKet)
type comap = SCKetBuckets.t COMap.t

let comap_to_lists : comap → (SCM.orders × SCA.braket list) list =
  fun comap →
  List.rev (COMap.fold (fun orders brakets acc → (orders, SCKetBuckets.to_lists brakets) :: acc) comap [])
```

Add *ket* to the set indexed by *bra* in the map from bras to sets of kets indexed by *orders* in *omap*. Initialize the inner map if it doesn't exist yet.

```
let addto_orders_map : comap → SCM.orders → SCA.wf → SCA.rhs → comap =
  fun omap orders bra ket →
```

```
    let bra_ket_map =
      match COMap.find_opt orders omap with
      | None → SCKetBuckets.empty
      | Some bkmap → bkmap in
    COMap.add orders (SCKetBuckets.add bra ket bra_ket_map) omap

  let _find_sliced fibered_dag wf =
    let wf_list = find_sliced fibered_dag wf in
    Printf.eprintf "find_sliced␣%s␣->␣%s\n"
      (CM.flavor_to_string (CA.flavor wf))
      (ThoList.to_string
         (fun wf → SCM.flavor_to_string (SCA.flavor wf))
         wf_list);
    wf_list
```

Take a left hand side and a right hand side, construct all allowed combinations of coupling orders and add them to our collection.

```
  let slice_braket1 : COC.t → sliced_fibered_dag → CA.wf → CA.rhs → comap → comap =
    fun conditions fibered_dag wf (coupling, children) comap →
    Product.fold2
      (fun bra children comap →
        let bra_bar = unslice_wf (SCA.conjugate bra) in
        List.fold_left
          (fun comap (f, c) →
            let orders = SCM.add_orders (SCM.orders bra.SCA.flavor) (SCM.orders f) in
            match COC.braket conditions orders with
            | Some orders → addto_orders_map comap orders bra (slice_coupling c coupling, children)
            | None → comap)
          comap (fuse_s_wf conditions bra_bar children))
      (find_sliced fibered_dag wf) (PT.product (PT.map (find_sliced fibered_dag) children)) comap

  let slice_braket : COC.t → sliced_fibered_dag → CA.braket → comap → comap =
    fun slicings fibered_dag (wf, rhs_list) comap →
    List.fold_right (slice_braket1 slicings fibered_dag wf) rhs_list comap

  let slice_brakets : COC.t → sliced_fibered_dag → CA.braket list → (SCM.orders × SCA.braket list) list =
    fun slicings fibered_dag brakets →
    comap_to_lists (List.fold_right (slice_braket slicings fibered_dag) brakets COMap.empty)

  let slice_amplitude slicings a =
    let trivial = List.map (fun co → (co, 0)) (COC.exclusive_fusion slicings) in
    let fin, fout = SCM.amplitude trivial a.CA.incoming a.CA.outgoing in
    let f = fin @ List.map SCM.conjugate fout in
    let nin, nout = List.length fin, List.length fout in
    let n = nin + nout in
    let externals = List.combine f (ThoList.range 1 n) in
    let external_wfs = SCA.external_wfs n externals in
    let wf_bundle = SCWFBundle.of_list external_wfs in
    let fibered_dag = slice_dag (slice_fusion slicings) slice_external a.CA.fusion_dag wf_bundle in
    let sliced_brakets = slice_brakets slicings fibered_dag a.CA.brakets in
    let brakets = ThoList.flatmap snd sliced_brakets in
    let dag = SCA.D.harvest_list fibered_dag.sliced_dag (SCA.wavefunctions brakets) in
    let fusions = List.filter (function (_, []) → false | _ → true) (SCA.D.lists dag) in
    let dependencies_map =
      SCA.D.fold (fun wf _ → SCBraMap.add wf (SCA.D.dependencies dag wf)) dag SCBraMap.empty in
    { SCA.fusions = fusions;
      SCA.brakets = sliced_brakets;
      SCA.constraints = a.CA.constraints;
      SCA.slicings = COC.to_strings slicings;
      SCA.incoming = fin;
      SCA.outgoing = fout;
      SCA.externals = external_wfs;
      SCA.fusion_dag = dag;
```

```
        SCA.fusion_tower = dag;
        SCA.symmetry = a.CA.symmetry;
        SCA.on_shell = (fun wf → a.CA.on_shell (unslice_wf wf));
        SCA.is_gauss = (fun wf → a.CA.is_gauss (unslice_wf wf));
        SCA.dependencies = (fun wf → SCBraMap.find wf dependencies_map) }

  let slice_amplitudes slicings amplitudes =
    List.map (slice_amplitude slicings) amplitudes
```

For the benefit of *Targets*, we also copy the amplitudes to equivalent sliced amplitudes with empty coupling orders. This way, we can use the same output routines for the sliced and unsliced amplitudes.
lift_amplitude is equivalent to *slice_amplitude Orders.Condition.trivial*, but it can shortcut *SCM.fuse*, since all fusions and brakets are known.

```
  let lift_wf wf =
    slice_wf (SCM.trivial wf.CA.flavor) wf

  let lift_coupling coupling =
    { SCA.sign = coupling.CA.sign;
      SCA.coupling = coupling.CA.coupling }

  let lift_external : wf_slicer =
    fun wf fibered_dag →
    (lift_wf wf, fibered_dag.sliced_bundle)

  let lift_fusion : node_slicer =
    fun wf (coupling, children) fibered_dag →
    let wf = lift_wf wf
    and coupling = lift_coupling coupling
    and children = PT.map lift_wf children in
    let sliced_bundle = SCWFBundle.add fibered_dag.sliced_bundle wf in
    ( [ (wf, (coupling, children)) ], sliced_bundle )

  let lift_dag : CA.D.t → SCWFBundle.t → sliced_fibered_dag =
    fun dag wf_bundle →
    slice_dag lift_fusion lift_external dag wf_bundle

  let lift_braket : CA.braket → SCA.braket =
    fun (wf, rhs) →
    let wf = lift_wf wf
    and rhs =
      List.map
        (fun (coupling, children) → (lift_coupling coupling, PT.map lift_wf children))
        rhs in
    (wf, rhs)

  let lift_amplitude a =
    let fin = List.map SCM.trivial a.CA.incoming
    and fout = List.map SCM.trivial a.CA.outgoing in
    let f = fin @ List.map SCM.conjugate fout in
    let nin, nout = List.length fin, List.length fout in
    let n = nin + nout in
    let externals = List.combine f (ThoList.range 1 n) in
    let external_wfs = SCA.external_wfs n externals in
    let wf_bundle = SCWFBundle.of_list external_wfs in
    let fibered_dag = lift_dag a.CA.fusion_dag wf_bundle in
    let brakets = List.map lift_braket a.CA.brakets in
    let dag = SCA.D.harvest_list fibered_dag.sliced_dag (SCA.wavefunctions brakets) in
    let fusions = List.filter (function (_, []) → false | _ → true) (SCA.D.lists dag) in
    let dependencies_map =
      SCA.D.fold (fun wf _ → SCBraMap.add wf (SCA.D.dependencies dag wf)) dag SCBraMap.empty in
    { SCA.fusions = fusions;
      SCA.brakets = SCA.unsliced brakets;
      SCA.constraints = a.CA.constraints;
      SCA.slicings = [];
      SCA.incoming = fin;
      SCA.outgoing = fout;
      SCA.externals = external_wfs;
      SCA.fusion_dag = dag;
      SCA.fusion_tower = dag;
      SCA.symmetry = a.CA.symmetry;
      SCA.on_shell = (fun wf → a.CA.on_shell (unslice_wf wf));
      SCA.is_gauss = (fun wf → a.CA.is_gauss (unslice_wf wf));
      SCA.dependencies = (fun wf → SCBraMap.find wf dependencies_map) }

let lift_amplitudes amplitudes =
  List.map lift_amplitude amplitudes

let amplitudes goldstones selectors slicings fin fout =
  let a = amplitudes_unsliced goldstones selectors fin fout in
  match slicings with
  | None → lift_amplitudes a
  | Some slicings → slice_amplitudes slicings a

 let amplitudes_all_orders goldstones selectors fin fout =
   lift_amplitudes (amplitudes_unsliced goldstones selectors fin fout)

let children_to_string children =
  "(" ^
    String.concat "*"
      (List.map (fun wf → SCM.flavor_to_string (SCA.flavor wf)) children) ^ ")"

let dump_sliced_amplitudes slicings sliced =
  List.iter
    (fun amplitude →
      Printf.eprintf "amplitude␣%s␣->␣%s\n"
        (String.concat "␣" (List.map SCM.flavor_to_string amplitude.SCA.incoming))
        (String.concat "␣" (List.map SCM.flavor_to_string amplitude.SCA.outgoing));
      List.iter
        (fun (orders, brakets) →
          Printf.eprintf "␣␣order␣%s\n" (coupling_orders_to_string orders);
          List.iter
            (fun braket →
              Printf.eprintf
                "␣␣␣␣braket␣(%s,␣[%s])\n"
                (SCM.flavor_to_string (SCA.flavor (SCA.bra braket)))
                (String.concat ";"
                   (List.map
                      (fun ket →
                        coupling_orders_to_string (coupling_orders (SCA.coupling ket)) ^
                          children_to_string (SCA.children ket))
                      (SCA.ket braket))))
            brakets)
        amplitude.brakets)
    sliced

let _amplitudes goldstones selectors slicings fin fout =
  let a = amplitudes goldstones selectors slicings fin fout in
  match slicings with
  | None → a
  | Some slicings →
     dump_sliced_amplitudes slicings a;
     begin match COC.to_strings slicings with
     | [] → ()
     | slicings →
        Printf.eprintf "!!!!!!!!!!!!!!!!!!!!!!!!!!\n";
        Printf.eprintf "!␣coupling␣orders␣selected\n";
        List.iter (Printf.eprintf "!␣%s\n") slicings;
        Printf.eprintf "!!!!!!!!!!!!!!!!!!!!!!!!!!\n"
     end;
```

a

Checking Conservation Laws

```
let check_charges () =
  let vlist3, vlist4, vlistn = M.vertices () in
  List.filter
    (fun flist → ¬ (M.Ch.is_null (M.Ch.sum (List.map M.charges flist))))
    (List.map (fun ((f1, f2, f3), _, _) → [f1; f2; f3]) vlist3
     @ List.map (fun ((f1, f2, f3, f4), _, _) → [f1; f2; f3; f4]) vlist4
     @ List.map (fun (flist, _, _) → flist) vlistn)
```

Diagnostics

```
let all_brakets a =
  ThoList.flatmap snd a.SCA.brakets

let count_propagators a =
  List.length a.SCA.fusions

let count_fusions a =
  let brakets = all_brakets a in
  List.fold_left (fun n (_, a) → n + List.length a) 0 a.SCA.fusions
    + List.fold_left (fun n (_, t) → n + List.length t) 0 brakets
    + List.length brakets
```

This brute force approach blows up for more than ten particles. Find a smarter algorithm.

```
let count_diagrams a =
  List.fold_left (fun n (wf1, wf23) →
    n + SCA.D.count_trees wf1 a.SCA.fusion_dag ×
      (List.fold_left (fun n′ (_, wfs) →
        n′ + PT.fold_left (fun n″ wf →
          n″ × SCA.D.count_trees wf a.SCA.fusion_dag) 1 wfs) 0 wf23))
    0 (all_brakets a)

exception Impossible

let forest′ a =
  let below wf = SCA.D.forest_memoized wf a.SCA.fusion_dag in
  ThoList.flatmap
    (fun (bra, ket) →
      (Product.list2 (fun bra′ ket′ → bra′ :: ket′)
         (below bra)
         (ThoList.flatmap
            (fun (_, wfs) →
              Product.list (fun w → w) (PT.to_list (PT.map below wfs)))
            ket)))
    (all_brakets a)

let cross wf =
  { SCA.flavor = SCM.conjugate wf.SCA.flavor;
    SCA.momentum = P.neg wf.SCA.momentum }

let fuse_trees wf ts =
  Tree.fuse (fun (wf′, e) → (cross wf′, e))
    wf (fun t → List.mem wf (Tree.leafs t)) ts

let forest wf a =
  List.map (fuse_trees wf) (forest′ a)
```

There's a lot of redundancy here. This is not harmful, but very confusing and should be cleaned up.

```
let poles_beneath wf dag =
  SCA.D.eval_memoized (fun wf' → [[ ]])
    (fun wf' _ p → List.map (fun p' → wf' :: p') p)
    (fun wf1 wf2 →
      Product.fold2 (fun wf' wfs' wfs'' → (wf' @ wfs') :: wfs'') wf1 wf2 [ ])
    (@) [[ ]] [[ ]] wf dag

let poles a =
  ThoList.flatmap (fun (wf1, wf23) →
    let poles_wf1 = poles_beneath wf1 a.SCA.fusion_dag in
    (ThoList.flatmap (fun (_, wfs) →
      Product.list List.flatten
        (PT.to_list (PT.map (fun wf →
          poles_wf1 @ poles_beneath wf a.SCA.fusion_dag) wfs)))
       wf23))
    (all_brakets a)

let s_channel a =
  SCWFSet.elements
    (ThoList.fold_right2
       (fun wf wfs →
         if P.Scattering.timelike wf.SCA.momentum then
           SCWFSet.add wf wfs
         else
           wfs) (poles a) SCWFSet.empty)
```

This should be much faster! Is it correct? Is it faster indeed?

```
let poles' a =
  List.map SCA.lhs a.SCA.fusions

let s_channel a =
  SCWFSet.elements
    (List.fold_right
       (fun wf wfs →
         if P.Scattering.timelike wf.SCA.momentum then
           SCWFSet.add wf wfs
         else
           wfs) (poles' a) SCWFSet.empty)
```

Pictures

Export the DAG in the `dot(1)` file format so that we can draw pretty pictures to impress audiences . . .

```
let p2s p =
  if p ≥ 0 ∧ p ≤ 9 then
    string_of_int p
  else if p ≤ 36 then
    String.make 1 (Char.chr (Char.code 'A' + p − 10))
  else
    "_"

let variable wf =
  SCM.flavor_symbol wf.SCA.flavor ^
    "_p" ^ String.concat "" (List.map p2s (P.to_ints wf.SCA.momentum))

let add_to_list i n m =
  IMap.add i (n :: try IMap.find i m with Not_found → [ ]) m

let classify_nodes dag =
  IMap.fold (fun i n acc → (i, n) :: acc)
    (SCA.D.fold_nodes (fun wf → add_to_list (P.rank wf.SCA.momentum) wf)
       dag IMap.empty) [ ]

let dag_to_dot ch brakets dag =
  Printf.fprintf ch "digraph␣OMEGA␣{\n";
  SCA.D.iter_nodes (fun wf →
    Printf.fprintf ch "␣␣\"%s\"␣[␣label␣=␣\"%s\"␣];\n"
      (variable wf) (variable wf)) dag;
  List.iter (fun (_, wfs) →
    Printf.fprintf ch "␣␣{␣rank␣=␣same;";
    List.iter (fun n →
      Printf.fprintf ch "␣\"%s\";" (variable n)) wfs;
    Printf.fprintf ch "␣};\n") (classify_nodes dag);
  List.iter (fun n →
    Printf.fprintf ch "␣\"*\"␣->␣\"%s\";\n" (variable n))
    (flatten_keystones brakets);
  SCA.D.iter (fun n (_, ns) →
    let p = variable n in
    PT.iter (fun n' →
      Printf.fprintf ch "␣␣\"%s\"␣->␣\"%s\";\n" p (variable n')) ns) dag;
  Printf.fprintf ch "}\n"

let tower_to_dot ch a =
  dag_to_dot ch (all_brakets a) a.SCA.fusion_tower

let amplitude_to_dot ch a =
  dag_to_dot ch (all_brakets a) a.SCA.fusion_dag
```

Phasespace

```
  let variable wf =
    M.flavor_to_string wf.A.flavor ^
      "[" ^ String.concat "/" (List.map p2s (P.to_ints wf.A.momentum)) ^ "]"

  let below_to_channel transform ch dag wf =
    let n2s wf = variable (transform wf)
    and e2s c = "" in
    Tree2.to_channel ch n2s e2s (A.D.dependencies dag wf)

  let bra_to_channel transform ch dag wf =
    let tree = A.D.dependencies dag wf in
    if Tree2.is_singleton tree then
      let n2s wf = variable (transform wf)
      and e2s c = "" in
      Tree2.to_channel ch n2s e2s tree
    else
      failwith "Fusion.phase_space_channels:␣wrong␣topology!"

  let ket_to_channel transform ch dag ket =
    Printf.fprintf ch "(";
    begin match A.children ket with
    | [] → ()
    | [child] → below_to_channel transform ch dag child
    | child :: children →
        below_to_channel transform ch dag child;
        List.iter
          (fun child →
            Printf.fprintf ch ",";
            below_to_channel transform ch dag child)
          children
    end;
    Printf.fprintf ch ")"

  let phase_space_braket transform ch (bra, ket) dag =
    bra_to_channel transform ch dag bra;
    Printf.fprintf ch ":␣{";
    begin match ket with
```

```
      | [] → ()
      | [ket1] →
          Printf.fprintf ch "␣";
          ket_to_channel transform ch dag ket1
      | ket1 :: kets →
          Printf.fprintf ch "␣";
          ket_to_channel transform ch dag ket1;
          List.iter
            (fun k →
              Printf.fprintf ch "␣\\\n␣␣␣|␣";
              ket_to_channel transform ch dag k)
            kets
      end;
      Printf.fprintf ch "␣}\n"

    let phase_space_channels_transformed transform ch a =
      List.iter
        (fun braket → phase_space_braket transform ch braket a.A.fusion_dag)
        a.A.brakets

    let phase_space_channels ch a =
      phase_space_channels_transformed (fun wf → wf) ch a

    let exchange_momenta_list p1 p2 p =
      List.map
        (fun pi →
          if pi = p1 then
            p2
          else if pi = p2 then
            p1
          else
            pi)
        p

    let exchange_momenta p1 p2 p =
      P.of_ints (P.dim p) (exchange_momenta_list p1 p2 (P.to_ints p))

    let flip_momenta wf =
      { wf with A.momentum = exchange_momenta 1 2 wf.A.momentum }

    let phase_space_channels_flipped ch a =
      phase_space_channels_transformed flip_momenta ch a

  end

module Binary = Make(Tuple.Binary)(Stat_Dirac)(Topology.Binary)
```

15.2.5 Fusions with Majorana Fermions

```
let majorana_log silent logging = logging
let majorana_log silent logging = silent
let force_legacy = true
let force_legacy = false

module Stat_Majorana (M : Model.T) : (Stat with type flavor = M.flavor) =
  struct

    exception Impossible

    type flavor = M.flavor
```

Keeping Track of Fermion Lines

JRR's algorithm doesn't use lists of pairs representing directed arrows as in *Stat_Dirac*().*stat* above, but a list of integers denoting the external leg a fermion line connects to:

```
type stat =
  | Fermion of int × int list
  | AntiFermion of int × int list
  | Boson of int list
  | Majorana of int × int list

let sign_of_permutation lines = fst (Combinatorics.sort_signed lines)

let lines_equivalent l1 l2 =
  sign_of_permutation l1 = sign_of_permutation l2

let stat_to_string s =
  let open Printf in
  let l2s = ThoList.to_string string_of_int in
  match s with
  | Boson lines → sprintf "B%s" (l2s lines)
  | Fermion (p, lines) → sprintf "F(%d,␣%s)" p (l2s lines)
  | AntiFermion (p, lines) → sprintf "A(%d,␣%s)" p (l2s lines)
  | Majorana (p, lines) → sprintf "M(%d,␣%s)" p (l2s lines)
```

Writing all cases explicitely is tedious, but allows exhaustiveness checking.

```
let equal s1 s2 =
  match s1, s2 with
  | Boson l1, Boson l2 →
     lines_equivalent l1 l2
  | Majorana (p1, l1), Majorana (p2, l2)
  | Fermion (p1, l1), Fermion (p2, l2)
  | AntiFermion (p1, l1), AntiFermion (p2, l2) →
     p1 = p2 ∧ lines_equivalent l1 l2
  | Boson _, (Fermion _ | AntiFermion _ | Majorana _ )
  | (Fermion _ | AntiFermion _ | Majorana _ ), Boson _
  | Majorana _, (Fermion _ | AntiFermion _)
  | (Fermion _ | AntiFermion _), Majorana _
  | Fermion _ , AntiFermion _
  | AntiFermion _ , Fermion _ → false
```

The final amplitude must not be fermionic!

```
let saturated = function
  | Boson _ → true
  | Fermion _ | AntiFermion _ | Majorana _ → false
```

stat f p interprets the numeric fermion numbers of flavor f at external leg p at creates a leaf:

```
let stat f p =
  match M.fermion f with
  | 0 → Boson []
  | 1 → Fermion (p, [])
  | −1 → AntiFermion (p, [])
  | 2 → Majorana (p, [])
  | _ → invalid_arg "Fusion.Stat_Majorana:␣invalid␣fermion␣number"
```

The formalism of [7] does not distinguish spinors from conjugate spinors, it is only important to know in which direction a fermion line is calculated. So the sign is made by the calculation together with an aditional one due to the permuation of the pairs of endpoints of fermion lines in the direction they are calculated. We propose a "canonical" direction from the right to the left child at a fusion point so we only have to keep in mind which external particle hangs at each side. Therefore we need not to have a list of pairs of conjugate spinors and spinors but just a list in which the pairs are right-left-right-left and so on. Unfortunately it is unavoidable to have couplings with clashing arrows in supersymmetric theories so we need transmutations from fermions in antifermions and vice versa as well.

Merge Fermion Lines for Legacy Models with Implied Fermion Connections

In the legacy case with at most one fermion line, it was straight forward to determine the kind of outgoing line from the corresponding flavor. In the general case, it is not possible to maintain this constraint, when constructing the n-ary fusion from binary ones.

We can break up the process into two steps however: first perform unconstrained fusions pairwise . . .

```
let stat_fuse_pair_unconstrained s1 s2 =
  match s1, s2 with
  | Boson l1, Boson l2 → Boson (l1 @ l2)
  | (Majorana (p1, l1) | Fermion (p1, l1) | AntiFermion (p1, l1)),
    (Majorana (p2, l2) | Fermion (p2, l2) | AntiFermion (p2, l2)) →
      Boson ([p2; p1] @ l1 @ l2)
  | Boson l1, Majorana (p, l2) → Majorana (p, l1 @ l2)
  | Boson l1, Fermion (p, l2) → Fermion (p, l1 @ l2)
  | Boson l1, AntiFermion (p, l2) → AntiFermion (p, l1 @ l2)
  | Majorana (p, l1), Boson l2 → Majorana (p, l1 @ l2)
  | Fermion (p, l1), Boson l2 → Fermion (p, l1 @ l2)
  | AntiFermion (p, l1), Boson l2 → AntiFermion (p, l1 @ l2)
```

. . . and only apply the constraint to the outgoing leg.

```
let constrain_stat_fusion s f =
  match s, M.lorentz f with
  | (Majorana (p, l) | Fermion (p, l) | AntiFermion (p, l)),
    (Coupling.Majorana | Coupling.Vectorspinor | Coupling.Maj_Ghost) →
     Majorana (p, l)
  | (Majorana (p, l) | Fermion (p, l) | AntiFermion (p, l)),
    Coupling.Spinor → Fermion (p, l)
  | (Majorana (p, l) | Fermion (p, l) | AntiFermion (p, l)),
    Coupling.ConjSpinor → AntiFermion (p, l)
  | (Majorana _ | Fermion _ | AntiFermion _ as s),
    (Coupling.Scalar | Coupling.Vector | Coupling.Massive_Vector
     | Coupling.Tensor_1 | Coupling.Tensor_2 | Coupling.BRS _) →
     invalid_arg
       (Printf.sprintf
          "Fusion.stat_fuse_pair_constrained:␣expected␣boson,␣got␣%s"
          (stat_to_string s))
  | Boson l as s,
    (Coupling.Majorana | Coupling.Vectorspinor | Coupling.Maj_Ghost
     | Coupling.Spinor | Coupling.ConjSpinor) →
     invalid_arg
       (Printf.sprintf
          "Fusion.stat_fuse_pair_constrained:␣expected␣fermion,␣got␣%s"
          (stat_to_string s))
  | Boson l,
    (Coupling.Scalar | Coupling.Vector | Coupling.Massive_Vector
     | Coupling.Tensor_1 | Coupling.Tensor_2 | Coupling.BRS _) →
     Boson l

let stat_fuse_pair_legacy f s1 s2 =
  stat_fuse_pair_unconstrained s1 s2

let stat_fuse_pair_legacy_logging f s1 s2 =
  let stat = stat_fuse_pair_legacy f s1 s2 in
  Printf.eprintf
    "stat_fuse_pair_legacy:␣(%s,␣%s)␣->␣%s␣=␣%s\n"
    (stat_to_string s1) (stat_to_string s2) (stat_to_string stat)
    (M.flavor_to_string f);
  stat

let stat_fuse_pair_legacy =
  majorana_log stat_fuse_pair_legacy stat_fuse_pair_legacy_logging
```

Note that we are using *List.fold_left*, therefore we perform the fusions as $f(f(\ldots(f(s_1, s_2), s_3), \ldots), s_n)$. Had we used *List.fold_right* instead, we would compute $f(s_1, f(s_2, \ldots f(s_{n-1}, s_n)))$. For our Dirac algorithm, this makes no difference, but JRR's Majorana algorithm depends on the order!
Also not that we *must not* apply *constrain_stat_fusion* here, because *stat_fuse_legacy* will be used in *stat_keystone_legacy* again, where we always expect *Boson _*.

```
let stat_fuse_legacy s1 s23__n f =
  List.fold_left (stat_fuse_pair_legacy f) s1 s23__n

let stat_fuse_legacy_logging s1 s23__n f =
  let stat = stat_fuse_legacy s1 s23__n f in
  Printf.eprintf
    "stat_fuse_legacy:␣␣␣␣␣␣%s␣->␣%s␣=␣%s\n"
    (ThoList.to_string stat_to_string (s1 :: s23__n))
    (stat_to_string stat)
    (M.flavor_to_string f);
  stat

let stat_fuse_legacy =
  majorana_log stat_fuse_legacy stat_fuse_legacy_logging
```

Merge Fermion Lines using Explicit Fermion Connections

Partially combined *stat*s of the incoming propagators and keeping track of the fermion lines, while we're scanning them.

```
type partial =
  { stat : stat (* the stat accumulated so far *);
    fermions : int IMap.t (* a map from the indices in the vertex to open (anti)fermion lines *);
    n : int (* the number of incoming propagators *) }
```

We will perform two passes:

1. collect the saturated fermion lines in a *Boson*, while building a map from the indices in the vertex to the open fermion lines
2. connect the open fermion lines using the *int* → *int* map *fermions*.

```
let empty_partial =
  { stat = Boson [];
    fermions = IMap.empty;
    n = 0 }
```

Only for debugging:

```
let partial_to_string p =
  Printf.sprintf
    "{␣fermions=%s,␣stat=%s,␣#=%d␣}"
    (ThoList.to_string
       (fun (i, particle) → Printf.sprintf "%d@%d" particle i)
       (IMap.bindings p.fermions))
    (stat_to_string p.stat)
    p.n
```

Add a list of saturated fermion lines at the top of the list of lines in a *stat*.

```
let add_lines l = function
  | Boson l' → Boson (l @ l')
  | Fermion (n, l') → Fermion (n, l @ l')
  | AntiFermion (n, l') → AntiFermion (n, l @ l')
  | Majorana (n, l') → Majorana (n, l @ l')
```

Process one line in the first pass: add the saturated fermion lines to the partial stat *p.stat* and add a pointer to an open fermion line in case of a fermion.

```
let add_lines_to_partial p stat =
  let n = succ p.n in
  match stat with
  | Boson l →
     { fermions = p.fermions;
       stat = add_lines l p.stat;
       n }
```

```
    | Majorana (f, l) →
       { fermions = IMap.add n f p.fermions;
         stat = add_lines l p.stat;
         n }
    | Fermion (p, l) →
       invalid_arg
         "add_lines_to_partial:␣unexpected␣Fermion"
    | AntiFermion (p, l) →
       invalid_arg
         "add_lines_to_partial:␣unexpected␣AntiFermion"
```

Do it for all lines:

```
let partial_of_slist stat_list =
  List.fold_left add_lines_to_partial empty_partial stat_list

let partial_of_rev_slist stat_list =
  List.fold_left add_lines_to_partial empty_partial (List.rev stat_list)
```

The building blocks for a single step of the second pass: saturate a fermion line or pass it through.
The indices i and j refer to incoming lines: add a saturated line to $p.stat$ and remove the corresponding open lines from the map.

```
let saturate_fermion_line p i j =
  match IMap.find_opt i p.fermions, IMap.find_opt j p.fermions with
  | Some f, Some f' →
     { stat = add_lines [f'; f] p.stat;
       fermions = IMap.remove i (IMap.remove j p.fermions);
       n = p.n }
  | Some _, None →
     invalid_arg "saturate_fermion_line:␣no␣open␣outgoing␣fermion␣line"
  | None, Some _ →
     invalid_arg "saturate_fermion_line:␣no␣open␣incoming␣fermion␣line"
  | None, None →
     invalid_arg "saturate_fermion_line:␣no␣open␣fermion␣lines"
```

The index i refers to an incoming line: add the open line to $p.stat$ and remove it from the map.

```
let pass_through_fermion_line p i =
  match IMap.find_opt i p.fermions, p.stat with
  | Some f, Boson l →
     { stat = Majorana (f, l);
       fermions = IMap.remove i p.fermions;
       n = p.n }
  | Some _ , (Majorana _ | Fermion _ | AntiFermion _) →
     invalid_arg "pass_through_fermion_line:␣more␣than␣one␣open␣line"
  | None, _ →
     invalid_arg "pass_through_fermion_line:␣expected␣fermion␣not␣found"
```

Ignoring the direction of the fermion line reproduces JRR's algorithm.

```
let sort_pair (i, j) =
  if i < j then
    (i, j)
  else
    (j, i)
```

The index $p.n + 1$ corresponds to the outgoing line:

```
let is_incoming p i =
  i ≤ p.n

let match_fermion_line p (i, j) =
  let i, j = sort_pair (i, j) in
  if is_incoming p i ∧ is_incoming p j then
    saturate_fermion_line p i j
  else if is_incoming p i then
```

```
      pass_through_fermion_line p i
    else if is_incoming p j then
      pass_through_fermion_line p j
    else
      failwith "match_fermion_line: both lines outgoing"

let match_fermion_line_logging p (i, j) =
  Printf.eprintf
    "match_fermion_line      %s [%d->%d]"
    (partial_to_string p) i j;
  let p' = match_fermion_line p (i, j) in
  Printf.eprintf " >> %s\n" (partial_to_string p');
  p'

let match_fermion_line =
  majorana_log match_fermion_line match_fermion_line_logging
```

Combine the passes ...

```
let match_fermion_lines flines s1 s23__n =
  List.fold_left match_fermion_line (partial_of_slist (s1 :: s23__n)) flines
```

... and keep only the *stat*.

```
let stat_fuse_new flines s1 s23__n _ =
  (match_fermion_lines flines s1 s23__n).stat
```

If there is at most a single fermion line, we can compare *stat* against the result of *stat_fuse_legacy* for checking *stat_fuse_new* (admittedly, this case is rather trivial) ...

```
let stat_fuse_new_check stat flines s1 s23__n f =
  if List.length flines < 2 then
    begin
      let legacy = stat_fuse_legacy s1 s23__n f in
      if ¬ (equal stat legacy) then
        failwith
          (Printf.sprintf
             "stat_fuse_new: %s <> %s!"
             (stat_to_string stat)
             (stat_to_string legacy))
    end
```

... do it, but only when we are writing debugging output.

```
let stat_fuse_new_logging flines s1 s23__n f =
  let stat = stat_fuse_new flines s1 s23__n f in
  Printf.eprintf
    "stat_fuse_new: %s: %s -> %s = %s\n"
    (UFO_Lorentz.fermion_lines_to_string flines)
    (ThoList.to_string stat_to_string (s1 :: s23__n))
    (stat_to_string stat)
    (M.flavor_to_string f);
  stat_fuse_new_check stat flines s1 s23__n f;
  stat

let stat_fuse_new =
  majorana_log stat_fuse_new stat_fuse_new_logging
```

Use *stat_fuse_new*, whenever fermion connections are available. NB: *Some* [] is *not* the same as *None*!

```
let stat_fuse flines_opt slist f =
  match slist with
  | [] → invalid_arg "stat_fuse: empty"
  | s1 :: s23__n →
     constrain_stat_fusion
       (match flines_opt with
        | Some flines → stat_fuse_new flines s1 s23__n f
        | None → stat_fuse_legacy s1 s23__n f)
        f

let stat_fuse_logging flines_opt slist f =
  let stat = stat_fuse flines_opt slist f in
  Printf.eprintf
    "stat_fuse:␣␣␣␣␣␣␣␣␣␣␣␣␣%s␣->␣%s␣=␣%s\n"
    (ThoList.to_string stat_to_string slist)
    (stat_to_string stat)
    (M.flavor_to_string f);
  stat

let stat_fuse =
  majorana_log stat_fuse stat_fuse_logging
```

Final Step using Implied Fermion Connections

```
let stat_keystone_legacy s1 s23__n f =
  stat_fuse_legacy s1 s23__n f

let stat_keystone_legacy_logging s1 s23__n f =
  let s = stat_keystone_legacy s1 s23__n f in
  Printf.eprintf
    "stat_keystone_legacy:␣%s␣(%s)␣%s␣->␣%s\n"
    (stat_to_string s1)
    (M.flavor_to_string f)
    (ThoList.to_string stat_to_string s23__n)
    (stat_to_string s);
  s

let stat_keystone_legacy =
  majorana_log stat_keystone_legacy stat_keystone_legacy_logging
```

Final Step using Explicit Fermion Connections

```
let stat_keystone_new flines slist f =
  match slist with
  | [] → invalid_arg "stat_keystone:␣empty"
  | [s] → invalid_arg "stat_keystone:␣singleton"
  | s1 :: s2 :: s34__n →
     let stat =
       stat_fuse_pair_unconstrained s1 (stat_fuse_new flines s2 s34__n f) in
     if saturated stat then
       stat
     else
       failwith
         (Printf.sprintf
            "stat_keystone:␣incomplete␣%s!"
            (stat_to_string stat))

let stat_keystone_new_check stat slist f =
  match slist with
  | [] → invalid_arg "stat_keystone_check:␣empty"
  | s1 :: s23__n →
     let legacy = stat_keystone_legacy s1 s23__n f in
     if ¬ (equal stat legacy) then
       failwith
         (Printf.sprintf
            "stat_keystone_check:␣%s␣<>␣%s!"
            (stat_to_string stat)
            (stat_to_string legacy))

let stat_keystone flines_opt slist f =
  match flines_opt with
```

```
      | Some flines → stat_keystone_new flines slist f
      | None →
          begin match slist with
          | [] → invalid_arg "stat_keystone: empty"
          | s1 :: s23__n → stat_keystone_legacy s1 s23__n f
          end

  let stat_keystone_logging flines_opt slist f =
    let stat = stat_keystone flines_opt slist f in
    Printf.eprintf
      "stat_keystone:        %s (%s) %s -> %s\n"
      (stat_to_string (List.hd slist))
      (M.flavor_to_string f)
      (ThoList.to_string stat_to_string (List.tl slist))
      (stat_to_string stat);
    stat_keystone_new_check stat slist f;
    stat

  let stat_keystone =
    majorana_log stat_keystone stat_keystone_logging
```

Force the legacy version w/o checking against the new implementation for comparing generated code against the hard coded models:

```
  let stat_fuse flines_opt slist f =
    if force_legacy then
      stat_fuse_legacy (List.hd slist) (List.tl slist) f
    else
      stat_fuse flines_opt slist f

  let stat_keystone flines_opt slist f =
    if force_legacy then
      stat_keystone_legacy (List.hd slist) (List.tl slist) f
    else
      stat_keystone flines_opt slist f
```

Evaluate Signs from Fermion Permuations

```
  let stat_sign = function
    | Boson lines → sign_of_permutation lines
    | Fermion (p, lines) → sign_of_permutation (p :: lines)
    | AntiFermion (pbar, lines) → sign_of_permutation (pbar :: lines)
    | Majorana (pm, lines) → sign_of_permutation (pm :: lines)

  let stat_sign_logging stat =
    let sign = stat_sign stat in
    Printf.eprintf
      "stat_sign: %s -> %d\n"
      (stat_to_string stat) sign;
    sign

  let stat_sign =
    majorana_log stat_sign stat_sign_logging

end

module Binary_Majorana =
  Make(Tuple.Binary)(Stat_Majorana)(Topology.Binary)

module Nary (B : Tuple.Bound) =
  Make(Tuple.Nary(B))(Stat_Dirac)(Topology.Nary(B))
module Nary_Majorana (B : Tuple.Bound) =
  Make(Tuple.Nary(B))(Stat_Majorana)(Topology.Nary(B))

module Mixed23 =
  Make(Tuple.Mixed23)(Stat_Dirac)(Topology.Mixed23)
```

```
module Mixed23_Majorana =
  Make(Tuple.Mixed23)(Stat_Majorana)(Topology.Mixed23)

module Helac (B : Tuple.Bound) =
  Make(Tuple.Nary(B))(Stat_Dirac)(Topology.Helac(B))
module Helac_Majorana (B : Tuple.Bound) =
  Make(Tuple.Nary(B))(Stat_Majorana)(Topology.Helac(B))

module B2 = struct let max_arity () = 2 end
module B3 = struct let max_arity () = 3 end
module Helac_Binary = Helac(B2)
module Helac_Binary_Majorana = Helac(B2)
module Helac_Mixed23 = Helac(B3)
module Helac_Mixed23_Majorana = Helac(B3)
```

15.2.6 Multiple Amplitudes

```
module type Multi =
  sig
    exception Mismatch
    val options : Options.t
    type flavor
    type process = flavor list × flavor list
    type amplitude
    type fusion
    type wf
    type selectors
    type slicings
    type coupling_order
    type amplitudes
    val amplitudes : bool → int option →
      selectors → slicings option → process list → amplitudes
    val empty : amplitudes
    val flavors : amplitudes → process list
    val vanishing_flavors : amplitudes → process list
    val color_flows : amplitudes → Color.Flow.t list
    val coupling_orders : amplitudes → (coupling_order list × int list list) option
    val helicities : amplitudes → (int list × int list) list
    val processes : amplitudes → amplitude list
    val process_table : amplitudes → amplitude option array array
    val process_table_new : amplitudes → amplitude option array array array
    val fusions : amplitudes → (fusion × amplitude) list
    val multiplicity : amplitudes → wf → int
    val dictionary : amplitudes → amplitude → wf → int
    val color_factors : amplitudes → Color.Flow.factor array array
    val constraints : amplitudes → string option
    val slicings : amplitudes → string list
  end

module type Multi_Maker = functor (Fusion_Maker : Maker) →
  functor (P : Momentum.T) →
    functor (M : Model.T) →
      Multi with type flavor = M.flavor
      and type amplitude = Fusion_Maker(P)(M).amplitude
      and type fusion = Fusion_Maker(P)(M).fusion
      and type wf = Fusion_Maker(P)(M).wf
      and type selectors = Fusion_Maker(P)(M).selectors
      and type slicings = Orders.Conditions(Colorize.It(M)).t
      and type coupling_order = Orders.Slice(Colorize.It(M)).coupling_order

module Multi (Fusion_Maker : Maker) (P : Momentum.T) (M : Model.T) =
  struct
```

```
exception Mismatch

type progress_mode =
  | Quiet
  | Channel of out_channel
  | File of string

let progress_option = ref Quiet

module CM = Colorize.It(M)
module SCM = Orders.Slice(Colorize.It(M))
module F = Fusion_Maker(P)(M)
module C = Cascade.Make(M)(P)
module COC = Orders.Conditions(Colorize.It(M))
```

A kludge, at best ...

```
let options = Options.extend F.options
    [ "progress", Arg.Unit (fun () → progress_option := Channel stderr),
      "␣report␣progress␣to␣the␣standard␣error␣stream";
      "progress_file", Arg.String (fun s → progress_option := File s),
      "file␣write␣progress␣report␣to␣file" ]

type flavor = M.flavor
type p = F.p
type process = flavor list × flavor list
type amplitude = F.amplitude
type fusion = F.fusion
type wf = F.wf
type selectors = F.selectors
type slicings = COC.t
type coupling_order = SCM.coupling_order

type flavors = flavor list array
type helicities = int list array
type colors = Color.Flow.t array

type amplitudes =
    { flavors : process list;
      vanishing_flavors : process list;
      color_flows : Color.Flow.t list;
      helicities : (int list × int list) list;
      coupling_orders : (coupling_order list × int list list) option;
      processes : amplitude list;
      process_table : amplitude option array array;
      process_table_new : amplitude option array array array;
      fusions : (fusion × amplitude) list;
      multiplicity : (wf → int);
      dictionary : (amplitude → wf → int);
      color_factors : Color.Flow.factor array array;
      constraints : string option;
      slicings : string list }

let flavors a = a.flavors
let vanishing_flavors a = a.vanishing_flavors
let color_flows a = a.color_flows
let helicities a = a.helicities
let coupling_orders a = a.coupling_orders
let processes a = a.processes
let process_table a = a.process_table
let process_table_new a = a.process_table_new
let fusions a = a.fusions
let multiplicity a = a.multiplicity
let dictionary a = a.dictionary
let color_factors a = a.color_factors
let constraints a = a.constraints
let slicings a = a.slicings

let sans_colors f =
  List.map CM.flavor_sans_color (List.map SCM.flavor_all_orders f)

let colors (fin, fout) =
  List.map M.color (fin @ fout)

let process_sans_color a =
  (sans_colors (F.incoming a), sans_colors (F.outgoing a))

let color_flow a =
  SCM.flow (F.incoming a) (F.outgoing a)

let process_to_string fin fout =
  String.concat "␣" (List.map M.flavor_to_string fin)
  ^ "␣->␣" ^ String.concat "␣" (List.map M.flavor_to_string fout)

let count_processes colored_processes =
  List.length colored_processes

module FMap =
  Map.Make (struct type t = process let compare = compare end)

module CMap =
  Map.Make (struct type t = Color.Flow.t let compare = compare end)
```

Recently *Product.list* began to guarantee lexicographic order for sorted arguments. Anyway, we still force a lexicographic order.

```
let rec order_spin_table1 s1 s2 =
  match s1, s2 with
  | h1 :: t1, h2 :: t2 →
      let c = compare h1 h2 in
      if c ≠ 0 then
        c
      else
        order_spin_table1 t1 t2
  | [], [] → 0
  | _ → invalid_arg "order_spin_table:␣inconsistent␣lengths"

let order_spin_table (s1_in, s1_out) (s2_in, s2_out) =
  let c = compare s1_in s2_in in
  if c ≠ 0 then
    c
  else
    order_spin_table1 s1_out s2_out

let sort_spin_table table =
  List.sort order_spin_table table

let id x = x

let pair x y = (x, y)
```

Improve support for on shell Ward identities: *Coupling.Vector* → [4] for one and only one external vector.

```
let rec hs_of_lorentz = function
  | Coupling.Scalar → [0]
  | Coupling.Spinor | Coupling.ConjSpinor
  | Coupling.Majorana | Coupling.Maj_Ghost → [−1; 1]
  | Coupling.Vector → [−1; 1]
  | Coupling.Massive_Vector → [−1; 0; 1]
  | Coupling.Tensor_1 → [−1; 0; 1]
  | Coupling.Vectorspinor → [−2; − 1; 1; 2]
  | Coupling.Tensor_2 → [−2; − 1; 0; 1; 2]
  | Coupling.BRS f → hs_of_lorentz f

let hs_of_flavor f =
  hs_of_lorentz (M.lorentz f)

let hs_of_flavors (fin, fout) =
  (List.map hs_of_flavor fin, List.map hs_of_flavor fout)

let rec unphysical_of_lorentz = function
  | Coupling.Vector → [4]
  | Coupling.Massive_Vector → [4]
  | _ → invalid_arg "unphysical_of_lorentz:␣not␣a␣vector␣particle"

let unphysical_of_flavor f =
  unphysical_of_lorentz (M.lorentz f)

let unphysical_of_flavors1 n f_list =
  ThoList.mapi
    (fun i f → if i = n then unphysical_of_flavor f else hs_of_flavor f)
    1 f_list

let unphysical_of_flavors n (fin, fout) =
  (unphysical_of_flavors1 n fin, unphysical_of_flavors1 (n − List.length fin) fout)

let helicity_table unphysical flavors =
  let hs =
    begin match unphysical with
    | None → List.map hs_of_flavors flavors
    | Some n → List.map (unphysical_of_flavors n) flavors
    end in
  if ¬ (ThoList.homogeneous hs) then
    invalid_arg "Fusion.helicity_table:␣not␣all␣flavors␣have␣the␣same␣helicity␣states!"
  else
    match hs with
    | [] → []
    | (hs_in, hs_out) :: _ →
        sort_spin_table (Product.list2 pair (Product.list id hs_in) (Product.list id hs_out))

module Proc = Process.Make(M)

module WFMap = Map.Make (struct type t = F.wf let compare = compare end)
module WFSet2 =
  Set.Make (struct type t = F.wf × (F.wf, F.coupling) Tree2.t let compare = compare end)
module WFMap2 =
  Map.Make (struct type t = F.wf × (F.wf, F.coupling) Tree2.t let compare = compare end)
module WFTSet =
  Set.Make (struct type t = (F.wf, F.coupling) Tree2.t let compare = compare end)
```

All wavefunctions are unique per amplitude. So we can use per-amplitude dependency trees without additional *internal* tags to identify identical wave functions.
NB: we miss potential optimizations, because we assume all coupling to be different, while in fact we have horizontal/family symmetries and non abelian gauge couplings are universal anyway.

```
let disambiguate_fusions amplitudes =
  let fusions =
    ThoList.flatmap (fun amplitude →
      List.map
        (fun fusion → (fusion, F.dependencies amplitude (F.lhs fusion)))
        (F.fusions amplitude))
      amplitudes in
  let duplicates =
    List.fold_left
      (fun map (fusion, dependencies) →
        let wf = F.lhs fusion in
        let set = try WFMap.find wf map with Not_found → WFTSet.empty in
        WFMap.add wf (WFTSet.add dependencies set) map)
      WFMap.empty fusions in
    let multiplicity_map =
      WFMap.fold (fun wf dependencies acc →
        let cardinal = WFTSet.cardinal dependencies in
        if cardinal ≤ 1 then
          acc
        else
          WFMap.add wf cardinal acc)
        duplicates WFMap.empty
    and dictionary_map =
      WFMap.fold (fun wf dependencies acc →
        let cardinal = WFTSet.cardinal dependencies in
        if cardinal ≤ 1 then
          acc
        else
          snd (WFTSet.fold
                 (fun dependency (i', acc') →
                    (succ i', WFMap2.add (wf, dependency) i' acc'))
                 dependencies (1, acc)))
        duplicates WFMap2.empty in
    let multiplicity wf =
      WFMap.find wf multiplicity_map
    and dictionary amplitude wf =
      WFMap2.find (wf, F.dependencies amplitude wf) dictionary_map in
    (multiplicity, dictionary)

  let eliminate_common_fusions1 seen_wfs amplitude =
    List.fold_left
      (fun (seen, acc) f →
        let wf = F.lhs f in
        let dependencies = F.dependencies amplitude wf in
        if WFSet2.mem (wf, dependencies) seen then
          (seen, acc)
        else
          (WFSet2.add (wf, dependencies) seen, (f, amplitude) :: acc))
      seen_wfs (F.fusions amplitude)

  let eliminate_common_fusions processes =
    let _, rev_fusions =
      List.fold_left
        eliminate_common_fusions1
        (WFSet2.empty, []) processes in
    List.rev rev_fusions

  module COPMap = Map.Make(struct type t = int list let compare = ThoList.compare ~cmp :
Stdlib.compare end)

  module COBundle = Bundle.Make
     (struct
        type elt = (coupling_order × int) list
        let compare_elt = compare
        type base = coupling_order list
        let compare_base = compare
        let pi = List.map fst
      end)

  let collect_coupling_orders processes =
    let bundle =
      List.fold_right
        (fun process →
          List.fold_right (fun (orders, _) bundle → COBundle.add bundle orders) (F.brakets process))
        processes COBundle.empty in
    match COBundle.fibers bundle with
    | [] | [([], _)] → None
    | [(coupling_orders, orders)] → Some (coupling_orders, List.map (List.map snd) orders)
    | _ → invalid_arg "Fusion.Multi().exclusive_coupling_orders:␣not␣unique"
```

Calculate All The Amplitudes

```
  let amplitudes goldstones unphysical select_wf slicings processes =
```

Eventually, we might want to support inhomogeneous helicities. However, this makes little physics sense for external particles on the mass shell, unless we have a model with degenerate massive fermions and bosons.

```
    if ¬ (ThoList.homogeneous (List.map hs_of_flavors processes)) then
      invalid_arg "Fusion.Multi.amplitudes:␣incompatible␣helicities";

    let unique_uncolored_processes =
      Proc.remove_duplicate_final_states (C.partition select_wf) processes in

    let progress =
      match !progress_option with
      | Quiet → Progress.dummy
      | Channel oc → Progress.channel oc (count_processes unique_uncolored_processes)
      | File name → Progress.file name (count_processes unique_uncolored_processes) in

    let allowed =
      ThoList.flatmap
        (fun (fi, fo) →
          Progress.begin_step progress (process_to_string fi fo);
          let amps = F.amplitudes goldstones select_wf slicings fi fo in
          begin match amps with
          | [] → Progress.end_step progress "forbidden"
          | _ → Progress.end_step progress "allowed"
          end;
          amps) unique_uncolored_processes in

    Progress.summary progress "all␣processes␣done";

    let color_flows =
      ThoList.uniq (List.sort compare (List.map color_flow allowed))
    and flavors =
      ThoList.uniq (List.sort compare (List.map process_sans_color allowed)) in

    let vanishing_flavors =
      Proc.diff processes flavors in

    let helicities =
      helicity_table unphysical flavors in

    let allowed_coupling_orders =
      collect_coupling_orders allowed in

    let f_index =
      fst (List.fold_left
             (fun (m, i) f → (FMap.add f i m, succ i))
             (FMap.empty, 0) flavors)
    and c_index =
      fst (List.fold_left
             (fun (m, i) c → (CMap.add c i m, succ i))
             (CMap.empty, 0) color_flows)
    and co_index =
      match allowed_coupling_orders with
      | None → COPMap.empty
      | Some (_, powers) →
         fst (List.fold_left
                (fun (m, i) c → (COPMap.add c i m, succ i))
                (COPMap.empty, 0) powers) in

    let table =
      Array.make_matrix (List.length flavors) (List.length color_flows) None in
    List.iter
      (fun a →
        let f = FMap.find (process_sans_color a) f_index
        and c = CMap.find (color_flow a) c_index in
        table.(f).(c) ← Some (a))
      allowed;

    let table_new =
      ThoArray.rank3 1 (List.length flavors) (List.length color_flows) None in
    List.iter
      (fun a →
        let co = 0
        and f = FMap.find (process_sans_color a) f_index
        and c = CMap.find (color_flow a) c_index in
        table_new.(co).(f).(c) ← Some (a))
      allowed;

    let color_factor_table = Color.Flow.factor_table color_flows in

    let fusions = eliminate_common_fusions allowed
    and multiplicity, dictionary = disambiguate_fusions allowed in

    let slicings =
      match slicings with
      | None → []
      | Some slicings → COC.to_strings slicings in

    { flavors = flavors;
      vanishing_flavors = vanishing_flavors;
      color_flows = color_flows;
      helicities = helicities;
      coupling_orders = allowed_coupling_orders;
      processes = allowed;
      process_table = table;
      process_table_new = table_new;
      fusions = fusions;
      multiplicity = multiplicity;
      dictionary = dictionary;
      color_factors = color_factor_table;
      constraints = C.description select_wf;
      slicings = slicings }

  let empty =
    { flavors = [];
      vanishing_flavors = [];
      color_flows = [];
      helicities = [];
      coupling_orders = None;
      processes = [];
      process_table = Array.make_matrix 0 0 None;
      process_table_new = ThoArray.rank3 0 0 0 None;
      fusions = [];
      multiplicity = (fun _ → 1);
      dictionary = (fun _ _ → 1);
      color_factors = Array.make_matrix 0 0 Color.Flow.zero;
      constraints = None;
      slicings = [] }

end
```

—16— Lorentz Representations, Couplings, Models and Targets

16.1 Interface of Coupling

The enumeration types used for communication from *Models* to *Targets*. On the physics side, the modules in *Models* must implement the Feynman rules according to the conventions set up here. On the numerics side, the modules in *Targets* must handle all cases according to the same conventions.

16.1.1 Propagators

The Lorentz representation of the particle. NB: O'Mega treats all lines as *outgoing* and particles are therefore transforming as *ConjSpinor* and antiparticles as *Spinor*.

```
type lorentz =
  | Scalar
  | Spinor (* ψ *)
  | ConjSpinor (* ψ̄ *)
  | Majorana (* χ *)
  | Maj_Ghost (* SUSY ghosts *)
  | Vector
  | Massive_Vector
  | Vectorspinor (* supersymmetric currents and gravitinos *)
  | Tensor_1
  | Tensor_2 (* massive gravitons (large extra dimensions) *)
  | BRS of lorentz
```

```
type lorentz3 = lorentz × lorentz × lorentz
type lorentz4 = lorentz × lorentz × lorentz × lorentz
type lorentzn = lorentz list
```

```
type fermion_lines = (int × int) list
```

If there were no vectors or auxiliary fields, we could deduce the propagator from the Lorentz representation. While we're at it, we can introduce "propagators" for the contact interactions of auxiliary fields as well. *Prop_Gauge* and *Prop_Feynman* are redundant as special cases of *Prop_Rxi*.

The special case *Only_Insertion* corresponds to operator insertions that do not correspond to a propagating field all. These are used for checking Slavnov-Taylor identities

$$\partial_\mu \langle \text{out}|W^\mu(x)|\text{in}\rangle = m_W \langle \text{out}|\phi(x)|\text{in}\rangle \tag{16.1}$$

of gauge theories in unitarity gauge where the Goldstone bosons are not propagating. Numerically, it would suffice to use a vanishing propagator, but then superflous fusions would be calculated in production code in which the Slavnov-Taylor identities are not tested.

```
type α propagator =
  | Prop_Scalar | Prop_Ghost
  | Prop_Spinor | Prop_ConjSpinor | Prop_Majorana
  | Prop_Unitarity | Prop_Feynman | Prop_Gauge of α | Prop_Rxi of α
  | Prop_Tensor_2 | Prop_Tensor_pure | Prop_Vector_pure
  | Prop_Vectorspinor
  | Prop_Col_Scalar | Prop_Col_Feynman | Prop_Col_Majorana
```

	only Dirac fermions	incl. Majorana fermions
Prop_Scalar	$\phi(p) \leftarrow \frac{\mathrm{i}}{p^2 - m^2 + \mathrm{i}m\Gamma}\phi(p)$	
Prop_Spinor	$\psi(p) \leftarrow \frac{\mathrm{i}(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\psi(p)$	$\psi(p) \leftarrow \frac{\mathrm{i}(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\psi(p)$
Prop_ConjSpinor	$\bar\psi(p) \leftarrow \bar\psi(p)\frac{\mathrm{i}(\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}$	$\psi(p) \leftarrow \frac{\mathrm{i}(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\psi(p)$
Prop_Majorana	N/A	$\chi(p) \leftarrow \frac{\mathrm{i}(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\chi(p)$
Prop_Unitarity	$\epsilon_\mu(p) \leftarrow \frac{\mathrm{i}}{p^2 - m^2 + \mathrm{i}m\Gamma}\left(-g_{\mu\nu} + \frac{p_\mu p_\nu}{m^2}\right)\epsilon^\nu(p)$	
Prop_Feynman	$\epsilon^\nu(p) \leftarrow \frac{-\mathrm{i}}{p^2 - m^2 + \mathrm{i}m\Gamma}\epsilon^\nu(p)$	
Prop_Gauge	$\epsilon_\mu(p) \leftarrow \frac{\mathrm{i}}{p^2}\left(-g_{\mu\nu} + (1-\xi)\frac{p_\mu p_\nu}{p^2}\right)\epsilon^\nu(p)$	
Prop_Rxi	$\epsilon_\mu(p) \leftarrow \frac{\mathrm{i}}{p^2 - m^2 + \mathrm{i}m\Gamma}\left(-g_{\mu\nu} + (1-\xi)\frac{p_\mu p_\nu}{p^2 - \xi m^2}\right)\epsilon^\nu(p)$	

Table 16.1: Propagators. NB: The sign of the momenta in the spinor propagators comes about because O'Mega treats all momenta as *outgoing* and the charge flow for *Spinor* is therefore opposite to the momentum, while the charge flow for *ConjSpinor* is parallel to the momentum.

Aux_Scalar	$\phi(p) \leftarrow \mathrm{i}\phi(p)$
Aux_Spinor	$\psi(p) \leftarrow \mathrm{i}\psi(p)$
Aux_ConjSpinor	$\bar\psi(p) \leftarrow \mathrm{i}\bar\psi(p)$
Aux_Vector	$\epsilon^\mu(p) \leftarrow \mathrm{i}\epsilon^\mu(p)$
Aux_Tensor_1	$T^{\mu\nu}(p) \leftarrow \mathrm{i}T^{\mu\nu}(p)$
Only_Insertion	N/A

Table 16.2: Auxiliary and non propagating fields

```
  | Prop_Col_Unitarity
  | Aux_Scalar | Aux_Vector | Aux_Tensor_1
  | Aux_Col_Scalar | Aux_Col_Vector | Aux_Col_Tensor_1
  | Aux_Spinor | Aux_ConjSpinor | Aux_Majorana
  | Only_Insertion
  | Prop_UFO of string
```

JR sez' (regarding the Majorana Feynman rules): We don't need different fermionic propagators as supposed by the variable names *Prop_Spinor*, *Prop_ConjSpinor* or *Prop_Majorana*. The propagator in all cases has to be multiplied on the left hand side of the spinor out of which a new one should be built. All momenta are treated as *outgoing*, so for the propagation of the different fermions the following table arises, in which the momentum direction is always downwards and the arrows show whether the momentum and the fermion line, respectively are parallel or antiparallel to the direction of calculation:

Fermion type	fermion arrow	mom.	calc.	sign
Dirac fermion	$\uparrow$	$\uparrow\ \downarrow$	$\uparrow\ \uparrow$	negative
Dirac antifermion	$\downarrow$	$\downarrow\ \downarrow$	$\uparrow\ \downarrow$	negative
Majorana fermion	-	$\uparrow\ \downarrow$	-	negative

So the sign of the momentum is always negative and no further distinction is needed. *(JR's probably right, but I need to check myself ...)*

```
type width =
  | Vanishing
  | Constant
  | Timelike
  | Running
  | Fudged
  | Complex_Mass
  | Custom of string
```

16.1.2 Vertices

The combined $S-P$ and $V-A$ couplings (see tables 16.5, 16.6, 16.8 and 16.12) are redundant, of course, but they allow some targets to create more efficient numerical code.[1] Choosing VA2 over VA will cause the FORTRAN backend to pass the coupling as a whole array

```
type fermion = Psi | Chi | Grav
type fermionbar = Psibar | Chibar | Gravbar
type boson =
  | SP | SPM | S | P | SL | SR | SLR | VA | V | A | VL | VR | VLR | VLRM | VAM
  | TVA | TLR | TRL | TVAM | TLRM | TRLM
  | POT | MOM | MOM5 | MOML | MOMR | LMOM | RMOM | VMOM | VA2 | VA3 | VA3M
type boson2 = S2 | P2 | S2P | S2L | S2R | S2LR
  | SV | PV | SLV | SRV | SLRV | V2 | V2LR
```

The integer is an additional coefficient that multiplies the respective coupling constant. This allows to reduce the number of required coupling constants in manifestly symmetrc cases. Most of times it will be equal unity, though.
The two vertex types *PBP* and *BBB* for the couplings of two fermions or two antifermions ("clashing arrows") is unavoidable in supersymmetric theories.

... tho doesn't like the names and has promised to find a better mnemonics!

```
type α vertex3 =
  | FBF of int × fermionbar × boson × fermion
  | PBP of int × fermion × boson × fermion
  | BBB of int × fermionbar × boson × fermionbar
  | GBG of int × fermionbar × boson × fermion (∗ gravitino-boson-fermion ∗)
```

[1]An additional benefit is that the counting of Feynman diagrams is not upset by a splitting of the vectorial and axial pieces of gauge bosons.

| Gauge_Gauge_Gauge of int | Aux_Gauge_Gauge of int
| I_Gauge_Gauge_Gauge of int
| Scalar_Vector_Vector of int
| Aux_Vector_Vector of int | Aux_Scalar_Vector of int
| Scalar_Scalar_Scalar of int | Aux_Scalar_Scalar of int
| Vector_Scalar_Scalar of int
| Graviton_Scalar_Scalar of int
| Graviton_Vector_Vector of int
| Graviton_Spinor_Spinor of int
| Dim4_Vector_Vector_Vector_T of int
| Dim4_Vector_Vector_Vector_L of int
| Dim4_Vector_Vector_Vector_T5 of int
| Dim4_Vector_Vector_Vector_L5 of int
| Dim6_Gauge_Gauge_Gauge of int
| Dim6_Gauge_Gauge_Gauge_5 of int
| Aux_DScalar_DScalar of int | Aux_Vector_DScalar of int
| Dim5_Scalar_Gauge2 of int (* $\frac{1}{2}\phi F_{1,\mu\nu}F_2^{\mu\nu} = -\frac{1}{2}\phi(\mathrm{i}\partial_{[\mu,}V_{1,\nu]})(\mathrm{i}\partial^{[\mu,}V_2^{\nu]})$ *)
| Dim5_Scalar_Gauge2_Skew of int
  (* $\frac{1}{4}\phi F_{1,\mu\nu}\tilde{F}_2^{\mu\nu} = -\phi(\mathrm{i}\partial_\mu V_{1,\nu})(\mathrm{i}\partial_\rho V_{2,\sigma})\epsilon^{\mu\nu\rho\sigma}$ *)
| Dim5_Scalar_Scalar2 of int (* $\phi_1\partial_\mu\phi_2\partial^\mu\phi_3$ *)
| Dim5_Scalar_Vector_Vector_T of int (* $\phi(\mathrm{i}\partial_\mu V_1^\nu)(\mathrm{i}\partial_\nu V_2^\mu)$ *)
| Dim5_Scalar_Vector_Vector_TU of int (* $(\mathrm{i}\partial_\nu\phi)(\mathrm{i}\partial_\mu V_1^\nu)V_2^\mu$ *)
| Dim5_Scalar_Vector_Vector_U of int (* $(\mathrm{i}\partial_\nu\phi)(\mathrm{i}\partial_\mu V^\nu)V^\mu$ *)
| Scalar_Vector_Vector_t of int (* $(\partial_\mu V_\nu - \partial_\nu V_\mu)^2$ *)
| Dim6_Vector_Vector_Vector_T of int (* $V_1^\mu((\mathrm{i}\partial_\nu V_2^\rho)\mathrm{i}\overleftrightarrow{\partial_\mu}(\mathrm{i}\partial_\rho V_3^\nu))$ *)
| Tensor_2_Vector_Vector of int (* $T^{\mu\nu}(V_{1,\mu}V_{2,\nu} + V_{1,\nu}V_{2,\mu})$ *)
| Tensor_2_Vector_Vector_1 of int (* $T^{\mu\nu}(V_{1,\mu}V_{2,\nu} + V_{1,\nu}V_{2,\mu} - g_{\mu,\nu}V_1^\rho V_{2,\rho})$ *)
| Tensor_2_Vector_Vector_cf of int (* $T^{\mu\nu}(-\frac{c_f}{2}g_{\mu,\nu}V_1^\rho V_{2,\rho})$ *)
| Tensor_2_Scalar_Scalar of int (* $T^{\mu\nu}(\partial_\mu\phi_1\partial_\nu\phi_2 + \partial_\nu\phi_1\partial_\mu\phi_2)$ *)
| Tensor_2_Scalar_Scalar_cf of int (* $T^{\mu\nu}(-\frac{c_f}{2}g_{\mu,\nu}\partial_\rho\phi_1\partial_\rho\phi_2)$ *)
| Tensor_2_Vector_Vector_t of int (* $T^{\mu\nu}(V_{1,\mu}V_{2,\nu} + V_{1,\nu}V_{2,\mu} - g_{\mu,\nu}V_1^\rho V_{2,\rho})$ *)
| Dim5_Tensor_2_Vector_Vector_1 of int (* $T^{\alpha\beta}(V_1^\mu\mathrm{i}\overleftrightarrow{\partial}_\alpha\mathrm{i}\overleftrightarrow{\partial}_\beta V_{2,\mu}$ *)
| Dim5_Tensor_2_Vector_Vector_2 of int
  (* $T^{\alpha\beta}(V_1^\mu\mathrm{i}\overleftrightarrow{\partial}_\beta(\mathrm{i}\partial_\mu V_{2,\alpha}) + V_1^\mu\mathrm{i}\overleftrightarrow{\partial}_\alpha(\mathrm{i}\partial_\mu V_{2,\beta}))$ *)
| Dim7_Tensor_2_Vector_Vector_T of int (* $T^{\alpha\beta}((\mathrm{i}\partial^\mu V_1^\nu)\mathrm{i}\overleftrightarrow{\partial}_\alpha\mathrm{i}\overleftrightarrow{\partial}_\beta(\mathrm{i}\partial_\nu V_{2,\mu}))$ *)
| Dim6_Scalar_Vector_Vector_D of int
  (* $\mathrm{i}\phi(-(\partial^\mu\partial^\nu W_\mu^-)W_\nu^+ - (\partial^\mu\partial^\nu W_\nu^+)W_\mu^-$
  $+ ((\partial^\rho\partial_\rho W_\mu^-)W_\nu^+ + (\partial^\rho\partial_\rho W_\nu^+)W_\mu^-)g^{\mu\nu})$ *)
| Dim6_Scalar_Vector_Vector_DP of int
  (* $\mathrm{i}((\partial^\mu H)(\partial^\nu W_\mu^-)W_\nu^+ + (\partial^\nu H)(\partial^\mu W_\nu^+)W_\mu^-$
  $- ((\partial^\rho H)(\partial_\rho W_\mu^-)W_\nu^+(\partial^\rho H)(\partial^\rho W_\nu^+)W_\mu^-)g^{\mu\nu})$*)
| Dim6_HAZ_D of int (* $\mathrm{i}((\partial^\mu\partial^\nu A_\mu)Z_\nu + (\partial^\rho\partial_\rho A_\mu)Z_\nu g^{\mu\nu})$ *)
| Dim6_HAZ_DP of int (* $\mathrm{i}((\partial^\nu A_\mu)(\partial^\mu H)Z_\nu - (\partial^\rho A_\mu)(\partial_\rho H)Z_\nu g^{\mu\nu})$ *)
| Dim6_AWW_DP of int (* $\mathrm{i}((\partial^\rho A_\mu)W_\nu^- W_\rho^+ g^{\mu\nu} - (\partial^\nu A_\mu)W_\nu^- W_\rho^+ g^{\mu\rho})$ *)
| Dim6_AWW_DW of int
  (*$\mathrm{i}[(3(\partial^\rho A_\mu)W_\nu^- W_\rho^+ - (\partial^\rho W_\nu^-)A_\mu W_\rho^+ + (\partial^\rho W_\rho^+)A_\mu W_\nu^-)g^{\mu\nu}$
  $+ (-3(\partial^\nu A_\mu)W_\nu^- W_\rho^+ - (\partial^\nu W_\nu^-)A_\mu W_\rho^+ + (\partial^\nu W_\rho^+)A_\mu W_\nu^-)g^{\mu\rho}$
  $+ (2(\partial^\mu W_\nu^-)A_\mu W_\rho^+ - 2(\partial^\mu W_\rho^+)A_\mu W_\nu^-)g^{\nu\rho}]$ *)
| Dim6_HHH of int (*$\mathrm{i}(-(\partial^\mu H_1)(\partial_\mu H_2)H_3 - (\partial^\mu H_1)H_2(\partial_\mu H_3) - H_1(\partial^\mu H_2)(\partial_\mu H_3))$ *)
| Dim6_Gauge_Gauge_Gauge_i of int
  (*$\mathrm{i}(-(\partial^\nu V_\mu)(\partial^\rho V_\nu)(\partial^\mu V_\rho) + (\partial^\rho V_\mu)(\partial^\mu V_\nu)(\partial^\nu V_\rho)$
  $+ (-\partial^\nu V_\rho g^{\mu\rho} + \partial^\mu V_\rho g^{\nu\rho})(\partial^\sigma V_\mu)(\partial_\sigma V_\nu) + (\partial^\rho V_\nu g^{\mu\nu} - \partial^\mu V_\nu g^{\nu\rho})(\partial^\sigma V_\mu)(\partial_\sigma V_\rho)$
  $+ (-\partial^\rho V_\mu g^{\mu\nu} + \partial^\mu V_\mu g^{\mu\rho})(\partial^\sigma V_\nu)(\partial_\sigma V_\rho))$ *)
| Gauge_Gauge_Gauge_i of int
| Dim6_GGG of int
| Dim6_WWZ_DPWDW of int
  (* $\mathrm{i}(((\partial^\rho V_\mu)V_\nu V_\rho - (\partial^\rho V_\nu)V_\mu V_\rho)g^{\mu\nu} - (\partial^\nu V_\mu)V_\nu V_\rho g^{\mu\rho} + (\partial^\mu V_\nu)V_\mu V_\rho)g^{\rho\nu})$ *)
| Dim6_WWZ_DW of int
  (* $\mathrm{i}(((\partial^\mu V_\mu)V_\nu V_\rho + V_\mu(\partial^\mu V_\nu)V_\rho)g^{\nu\rho} - ((\partial^\nu V_\mu)V_\nu V_\rho + V_\mu(\partial^\nu V_\nu)V_\rho)g^{\mu\rho})$ *)
| Dim6_WWZ_D of int (* $\mathrm{i}(V_\mu)V_\nu(\partial^\nu V_\rho)g^{\mu\rho} + V_\mu V_\nu(\partial^\mu V_\rho)g^{\nu\rho})$ *)

| *TensorVector_Vector_Vector* of *int*
| *TensorVector_Vector_Vector_cf* of *int*
| *TensorVector_Scalar_Scalar* of *int*
| *TensorVector_Scalar_Scalar_cf* of *int*
| *TensorScalar_Vector_Vector* of *int*
| *TensorScalar_Vector_Vector_cf* of *int*
| *TensorScalar_Scalar_Scalar* of *int*
| *TensorScalar_Scalar_Scalar_cf* of *int*

As long as we stick to renormalizable couplings, there are only three types of quartic couplings: *Scalar4*, *Scalar2_Vector2* and *Vector4*. However, there are three inequivalent contractions for the latter and the general vertex will be a linear combination with integer coefficients:

$$\textit{Scalar4}\ 1: \quad \phi_1\phi_2\phi_3\phi_4 \tag{16.2a}$$

$$\textit{Scalar2\_Vector2}\ 1: \quad \phi_1\phi_2 V_3^\mu V_{4,\mu} \tag{16.2b}$$

$$\textit{Vector4}\ [1, C\_12\_34]: \quad V_1^\mu V_{2,\mu} V_3^\nu V_{4,\nu} \tag{16.2c}$$

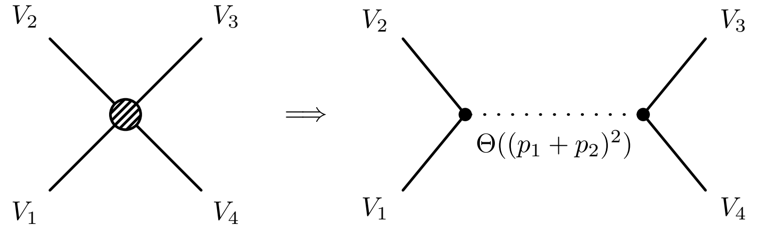
$$\textit{Vector4}\ [1, C\_13\_42]: \quad V_1^\mu V_2^\nu V_{3,\mu} V_{4,\nu} \tag{16.2d}$$

$$\textit{Vector4}\ [1, C\_14\_23]: \quad V_1^\mu V_2^\nu V_{3,\nu} V_{4,\mu} \tag{16.2e}$$

type *contract4* = *C_12_34* | *C_13_42* | *C_14_23*

type α *vertex4* =
| *Scalar4* of *int*
| *Scalar2_Vector2* of *int*
| *Vector4* of (*int* × *contract4*) *list*
| *DScalar4* of (*int* × *contract4*) *list*
| *DScalar2_Vector2* of (*int* × *contract4*) *list*
| *Dim8_Scalar2_Vector2_1* of *int*
| *Dim8_Scalar2_Vector2_2* of *int*
| *Dim8_Scalar2_Vector2_m_0* of *int*
| *Dim8_Scalar2_Vector2_m_1* of *int*
| *Dim8_Scalar2_Vector2_m_7* of *int*
| *Dim8_Scalar4* of *int*
| *Dim8_Vector4_t_0* of (*int* × *contract4*) *list*
| *Dim8_Vector4_t_1* of (*int* × *contract4*) *list*
| *Dim8_Vector4_t_2* of (*int* × *contract4*) *list*
| *Dim8_Vector4_m_0* of (*int* × *contract4*) *list*
| *Dim8_Vector4_m_1* of (*int* × *contract4*) *list*
| *Dim8_Vector4_m_7* of (*int* × *contract4*) *list*
| *GBBG* of *int* × *fermionbar* × *boson2* × *fermion*

In some applications, we have to allow for contributions outside of perturbation theory. The most prominent example is heavy gauge boson scattering at very high energies, where the perturbative expression violates unitarity.
One solution is the 'K-matrix' ansatz. Such unitarizations typically introduce effective propagators and/or vertices that violate crossing symmetry and vanish in the t-channel. This can be taken care of in *Fusion* by filtering out vertices that have the wrong momenta.
In this case the ordering of the fields in a vertex of the Feynman rules becomes significant. In particular, we assume that (V_1, V_2, V_3, V_4) implies



(16.3)

The list of pairs of parameters denotes the location and strengths of the poles in the K-matrix ansatz:

$$(c_1, a_1, c_2, a_2, \ldots, c_n, a_n) \Longrightarrow f(s) = \sum_{i=1}^{n} \frac{c_i}{s - a_i} \tag{16.4}$$

| *Vector4_K_Matrix_tho* of *int* $\times$ (α $\times$ α) *list*
| *Vector4_K_Matrix_jr* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_t0* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_t1* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_t2* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_t_rsi* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_m0* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_m1* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Vector4_K_Matrix_cf_m7* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *DScalar2_Vector2_K_Matrix_ms* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *DScalar2_Vector2_m_0_K_Matrix_cf* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *DScalar2_Vector2_m_1_K_Matrix_cf* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *DScalar2_Vector2_m_7_K_Matrix_cf* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *DScalar4_K_Matrix_ms* of *int* $\times$ (*int* $\times$ *contract4*) *list*
| *Dim6_H4_P2* of *int*
$(*\ \mathrm{i}(-(\partial^\mu H_1)(\partial_\mu H_2)H_3H_4 - (\partial^\mu H_1)H_2(\partial_\mu H_3)H_4 - (\partial^\mu H_1)H_2H_3(\partial_{mu}H_4)$
$- H_1(\partial^\mu H_2)(\partial_\mu H_3)H_4 - H_1(\partial^\mu H_2)H_3(\partial_\mu H_4) - H_1H_2(\partial^\mu H_3)(\partial_\mu H_4))\ *)$
| *Dim6_AHWW_DPB* of *int* $(*\ \mathrm{i}H((\partial^\rho A_\mu)W_\nu W_\rho g^{\mu\nu} - (\partial^\nu A_\mu)W_\nu W_\rho g^{\mu\rho})\ *)$
| *Dim6_AHWW_DPW* of *int*
$(*\ \mathrm{i}(((\partial^\rho A_\mu)W_\nu W_\rho - (\partial^\rho H)A_\mu W_\nu W_\rho)g^{\mu\nu}$
$(-(\partial^\nu A_\mu)W_\nu W_\rho + (\partial^\nu H)A_\mu W_\nu W_\rho)g^{\mu\rho})\ *)$
| *Dim6_AHWW_DW* of *int*
$(*\ \mathrm{i}H((3(\partial^\rho A_\mu)W_\nu W_\rho - A_\mu(\partial^\rho W_\nu)W_\rho + A_\mu W_\nu(\partial^\rho W_\rho))g^{\mu\nu}$
$+ (-3(\partial^\nu A_\mu)W_\nu W_\rho - A_\mu(\partial^\nu W_\nu)W_\rho + A_\mu W_\nu(\partial^\nu W_\rho))g^{\mu\rho}$
$+ 2(A_\mu(\partial^\mu W_\nu)W_\rho + A_\mu W_\nu(\partial^\mu W_\rho)))g^{\nu\rho})\ *)$
| *Dim6_Vector4_DW* of *int* $(*\mathrm{i}(-V_{1,\mu}V_{2,\nu}V^{3,\nu}V^{4,\mu} - V_{1,\mu}V_{2,\nu}V^{3,\mu}V^{4,\nu}$
$+ 2V_{1,\mu}V^{2,\mu}V_{3,\nu}V^{4,\nu}\ *)$
| *Dim6_Vector4_W* of *int*
$(*\ \mathrm{i}(((\partial^\rho V_{1,\mu})V_2^\mu(\partial^\sigma V_{3,\rho})V_{4,\sigma} + V_{1,\mu}(\partial^\rho V_2^\mu)(\partial^\sigma V_{3,\rho})V_{4,\sigma}$
$+ (\partial^\sigma V_{1,\mu})V_2^\mu V_{3,\rho}(\partial^\rho V_{4,\sigma}) + V_{1,\mu}(\partial^\sigma V_2^\mu)V_{3,\rho}(\partial^\rho V_{4,\sigma}))$
$+ ((\partial^\sigma V_{1,\mu})V_{2,\nu}(\partial^\nu V_3^\mu)V_{4,\sigma} - V_{1,\mu}(\partial^\sigma V_{2,\nu})(\partial^\nu V_3^\mu)V_{4,\sigma}$
$- (\partial^\nu V_1^\mu)V_{2,\nu}(\partial^\sigma V_{3,\mu})V_{4,\sigma} - (\partial^\sigma V_{1,\mu})V_{2,\nu}V_3^\mu(\partial^\nu V_{4,\sigma}))$
$+ (-(\partial^\rho V_{1,\mu})V_{2,\nu}(\partial^\nu V_{3,\rho})V_4^\mu + (\partial^\rho V_{1,\mu})V_{2,\nu}V_{3,\rho}(\partial^\nu V_4^\mu)$
$- V_{1,\mu}(\partial^\rho V_{2,\nu})V_{3,\rho}(\partial^\nu V_4^\mu) - (\partial^\nu V_{1,\mu})V_{2,\nu}V_{3,\rho}(\partial^\rho V_4^\mu))$
$+ (-(\partial^\sigma V_{1,\mu})V_{2,\nu}(\partial^\mu V_3^\nu)V_{4,\sigma} + V_{1,\mu}(\partial^\sigma V_{2,\nu})(\partial^\mu V_3^\nu)V_{4,\sigma}$
$- V_{1,\mu}(\partial^\mu V_{2,\nu})(\partial^\sigma V_3^\nu)V_{4,\sigma} - V_{1,\mu}(\partial^\sigma V_{2,\nu})V_3^\nu(\partial^\mu V_{4,\sigma})$
$+ (-V_{1,\mu}(\partial^\rho V_{2,\nu})(\partial^\mu V_{3,\rho})V_4^\nu - (\partial^\rho V_{1,\mu})V_{2,\nu}V_{3,\rho}(\partial^\mu V_4^\nu)$
$+ V_{1,\mu}(\partial^\rho V_{2,\nu})V_{3,\rho}(\partial^\mu V_4^\nu) - V_{1,\mu}(\partial^\mu V_{2,\nu})V_{3,\rho}(\partial^\rho V_4^\nu))$
$+ ((\partial^\nu V_{1,\mu})V_{2,\nu}(\partial^\mu V_{3,\rho})V_4^\rho + V_{1,\mu}(\partial^\mu V_{2,\nu})(\partial^\nu V_{3,\rho})V_4^\rho$
$+ (\partial^\nu V_{1,\mu})V_{2,\nu}V_{3,\rho}(\partial^\mu V_4^\rho) + V_{1,\mu}(\partial^\mu V_{2,\nu})V_{3,\rho}(\partial^\nu V_4^\rho))$
$+ (\partial^\rho V_{1,\mu})V_{2,\nu}V_3^\mu(\partial_\rho V_4^\nu) - (\partial^\rho V_{1,\mu})V_2^\mu V_{3,\nu}(\partial_\rho V_4^\nu)$
$+ V_{1,\mu}(\partial^\rho V_{2,\nu})(\partial_\rho V_3^\mu)V_4^\nu - V_{1,\mu}(\partial^\rho V_2^\mu)(\partial_\rho V_{3,\nu})V_4^\nu$
$+ (\partial^\rho V_{1,\mu})V_{2,\nu}(\partial_\rho V_3^\nu)V_4^\mu - (\partial^\rho V_{1,\mu})V_2^\mu(\partial_\rho V_{3,\nu})V_4^\nu$
$+ V_{1,\mu}(\partial^\rho V_{2,\nu})V_3^\nu(\partial_\rho V_4^\mu) - V_{1,\mu}(\partial^\rho V_2^\mu)V_{3,\nu}(\partial_\rho V_4^\nu))\ *)$
| *Dim6_Scalar2_Vector2_D* of *int*
$(*\mathrm{i}H_1H_2(-(\partial^\mu\partial^\nu V_{3,\mu})V_{4,\nu} + (\partial^\mu\partial_\mu V_{3,\nu})V_4^\nu$
$- V_{3,\mu}(\partial^\mu\partial^\nu V_{4,\nu}) + V_{3,\mu}(\partial^\nu\partial_\nu V_4^\mu))\ *)$
| *Dim6_Scalar2_Vector2_DP* of *int*
$(*\mathrm{i}((\partial^\mu H_1)H_2(\partial^\nu V_{3,\mu})V_{4,\nu} - (\partial^\nu H_1)H_2(\partial_\nu V_{3,\mu})V^{4,\mu} + H_1(\partial^\mu H_2)(\partial^\nu V_{3,\mu})V_{4,\nu}$
$- H_1(\partial^\nu H_2)(\partial_\nu V_{3,\mu})V^{4,\mu} + (\partial^\nu H_1)H_2V_{3,\mu}(\partial^\mu V_{4,\nu}) - (\partial^\nu H_1)H_2V_{3,\mu}(\partial_\nu V^{4,\mu})$
$+ H_1(\partial^\nu H_2)V_{3,\mu}(\partial^\mu V_{4,\nu}) - H_1(\partial^\nu H_2)V_{3,\mu}(\partial_\nu V^{4,\mu}))\ *)$
| *Dim6_Scalar2_Vector2_PB* of *int*
$(*\mathrm{i}(H_1H_2(\partial^\nu V_{3,\mu})(\partial^\mu V_{4,\nu}) - H_1H_2(\partial^\nu V_{3,\mu})(\partial_\nu V^{4,\mu}))\ *)$
| *Dim6_HHZZ_T* of *int* $(*\mathrm{i}H_1H_2V_{3,\mu}V^{4,\mu}\ *)$
| *Dim6_HWWZ_DW* of *int*
$(*\ \mathrm{i}(H_1(\partial^\rho W_{2,\mu})W^{3,\mu}Z_{4,\rho} - H_1W_{2,\mu}(\partial^\rho W^{3,\mu})Z_{4,\rho} - 2H_1(\partial^\nu W_{2,\mu})W_{3,\nu}Z^{4,\mu}$
$- H_1W_{2,\mu}(\partial^\nu W_{3,\nu})Z^{4,\mu} + H_1(\partial^\mu W_{2,\mu})W_{3,\nu}Z^{4,\nu} + 2H_1W_{2,\mu}(\partial^\mu W_{3,\nu})Z^{4,\nu})\ *)$
| *Dim6_HWWZ_DPB* of *int*
$(*\ \mathrm{i}(-H_1W_{2,\mu}W_{3,\nu}(\partial^\nu Z^{4,\mu}) + H_1W_{2,\mu}W_{3,\nu}(\partial^\mu Z^{4,\nu}))\ *)$
| *Dim6_HWWZ_DDPW* of *int*
$(*\ \mathrm{i}(H_1(\partial^\nu W_{2,\mu})W^{3,\mu}Z_{4,\nu} - H_1W_{2,\mu}(\partial^\nu W^{3,\mu})Z_{4,\nu} - H_1(\partial^\nu W_{2,\mu})W_{3,\nu}Z^{4,\mu}$

$+ H_1 W_{2,\mu} W_{3,\nu} (\partial^\nu Z^{4,\mu}) + H_1 W_{2,\mu} (\partial^\mu W_{3,\nu}) Z^{4,\nu} - H_1 W_{2,\mu} W_{3,\nu} (\partial^\mu Z^{4,\nu}))$ *)
| *Dim6_HWWZ_DPW* of *int*
(* $\mathrm{i}(H_1(\partial^\nu W_{2,\mu}) W^{3,\mu} Z_{4,\nu} - H_1 W_{2,\mu} (\partial^\nu W^{3,\mu}) Z_{4,\nu} + (\partial^\nu H_1) W_{2,\mu} W_{3,\nu} Z^{4,\mu}$
$- H_1 (\partial^\nu W_{2,\mu}) W_{3,\nu} Z^{4,\mu} - (\partial^\mu H_1) W_{2,\mu} W_{3,\nu} Z^{4,\nu} + H_1 W_{2,\mu} (\partial^\mu W_{3,\nu}) Z^{4,\nu})$ *)
| *Dim6_AHHZ_D* of *int*
(* $\mathrm{i}(H_1 H_2 (\partial^\mu \partial^\nu A_\mu) Z_\nu - H_1 H_2 (\partial^\nu \partial_\nu A_\mu) Z^\mu)$ *)
| *Dim6_AHHZ_DP* of *int*
(* $\mathrm{i}((\partial^\mu H_1) H_2 (\partial^\nu A_\mu) Z_\nu + H_1 (\partial^\mu H_2)(\partial^\nu A_\mu) Z_\nu$
$- (\partial^\nu H_1) H_2 (\partial_\nu A_\mu) Z^\mu - H_1 (\partial^\nu H_2)(\partial_\nu A_\mu) Z^\mu)$ *)
| *Dim6_AHHZ_PB* of *int*
(* $\mathrm{i}(H_1 H_2 (\partial^\nu A_\mu)(\partial_\nu Z^\mu) - H_1 H_2 (\partial^\nu A_\mu)(\partial^\mu Z_\nu))$ *)

type α *vertexn* =
| *UFO* of *Algebra.QC.t* $\times$ *string* $\times$ *lorentzn* $\times$ *fermion_lines* $\times$ *Color.Vertex.t*

An obvious candidate for addition to *boson* is T, of course.

This list is sufficient for the minimal standard model, but not comprehensive enough for most of its extensions, supersymmetric or otherwise. In particular, we need a *general* parameterization for all trilinear vertices. One straightforward possibility are polynomials in the momenta for each combination of fields.

JR sez' (regarding the Majorana Feynman rules): Here we use the rules which can be found in [7] and are more properly described in *Targets* where the performing of the fusion rules in analytical expressions is encoded. *(JR's probably right, but I need to check myself ...)*

Signify which two of three fields are fused:

type *fuse2* = *F23* | *F32* | *F31* | *F13* | *F12* | *F21*

Signify which three of four fields are fused:

type *fuse3* =
| *F123* | *F231* | *F312* | *F132* | *F321* | *F213*
| *F124* | *F241* | *F412* | *F142* | *F421* | *F214*
| *F134* | *F341* | *F413* | *F143* | *F431* | *F314*
| *F234* | *F342* | *F423* | *F243* | *F432* | *F324*

Explicit enumeration types make no sense for higher degrees.

type *fusen* = *int list*

The third member of the triplet will contain the coupling constant:

type α *t* =
| *V3* of α *vertex3* $\times$ *fuse2* $\times$ α
| *V4* of α *vertex4* $\times$ *fuse3* $\times$ α
| *Vn* of α *vertexn* $\times$ *fusen* $\times$ α

16.1.3 Gauge Couplings

Dimension-4 trilinear vector boson couplings

$$\begin{aligned} f_{abc}\partial^\mu A^{a,\nu} A^b_\mu A^c_\nu &\to \mathrm{i} f_{abc} k_1^\mu A^{a,\nu}(k_1) A^b_\mu(k_2) A^c_\nu(k_3) \\ &= -\frac{\mathrm{i}}{3!} f_{a_1 a_2 a_3} C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) A^{a_1}_{\mu_1}(k_1) A^{a_2}_{\mu_2}(k_2) A^{a_3}_{\mu_3}(k_3) \end{aligned} \tag{16.5a}$$

with the totally antisymmetric tensor (under simultaneous permutations of all quantum numbers μ_i and k_i) and all momenta *outgoing*

$$C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) = (g^{\mu_1\mu_2}(k_1^{\mu_3} - k_2^{\mu_3}) + g^{\mu_2\mu_3}(k_2^{\mu_1} - k_3^{\mu_1}) + g^{\mu_3\mu_1}(k_3^{\mu_2} - k_1^{\mu_2})) \tag{16.5b}$$

Since $f_{a_1 a_2 a_3} C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3)$ is totally symmetric (under simultaneous permutations of all quantum numbers a_i, μ_i and k_i), it is easy to take the partial derivative

$$A^{a,\mu}(k_2+k_3) = -\frac{\mathrm{i}}{2!} f_{abc} C^{\mu\rho\sigma}(-k_2-k_3,k_2,k_3) A^b_\rho(k_2) A^c_\sigma(k_3) \tag{16.6a}$$

	only Dirac fermions	incl. Majorana fermions
FBF (*Psibar*, *S*, *Psi*): $\mathcal{L}_I = g_S\bar\psi_1 S\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_S\bar\psi_1 S$	$\psi_2 \leftarrow \mathrm{i}\cdot g_S\psi_1 S$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_S S\bar\psi_1$	$\psi_2 \leftarrow \mathrm{i}\cdot g_S S\psi_1$
F13	$S \leftarrow \mathrm{i}\cdot g_S\bar\psi_1\psi_2$	$S \leftarrow \mathrm{i}\cdot g_S\psi_1^T \mathrm{C}\psi_2$
F31	$S \leftarrow \mathrm{i}\cdot g_S\psi_{2,\alpha}\bar\psi_{1,\alpha}$	$S \leftarrow \mathrm{i}\cdot g_S\psi_2^T \mathrm{C}\psi_1$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_S S\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot g_S S\psi_2$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot g_S\psi_2 S$	$\psi_1 \leftarrow \mathrm{i}\cdot g_S\psi_2 S$
FBF (*Psibar*, *P*, *Psi*): $\mathcal{L}_I = g_P\bar\psi_1 P\gamma_5\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_P\bar\psi_1\gamma_5 P$	$\psi_2 \leftarrow \mathrm{i}\cdot g_P\gamma_5\psi_1 P$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_P P\bar\psi_1\gamma_5$	$\psi_2 \leftarrow \mathrm{i}\cdot g_P P\gamma_5\psi_1$
F13	$P \leftarrow \mathrm{i}\cdot g_P\bar\psi_1\gamma_5\psi_2$	$P \leftarrow \mathrm{i}\cdot g_P\psi_1^T \mathrm{C}\gamma_5\psi_2$
F31	$P \leftarrow \mathrm{i}\cdot g_P[\gamma_5\psi_2]_\alpha\bar\psi_{1,\alpha}$	$P \leftarrow \mathrm{i}\cdot g_P\psi_2^T \mathrm{C}\gamma_5\psi_1$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_P P\gamma_5\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot g_P P\gamma_5\psi_2$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot g_P\gamma_5\psi_2 P$	$\psi_1 \leftarrow \mathrm{i}\cdot g_P\gamma_5\psi_2 P$
FBF (*Psibar*, *V*, *Psi*): $\mathcal{L}_I = g_V\bar\psi_1\not{V}\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_V\bar\psi_1\not{V}$	$\psi_{2,\alpha} \leftarrow \mathrm{i}\cdot(-g_V)\psi_{1,\beta}\not{V}_{\alpha\beta}$
F21	$\bar\psi_{2,\beta} \leftarrow \mathrm{i}\cdot g_V\not{V}_{\alpha\beta}\bar\psi_{1,\alpha}$	$\psi_2 \leftarrow \mathrm{i}\cdot(-g_V)\not{V}\psi_1$
F13	$V_\mu \leftarrow \mathrm{i}\cdot g_V\bar\psi_1\gamma_\mu\psi_2$	$V_\mu \leftarrow \mathrm{i}\cdot g_V(\psi_1)^T \mathrm{C}\gamma_\mu\psi_2$
F31	$V_\mu \leftarrow \mathrm{i}\cdot g_V[\gamma_\mu\psi_2]_\alpha\bar\psi_{1,\alpha}$	$V_\mu \leftarrow \mathrm{i}\cdot(-g_V)(\psi_2)^T \mathrm{C}\gamma_\mu\psi_1$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_V\not{V}\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot g_V\not{V}\psi_2$
F32	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_V\psi_{2,\beta}\not{V}_{\alpha\beta}$	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_V\psi_{2,\beta}\not{V}_{\alpha\beta}$
FBF (*Psibar*, *A*, *Psi*): $\mathcal{L}_I = g_A\bar\psi_1\gamma_5\not{A}\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_A\bar\psi_1\gamma_5\not{A}$	$\psi_{2,\alpha} \leftarrow \mathrm{i}\cdot g_A\psi_\beta[\gamma_5\not{A}]_{\alpha\beta}$
F21	$\bar\psi_{2,\beta} \leftarrow \mathrm{i}\cdot g_A[\gamma_5\not{A}]_{\alpha\beta}\bar\psi_{1,\alpha}$	$\psi_2 \leftarrow \mathrm{i}\cdot g_A\gamma_5\not{A}\psi$
F13	$A_\mu \leftarrow \mathrm{i}\cdot g_A\bar\psi_1\gamma_5\gamma_\mu\psi_2$	$A_\mu \leftarrow \mathrm{i}\cdot g_A\psi_1^T \mathrm{C}\gamma_5\gamma_\mu\psi_2$
F31	$A_\mu \leftarrow \mathrm{i}\cdot g_A[\gamma_5\gamma_\mu\psi_2]_\alpha\bar\psi_{1,\alpha}$	$A_\mu \leftarrow \mathrm{i}\cdot g_A\psi_2^T \mathrm{C}\gamma_5\gamma_\mu\psi_1$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_A\gamma_5\not{A}\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot g_A\gamma_5\not{A}\psi_2$
F32	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_A\psi_{2,\beta}[\gamma_5\not{A}]_{\alpha\beta}$	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_A\psi_{2,\beta}[\gamma_5\not{A}]_{\alpha\beta}$

Table 16.3: Dimension-4 trilinear fermionic couplings. The momenta are unambiguous, because there are no derivative couplings and all participating fields are different.

	only Dirac fermions	incl. Majorana fermions
FBF (*Psibar*, *T*, *Psi*): $\mathcal{L}_I = g_T T_{\mu\nu}\bar\psi_1[\gamma^\mu,\gamma^\nu]_-\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_T\bar\psi_1[\gamma^\mu,\gamma^\nu]_- T_{\mu\nu}$	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_T\cdots$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_T T_{\mu\nu}\bar\psi_1[\gamma^\mu,\gamma^\nu]_-$	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_T\cdots$
F13	$T_{\mu\nu} \leftarrow \mathrm{i}\cdot g_T\bar\psi_1[\gamma_\mu,\gamma_\nu]_-\psi_2$	$T_{\mu\nu} \leftarrow \mathrm{i}\cdot g_T\cdots$
F31	$T_{\mu\nu} \leftarrow \mathrm{i}\cdot g_T[[\gamma_\mu,\gamma_\nu]_-\psi_2]_\alpha\bar\psi_{1,\alpha}$	$T_{\mu\nu} \leftarrow \mathrm{i}\cdot g_T\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_T T_{\mu\nu}[\gamma^\mu,\gamma^\nu]_-\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot g_T\cdots$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot g_T[\gamma^\mu,\gamma^\nu]_-\psi_2 T_{\mu\nu}$	$\psi_1 \leftarrow \mathrm{i}\cdot g_T\cdots$

Table 16.4: Dimension-5 trilinear fermionic couplings (NB: the coefficients and signs are not fixed yet). The momenta are unambiguous, because there are no derivative couplings and all participating fields are different.

	only Dirac fermions	incl. Majorana fermions
FBF (*Psibar*, *SP*, *Psi*): $\mathcal{L}_I = \bar\psi_1\phi(g_S + g_P\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot\bar\psi_1(g_S + g_P\gamma_5)\phi$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot\phi\bar\psi_1(g_S + g_P\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$\phi \leftarrow \mathrm{i}\cdot\bar\psi_1(g_S + g_P\gamma_5)\psi_2$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F31	$\phi \leftarrow \mathrm{i}\cdot[(g_S + g_P\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot\phi(g_S + g_P\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot(g_S + g_P\gamma_5)\psi_2\phi$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *SL*, *Psi*): $\mathcal{L}_I = g_L\bar\psi_1\phi(1-\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_L\bar\psi_1(1-\gamma_5)\phi$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_L\phi\bar\psi_1(1-\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$\phi \leftarrow \mathrm{i}\cdot g_L\bar\psi_1(1-\gamma_5)\psi_2$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F31	$\phi \leftarrow \mathrm{i}\cdot g_L[(1-\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_L\phi(1-\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot g_L(1-\gamma_5)\psi_2\phi$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *SR*, *Psi*): $\mathcal{L}_I = g_R\bar\psi_1\phi(1+\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_R\bar\psi_1(1+\gamma_5)\phi$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_R\phi\bar\psi_1(1+\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$\phi \leftarrow \mathrm{i}\cdot g_R\bar\psi_1(1+\gamma_5)\psi_2$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F31	$\phi \leftarrow \mathrm{i}\cdot g_R[(1+\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$\phi \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_R\phi(1+\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_1 \leftarrow \mathrm{i}\cdot g_R(1+\gamma_5)\psi_2\phi$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *SLR*, *Psi*): $\mathcal{L}_I = g_L\bar\psi_1\phi(1-\gamma_5)\psi_2 + g_R\bar\psi_1\phi(1+\gamma_5)\psi_2$		

Table 16.5: Combined dimension-4 trilinear fermionic couplings.

	only Dirac fermions	incl. Majorana fermions
FBF (*Psibar*, *VA*, *Psi*): $\mathcal{L}_I = \bar\psi_1 \not{Z}(g_V - g_A\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot\bar\psi_1\not{Z}(g_V - g_A\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_{2,\beta} \leftarrow \mathrm{i}\cdot[\not{Z}(g_V - g_A\gamma_5)]_{\alpha\beta}\bar\psi_{1,\alpha}$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$Z_\mu \leftarrow \mathrm{i}\cdot\bar\psi_1\gamma_\mu(g_V - g_A\gamma_5)\psi_2$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F31	$Z_\mu \leftarrow \mathrm{i}\cdot[\gamma_\mu(g_V - g_A\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot\not{Z}(g_V - g_A\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot\psi_{2,\beta}[\not{Z}(g_V - g_A\gamma_5)]_{\alpha\beta}$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *VL*, *Psi*): $\mathcal{L}_I = g_L\bar\psi_1\not{Z}(1-\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_L\bar\psi_1\not{Z}(1-\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_{2,\beta} \leftarrow \mathrm{i}\cdot g_L[\not{Z}(1-\gamma_5)]_{\alpha\beta}\bar\psi_{1,\alpha}$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$Z_\mu \leftarrow \mathrm{i}\cdot g_L\bar\psi_1\gamma_\mu(1-\gamma_5)\psi_2$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F31	$Z_\mu \leftarrow \mathrm{i}\cdot g_L[\gamma_\mu(1-\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_L\not{Z}(1-\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_L\psi_{2,\beta}[\not{Z}(1-\gamma_5)]_{\alpha\beta}$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *VR*, *Psi*): $\mathcal{L}_I = g_R\bar\psi_1\not{Z}(1+\gamma_5)\psi_2$		
F12	$\bar\psi_2 \leftarrow \mathrm{i}\cdot g_R\bar\psi_1\not{Z}(1+\gamma_5)$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F21	$\bar\psi_{2,\beta} \leftarrow \mathrm{i}\cdot g_R[\not{Z}(1+\gamma_5)]_{\alpha\beta}\bar\psi_{1,\alpha}$	$\psi_2 \leftarrow \mathrm{i}\cdot\cdots$
F13	$Z_\mu \leftarrow \mathrm{i}\cdot g_R\bar\psi_1\gamma_\mu(1+\gamma_5)\psi_2$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F31	$Z_\mu \leftarrow \mathrm{i}\cdot g_R[\gamma_\mu(1+\gamma_5)\psi_2]_\alpha\bar\psi_{1,\alpha}$	$Z_\mu \leftarrow \mathrm{i}\cdot\cdots$
F23	$\psi_1 \leftarrow \mathrm{i}\cdot g_R\not{Z}(1+\gamma_5)\psi_2$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
F32	$\psi_{1,\alpha} \leftarrow \mathrm{i}\cdot g_R\psi_{2,\beta}[\not{Z}(1+\gamma_5)]_{\alpha\beta}$	$\psi_1 \leftarrow \mathrm{i}\cdot\cdots$
FBF (*Psibar*, *VLR*, *Psi*): $\mathcal{L}_I = g_L\bar\psi_1\not{Z}(1-\gamma_5)\psi_2 + g_R\bar\psi_1\not{Z}(1+\gamma_5)\psi_2$		

Table 16.6: Combined dimension-4 trilinear fermionic couplings continued.

FBF (*Psibar*, *S*, *Chi*): $\bar\psi S\chi$	
F12: $\chi \leftarrow \psi S$	*F21*: $\chi \leftarrow S\psi$
F13: $S \leftarrow \psi^T\mathrm{C}\chi$	*F31*: $S \leftarrow \chi^T\mathrm{C}\psi$
F23: $\psi \leftarrow S\chi$	*F32*: $\psi \leftarrow \chi S$
FBF (*Psibar*, *P*, *Chi*): $\bar\psi P\gamma_5\chi$	
F12: $\chi \leftarrow \gamma_5\psi P$	*F21*: $\chi \leftarrow P\gamma_5\psi$
F13: $P \leftarrow \psi^T\mathrm{C}\gamma_5\chi$	*F31*: $P \leftarrow \chi^T\mathrm{C}\gamma_5\psi$
F23: $\psi \leftarrow P\gamma_5\chi$	*F32*: $\psi \leftarrow \gamma_5\chi P$
FBF (*Psibar*, *V*, *Chi*): $\bar\psi\not{V}\chi$	
F12: $\chi_\alpha \leftarrow -\psi_\beta\not{V}_{\alpha\beta}$	*F21*: $\chi \leftarrow -\not{V}\psi$
F13: $V_\mu \leftarrow \psi^T\mathrm{C}\gamma_\mu\chi$	*F31*: $V_\mu \leftarrow \chi^T\mathrm{C}(-\gamma_\mu\psi)$
F23: $\psi \leftarrow \not{V}\chi$	*F32*: $\psi_\alpha \leftarrow \chi_\beta\not{V}_{\alpha\beta}$
FBF (*Psibar*, *A*, *Chi*): $\bar\psi\gamma^5\not{A}\chi$	
F12: $\chi_\alpha \leftarrow \psi_\beta[\gamma^5\not{A}]_{\alpha\beta}$	*F21*: $\chi \leftarrow \gamma^5\not{A}\psi$
F13: $A_\mu \leftarrow \psi^T\mathrm{C}\gamma^5\gamma_\mu\chi$	*F31*: $A_\mu \leftarrow \chi^T\mathrm{C}(\gamma^5\gamma_\mu\psi)$
F23: $\psi \leftarrow \gamma^5\not{A}\chi$	*F32*: $\psi_\alpha \leftarrow \chi_\beta[\gamma^5\not{A}]_{\alpha\beta}$

Table 16.7: Dimension-4 trilinear couplings including one Dirac and one Majorana fermion

FBF (*Psibar*, *SP*, *Chi*): $\bar\psi\phi(g_S+g_P\gamma_5)\chi$	
F12: $\chi\leftarrow(g_S+g_P\gamma_5)\psi\phi$	*F21*: $\chi\leftarrow\phi(g_S+g_P\gamma_5)\psi$
F13: $\phi\leftarrow\psi^T\mathrm{C}(g_S+g_P\gamma_5)\chi$	*F31*: $\phi\leftarrow\chi^T\mathrm{C}(g_S+g_P\gamma_5)\chi$
F23: $\psi\leftarrow\phi(g_S+g_P\gamma_5)\chi$	*F32*: $\psi\leftarrow(g_S+g_P\gamma_5)\chi\phi$
FBF (*Psibar*, *VA*, *Chi*): $\bar\psi\not{Z}(g_V-g_A\gamma_5)\chi$	
F12: $\chi_\alpha\leftarrow\psi_\beta[\not{Z}(-g_V-g_A\gamma_5)]_{\alpha\beta}$	*F21*: $\chi\leftarrow\not{Z}(-g_V-g_A\gamma_5)]\psi$
F13: $Z_\mu\leftarrow\psi^T\mathrm{C}\gamma_\mu(g_V-g_A\gamma_5)\chi$	*F31*: $Z_\mu\leftarrow\chi^T\mathrm{C}\gamma_\mu(-g_V-g_A\gamma_5)\psi$
F23: $\psi\leftarrow\not{Z}(g_V-g_A\gamma_5)\chi$	*F32*: $\psi_\alpha\leftarrow\chi_\beta[\not{Z}(g_V-g_A\gamma_5)]_{\alpha\beta}$

Table 16.8: Combined dimension-4 trilinear fermionic couplings including one Dirac and one Majorana fermion.

FBF (*Chibar*, *S*, *Psi*): $\bar\chi S\psi$	
F12: $\psi\leftarrow\chi S$	*F21*: $\psi\leftarrow S\chi$
F13: $S\leftarrow\chi^T\mathrm{C}\psi$	*F31*: $S\leftarrow\psi^T\mathrm{C}\chi$
F23: $\chi\leftarrow S\psi$	*F32*: $\chi\leftarrow\psi S$
FBF (*Chibar*, *P*, *Psi*): $\bar\chi P\gamma_5\psi$	
F12: $\psi\leftarrow\gamma_5\chi P$	*F21*: $\psi\leftarrow P\gamma_5\chi$
F13: $P\leftarrow\chi^T\mathrm{C}\gamma_5\psi$	*F31*: $P\leftarrow\psi^T\mathrm{C}\gamma_5\chi$
F23: $\chi\leftarrow P\gamma_5\psi$	*F32*: $\chi\leftarrow\gamma_5\psi P$
FBF (*Chibar*, *V*, *Psi*): $\bar\chi\not{V}\psi$	
F12: $\psi_\alpha\leftarrow-\chi_\beta\not{V}_{\alpha\beta}$	*F21*: $\psi\leftarrow-\not{V}\chi$
F13: $V_\mu\leftarrow\chi^T\mathrm{C}\gamma_\mu\psi$	*F31*: $V_\mu\leftarrow\psi^T\mathrm{C}(-\gamma_\mu\chi)$
F23: $\chi\leftarrow\not{V}\psi$	*F32*: $\chi_\alpha\leftarrow\psi_\beta\not{V}_{\alpha\beta}$
FBF (*Chibar*, *A*, *Psi*): $\bar\chi\gamma^5\not{A}\psi$	
F12: $\psi_\alpha\leftarrow\chi_\beta[\gamma^5\not{A}]_{\alpha\beta}$	*F21*: $\psi\leftarrow\gamma^5\not{A}\chi$
F13: $A_\mu\leftarrow\chi^T\mathrm{C}(\gamma^5\gamma_\mu\psi)$	*F31*: $A_\mu\leftarrow\psi^T\mathrm{C}\gamma^5\gamma_\mu\chi$
F23: $\chi\leftarrow\gamma^5\not{A}\psi$	*F32*: $\chi_\alpha\leftarrow\psi_\beta[\gamma^5\not{A}]_{\alpha\beta}$

Table 16.9: Dimension-4 trilinear couplings including one Dirac and one Majorana fermion

FBF (*Chibar*, *SP*, *Psi*): $\bar\chi\phi(g_S+g_P\gamma_5)\psi$	
F12: $\psi\leftarrow(g_S+g_P\gamma_5)\chi\phi$	*F21*: $\psi\leftarrow\phi(g_S+g_P\gamma_5)\chi$
F13: $\phi\leftarrow\chi^T\mathrm{C}(g_S+g_P\gamma_5)\psi$	*F31*: $\phi\leftarrow\psi^T\mathrm{C}(g_S+g_P\gamma_5)\chi$
F23: $\chi\leftarrow\phi(g_S+g_P\gamma_5)\psi$	*F32*: $\chi\leftarrow(g_S+g_P\gamma_5)\psi\phi$
FBF (*Chibar*, *VA*, *Psi*): $\bar\chi\not{Z}(g_V-g_A\gamma_5)\psi$	
F12: $\psi_\alpha\leftarrow\chi_\beta[\not{Z}(-g_V-g_A\gamma_5)]_{\alpha\beta}$	*F21*: $\psi\leftarrow\not{Z}(-g_V-g_A\gamma_5)\chi$
F13: $Z_\mu\leftarrow\chi^T\mathrm{C}\gamma_\mu(g_V-g_A\gamma_5)\psi$	*F31*: $Z_\mu\leftarrow\psi^T\mathrm{C}\gamma_\mu(-g_V-g_A\gamma_5)\chi$
F23: $\chi\leftarrow\not{Z}(g_V-g_A\gamma_5)]\psi$	*F32*: $\chi_\alpha\leftarrow\psi_\beta[\not{Z}(g_V-g_A\gamma_5)]_{\alpha\beta}$

Table 16.10: Combined dimension-4 trilinear fermionic couplings including one Dirac and one Majorana fermion.

FBF (*Chibar*, *S*, *Chi*): $\bar{\chi}_a S \chi_b$	
F12: $\chi_b \leftarrow \chi_a S$	*F21*: $\chi_b \leftarrow S\chi_a$
F13: $S \leftarrow \chi_a^T \mathrm{C} \chi_b$	*F31*: $S \leftarrow \chi_b^T \mathrm{C} \chi_a$
F23: $\chi_a \leftarrow S\chi_b$	*F32*: $\chi_a \leftarrow \chi S_b$
FBF (*Chibar*, *P*, *Chi*): $\bar{\chi}_a P \gamma_5 \psi_b$	
F12: $\chi_b \leftarrow \gamma_5 \chi_a P$	*F21*: $\chi_b \leftarrow P\gamma_5\chi_a$
F13: $P \leftarrow \chi_a^T \mathrm{C} \gamma_5 \chi_b$	*F31*: $P \leftarrow \chi_b^T \mathrm{C}\gamma_5 \chi_a$
F23: $\chi_a \leftarrow P\gamma_5\chi_b$	*F32*: $\chi_a \leftarrow \gamma_5 \chi_b P$
FBF (*Chibar*, *V*, *Chi*): $\bar{\chi}_a \slashed{V} \chi_b$	
F12: $\chi_{b,\alpha} \leftarrow -\chi_{a,\beta} \slashed{V}_{\alpha\beta}$	*F21*: $\chi_b \leftarrow -\slashed{V}\chi_a$
F13: $V_\mu \leftarrow \chi_a^T \mathrm{C}\gamma_\mu \chi_b$	*F31*: $V_\mu \leftarrow -\chi_b^T \mathrm{C}\gamma_\mu \chi_a$
F23: $\chi_a \leftarrow \slashed{V}\chi_b$	*F32*: $\chi_{a,\alpha} \leftarrow \chi_{b,\beta} \slashed{V}_{\alpha\beta}$
FBF (*Chibar*, *A*, *Chi*): $\bar{\chi}_a \gamma^5 \slashed{A} \chi_b$	
F12: $\chi_{b,\alpha} \leftarrow \chi_{a,\beta} [\gamma^5 \slashed{A}]_{\alpha\beta}$	*F21*: $\chi_b \leftarrow \gamma^5 \slashed{A} \chi_a$
F13: $A_\mu \leftarrow \chi_a^T \mathrm{C}\gamma^5 \gamma_\mu \chi_b$	*F31*: $A_\mu \leftarrow \chi_b^T \mathrm{C}(\gamma^5\gamma_\mu \chi_a)$
F23: $\chi_a \leftarrow \gamma^5 \slashed{A} \chi_b$	*F32*: $\chi_{a,\alpha} \leftarrow \chi_{b,\beta} [\gamma^5 \slashed{A}]_{\alpha\beta}$

Table 16.11: Dimension-4 trilinear couplings of two Majorana fermions

FBF (*Chibar*, *SP*, *Chi*): $\bar{\chi}\phi_a (g_S + g_P\gamma_5)\chi_b$	
F12: $\chi_b \leftarrow (g_S + g_P\gamma_5)\chi_a \phi$	*F21*: $\chi_b \leftarrow \phi(g_S + g_P\gamma_5)\chi_a$
F13: $\phi \leftarrow \chi_a^T \mathrm{C}(g_S + g_P\gamma_5)\chi_b$	*F31*: $\phi \leftarrow \chi_b^T \mathrm{C}(g_S + g_P\gamma_5)\chi_a$
F23: $\chi_a \leftarrow \phi(g_S + g_P\gamma_5)\chi_b$	*F32*: $\chi_a \leftarrow (g_S + g_P\gamma_5)\chi_b \phi$
FBF (*Chibar*, *VA*, *Chi*): $\bar{\chi}_a \slashed{Z}(g_V - g_A\gamma_5)\chi_b$	
F12: $\chi_{b,\alpha} \leftarrow \chi_{a,\beta}[\slashed{Z}(-g_V - g_A\gamma_5)]_{\alpha\beta}$	*F21*: $\chi_b \leftarrow \slashed{Z}(-g_V - g_A\gamma_5)]\chi_a$
F13: $Z_\mu \leftarrow \chi_a^T \mathrm{C}\gamma_\mu (g_V - g_A\gamma_5)\chi_b$	*F31*: $Z_\mu \leftarrow \chi_b^T \mathrm{C}\gamma_\mu(-g_V - g_A\gamma_5)\chi_a$
F23: $\chi_a \leftarrow \slashed{Z}(g_V - g_A\gamma_5)\chi_b$	*F32*: $\chi_{a,\alpha} \leftarrow \chi_{b,\beta}[\slashed{Z}(g_V - g_A\gamma_5)]_{\alpha\beta}$

Table 16.12: Combined dimension-4 trilinear fermionic couplings of two Majorana fermions.

Gauge_Gauge_Gauge: $\mathcal{L}_I = g f_{abc} A_a^\mu A_b^\nu \partial_\mu A_{c,\nu}$
_: $A_a^\mu \leftarrow \mathrm{i}\cdot(-\mathrm{i}g/2)\cdot C_{abc}^{\mu\rho\sigma}(-k_2 - k_3, k_2, k_3) A_\rho^b A_\sigma^c$
Aux_Gauge_Gauge: $\mathcal{L}_I = g f_{abc} X_{a,\mu\nu}(k_1)(A_b^\mu(k_2) A_c^\nu(k_3) - A_b^\nu(k_2) A_c^\mu(k_3))$
F23$\vee$*F32*: $X_a^{\mu\nu}(k_2 + k_3) \leftarrow \mathrm{i}\cdot g f_{abc}(A_b^\mu(k_2) A_c^\nu(k_3) - A_b^\nu(k_2) A_c^\mu(k_3))$
F12$\vee$*F13*: $A_{a,\mu}(k_1 + k_{2/3}) \leftarrow \mathrm{i}\cdot g f_{abc} X_{b,\nu\mu}(k_1) A_c^\nu(k_{2/3})$
F21$\vee$*F31*: $A_{a,\mu}(k_{2/3} + k_1) \leftarrow \mathrm{i}\cdot g f_{abc} A_b^\nu(k_{2/3}) X_{c,\mu\nu}(k_1)$

Table 16.13: Dimension-4 Vector Boson couplings with *outgoing* momenta. See (16.5b) and (16.6b) for the definition of the antisymmetric tensor $C^{\mu_1\mu_2\mu_3}(k_1, k_2, k_3)$.

Scalar_Vector_Vector: $\mathcal{L}_I = g\phi V_1^\mu V_{2,\mu}$	
F13: $\leftarrow \mathrm{i}\cdot g\cdots$	*F31*: $\leftarrow \mathrm{i}\cdot g\cdots$
F12: $\leftarrow \mathrm{i}\cdot g\cdots$	*F21*: $\leftarrow \mathrm{i}\cdot g\cdots$
F23: $\phi \leftarrow \mathrm{i}\cdot gV_1^\mu V_{2,\mu}$	*F32*: $\phi \leftarrow \mathrm{i}\cdot gV_{2,\mu}V_1^\mu$
Aux_Vector_Vector: $\mathcal{L}_I = gXV_1^\mu V_{2,\mu}$	
F13: $\leftarrow \mathrm{i}\cdot g\cdots$	*F31*: $\leftarrow \mathrm{i}\cdot g\cdots$
F12: $\leftarrow \mathrm{i}\cdot g\cdots$	*F21*: $\leftarrow \mathrm{i}\cdot g\cdots$
F23: $X \leftarrow \mathrm{i}\cdot gV_1^\mu V_{2,\mu}$	*F32*: $X \leftarrow \mathrm{i}\cdot gV_{2,\mu}V_1^\mu$
Aux_Scalar_Vector: $\mathcal{L}_I = gX^\mu\phi V_\mu$	
F13: $\leftarrow \mathrm{i}\cdot g\cdots$	*F31*: $\leftarrow \mathrm{i}\cdot g\cdots$
F12: $\leftarrow \mathrm{i}\cdot g\cdots$	*F21*: $\leftarrow \mathrm{i}\cdot g\cdots$
F23: $\leftarrow \mathrm{i}\cdot g\cdots$	*F32*: $\leftarrow \mathrm{i}\cdot g\cdots$

Table 16.14: ...

Scalar_Scalar_Scalar: $\mathcal{L}_I = g\phi_1\phi_2\phi_3$	
F13: $\phi_2 \leftarrow \mathrm{i}\cdot g\phi_1\phi_3$	*F31*: $\phi_2 \leftarrow \mathrm{i}\cdot g\phi_3\phi_1$
F12: $\phi_3 \leftarrow \mathrm{i}\cdot g\phi_1\phi_2$	*F21*: $\phi_3 \leftarrow \mathrm{i}\cdot g\phi_2\phi_1$
F23: $\phi_1 \leftarrow \mathrm{i}\cdot g\phi_2\phi_3$	*F32*: $\phi_1 \leftarrow \mathrm{i}\cdot g\phi_3\phi_2$
Aux_Scalar_Scalar: $\mathcal{L}_I = gX\phi_1\phi_2$	
F13: $\leftarrow \mathrm{i}\cdot g\cdots$	*F31*: $\leftarrow \mathrm{i}\cdot g\cdots$
F12: $\leftarrow \mathrm{i}\cdot g\cdots$	*F21*: $\leftarrow \mathrm{i}\cdot g\cdots$
F23: $X \leftarrow \mathrm{i}\cdot g\phi_1\phi_2$	*F32*: $X \leftarrow \mathrm{i}\cdot g\phi_2\phi_1$

Table 16.15: ...

Vector_Scalar_Scalar: $\mathcal{L}_I = gV^\mu\phi_1\mathrm{i}\overleftrightarrow{\partial_\mu}\phi_2$
F23: $V^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)\phi_1(k_2)\phi_2(k_3)$
F32: $V^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)\phi_2(k_3)\phi_1(k_2)$
F12: $\phi_2(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\mu + 2k_2^\mu)V_\mu(k_1)\phi_1(k_2)$
F21: $\phi_2(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\mu + 2k_2^\mu)\phi_1(k_2)V_\mu(k_1)$
F13: $\phi_1(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\mu - 2k_3^\mu)V_\mu(k_1)\phi_2(k_3)$
F31: $\phi_1(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\mu - 2k_3^\mu)\phi_2(k_3)V_\mu(k_1)$

Table 16.16: ...

Aux_DScalar_DScalar: $\mathcal{L}_I = g\chi(\mathrm{i}\partial_\mu\phi_1)(\mathrm{i}\partial^\mu\phi_2)$
F23: $\chi(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2\cdot k_3)\phi_1(k_2)\phi_2(k_3)$
F32: $\chi(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_3\cdot k_2)\phi_2(k_3)\phi_1(k_2)$
F12: $\phi_2(k_1+k_2) \leftarrow \mathrm{i}\cdot g((-k_1-k_2)\cdot k_2)\chi(k_1)\phi_1(k_2)$
F21: $\phi_2(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_2\cdot(-k_1-k_2))\phi_1(k_2)\chi(k_1)$
F13: $\phi_1(k_1+k_3) \leftarrow \mathrm{i}\cdot g((-k_1-k_3)\cdot k_3)\chi(k_1)\phi_2(k_3)$
F31: $\phi_1(k_1+k_3) \leftarrow \mathrm{i}\cdot g(k_3\cdot(-k_1-k_3))\phi_2(k_3)\chi(k_1)$

Table 16.17: ...

Aux_Vector_DScalar: $\mathcal{L}_I = g\chi V_\mu(\mathrm{i}\partial^\mu\phi)$	
F23:	$\chi(k_2+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu V_\mu(k_2)\phi(k_3)$
F32:	$\chi(k_2+k_3) \leftarrow \mathrm{i}\cdot g\phi(k_3)k_3^\mu V_\mu(k_2)$
F12:	$\phi(k_1+k_2) \leftarrow \mathrm{i}\cdot g\chi(k_1)(-k_1-k_2)^\mu V_\mu(k_2)$
F21:	$\phi(k_1+k_2) \leftarrow \mathrm{i}\cdot g(-k_1-k_2)^\mu V_\mu(k_2)\chi(k_1)$
F13:	$V_\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1-k_3)_\mu\chi(k_1)\phi(k_3)$
F31:	$V_\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1-k_3)_\mu\phi(k_3)\chi(k_1)$

Table 16.18: ...

with

$$C^{\mu\rho\sigma}(-k_2-k_3,k_2,k_3) = (g^{\rho\sigma}(k_2^\mu - k_3^\mu) + g^{\mu\sigma}(2k_3^\rho + k_2^\rho) - g^{\mu\rho}(2k_2^\sigma + k_3^\sigma)) \tag{16.6b}$$

i. e.

$$A^{a,\mu}(k_2+k_3) = -\frac{\mathrm{i}}{2!}f_{abc}\big((k_2^\mu - k_3^\mu)A^b(k_2)\cdot A^c(k_3) \\ + (2k_3+k_2)\cdot A^b(k_2)A^{c,\mu}(k_3) - A^{b,\mu}(k_2)A^c(k_3)\cdot(2k_2+k_3)\big) \tag{16.6c}$$

Investigate the rearrangements proposed in [5] for improved numerical stability.

Non-Gauge Vector Couplings

As a basis for the dimension-4 couplings of three vector bosons, we choose "transversal" and "longitudinal" (with respect to the first vector field) tensors that are odd and even under permutation of the second and third argument

$$\mathcal{L}_T(V_1,V_2,V_3) = V_1^\mu(V_{2,\nu}\mathrm{i}\overleftrightarrow{\partial_\mu}V_3^\nu) = -\mathcal{L}_T(V_1,V_3,V_2) \tag{16.7a}$$
$$\mathcal{L}_L(V_1,V_2,V_3) = (\mathrm{i}\partial_\mu V_1^\mu)V_{2,\nu}V_3^\nu = \mathcal{L}_L(V_1,V_3,V_2) \tag{16.7b}$$

Using partial integration in $\mathcal{L}_L$, we find the convenient combinations

$$\mathcal{L}_T(V_1,V_2,V_3) + \mathcal{L}_L(V_1,V_2,V_3) = -2V_1^\mu\mathrm{i}\partial_\mu V_{2,\nu}V_3^\nu \tag{16.8a}$$
$$\mathcal{L}_T(V_1,V_2,V_3) - \mathcal{L}_L(V_1,V_2,V_3) = 2V_1^\mu V_{2,\nu}\mathrm{i}\partial_\mu V_3^\nu \tag{16.8b}$$

As an important example, we can rewrite the dimension-4 "anomalous" triple gauge couplings

$$\mathrm{i}\mathcal{L}_{\mathrm{TGC}}(g_1,\kappa,g_4)/g_{VWW} = g_1V^\mu(W^-_{\mu\nu}W^{+,\nu} - W^+_{\mu\nu}W^{-,\nu}) \\ + \kappa W^+_\mu W^-_\nu V^{\mu\nu} + g_4W^+_\mu W^-_\nu(\partial^\mu V^\nu + \partial^\nu V^\mu) \tag{16.9}$$

as

$$\mathcal{L}_{\mathrm{TGC}}(g_1,\kappa,g_4) = g_1\mathcal{L}_T(V,W^-,W^+) \\ - \frac{\kappa+g_1-g_4}{2}\mathcal{L}_T(W^-,V,W^+) + \frac{\kappa+g_1+g_4}{2}\mathcal{L}_T(W^+,V,W^-) \\ - \frac{\kappa-g_1-g_4}{2}\mathcal{L}_L(W^-,V,W^+) + \frac{\kappa-g_1+g_4}{2}\mathcal{L}_L(W^+,V,W^-) \tag{16.10}$$

CP Violation

$$\mathcal{L}_{\tilde T}(V_1,V_2,V_3) = V_{1,\mu}(V_{2,\rho}\mathrm{i}\overleftrightarrow{\partial_\nu}V_{3,\sigma})\epsilon^{\mu\nu\rho\sigma} = +\mathcal{L}_T(V_1,V_3,V_2) \tag{16.11a}$$
$$\mathcal{L}_{\tilde L}(V_1,V_2,V_3) = (\mathrm{i}\partial_\mu V_{1,\nu})V_{2,\rho}V_{3,\sigma}\epsilon^{\mu\nu\rho\sigma} = -\mathcal{L}_L(V_1,V_3,V_2) \tag{16.11b}$$

Here the notations $\tilde T$ and $\tilde L$ are clearly *abuse de langage*, because $\mathcal{L}_{\tilde L}(V_1,V_2,V_3)$ is actually the transversal combination, due to the antisymmetry of ϵ. Using partial integration in $\mathcal{L}_{\tilde L}$, we could again find combinations

$$\mathcal{L}_{\tilde T}(V_1,V_2,V_3) + \mathcal{L}_{\tilde L}(V_1,V_2,V_3) = -2V_{1,\mu}V_{2,\nu}\mathrm{i}\partial_\rho V_{3,\sigma}\epsilon^{\mu\nu\rho\sigma} \tag{16.12a}$$

Dim4_Vector_Vector_Vector_T: $\mathcal{L}_I = gV_1^\mu V_{2,\nu}\mathrm{i}\overleftrightarrow{\partial_\mu}V_3^\nu$
F23: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)V_{2,\nu}(k_2)V_3^\nu(k_3)$
F32: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)V_3^\nu(k_3)V_{2,\nu}(k_2)$
F12: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(2k_2^\nu + k_1^\nu)V_{1,\nu}(k_1)V_2^\mu(k_2)$
F21: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(2k_2^\nu + k_1^\nu)V_2^\mu(k_2)V_{1,\nu}(k_1)$
F13: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\nu - 2k_3^\nu)V_1^\nu(k_1)V_3^\mu(k_3)$
F31: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\nu - 2k_3^\nu)V_3^\mu(k_3)V_1^\nu(k_1)$
Dim4_Vector_Vector_Vector_L: $\mathcal{L}_I = g\mathrm{i}\partial_\mu V_1^\mu V_{2,\nu}V_3^\nu$
F23: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu + k_3^\mu)V_{2,\nu}(k_2)V_3^\nu(k_3)$
F32: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu + k_3^\mu)V_3^\nu(k_3)V_{2,\nu}(k_2)$
F12: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(-k_1^\nu)V_{1,\nu}(k_1)V_2^\mu(k_2)$
F21: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(-k_1^\nu)V_2^\mu(k_2)V_{1,\nu}(k_1)$
F13: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\nu)V_1^\nu(k_1)V_3^\mu(k_3)$
F31: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(-k_1^\nu)V_3^\mu(k_3)V_1^\nu(k_1)$

Table 16.19: ...

Dim4_Vector_Vector_Vector_T5: $\mathcal{L}_I = gV_{1,\mu}V_{2,\rho}\mathrm{i}\overleftrightarrow{\partial_\nu}V_{3,\sigma}\epsilon^{\mu\nu\rho\sigma}$
F23: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(k_{2,\nu} - k_{3,\nu})V_{2,\rho}(k_2)V_{3,\sigma}(k_3)$
F32: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(k_{2,\nu} - k_{3,\nu})V_{3,\sigma}(k_3)V_{2,\rho}(k_2)$
F12: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(2k_{2,\nu} + k_{1,\nu})V_{1,\rho}(k_1)V_{2,\sigma}(k_2)$
F21: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(2k_{2,\nu} + k_{1,\nu})V_{2,\sigma}(k_2)V_{1,\rho}(k_1)$
F13: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu} - 2k_{3,\nu})V_{1,\rho}(k_1)V_{3,\sigma}(k_3)$
F31: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu} - 2k_{3,\nu})V_{3,\sigma}(k_3)V_{1,\rho}(k_1)$
Dim4_Vector_Vector_Vector_L5: $\mathcal{L}_I = g\mathrm{i}\partial_\mu V_{1,\nu}V_{2,\nu}V_{3,\sigma}\epsilon^{\mu\nu\rho\sigma}$
F23: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(k_{2,\nu} + k_{3,\nu})V_{2,\rho}(k_2)V_{3,\sigma}(k_3)$
F32: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(k_{2,\nu} + k_{3,\nu})V_{2,\rho}(k_2)V_{3,\sigma}(k_3)$
F12: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu})V_{1,\rho}(k_1)V_{2,\sigma}(k_2)$
F21: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu})V_{2,\sigma}(k_2)V_{1,\rho}(k_1)$
F13: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu})V_{1,\rho}(k_1)V_{3,\sigma}(k_3)$
F31: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g\epsilon^{\mu\nu\rho\sigma}(-k_{1,\nu})V_{3,\sigma}(k_3)V_{1,\rho}(k_1)$

Table 16.20: ...

$$\mathcal{L}_{\tilde{T}}(V_1,V_2,V_3) - \mathcal{L}_{\tilde{L}}(V_1,V_2,V_3) = -2V_{1,\mu}\mathrm{i}\partial_\nu V_{2,\rho}V_{3,\sigma}\epsilon^{\mu\nu\rho\sigma} \tag{16.12b}$$

but we don't need them, since

$$\mathrm{i}\mathcal{L}_{\mathrm{TGC}}(g_5,\tilde{\kappa})/g_{VWW} = g_5\epsilon_{\mu\nu\rho\sigma}(W^{+,\mu}\mathrm{i}\overleftrightarrow{\partial^\rho}W^{-,\nu})V^\sigma \\ - \frac{\tilde{\kappa}_V}{2}W^-_\mu W^+_\nu\epsilon^{\mu\nu\rho\sigma}V_{\rho\sigma} \tag{16.13}$$

is immediately recognizable as

$$\mathcal{L}_{\mathrm{TGC}}(g_5,\tilde{\kappa})/g_{VWW} = -\mathrm{i}g_5\mathcal{L}_{\tilde{L}}(V,W^-,W^+) + \tilde{\kappa}\mathcal{L}_{\tilde{T}}(V,W^-,W^+) \tag{16.14}$$

Dim6_Gauge_Gauge_Gauge: $\mathcal{L}_I = gF_1^{\mu\nu}F_{2,\nu\rho}F_{3,\ \mu}^{\ \ \rho}$
_: $A_1^\mu(k_2+k_3) \leftarrow -\mathrm{i}\cdot\Lambda^{\mu\rho\sigma}(-k_2-k_3,k_2,k_3)A_{2,\rho}A_{c,\sigma}$

Table 16.21: ...

Dim6_Gauge_Gauge_Gauge_5: $\mathcal{L}_I = g/2\cdot\epsilon^{\mu\nu\lambda\tau}F_{1,\mu\nu}F_{2,\tau\rho}F_{3,\ \lambda}^{\ \ \rho}$
F23: $A_1^\mu(k_2+k_3) \leftarrow -\mathrm{i}\cdot\Lambda_5^{\mu\rho\sigma}(-k_2-k_3,k_2,k_3)A_{2,\rho}A_{3,\sigma}$
F32: $A_1^\mu(k_2+k_3) \leftarrow -\mathrm{i}\cdot\Lambda_5^{\mu\rho\sigma}(-k_2-k_3,k_2,k_3)A_{3,\sigma}A_{2,\rho}$
F12: $A_3^\mu(k_1+k_2) \leftarrow -\mathrm{i}\cdot$
F21: $A_3^\mu(k_1+k_2) \leftarrow -\mathrm{i}\cdot$
F13: $A_2^\mu(k_1+k_3) \leftarrow -\mathrm{i}\cdot$
F31: $A_2^\mu(k_1+k_3) \leftarrow -\mathrm{i}\cdot$

Table 16.22: ...

16.1.4 SU(2) Gauge Bosons

An important special case for table 16.13 are the two usual coordinates of SU(2)

$$W_\pm = \frac{1}{\sqrt{2}}(W_1 \mp \mathrm{i}W_2) \tag{16.15}$$

i. e.

$$W_1 = \frac{1}{\sqrt{2}}(W_+ + W_-) \tag{16.16a}$$

$$W_2 = \frac{\mathrm{i}}{\sqrt{2}}(W_+ - W_-) \tag{16.16b}$$

and

$$W_1^\mu W_2^\nu - W_2^\mu W_1^\nu = \mathrm{i}\left(W_-^\mu W_+^\nu - W_+^\mu W_-^\nu\right) \tag{16.17}$$

Thus the symmtry remains after the change of basis:

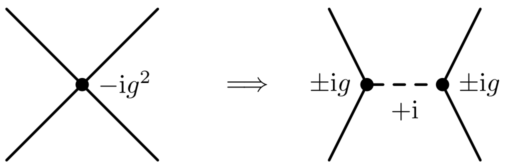
$$\epsilon^{abc}W_a^{\mu_1}W_b^{\mu_2}W_c^{\mu_3} = \mathrm{i}W_-^{\mu_1}(W_+^{\mu_2}W_3^{\mu_3} - W_3^{\mu_2}W_+^{\mu_3}) \\ + \mathrm{i}W_+^{\mu_1}(W_3^{\mu_2}W_-^{\mu_3} - W_-^{\mu_2}W_3^{\mu_3}) + \mathrm{i}W_3^{\mu_1}(W_-^{\mu_2}W_+^{\mu_3} - W_+^{\mu_2}W_-^{\mu_3}) \tag{16.18}$$

16.1.5 Quartic Couplings and Auxiliary Fields

Quartic couplings can be replaced by cubic couplings to a non-propagating auxiliary field. The quartic term should get a negative sign so that it the energy is bounded from below for identical fields. In the language of functional integrals

$$\mathcal{L}_{\phi^4} = -g^2\phi_1\phi_2\phi_3\phi_4 \Longrightarrow \\ \mathcal{L}_{X\phi^2} = X^*X \pm gX\phi_1\phi_2 \pm gX^*\phi_3\phi_4 = (X^* \pm g\phi_1\phi_2)(X \pm g\phi_3\phi_4) - g^2\phi_1\phi_2\phi_3\phi_4 \tag{16.19a}$$

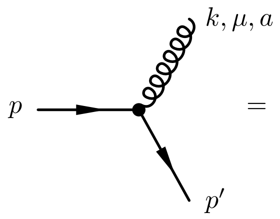
and in the language of Feynman diagrams



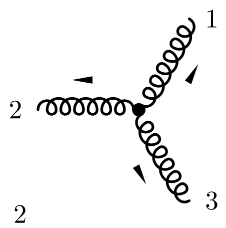
(16.19b)

The other choice of signs

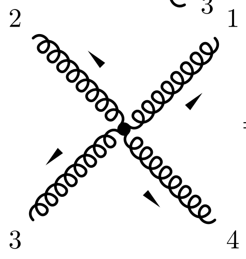
$$\mathcal{L}'_{X\phi^2} = -X^*X \pm gX\phi_1\phi_2 \mp gX^*\phi_3\phi_4 = -(X^* \pm g\phi_1\phi_2)(X \mp g\phi_3\phi_4) - g^2\phi_1\phi_2\phi_3\phi_4 \tag{16.20}$$



$$= \qquad +\mathrm{i}g\gamma_\mu T_a \tag{16.22a}$$

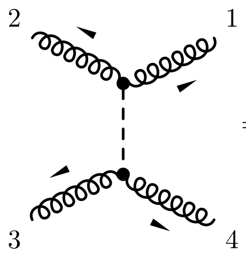


$$= \qquad gf_{a_1a_2a_3}C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3) \tag{16.22b}$$



$$\begin{aligned} = \qquad & -\mathrm{i}g^2 f_{a_1a_2b}f_{a_3a_4b}(g_{\mu_1\mu_3}g_{\mu_4\mu_2} - g_{\mu_1\mu_4}g_{\mu_2\mu_3}) \\ & -\mathrm{i}g^2 f_{a_1a_3b}f_{a_4a_2b}(g_{\mu_1\mu_4}g_{\mu_2\mu_3} - g_{\mu_1\mu_2}g_{\mu_3\mu_4}) \\ & -\mathrm{i}g^2 f_{a_1a_4b}f_{a_2a_3b}(g_{\mu_1\mu_2}g_{\mu_3\mu_4} - g_{\mu_1\mu_3}g_{\mu_4\mu_2}) \end{aligned} \tag{16.22c}$$

Figure 16.1: Gauge couplings. See (16.5b) for the definition of the antisymmetric tensor $C^{\mu_1\mu_2\mu_3}(k_1,k_2,k_3)$.



$$= -\mathrm{i}g^2 f_{a_1a_2b}f_{a_3a_4b}(g_{\mu_1\mu_3}g_{\mu_4\mu_2} - g_{\mu_1\mu_4}g_{\mu_2\mu_3}) \tag{16.23}$$

Figure 16.2: Gauge couplings.

can not be extended easily to identical particles and is therefore not used. For identical particles we have

$$\mathcal{L}_{\phi^4} = -\frac{g^2}{4!}\phi^4 \Longrightarrow$$

$$\mathcal{L}_{X\phi^2} = \frac{1}{2}X^2 \pm \frac{g}{2}X\phi^2 \pm \frac{g}{2}X\phi^2 = \frac{1}{2}\left(X \pm \frac{g}{2}\phi^2\right)\left(X \pm \frac{g}{2}\phi^2\right) - \frac{g^2}{4!}\phi^4 \tag{16.21}$$

Explain the factor 1/3 in the functional setting and its relation to the three diagrams in the graphical setting?

Quartic Gauge Couplings

The three crossed versions of figure 16.2 reproduces the quartic coupling in figure 16.1, because

$$\begin{aligned} & -\mathrm{i}g^2 f_{a_1a_2b}f_{a_3a_4b}(g_{\mu_1\mu_3}g_{\mu_4\mu_2} - g_{\mu_1\mu_4}g_{\mu_2\mu_3}) \\ & \qquad = (\mathrm{i}gf_{a_1a_2b}T_{\mu_1\mu_2,\nu_1\nu_2})\left(\frac{\mathrm{i}g^{\nu_1\nu_3}g^{\nu_2\nu_4}}{2}\right)(\mathrm{i}gf_{a_3a_4b}T_{\mu_3\mu_4,\nu_3\nu_4}) \end{aligned} \tag{16.24}$$

with $T_{\mu_1\mu_2,\mu_3\mu_4} = g_{\mu_1\mu_3}g_{\mu_4\mu_2} - g_{\mu_1\mu_4}g_{\mu_2\mu_3}$.

16.1.6 Gravitinos and supersymmetric currents

In supergravity theories there is a fermionic partner of the graviton, the gravitino. Therefore we have introduced the Lorentz type *Vectorspinor*.

GBG (*Fermbar*, *MOM*, *Ferm*): $\bar\psi_1(\mathrm{i}\not\partial \pm m)\phi\psi_2$	
F12: $\psi_2 \leftarrow -(\not k \mp m)\psi_1 S$	*F21*: $\psi_2 \leftarrow -S(\not k \mp m)\psi_1$
F13: $S \leftarrow \psi_1^T \mathrm{C}(\not k \pm m)\psi_2$	*F31*: $S \leftarrow \psi_2^T \mathrm{C}(-(\not k \mp m)\psi_1)$
F23: $\psi_1 \leftarrow S(\not k \pm m)\psi_2$	*F32*: $\psi_1 \leftarrow (\not k \pm m)\psi_2 S$
GBG (*Fermbar*, *MOM5*, *Ferm*): $\bar\psi_1(\mathrm{i}\not\partial \pm m)\phi\gamma^5\psi_2$	
F12: $\psi_2 \leftarrow (\not k \pm m)\gamma^5\psi_1 P$	*F21*: $\psi_2 \leftarrow P(\not k \pm m)\gamma^5\psi_1$
F13: $P \leftarrow \psi_1^T \mathrm{C}(\not k \pm m)\gamma^5\psi_2$	*F31*: $P \leftarrow \psi_2^T \mathrm{C}(\not k \pm m)\gamma^5\psi_1$
F23: $\psi_1 \leftarrow P(\not k \pm m)\gamma^5\psi_2$	*F32*: $\psi_1 \leftarrow (\not k \pm m)\gamma^5\psi_2 P$
GBG (*Fermbar*, *MOML*, *Ferm*): $\bar\psi_1(\mathrm{i}\not\partial \pm m)\phi(1-\gamma^5)\psi_2$	
F12: $\psi_2 \leftarrow -(1-\gamma^5)(\not k \mp m)\psi_1\phi$	*F21*: $\psi_2 \leftarrow -\phi(1-\gamma^5)(\not k \mp m)\psi_1$
F13: $\phi \leftarrow \psi_1^T \mathrm{C}(\not k \pm m)(1-\gamma^5)\psi_2$	*F31*: $\phi \leftarrow \psi_2^T \mathrm{C}(1-\gamma^5)(-(\not k \mp m)\psi_1)$
F23: $\psi_1 \leftarrow \phi(\not k \pm m)(1-\gamma^5)\psi_2$	*F32*: $\psi_1 \leftarrow (\not k \pm m)(1-\gamma^5)\psi_2\phi$
GBG (*Fermbar*, *LMOM*, *Ferm*): $\bar\psi_1\phi(1-\gamma^5)(\mathrm{i}\not\partial \pm m)\psi_2$	
F12: $\psi_2 \leftarrow -(\not k \mp m)\psi_1(1-\gamma^5)\phi$	*F21*: $\psi_2 \leftarrow -\phi(\not k \mp m)(1-\gamma^5)\psi_1$
F13: $\phi \leftarrow \psi_1^T \mathrm{C}(1-\gamma^5)(\not k \pm m)\psi_2$	*F31*: $\phi \leftarrow \psi_2^T \mathrm{C}(-(\not k \mp m)(1-\gamma^5)\psi_1)$
F23: $\psi_1 \leftarrow \phi(1-\gamma^5)(\not k \pm m)\psi_2$	*F32*: $\psi_1 \leftarrow (1-\gamma^5)(\not k \pm m)\psi_2\phi$
GBG (*Fermbar*, *VMOM*, *Ferm*): $\bar\psi_1\mathrm{i}\not\partial_\alpha V_\beta[\gamma^\alpha, \gamma^\beta]\psi_2$	
F12: $\psi_2 \leftarrow -[\not k, \gamma^\alpha]\psi_1 V_\alpha$	*F21*: $\psi_2 \leftarrow -[\not k, \not V]\psi_1$
F13: $V_\alpha \leftarrow \psi_1^T \mathrm{C}[\not k, \gamma_\alpha]\psi_2$	*F31*: $V_\alpha \leftarrow \psi_2^T \mathrm{C}(-[\not k, \gamma_\alpha]\psi_1)$
F23: $\psi_1 \leftarrow]\not k, \not V]\psi_2$	*F32*: $\psi_1 \leftarrow [\not k, \gamma^\alpha]\psi_2 V_\alpha$

Table 16.23: Combined dimension-4 trilinear fermionic couplings including a momentum. $Ferm$ stands for Psi and Chi. The case of $MOMR$ is identical to $MOML$ if one substitutes $1+\gamma^5$ for $1-\gamma^5$, as well as for $LMOM$ and $RMOM$. The mass term forces us to keep the chiral projector always on the left after "inverting the line" for $MOML$ while on the right for $LMOM$.

GBBG (*Fermbar*, *S2LR*, *Ferm*): $\bar\psi_1 S_1 S_2 (g_L P_L + g_R P_R)\psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow S_1 S_2 (g_R P_L + g_L P_R)\psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow S_1 S_2 (g_L P_L + g_R P_R)\psi_2$
F134 F143 F314:	$S_1 \leftarrow \psi_1^T C S_2 (g_L P_L + g_R P_R)\psi_2$
F124 F142 F214:	$S_2 \leftarrow \psi_1^T C S_1 (g_L P_L + g_R P_R)\psi_2$
F413 F431 F341:	$S_1 \leftarrow \psi_2^T C S_2 (g_R P_L + g_L P_R)\psi_1$
F412 F421 F241:	$S_2 \leftarrow \psi_2^T C S_1 (g_R P_L + g_L P_R)\psi_1$
GBBG (*Fermbar*, *S2*, *Ferm*): $\bar\psi_1 S_1 S_2 \gamma^5 \psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow S_1 S_2 \gamma^5 \psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow S_1 S_2 \gamma^5 \psi_2$
F134 F143 F314:	$S_1 \leftarrow \psi_1^T C S_2 \gamma^5 \psi_2$
F124 F142 F214:	$S_2 \leftarrow \psi_1^T C S_1 \gamma^5 \psi_2$
F413 F431 F341:	$S_1 \leftarrow \psi_2^T C S_2 \gamma^5 \psi_1$
F412 F421 F241:	$S_2 \leftarrow \psi_2^T C S_1 \gamma^5 \psi_1$
GBBG (*Fermbar*, *V2*, *Ferm*): $\bar\psi_1 [\not{V}_1, \not{V}_2] \psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow -[\not{V}_1, \not{V}_2]\psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow [\not{V}_1, \not{V}_2]\psi_2$
F134 F143 F314:	$V_{1\,\alpha} \leftarrow \psi_1^T C [\gamma_\alpha, \not{V}_2]\psi_2$
F124 F142 F214:	$V_{2\,\alpha} \leftarrow \psi_1^T C (-[\gamma_\alpha, \not{V}_1])\psi_2$
F413 F431 F341:	$V_{1\,\alpha} \leftarrow \psi_2^T C (-[\gamma_\alpha, \not{V}_2])\psi_1$
F412 F421 F241:	$V_{2\,\alpha} \leftarrow \psi_2^T C [\gamma_\alpha, \not{V}_1]\psi_1$

Table 16.24: Vertices with two fermions (*Ferm* stands for *Psi* and *Chi*, but not for *Grav*) and two bosons (two scalars, scalar/vector, two vectors) for the BRST transformations. Part I

GBBG (*Fermbar*, *SV*, *Ferm*): $\bar\psi_1 \slashed{V} S\psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow -\slashed{V} S\psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow \slashed{V} S\psi_2$
F134 F143 F314:	$V_\alpha \leftarrow \psi_1^T C\gamma_\alpha S\psi_2$
F124 F142 F214:	$S \leftarrow \psi_1^T C\slashed{V}\psi_2$
F413 F431 F341:	$V_\alpha \leftarrow \psi_2^T C(-\gamma_\alpha S\psi_1)$
F412 F421 F241:	$S \leftarrow \psi_2^T C(-\slashed{V}\psi_1)$
GBBG (*Fermbar*, *PV*, *Ferm*): $\bar\psi_1 \slashed{V}\gamma^5 P\psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow \slashed{V}\gamma^5 P\psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow \slashed{V}\gamma^5 P\psi_2$
F134 F143 F314:	$V_\alpha \leftarrow \psi_1^T C\gamma_\alpha\gamma^5 P\psi_2$
F124 F142 F214:	$P \leftarrow \psi_1^T C\slashed{V}\gamma^5\psi_2$
F413 F431 F341:	$V_\alpha \leftarrow \psi_2^T C\gamma_\alpha\gamma^5 P\psi_1$
F412 F421 F241:	$P \leftarrow \psi_2^T C\slashed{V}\gamma^5\psi_1$
GBBG (*Fermbar*, *S(L/R)V*, *Ferm*): $\bar\psi_1 \slashed{V}(1\mp\gamma^5)\phi\psi_2$	
F123 F213 F132 F231 F312 F321:	$\psi_2 \leftarrow -\slashed{V}(1\pm\gamma^5)\phi\psi_1$
F423 F243 F432 F234 F342 F324:	$\psi_1 \leftarrow \slashed{V}(1\mp\gamma^5)\phi\psi_2$
F134 F143 F314:	$V_\alpha \leftarrow \psi_1^T C\gamma_\alpha(1\mp\gamma^5)\phi\psi_2$
F124 F142 F214:	$\phi \leftarrow \psi_1^T C\slashed{V}(1\mp\gamma^5)\psi_2$
F413 F431 F341:	$V_\alpha \leftarrow \psi_2^T C\gamma_\alpha(-(1\pm\gamma^5)\phi\psi_1)$
F412 F421 F241:	$\phi \leftarrow \psi_2^T C\slashed{V}(-(1\pm\gamma^5)\psi_1)$

Table 16.25: Vertices with two fermions (*Ferm* stands for *Psi* and *Chi*, but not for *Grav*) and two bosons (two scalars, scalar/vector, two vectors) for the BRST transformations. Part II

GBG (*Gravbar*, *POT*, *Psi*): $\bar{\psi}_\mu S\gamma^\mu\psi$	
F12: $\psi \leftarrow -\gamma^\mu\psi_\mu S$	*F21*: $\psi \leftarrow -S\gamma^\mu\psi_\mu$
F13: $S \leftarrow \psi_\mu^T \mathrm{C}\gamma^\mu\psi$	*F31*: $S \leftarrow \psi^T \mathrm{C}(-\gamma^\mu)\psi_\mu$
F23: $\psi_\mu \leftarrow S\gamma_\mu\psi$	*F32*: $\psi_\mu \leftarrow \gamma_\mu\psi S$
GBG (*Gravbar*, *S*, *Psi*): $\bar{\psi}_\mu \not{k}_S S\gamma^\mu\psi$	
F12: $\psi \leftarrow \gamma^\mu \not{k}_S \psi_\mu S$	*F21*: $\psi \leftarrow S\gamma^\mu \not{k}_S \psi_\mu$
F13: $S \leftarrow \psi_\mu^T \mathrm{C}\not{k}_S \gamma^\mu\psi$	*F31*: $S \leftarrow \psi^T \mathrm{C}\gamma^\mu \not{k}_S \psi_\mu$
F23: $\psi_\mu \leftarrow S\not{k}_S \gamma_\mu\psi$	*F32*: $\psi_\mu \leftarrow \not{k}_S \gamma_\mu\psi S$
GBG (*Gravbar*, *P*, *Psi*): $\bar{\psi}_\mu \not{k}_P P\gamma^\mu\gamma_5\psi$	
F12: $\psi \leftarrow \gamma^\mu \not{k}_P \gamma_5\psi_\mu P$	*F21*: $\psi \leftarrow P\gamma^\mu \not{k}_P \gamma_5\psi_\mu$
F13: $P \leftarrow \psi_\mu^T \mathrm{C}\not{k}_P \gamma^\mu\gamma_5\psi$	*F31*: $P \leftarrow \psi^T \mathrm{C}\gamma^\mu \not{k}_P \gamma_5\psi_\mu$
F23: $\psi_\mu \leftarrow P\not{k}_P \gamma_\mu\gamma_5\psi$	*F32*: $\psi_\mu \leftarrow \not{k}_P \gamma_\mu\gamma_5\psi P$
GBG (*Gravbar*, *V*, *Psi*): $\bar{\psi}_\mu [\not{k}_V, \not{V}]\gamma^\mu\gamma^5\psi$	
F12: $\psi \leftarrow \gamma^5\gamma^\mu[\not{k}_V, \gamma^\alpha]\psi_\mu V_\alpha$	*F21*: $\psi \leftarrow \gamma^5\gamma^\mu[\not{k}_V, \not{V}]\psi_\mu$
F13: $V_\mu \leftarrow \psi_\rho^T \mathrm{C}[\not{k}_V, \gamma_\mu]\gamma^\rho\gamma^5\psi$	*F31*: $V_\mu \leftarrow \psi^T \mathrm{C}\gamma^5\gamma^\rho[\not{k}_V, \gamma_\mu]\psi_\rho$
F23: $\psi_\mu \leftarrow [\not{k}_V, \not{V}]\gamma_\mu\gamma^5\psi$	*F32*: $\psi_\mu \leftarrow [\not{k}_V, \gamma^\alpha]\gamma_\mu\gamma^5\psi V_\alpha$

Table 16.26: Dimension-5 trilinear couplings including one Dirac, one Gravitino fermion and one additional particle.The option *POT* is for the coupling of the supersymmetric current to the derivative of the quadratic terms in the superpotential.

GBG (*Psibar*, *POT*, *Grav*): $\bar{\psi}\gamma^\mu S\psi_\mu$	
F12: $\psi_\mu \leftarrow -\gamma_\mu\psi S$	*F21*: $\psi_\mu \leftarrow -S\gamma_\mu\psi$
F13: $S \leftarrow \psi^T \mathrm{C}\gamma^\mu\psi_\mu$	*F31*: $S \leftarrow \psi_\mu^T \mathrm{C}(-\gamma^\mu)\psi$
F23: $\psi \leftarrow S\gamma^\mu\psi_\mu$	*F32*: $\psi \leftarrow \gamma^\mu\psi_\mu S$
GBG (*Psibar*, *S*, *Grav*): $\bar{\psi}\gamma^\mu \not{k}_S S\psi_\mu$	
F12: $\psi_\mu \leftarrow \not{k}_S \gamma_\mu\psi S$	*F21*: $\psi_\mu \leftarrow S\not{k}_S \gamma_\mu\psi$
F13: $S \leftarrow \psi^T \mathrm{C}\gamma^\mu \not{k}_S \psi_\mu$	*F31*: $S \leftarrow \psi_\mu^T \mathrm{C}\not{k}_S \gamma^\mu\psi$
F23: $\psi \leftarrow S\gamma^\mu \not{k}_S \psi_\mu$	*F32*: $\psi \leftarrow \gamma^\mu \not{k}_S \psi_\mu S$
GBG (*Psibar*, *P*, *Grav*): $\bar{\psi}\gamma^\mu\gamma^5 P\not{k}_P \psi_\mu$	
F12: $\psi_\mu \leftarrow -\not{k}_P \gamma_\mu\gamma^5\psi P$	*F21*: $\psi_\mu \leftarrow -P\not{k}_P \gamma_\mu\gamma^5\psi$
F13: $P \leftarrow \psi^T \mathrm{C}\gamma^\mu\gamma^5\not{k}_P \psi_\mu$	*F31*: $P \leftarrow -\psi_\mu^T \mathrm{C}\not{k}_P \gamma^\mu\gamma_5\psi$
F23: $\psi \leftarrow P\gamma^\mu\gamma^5\not{k}_P \psi_\mu$	*F32*: $\psi \leftarrow \gamma^\mu\gamma^5\not{k}_P \psi_\mu P$
GBG (*Psibar*, *V*, *Grav*): $\bar{\psi}\gamma^5\gamma^\mu[\not{k}_V, \not{V}]\psi_\mu$	
F12: $\psi_\mu \leftarrow [\not{k}_V, \gamma^\alpha]\gamma_\mu\gamma^5\psi V_\alpha$	*F21*: $\psi_\mu \leftarrow [\not{k}_V, \not{V}]\gamma_\mu\gamma^5\psi$
F13: $V_\mu \leftarrow \psi^T \mathrm{C}\gamma^5\gamma^\rho[\not{k}_V, \gamma_\mu]\psi_\rho$	*F31*: $V_\mu \leftarrow \psi_\rho^T \mathrm{C}[\not{k}_V, \gamma_\mu]\gamma^\rho\gamma^5\psi$
F23: $\psi \leftarrow \gamma^5\gamma^\mu[\not{k}_V, \not{V}]\psi_\mu$	*F32*: $\psi \leftarrow \gamma^5\gamma^\mu[\not{k}_V, \gamma^\alpha]\psi_\mu V_\alpha$

Table 16.27: Dimension-5 trilinear couplings including one conjugated Dirac, one Gravitino fermion and one additional particle.

GBG (*Gravbar*, *POT*, *Chi*): $\bar{\psi}_\mu S\gamma^\mu\chi$	
F12: $\chi \leftarrow -\gamma^\mu\psi_\mu S$	*F21*: $\chi \leftarrow -S\gamma^\mu\psi_\mu$
F13: $S \leftarrow \psi_\mu^T \mathrm{C}\gamma^\mu\chi$	*F31*: $S \leftarrow \chi^T\mathrm{C}(-\gamma^\mu)\psi_\mu$
F23: $\psi_\mu \leftarrow S\gamma_\mu\chi$	*F32*: $\psi_\mu \leftarrow \gamma_\mu\chi S$
GBG (*Gravbar*, *S*, *Chi*): $\bar{\psi}_\mu \not{k}_S S\gamma^\mu\chi$	
F12: $\chi \leftarrow \gamma^\mu\not{k}_S\psi_\mu S$	*F21*: $\chi \leftarrow S\gamma^\mu\not{k}_S\psi_\mu$
F13: $S \leftarrow \psi_\mu^T\mathrm{C}\not{k}_S\gamma^\mu\chi$	*F31*: $S \leftarrow \chi^T\mathrm{C}\gamma^\mu\not{k}_S\psi_\mu$
F23: $\psi_\mu \leftarrow S\not{k}_S\gamma_\mu\chi$	*F32*: $\psi_\mu \leftarrow \not{k}_S\gamma_\mu\chi S$
GBG (*Gravbar*, *P*, *Chi*): $\bar{\psi}_\mu\not{k}_P P\gamma^\mu\gamma_5\chi$	
F12: $\chi \leftarrow \gamma^\mu\not{k}_P\gamma_5\psi_\mu P$	*F21*: $\chi \leftarrow P\gamma^\mu\not{k}_P\gamma_5\psi_\mu$
F13: $P \leftarrow \psi_\mu^T\mathrm{C}\not{k}_P\gamma^\mu\gamma_5\chi$	*F31*: $P \leftarrow \chi^T\mathrm{C}\gamma^\mu\not{k}_P\gamma_5\psi_\mu$
F23: $\psi_\mu \leftarrow P\not{k}_P\gamma_\mu\gamma_5\chi$	*F32*: $\psi_\mu \leftarrow \not{k}_P\gamma_\mu\gamma_5\chi P$
GBG (*Gravbar*, *V*, *Chi*): $\bar{\psi}_\mu[\not{k}_V,\not{V}]\gamma^\mu\gamma^5\chi$	
F12: $\chi \leftarrow \gamma^5\gamma^\mu[\not{k}_V,\gamma^\alpha]\psi_\mu V_\alpha$	*F21*: $\chi \leftarrow \gamma^5\gamma^\mu[\not{k}_V,\not{V}]\psi_\mu$
F13: $V_\mu \leftarrow \psi_\rho^T\mathrm{C}[\not{k}_V,\gamma_\mu]\gamma^\rho\gamma^5\chi$	*F31*: $V_\mu \leftarrow \chi^T\mathrm{C}\gamma^5\gamma^\rho[\not{k}_V,\gamma_\mu]\psi_\rho$
F23: $\psi_\mu \leftarrow [\not{k}_V,\not{V}]\gamma_\mu\gamma^5\chi$	*F32*: $\psi_\mu \leftarrow [\not{k}_V,\gamma^\alpha]\gamma_\mu\gamma^5\chi V_\alpha$

Table 16.28: Dimension-5 trilinear couplings including one Majorana, one Gravitino fermion and one additional particle. The table is essentially the same as the one with the Dirac fermion and only written for the sake of completeness.

GBG (*Chibar*, *POT*, *Grav*): $\bar{\chi}\gamma^\mu S\psi_\mu$	
F12: $\psi_\mu \leftarrow -\gamma_\mu\chi S$	*F21*: $\psi_\mu \leftarrow -S\gamma_\mu\chi$
F13: $S \leftarrow \chi^T\mathrm{C}\gamma^\mu\psi_\mu$	*F31*: $S \leftarrow \psi_\mu^T\mathrm{C}(-\gamma^\mu)\chi$
F23: $\chi \leftarrow S\gamma^\mu\psi_\mu$	*F32*: $\chi \leftarrow \gamma^\mu\psi_\mu S$
GBG (*Chibar*, *S*, *Grav*): $\bar{\chi}\gamma^\mu\not{k}_S S\psi_\mu$	
F12: $\psi_\mu \leftarrow \not{k}_S\gamma_\mu\chi S$	*F21*: $\psi_\mu \leftarrow S\not{k}_S\gamma_\mu\chi$
F13: $S \leftarrow \chi^T\mathrm{C}\gamma^\mu\not{k}_S\psi_\mu$	*F31*: $S \leftarrow \psi_\mu^T\mathrm{C}\not{k}_S\gamma^\mu\chi$
F23: $\chi \leftarrow S\gamma^\mu\not{k}_S\psi_\mu$	*F32*: $\chi \leftarrow \gamma^\mu\not{k}_S\psi_\mu S$
GBG (*Chibar*, *P*, *Grav*): $\bar{\chi}\gamma^\mu\gamma^5 P\not{k}_P\psi_\mu$	
F12: $\psi_\mu \leftarrow -\not{k}_P\gamma_\mu\gamma^5\chi P$	*F21*: $\psi_\mu \leftarrow -P\not{k}_P\gamma_\mu\gamma^5\chi$
F13: $P \leftarrow \chi^T\mathrm{C}\gamma^\mu\gamma^5\not{k}_P\psi_\mu$	*F31*: $P \leftarrow -\psi_\mu^T\mathrm{C}\not{k}_P\gamma^\mu\gamma_5\chi$
F23: $\chi \leftarrow P\gamma^\mu\gamma^5\not{k}_P\psi_\mu$	*F32*: $\chi \leftarrow \gamma^\mu\gamma^5\not{k}_P\psi_\mu P$
GBG (*Chibar*, *V*, *Grav*): $\bar{\chi}\gamma^5\gamma^\mu[\not{k}_V,\not{V}]\psi_\mu$	
F12: $\psi_\mu \leftarrow [\not{k}_V,\gamma^\alpha]\gamma_\mu\gamma^5\chi V_\alpha$	*F21*: $\psi_\mu \leftarrow [\not{k}_V,\not{V}]\gamma_\mu\gamma^5\chi$
F13: $V_\mu \leftarrow \chi^T\mathrm{C}\gamma^5\gamma^\rho[\not{k}_V,\gamma_\mu]\psi_\rho$	*F31*: $V_\mu \leftarrow \psi_\rho^T\mathrm{C}[\not{k}_V,\gamma_\mu]\gamma^\rho\gamma^5\chi$
F23: $\chi \leftarrow \gamma^5\gamma^\mu[\not{k}_V,\not{V}]\psi_\mu$	*F32*: $\chi \leftarrow \gamma^5\gamma^\mu[\not{k}_V,\gamma^\alpha]\psi_\mu V_\alpha$

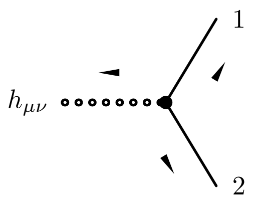
Table 16.29: Dimension-5 trilinear couplings including one conjugated Majorana, one Gravitino fermion and one additional particle. This table is not only the same as the one with the conjugated Dirac fermion but also the same part of the Lagrangian density as the one with the Majorana particle on the right of the gravitino.

GBBG (*Gravbar*, *S2*, *Psi*): $\bar\psi_\mu S_1 S_2 \gamma^\mu \psi$	
F123 F213 F132 F231 F312 F321:	$\psi \leftarrow -\gamma^\mu S_1 S_2 \psi_\mu$
F423 F243 F432 F234 F342 F324:	$\psi_\mu \leftarrow \gamma_\mu S_1 S_2 \psi$
F134 F143 F314:	$S_1 \leftarrow \psi_\mu^T C S_2 \gamma^\mu \psi$
F124 F142 F214:	$S_2 \leftarrow \psi_\mu^T C S_1 \gamma^\mu \psi$
F413 F431 F341:	$S_1 \leftarrow -\psi^T C S_2 \gamma^\mu \psi_\mu$
F412 F421 F241:	$S_2 \leftarrow -\psi^T C S_1 \gamma^\mu \psi_\mu$
GBBG (*Gravbar*, *SV*, *Psi*): $\bar\psi_\mu S \slashed{V} \gamma^\mu \gamma^5 \psi$	
F123 F213 F132 F231 F312 F321:	$\psi \leftarrow \gamma^5 \gamma^\mu S \slashed{V} \psi_\mu$
F423 F243 F432 F234 F342 F324:	$\psi_\mu \leftarrow \slashed{V} S \gamma_\mu \gamma^5 \psi$
F134 F143 F314:	$S \leftarrow \psi_\mu^T C \slashed{V} \gamma^\mu \gamma^5 \psi$
F124 F142 F214:	$V_\mu \leftarrow \psi_\rho^T C S \gamma_\mu \gamma^\rho \gamma^5 \psi$
F413 F431 F341:	$S \leftarrow \psi^T C \gamma^5 \gamma^\mu \slashed{V} \psi_\mu$
F412 F421 F241:	$V_\mu \leftarrow \psi^T C S \gamma^5 \gamma^\rho \gamma_\mu \psi_\rho$
GBBG (*Gravbar*, *PV*, *Psi*): $\bar\psi_\mu P \slashed{V} \gamma^\mu \psi$	
F123 F213 F132 F231 F312 F321:	$\psi \leftarrow \gamma^\mu P \slashed{V} \psi_\mu$
F423 F243 F432 F234 F342 F324:	$\psi_\mu \leftarrow \slashed{V} P \gamma_\mu \psi$
F134 F143 F314:	$P \leftarrow \psi_\mu^T C \slashed{V} \gamma^\mu \psi$
F124 F142 F214:	$V_\mu \leftarrow \psi_\rho^T C P \gamma_\mu \gamma^\rho \psi$
F413 F431 F341:	$P \leftarrow \psi^T C \gamma^\mu \slashed{V} \psi_\mu$
F412 F421 F241:	$V_\mu \leftarrow \psi^T C P \gamma^\rho \gamma_\mu \psi_\rho$
GBBG (*Gravbar*, *V2*, *Psi*): $\bar\psi_\mu f_{abc} [\slashed{V}^a, \slashed{V}^b] \gamma^\mu \gamma^5 \psi$	
F123 F213 F132 F231 F312 F321:	$\psi \leftarrow f_{abc} \gamma^5 \gamma^\mu [\slashed{V}^a, \slashed{V}^b] \psi_\mu$
F423 F243 F432 F234 F342 F324:	$\psi_\mu \leftarrow f_{abc} [\slashed{V}^a, \slashed{V}^b] \gamma_\mu \gamma^5 \psi$
F134 F143 F314 F124 F142 F214:	$V_\mu^a \leftarrow \psi_\rho^T C f_{abc} [\gamma_\mu, \slashed{V}^b] \gamma^\rho \gamma^5 \psi$
F413 F431 F341 F412 F421 F241:	$V_\mu^a \leftarrow \psi^T C f_{abc} \gamma^5 \gamma^\rho [\gamma_\mu, \slashed{V}^b] \psi_\rho$

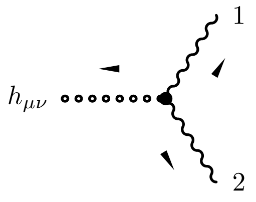
Table 16.30: Dimension-5 trilinear couplings including one Dirac, one Gravitino fermion and two additional bosons. In each lines we list the fusion possibilities with the same order of the fermions, but the order of the bosons is arbitrary (of course, one has to take care of this order in the mapping of the wave functions in *fusion*).

GBBG (*Psibar*, *S2*, *Grav*): $\bar\psi S_1 S_2 \gamma^\mu \psi_\mu$	
F123 F213 F132 F231 F312 F321:	$\psi_\mu \leftarrow -\gamma_\mu S_1 S_2 \psi$
F423 F243 F432 F234 F342 F324:	$\psi \leftarrow \gamma^\mu S_1 S_2 \psi_\mu$
F134 F143 F314:	$S_1 \leftarrow \psi^T C S_2 \gamma^\mu \psi_\mu$
F124 F142 F214:	$S_2 \leftarrow \psi^T C S_1 \gamma^\mu \psi_\mu$
F413 F431 F341:	$S_1 \leftarrow -\psi_\mu^T C S_2 \gamma^\mu \psi$
F412 F421 F241:	$S_2 \leftarrow -\psi_\mu^T C S_1 \gamma^\mu \psi$
GBBG (*Psibar*, *SV*, *Grav*): $\bar\psi S \gamma^\mu \gamma^5 \not{V} \psi_\mu$	
F123 F213 F132 F231 F312 F321:	$\psi_\mu \leftarrow \not{V} S \gamma^5 \gamma^\mu \psi$
F423 F243 F432 F234 F342 F324:	$\psi \leftarrow \gamma^\mu \gamma^5 S \not{V} \psi_\mu$
F134 F143 F314:	$S \leftarrow \psi^T C \gamma^\mu \gamma^5 \not{V} \psi$
F124 F142 F214:	$V_\mu \leftarrow \psi^T C \gamma^\rho \gamma^5 S \gamma_\mu \psi_\rho$
F413 F431 F341:	$S \leftarrow \psi_\mu^T C \not{V} \gamma^5 \gamma^\mu \psi$
F412 F421 F241:	$V_\mu \leftarrow \psi_\rho^T C S \gamma_\mu \gamma^5 \gamma^\rho \psi$
GBBG (*Psibar*, *PV*, *Grav*): $\bar\psi P \gamma^\mu \not{V} \psi_\mu$	
F123 F213 F132 F231 F312 F321:	$\psi_\mu \leftarrow \not{V} \gamma_\mu P \psi$
F423 F243 F432 F234 F342 F324:	$\psi \leftarrow \gamma^\mu \not{V} P \psi_\mu$
F134 F143 F314:	$P \leftarrow \psi^T C \gamma^\mu \not{V} \psi_\mu$
F124 F142 F214:	$V_\mu \leftarrow \psi^T C P \gamma^\rho \gamma_\mu \psi_\rho$
F413 F431 F341:	$P \leftarrow \psi_\mu^T C \not{V} \gamma^\mu \psi$
F412 F421 F241:	$V_\mu \leftarrow \psi_\rho^T C P \gamma_\mu \gamma^\rho \psi$
GBBG (*Psibar*, *V2*, *Grav*): $\bar\psi f_{abc} \gamma^5 \gamma^\mu [\not{V}^a, \not{V}^b] \psi_\mu$	
F123 F213 F132 F231 F312 F321:	$\psi_\mu \leftarrow f_{abc} [\not{V}^a, \not{V}^b] \gamma_\mu \gamma^5 \psi$
F423 F243 F432 F234 F342 F324:	$\psi \leftarrow f_{abc} \gamma^5 \gamma^\mu [\not{V}^a, \not{V}^b] \psi_\mu$
F134 F143 F314 F124 F142 F214:	$V_\mu^a \leftarrow \psi^T C f_{abc} \gamma^5 \gamma^\rho [\gamma_\mu, \not{V}^b] \psi_\rho$
F413 F431 F341 F412 F421 F241:	$V_\mu^a \leftarrow \psi_\rho^T C f_{abc} [\gamma_\mu, \not{V}^b] \gamma^\rho \gamma^5 \psi$

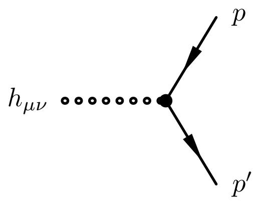
Table 16.31: Dimension-5 trilinear couplings including one conjugated Dirac, one Gravitino fermion and two additional bosons. The couplings of Majorana fermions to the gravitino and two bosons are essentially the same as for Dirac fermions and they are omitted here.



$$= \quad -\mathrm{i}\frac{\kappa}{2}g_{\mu\nu}m^2 + \mathrm{i}\frac{\kappa}{2}C_{\mu\nu,\mu_1\mu_2}k_1^{\mu_1}k_2^{\mu_2} \qquad (16.27a)$$



$$= \quad \begin{aligned} -\mathrm{i}\frac{\kappa}{2}m^2 C_{\mu\nu,\mu_1\mu_2} - \mathrm{i}\frac{\kappa}{2}(k_1k_2C_{\mu\nu,\mu_1\mu_2} & \\ + D_{\mu\nu,\mu_1\mu_2}(k_1,k_2) & \\ + \xi^{-1}E_{\mu\nu,\mu_1\mu_2}(k_1,k_2)) & \end{aligned} \qquad (16.27b)$$



$$= \quad \begin{aligned} -\mathrm{i}\frac{\kappa}{2}mg_{\mu\nu} - \mathrm{i}\frac{\kappa}{8}(\gamma_\mu(p+p')_\nu + \gamma_\nu(p+p')_\mu & \\ - 2g_{\mu\nu}(\not{p} + \not{p}')) & \end{aligned} \qquad (16.27c)$$

Figure 16.3: Three-point graviton couplings.

16.1.7 Perturbative Quantum Gravity and Kaluza-Klein Interactions

The gravitational coupling constant and the relative strength of the dilaton coupling are abbreviated as

$$\kappa = \sqrt{16\pi G_N} \qquad (16.25a)$$

$$\omega = \sqrt{\frac{2}{3(n+2)}} = \sqrt{\frac{2}{3(d-2)}}\,, \qquad (16.25b)$$

where $n = d - 4$ is the number of extra space dimensions.
In (16.27-16.34), we use the notation of [14]:

$$C_{\mu\nu,\rho\sigma} = g_{\mu\rho}g_{\nu\sigma} + g_{\mu\sigma}g_{\nu\rho} - g_{\mu\nu}g_{\rho\sigma} \qquad (16.26a)$$

$$\begin{aligned} D_{\mu\nu,\rho\sigma}(k_1,k_2) = g_{\mu\nu}k_{1,\sigma}k_{2,\rho} & \\ - (g_{\mu\sigma}k_{1,\nu}k_{2,\rho} + g_{\mu\rho}k_{1,\sigma}k_{2,\nu} - g_{\rho\sigma}k_{1,\mu}k_{2,\nu} + (\mu \leftrightarrow \nu)) & \end{aligned} \qquad (16.26b)$$

$$\begin{aligned} E_{\mu\nu,\rho\sigma}(k_1,k_2) = g_{\mu\nu}(k_{1,\rho}k_{1,\sigma} + k_{2,\rho}k_{2,\sigma} + k_{1,\rho}k_{2,\sigma}) & \\ - (g_{\nu\sigma}k_{1,\mu}k_{1,\rho} + g_{\nu\rho}k_{2,\mu}k_{2,\sigma} + (\mu \leftrightarrow \nu)) & \end{aligned} \qquad (16.26c)$$

$$\begin{aligned} F_{\mu\nu,\rho\sigma\lambda}(k_1,k_2,k_3) = & \\ g_{\mu\rho}g_{\sigma\lambda}(k_2-k_3)_\nu + g_{\mu\sigma}g_{\lambda\rho}(k_3-k_1)_\nu + g_{\mu\lambda}g_{\rho\sigma}(k_1-k_2)_\nu + (\mu \leftrightarrow \nu) & \end{aligned} \qquad (16.26d)$$

$$\begin{aligned} G_{\mu\nu,\rho\sigma\lambda\delta} = g_{\mu\nu}(g_{\rho\sigma}g_{\lambda\delta} - g_{\rho\delta}g_{\lambda\sigma}) & \\ + (g_{\mu\rho}g_{\nu\delta}g_{\lambda\sigma} + g_{\mu\lambda}g_{\nu\sigma}g_{\rho\delta} - g_{\mu\rho}g_{\nu\sigma}g_{\lambda\delta} - g_{\mu\lambda}g_{\nu\delta}g_{\rho\sigma} + (\mu \leftrightarrow \nu)) & \end{aligned} \qquad (16.26e)$$

Derivation of (16.27a)

$$L = \frac{1}{2}(\partial_\mu\phi)(\partial^\mu\phi) - \frac{m^2}{2}\phi^2 \qquad (16.28a)$$

$$(\partial_\mu\phi)\frac{\partial L}{\partial(\partial^\nu\phi)} = (\partial_\mu\phi)(\partial_\nu\phi) \qquad (16.28b)$$

$$T_{\mu\nu} = -g_{\mu\nu}L + (\partial_\mu\phi)\frac{\partial L}{\partial(\partial^\nu\phi)} + \qquad (16.28c)$$

Graviton_Scalar_Scalar: $h_{\mu\nu}C_0^{\mu\nu}(k_1,k_2)\phi_1\phi_2$	
F12 | *F21*:	$\phi_2 \leftarrow \mathrm{i}\cdot h_{\mu\nu}C_0^{\mu\nu}(k_1,-k-k_1)\phi_1$
F13 | *F31*:	$\phi_1 \leftarrow \mathrm{i}\cdot h_{\mu\nu}C_0^{\mu\nu}(-k-k_2,k_2)\phi_2$
F23 | *F32*:	$h^{\mu\nu} \leftarrow \mathrm{i}\cdot C_0^{\mu\nu}(k_1,k_2)\phi_1\phi_2$
Graviton_Vector_Vector: $h_{\mu\nu}C_1^{\mu\nu,\mu_1\mu_2}(k_1,k_2,\xi)V_{\mu_1}V_{\mu_2}$	
F12 | *F21*:	$V_2^\mu \leftarrow \mathrm{i}\cdot h_{\kappa\lambda}C_1^{\kappa\lambda,\mu\nu}(-k-k_1,k_1\xi)V_{1,\nu}$
F13 | *F31*:	$V_1^\mu \leftarrow \mathrm{i}\cdot h_{\kappa\lambda}C_1^{\kappa\lambda,\mu\nu}(-k-k_2,k_2,\xi)V_{2,\nu}$
F23 | *F32*:	$h^{\mu\nu} \leftarrow \mathrm{i}\cdot C_1^{\mu\nu,\mu_1\mu_2}(k_1,k_2,\xi)V_{1,\mu_1}V_{2,\mu_2}$
Graviton_Spinor_Spinor: $h_{\mu\nu}\bar\psi_1 C_{\frac12}^{\mu\nu}(k_1,k_2)\psi_2$	
F12:	$\bar\psi_2 \leftarrow \mathrm{i}\cdot h_{\mu\nu}\bar\psi_1 C_{\frac12}^{\mu\nu}(k_1,-k-k_1)$
F21:	$\bar\psi_2 \leftarrow \mathrm{i}\cdot\ldots$
F13:	$\psi_1 \leftarrow \mathrm{i}\cdot h_{\mu\nu}C_{\frac12}^{\mu\nu}(-k-k_2,k_2)\psi_2$
F31:	$\psi_1 \leftarrow \mathrm{i}\cdot\ldots$
F23:	$h^{\mu\nu} \leftarrow \mathrm{i}\cdot\bar\psi_1 C_{\frac12}^{\mu\nu}(k_1,k_2)\psi_2$
F32:	$h^{\mu\nu} \leftarrow \mathrm{i}\cdot\ldots$

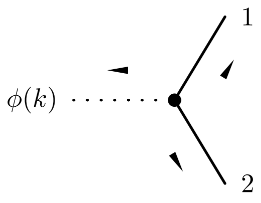
Table 16.32: ...

$$C_0^{\mu\nu}(k_1,k_2) = C^{\mu\nu,\mu_1\mu_2}k_{1,\mu_1}k_{2,\mu_2} \tag{16.29a}$$
$$C_1^{\mu\nu,\mu_1\mu_2}(k_1,k_2,\xi) = k_1k_2C^{\mu\nu,\mu_1\mu_2} + D^{\mu\nu,\mu_1\mu_2}(k_1,k_2) + \xi^{-1}E^{\mu\nu,\mu_1\mu_2}(k_1,k_2) \tag{16.29b}$$
$$C_{\frac12,\alpha\beta}^{\mu\nu}(p,p') = \gamma_{\alpha\beta}^\mu(p+p')^\nu + \gamma_{\alpha\beta}^\nu(p+p')^\mu - 2g^{\mu\nu}(\not p+\not p')_{\alpha\beta} \tag{16.29c}$$

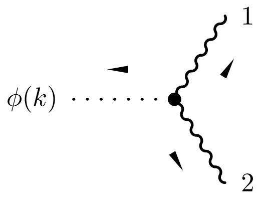
16.1.8 Dependent Parameters

This is a simple abstract syntax for parameter dependencies. Later, there will be a parser for a convenient concrete syntax as a part of a concrete syntax for models. There is no intention to do *any* symbolic manipulation with this. The expressions will be translated directly by *Targets* to the target language.

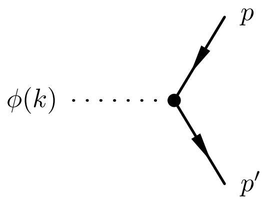
```
type α expr =
  | I
  | Integer of int
  | Float of float
  | Atom of α
  | Sum of α expr list
  | Diff of α expr × α expr
  | Neg of α expr
  | Prod of α expr list
  | Quot of α expr × α expr
  | Rec of α expr
  | Pow of α expr × int
  | PowX of α expr × α expr
  | Sqrt of α expr
  | Sin of α expr
  | Cos of α expr
  | Tan of α expr
```



$$= -\mathrm{i}\omega\kappa 2m^2 - \mathrm{i}\omega\kappa k_1 k_2 \tag{16.30a}$$



$$= -\mathrm{i}\omega\kappa g_{\mu_1\mu_2} m^2 - \mathrm{i}\omega\kappa\xi^{-1}(k_{1,\mu_1}k_{\mu_2} + k_{2,\mu_2}k_{\mu_1}) \tag{16.30b}$$

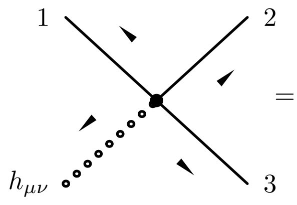


$$= -\mathrm{i}\omega\kappa 2m + \mathrm{i}\omega\kappa\frac{3}{4}(\not{p} + \not{p}') \tag{16.30c}$$

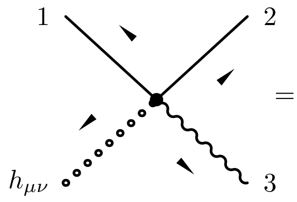
Figure 16.4: Three-point dilaton couplings.

Dilaton_Scalar_Scalar: $\phi \ldots k_1 k_2 \phi_1 \phi_2$	
F12 | *F21*:	$\phi_2 \leftarrow \mathrm{i} \cdot k_1(-k - k_1)\phi\phi_1$
F13 | *F31*:	$\phi_1 \leftarrow \mathrm{i} \cdot (-k - k_2)k_2\phi\phi_2$
F23 | *F32*:	$\phi \leftarrow \mathrm{i} \cdot k_1 k_2 \phi_1 \phi_2$
Dilaton_Vector_Vector: $\phi \ldots$	
F12:	$V_{2,\mu} \leftarrow \mathrm{i} \cdot \ldots$
F21:	$V_{2,\mu} \leftarrow \mathrm{i} \cdot \ldots$
F13:	$V_{1,\mu} \leftarrow \mathrm{i} \cdot \ldots$
F31:	$V_{1,\mu} \leftarrow \mathrm{i} \cdot \ldots$
F23:	$\phi \leftarrow \mathrm{i} \cdot \ldots$
F32:	$\phi \leftarrow \mathrm{i} \cdot \ldots$
Dilaton_Spinor_Spinor: $\phi \ldots$	
F12:	$\bar\psi_2 \leftarrow \mathrm{i} \cdot \ldots$
F21:	$\bar\psi_2 \leftarrow \mathrm{i} \cdot \ldots$
F13:	$\psi_1 \leftarrow \mathrm{i} \cdot \ldots$
F31:	$\psi_1 \leftarrow \mathrm{i} \cdot \ldots$
F23:	$\phi \leftarrow \mathrm{i} \cdot \ldots$
F32:	$\phi \leftarrow \mathrm{i} \cdot \ldots$

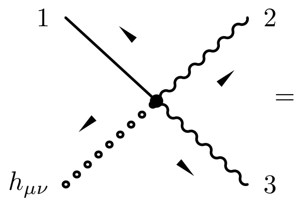
Table 16.33: ...



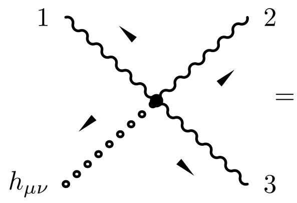
$$= \qquad ??? \tag{16.31a}$$



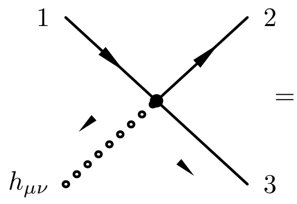
$$= \qquad -\mathrm{i}g\frac{\kappa}{2}C_{\mu\nu,\mu_3\rho}(k_1-k_2)^\rho T^{a_3}_{n_2n_1} \tag{16.31b}$$



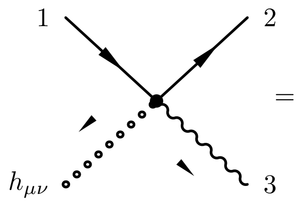
$$= \qquad ??? \tag{16.31c}$$



$$\begin{aligned} = \qquad -g\frac{\kappa}{2}f^{a_1a_2a_3}(&C_{\mu\nu,\mu_1\mu_2}(k_1-k_2)_{\mu_3} \\ &+C_{\mu\nu,\mu_2\mu_3}(k_2-k_3)_{\mu_1} \\ &+C_{\mu\nu,\mu_3\mu_1}(k_3-k_1)_{\mu_2} \\ &+F_{\mu\nu,\mu_1\mu_2\mu_3}(k_1,k_2,k_3)) \end{aligned} \tag{16.31d}$$

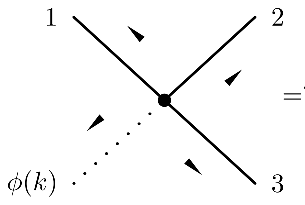


$$= \qquad ??? \tag{16.31e}$$

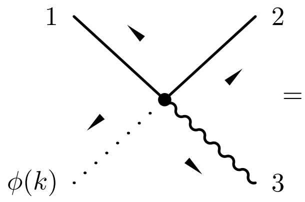


$$= \qquad \mathrm{i}g\frac{\kappa}{4}(C_{\mu\nu,\mu_3\rho}-g_{\mu\nu}g_{\mu_3\rho})\gamma^\rho T^{a_3}_{n_2n_1} \tag{16.31f}$$

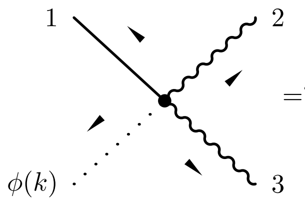
Figure 16.5: Four-point graviton couplings. (16.31a), (16.31c), and (**??** are missing in [14], but should be generated by standard model Higgs selfcouplings, Higgs-gaugeboson couplings, and Yukawa couplings.



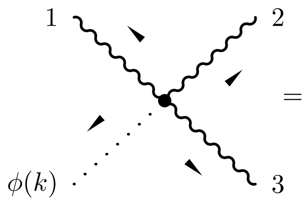
$$=??? \tag{16.32a}$$



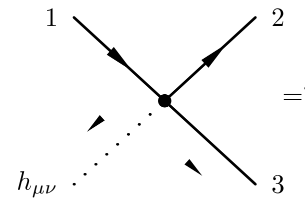
$$= -\mathrm{i}\omega\kappa(k_1+k_2)_{\mu_3}T^{a_3}_{n_1,n_2} \tag{16.32b}$$



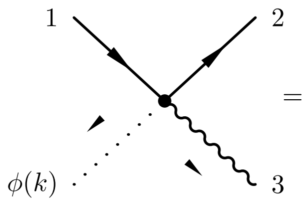
$$=??? \tag{16.32c}$$



$$= 0 \tag{16.32d}$$

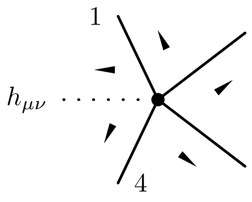


$$=??? \tag{16.32e}$$

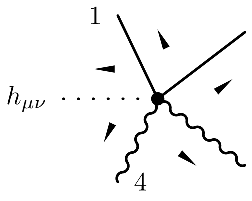


$$= -\mathrm{i}\frac{3}{2}\omega g\kappa\gamma_{\mu_3}T^{a_3}_{n_1n_2} \tag{16.32f}$$

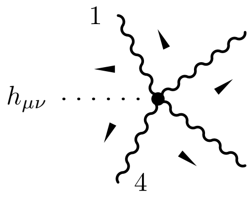
Figure 16.6: Four-point dilaton couplings. (16.32a), (16.32c) and (16.32e) are missing in [14], but could be generated by standard model Higgs selfcouplings, Higgs-gaugeboson couplings, and Yukawa couplings.



$$= \qquad ??? \tag{16.33a}$$

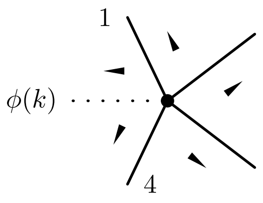


$$= -\mathrm{i}g^2\frac{\kappa}{2}C_{\mu\nu,\mu_3\mu_4}(T^{a_3}T^{a_4}+T^{a_4}T^{a_3})_{n_2n_1} \tag{16.33b}$$

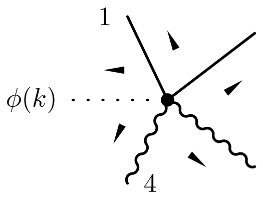


$$\begin{aligned} = -\mathrm{i}g^2\frac{\kappa}{2}(&f^{ba_1a_3}f^{ba_2a_4}G_{\mu\nu,\mu_1\mu_2\mu_3\mu_4} \\ &+f^{ba_1a_2}f^{ba_3a_4}G_{\mu\nu,\mu_1\mu_3\mu_2\mu_4} \\ &+f^{ba_1a_4}f^{ba_2a_3}G_{\mu\nu,\mu_1\mu_2\mu_4\mu_3}) \end{aligned} \tag{16.33c}$$

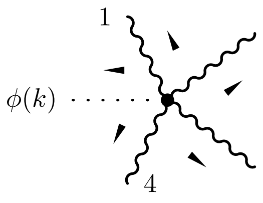
Figure 16.7: Five-point graviton couplings. (16.33a) is missing in [14], but should be generated by standard model Higgs selfcouplings.



$$=??? \tag{16.34a}$$



$$= \mathrm{i}\omega g^2\kappa g_{\mu_3\mu_4}(T^{a_3}T^{a_4}+T^{a_4}T^{a_3})_{n_2n_1} \tag{16.34b}$$



$$= 0 \tag{16.34c}$$

Figure 16.8: Five-point dilaton couplings. (16.34a) is missing in [14], but could be generated by standard model Higgs selfcouplings.

Dim5_Scalar_Vector_Vector_T: $\mathcal{L}_I = g\phi(\mathrm{i}\partial_\mu V_1^\nu)(\mathrm{i}\partial_\nu V_2^\mu)$
F23: $\phi(k_2+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu V_{1,\mu}(k_2)k_2^\nu V_{2,\nu}(k_3)$
F32: $\phi(k_2+k_3) \leftarrow \mathrm{i}\cdot gk_2^\mu V_{2,\mu}(k_3)k_3^\nu V_{1,\nu}(k_2)$
F12: $V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu \phi(k_1)(-k_1^\nu - k_2^\nu)V_{1,\nu}(k_2)$
F21: $V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu (-k_1^\nu - k_2^\nu)V_{1,\nu}(k_2)\phi(k_1)$
F13: $V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu \phi(k_1)(-k_1^\nu - k_3^\nu)V_{2,\nu}(k_3)$
F31: $V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu (-k_1^\nu - k_3^\nu)V_{2,\nu}(k_3)\phi(k_1)$

Table 16.34: ...

Dim6_Vector_Vector_Vector_T: $\mathcal{L}_I = gV_1^\mu((\mathrm{i}\partial_\nu V_2^\rho)\mathrm{i}\overleftrightarrow{\partial_\mu}(\mathrm{i}\partial_\rho V_3^\nu))$
F23: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)k_3^\nu V_{2,\nu}(k_2)k_2^\rho V_{3,\rho}(k_3)$
F32: $V_1^\mu(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\mu - k_3^\mu)k_2^\nu V_{3,\nu}(k_3)k_3^\rho V_{2,\rho}(k_2)$
F12: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu(k_1^\nu + 2k_2^\nu)V_{1,\nu}(k_1)(-k_1^\rho - k_2^\rho)V_{2,\rho}(k_2)$
F21: $V_3^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu(-k_1^\rho - k_2^\rho)V_{2,\rho}(k_2)(k_1^\nu + 2k_2^\nu)V_{1,\nu}(k_1)$
F13: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu(k_1^\nu + 2k_3^\nu)V_{1,\nu}(k_1)(-k_1^\rho - k_3^\rho)V_{3,\rho}(k_3)$
F31: $V_2^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu(-k_1^\rho - k_3^\rho)V_{3,\rho}(k_3)(k_1^\nu + 2k_3^\nu)V_{1,\nu}(k_1)$

Table 16.35: ...

```
  | Cot of α expr
  | Asin of α expr
  | Acos of α expr
  | Atan of α expr
  | Atan2 of α expr × α expr
  | Sinh of α expr
  | Cosh of α expr
  | Tanh of α expr
  | Exp of α expr
  | Log of α expr
  | Log10 of α expr
  | Conj of α expr
  | Abs of α expr

type α variable = Real of α | Complex of α
type α variable_array = Real_Array of α | Complex_Array of α

type α parameters =
    { input : (α × float) list;
      derived : (α variable × α expr) list;
      derived_arrays : (α variable_array × α expr list) list }
```

16.1.9 More Exotic Couplings

16.2 Interface of Model

16.2.1 General Quantum Field Theories

```
module type T =
  sig
```

flavor abstractly encodes all quantum numbers.

```
    type flavor
```

Tensor_2_Vector_Vector: $\mathcal{L}_I = gT^{\mu\nu}(V_{1,\mu}V_{2,\nu} + V_{1,\nu}V_{2,\mu})$	
F23:	$T^{\mu\nu}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(V_{1,\mu}(k_2)V_{2,\nu}(k_3) + V_{1,\nu}(k_2)V_{2,\mu}(k_3))$
F32:	$T^{\mu\nu}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(V_{2,\nu}(k_3)V_{1,\mu}(k_2) + V_{2,\mu}(k_3)V_{1,\nu}(k_2))$
F12:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(T^{\mu\nu}(k_1) + T^{\nu\mu}(k_1))V_{1,\nu}(k_2)$
F21:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gV_{1,\nu}(k_2)(T^{\mu\nu}(k_1) + T^{\nu\mu}(k_1))$
F13:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(T^{\mu\nu}(k_1) + T^{\nu\mu}(k_1))V_{2,\nu}(k_3)$
F31:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gV_{2,\nu}(k_3)(T^{\mu\nu}(k_1) + T^{\nu\mu}(k_1))$

Table 16.36: ...

Dim5_Tensor_2_Vector_Vector_1: $\mathcal{L}_I = gT^{\alpha\beta}(V_1^\mu \mathrm{i}\overleftrightarrow{\partial}_\alpha \mathrm{i}\overleftrightarrow{\partial}_\beta V_{2,\mu})$	
F23:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\alpha - k_3^\alpha)(k_2^\beta - k_3^\beta)V_1^\mu(k_2)V_{2,\mu}(k_3)$
F32:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\alpha - k_3^\alpha)(k_2^\beta - k_3^\beta)V_{2,\mu}(k_3)V_1^\mu(k_2)$
F12:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\alpha + 2k_2^\alpha)(k_1^\beta + 2k_2^\beta)T_{\alpha\beta}(k_1)V_1^\mu(k_2)$
F21:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\alpha + 2k_2^\alpha)(k_1^\beta + 2k_2^\beta)V_1^\mu(k_2)T_{\alpha\beta}(k_1)$
F13:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(k_1^\alpha + 2k_3^\alpha)(k_1^\beta + 2k_3^\beta)T_{\alpha\beta}(k_1)V_2^\mu(k_3)$
F31:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot g(k_1^\alpha + 2k_3^\alpha)(k_1^\beta + 2k_3^\beta)V_2^\mu(k_3)T_{\alpha\beta}(k_1)$

Table 16.37: ...

Dim5_Tensor_2_Vector_Vector_2: $\mathcal{L}_I = gT^{\alpha\beta}(V_1^\mu \mathrm{i}\overleftrightarrow{\partial}_\beta(\mathrm{i}\partial_\mu V_{2,\alpha}) + V_1^\mu \mathrm{i}\overleftrightarrow{\partial}_\alpha(\mathrm{i}\partial_\mu V_{2,\beta}))$	
F23:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_3^\beta - k_2^\beta)k_3^\mu V_{1,\mu}(k_2)V_2^\alpha(k_3) + (\alpha \leftrightarrow \beta)$
F32:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_3^\beta - k_2^\beta)V_2^\alpha(k_3)k_3^\mu V_{1,\mu}(k_2) + (\alpha \leftrightarrow \beta)$
F12:	$V_2^\alpha(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\beta + 2k_2^\beta)(T^{\alpha\beta}(k_1) + T^{\beta\alpha}(k_1))(k_1^\mu + k_2^\mu)V_{1,\mu}(k_2)$
F21:	$V_2^\alpha(k_1+k_2) \leftarrow \mathrm{i}\cdot g(k_1^\mu + k_2^\mu)V_{1,\mu}(k_2)(k_1^\beta + 2k_2^\beta)(T^{\alpha\beta}(k_1) + T^{\beta\alpha}(k_1))$
F13:	$V_1^\alpha(k_1+k_3) \leftarrow \mathrm{i}\cdot g(k_1^\beta + 2k_3^\beta)(T^{\alpha\beta}(k_1) + T^{\beta\alpha}(k_1))(k_1^\mu + k_3^\mu)V_{2,\mu}(k_3)$
F31:	$V_1^\alpha(k_1+k_3) \leftarrow \mathrm{i}\cdot g(k_1^\mu + k_3^\mu)V_{2,\mu}(k_3)(k_1^\beta + 2k_3^\beta)(T^{\alpha\beta}(k_1) + T^{\beta\alpha}(k_1))$

Table 16.38: ...

Dim7_Tensor_2_Vector_Vector_T: $\mathcal{L}_I = gT^{\alpha\beta}((\mathrm{i}\partial^\mu V_1^\nu)\mathrm{i}\overleftrightarrow{\partial}_\alpha \mathrm{i}\overleftrightarrow{\partial}_\beta(\mathrm{i}\partial_\nu V_{2,\mu}))$	
F23:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\alpha - k_3^\alpha)(k_2^\beta - k_3^\beta)k_3^\mu V_{1,\mu}(k_2)k_2^\nu V_{2,\nu}(k_3)$
F32:	$T^{\alpha\beta}(k_2+k_3) \leftarrow \mathrm{i}\cdot g(k_2^\alpha - k_3^\alpha)(k_2^\beta - k_3^\beta)k_2^\nu V_{2,\nu}(k_3)k_3^\mu V_{1,\mu}(k_2)$
F12:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu(k_1^\alpha + 2k_2^\alpha)(k_1^\beta + 2k_2^\beta)T_{\alpha\beta}(k_1)(-k_1^\nu - k_2^\nu)V_{1,\nu}(k_2)$
F21:	$V_2^\mu(k_1+k_2) \leftarrow \mathrm{i}\cdot gk_2^\mu(-k_1^\nu - k_2^\nu)V_{1,\nu}(k_2)(k_1^\alpha + 2k_2^\alpha)(k_1^\beta + 2k_2^\beta)T_{\alpha\beta}(k_1)$
F13:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu(k_1^\alpha + 2k_3^\alpha)(k_1^\beta + 2k_3^\beta)T_{\alpha\beta}(k_1)(-k_1^\nu - k_3^\nu)V_{2,\nu}(k_3)$
F31:	$V_1^\mu(k_1+k_3) \leftarrow \mathrm{i}\cdot gk_3^\mu(-k_1^\nu - k_3^\nu)V_{2,\nu}(k_3)(k_1^\alpha + 2k_3^\alpha)(k_1^\beta + 2k_3^\beta)T_{\alpha\beta}(k_1)$

Table 16.39: ...

Color.t encodes the (SU(N)) color representation.

```
val color : flavor → Color.t
val nc : unit → int
```

The set of conserved charges.

```
module Ch : Charges.T
val charges : flavor → Ch.t
```

The PDG particle code for interfacing with Monte Carlos.

```
val pdg : flavor → int
```

The Lorentz representation of the particle.

```
val lorentz : flavor → Coupling.lorentz
```

The propagator for the particle, which *can* depend on a gauge parameter.

```
type gauge
val propagator : flavor → gauge Coupling.propagator
```

Not the symbol for the numerical value, but the scheme or strategy.

```
val width : flavor → Coupling.width
```

Charge conjugation, with and without color.

```
val conjugate : flavor → flavor
```

Returns 1 for fermions, −1 for anti-fermions, 2 for Majoranas and 0 otherwise.

```
val fermion : flavor → int
```

The Feynman rules. *vertices* and (*fuse2*, *fuse3*, *fusen*) are redundant, of course. However, *vertices* is required for building functors for models and *vertices* can be recovered from (*fuse2*, *fuse3*, *fusen*) only at great cost.

Nevertheless: *vertices* is a candidate for removal, b/c we can build a smarter *Colorize* functor acting on (*fuse2*, *fuse3*, *fusen*). It can support an arbitrary numer of color lines. But we have to test whether it is efficient enough. And we have to make sure that this wouldn't break the UFO interface.

```
type constant

val max_degree : unit → int
val vertices : unit →
  ((((flavor × flavor × flavor) × constant Coupling.vertex3 × constant) list)
      × (((flavor × flavor × flavor × flavor) × constant Coupling.vertex4 × constant) list)
      × (((flavor list) × constant Coupling.vertexn × constant) list))
val fuse2 : flavor → flavor → (flavor × constant Coupling.t) list
val fuse3 : flavor → flavor → flavor → (flavor × constant Coupling.t) list
val fuse : flavor list → (flavor × constant Coupling.t) list
```

For counting coupling orders.

```
type coupling_order
val all_coupling_orders : unit → coupling_order list
val coupling_order_to_string : coupling_order → string
val coupling_orders : constant → (coupling_order × int) list
```

The list of all known flavors.

```
val flavors : unit → flavor list
```

The flavors that can appear in incoming or outgoing states, grouped in a way that is useful for user interfaces.

```
val external_flavors : unit → (string × flavor list) list
```

The Goldstone bosons corresponding to a gauge field, if any.

```
val goldstone : flavor → (flavor × constant Coupling.expr) option
```

The dependent parameters.

```
val parameters : unit → constant Coupling.parameters
```

Translate from and to convenient textual representations of flavors.

```
    val flavor_of_string : string → flavor
    val flavor_to_string : flavor → string
```

TeX and LaTeX

```
    val flavor_to_TeX : flavor → string
```

The following must return unique symbols that are acceptable as symbols in all programming languages under consideration as targets. Strings of alphanumeric characters (starting with a letter) should be safe. Underscores are also usable, but would violate strict Fortran77.

```
    val flavor_symbol : flavor → string
    val gauge_symbol : gauge → string
    val mass_symbol : flavor → string
    val width_symbol : flavor → string
    val constant_symbol : constant → string
```

Model specific options.

```
    val options : Options.t
```

Not ready for prime time or other warnings to be written to the source files for the amplitudes.

```
    val caveats : unit → string list

  end
```

In addition to hardcoded models, we can have models that are initialized at run time.

16.2.2 Mutable Quantum Field Theories

```
module type Mutable =
  sig
    include T
```

Pass initialization data to the model. Typically, this is the name of a UFO directory and we can specialize *Mutable* with type *init* = *string*

```
    type init
    val init : init → unit
    val write_whizard : out_channel → unit
```

Export only one big initialization function to discourage partial initializations. Labels make this usable.

```
    val setup :
        color : (flavor → Color.t) →
        nc : (unit → int) →
        pdg : (flavor → int) →
        lorentz : (flavor → Coupling.lorentz) →
        propagator : (flavor → gauge Coupling.propagator) →
        width : (flavor → Coupling.width) →
        goldstone : (flavor → (flavor × constant Coupling.expr) option) →
        conjugate : (flavor → flavor) →
        fermion : (flavor → int) →
        vertices :
          (unit →
           ((((flavor × flavor × flavor) × constant Coupling.vertex3 × constant) list)
             × (((flavor × flavor × flavor × flavor) × constant Coupling.vertex4 × constant) list)
             × (((flavor list) × constant Coupling.vertexn × constant) list))) →
        flavors : ((string × flavor list) list) →
        parameters : (unit → constant Coupling.parameters) →
        flavor_of_string : (string → flavor) →
        flavor_to_string : (flavor → string) →
        flavor_to_TeX : (flavor → string) →
        flavor_symbol : (flavor → string) →
        gauge_symbol : (gauge → string) →
        mass_symbol : (flavor → string) →
        width_symbol : (flavor → string) →
        constant_symbol : (constant → string) →
        all_coupling_orders : (unit → coupling_order list) →
        coupling_order_to_string : (coupling_order → string) →
        coupling_orders : (constant → (coupling_order × int) list) →
        unit
    end
```

16.2.3 Gauge Field Theories

The following signatures are used only for model building. The diagrammatics and numerics is supposed to be completely ignorant about the detail of the models and expected to rely on the interface T exclusively.

In the end, we might have functors $(M : T) \rightarrow$ *Gauge*, but we will need to add the quantum numbers to T.

```
module type Gauge =
  sig
    include T
```

Matter field carry conserved quantum numbers and can be replicated in generations without changing the gauge sector.

```
    type matter_field
```

Gauge bosons proper.

```
    type gauge_boson
```

Higgses, Goldstones and all the rest:

```
    type other
```

We can query the kind of field

```
    type field =
      | Matter of matter_field
      | Gauge of gauge_boson
      | Other of other
    val field : flavor → field
```

and we can build new fields of a given kind:

```
    val matter_field : matter_field → flavor
    val gauge_boson : gauge_boson → flavor
    val other : other → flavor
  end
```

16.2.4 Gauge Field Theories with Broken Gauge Symmetries

Both are carefully crafted as subtypes of *Gauge* so that they can be used in place of *Gauge* and T everywhere:

```
module type Broken_Gauge =
  sig
    include Gauge

    type massless
    type massive
    type goldstone

    type kind =
      | Massless of massless
      | Massive of massive
      | Goldstone of goldstone
    val kind : gauge_boson → kind

    val massless : massive → gauge_boson
    val massive : massive → gauge_boson
    val goldstone : goldstone → gauge_boson
  end
module type Unitarity_Gauge =
  sig
    include Gauge
    type massless
    type massive
    type kind =
      | Massless of massless
      | Massive of massive
    val kind : gauge_boson → kind
    val massless : massive → gauge_boson
    val massive : massive → gauge_boson
  end
module type Colorized =
  sig
    include T
    type flavor_sans_color
    val flavor_sans_color : flavor → flavor_sans_color
    val conjugate_sans_color : flavor_sans_color → flavor_sans_color
```

amplitude does *not* compute the amplitude, but returns all possible color combinations for the given flavor. These will be used by the functions in *Fusion*.

```
    val amplitude : flavor_sans_color list → flavor_sans_color list →
      (flavor list × flavor list) list
    val flow : flavor list → flavor list → Color.Flow.t
    val flavor_equal : flavor → flavor → bool
  end
module type Colorized_Gauge =
  sig
    include Gauge
    type flavor_sans_color
    val flavor_sans_color : flavor → flavor_sans_color
    val conjugate_sans_color : flavor_sans_color → flavor_sans_color
    val amplitude : flavor_sans_color list → flavor_sans_color list →
      (flavor list × flavor list) list
    val flow : flavor list → flavor list → Color.Flow.t
    val flavor_equal : flavor → flavor → bool
  end
module type Sliced_by_Orders =
  sig
    include Colorized
    type flavor_all_orders
    val flavor_all_orders : flavor → flavor_all_orders
    val conjugate_all_orders : flavor_all_orders → flavor_all_orders
    type orders
    val orders : flavor → orders
    val add_orders : orders → orders → orders
    val incr_orders : orders → orders → orders
    val orders_to_string : orders → string
    val orders_symbol : orders → string
    val trivial : flavor_all_orders → flavor
    val amplitude : orders → flavor_all_orders list → flavor_all_orders list →
                        flavor list × flavor list
    val flow : flavor list → flavor list → Color.Flow.t
  end
```

16.3 Interface of Dirac

16.3.1 Dirac γ-matrices

```
module type T =
  sig
```

Matrices with complex rational entries.

```
    type qc = Algebra.QC.t
    type t = qc array array
```

Complex rational constants.

```
    val zero : qc
    val one : qc
    val minus_one : qc
    val i : qc
    val minus_i : qc
```

Basic γ-matrices.

```
    val unit : t
    val null : t
    val gamma0 : t
    val gamma1 : t
    val gamma2 : t
    val gamma3 : t
    val gamma5 : t
```

$(\gamma_0, \gamma_1, \gamma_2, \gamma_3)$

```
    val gamma : t array
```

Charge conjugation

```
    val cc : t
```

Algebraic operations on γ-matrices

```
    val neg : t → t
    val add : t → t → t
    val sub : t → t → t
    val mul : t → t → t
    val times : qc → t → t
    val transpose : t → t
    val adjoint : t → t
    val conj : t → t
    val product : t list → t
```

Toplevel

```
    val pp : Format.formatter → t → unit
```

Unit tests

```
    val test_suite : OUnit.test
  end
```

```
module Chiral : T
module Dirac : T
module Majorana : T
```

16.4 Implementation of Dirac

16.4.1 Dirac γ-matrices

```
module type T =
  sig
    type qc = Algebra.QC.t
    type t = qc array array
    val zero : qc
    val one : qc
    val minus_one : qc
    val i : qc
    val minus_i : qc
    val unit : t
    val null : t
    val gamma0 : t
    val gamma1 : t
    val gamma2 : t
    val gamma3 : t
    val gamma5 : t
    val gamma : t array
    val cc : t
    val neg : t → t
    val add : t → t → t
    val sub : t → t → t
    val mul : t → t → t
    val times : qc → t → t
    val transpose : t → t
    val adjoint : t → t
    val conj : t → t
    val product : t list → t
    val pp : Format.formatter → t → unit
    val test_suite : OUnit.test
  end
```

Matrices with complex rational entries

```
module Q = Algebra.Q
module QC = Algebra.QC

type complex_rational = QC.t

let zero = QC.null
let one = QC.unit
let minus_one = QC.neg one
let i = QC.make Q.null Q.unit
let minus_i = QC.conj i

type matrix = complex_rational array array
```

Dirac γ-matrices

```
module type R =
  sig
    type qc = complex_rational
    type t = matrix
    val gamma0 : t
    val gamma1 : t
    val gamma2 : t
    val gamma3 : t
    val gamma5 : t
    val cc : t
    val cc_is_i_gamma2_gamma_0 : bool
  end

module Make (R : R) : T =
  struct

    type qc = complex_rational
    type t = matrix

    let zero = zero
    let one = one
    let minus_one = minus_one
    let i = i
    let minus_i = minus_i

    let null =
      [| [| zero; zero; zero; zero |];
         [| zero; zero; zero; zero |];
         [| zero; zero; zero; zero |];
         [| zero; zero; zero; zero |] |]

    let unit =
      [| [| one; zero; zero; zero |];
         [| zero; one; zero; zero |];
         [| zero; zero; one; zero |];
         [| zero; zero; zero; one |] |]

    let gamma0 = R.gamma0
    let gamma1 = R.gamma1
    let gamma2 = R.gamma2
    let gamma3 = R.gamma3
    let gamma5 = R.gamma5
    let gamma = [| gamma0; gamma1; gamma2; gamma3 |]
    let cc = R.cc

    let neg g =
      let g' = Array.make_matrix 4 4 zero in
      for i = 0 to 3 do
        for j = 0 to 3 do
          g'.(i).(j) ← QC.neg g.(i).(j)
        done
      done;
      g'

    let add g1 g2 =
      let g12 = Array.make_matrix 4 4 zero in
      for i = 0 to 3 do
        for j = 0 to 3 do
          g12.(i).(j) ← QC.add g1.(i).(j) g2.(i).(j)
        done
      done;
      g12

    let sub g1 g2 =
      let g12 = Array.make_matrix 4 4 zero in
      for i = 0 to 3 do
        for j = 0 to 3 do
          g12.(i).(j) ← QC.sub g1.(i).(j) g2.(i).(j)
        done
      done;
      g12

    let mul g1 g2 =
      let g12 = Array.make_matrix 4 4 zero in
      for i = 0 to 3 do
      for k = 0 to 3 do
        for j = 0 to 3 do
          g12.(i).(k) ← QC.add g12.(i).(k) (QC.mul g1.(i).(j) g2.(j).(k))
        done
      done
    done;
    g12

let times q g =
  let g' = Array.make_matrix 4 4 zero in
  for i = 0 to 3 do
    for j = 0 to 3 do
      g'.(i).(j) ← QC.mul q g.(i).(j)
    done
  done;
  g'

let transpose g =
  let g' = Array.make_matrix 4 4 zero in
  for i = 0 to 3 do
    for j = 0 to 3 do
      g'.(i).(j) ← g.(j).(i)
    done
  done;
  g'

let adjoint g =
  let g' = Array.make_matrix 4 4 zero in
  for i = 0 to 3 do
    for j = 0 to 3 do
      g'.(i).(j) ← QC.conj g.(j).(i)
    done
  done;
  g'

let conj g =
  let g' = Array.make_matrix 4 4 zero in
  for i = 0 to 3 do
    for j = 0 to 3 do
      g'.(i).(j) ← QC.conj g.(i).(j)
    done
  done;
  g'

let product glist =
  List.fold_right mul glist unit

let pp fmt g =
  let pp_row i =
    for j = 0 to 3 do
      Format.fprintf fmt "␣%8s" (QC.to_string g.(i).(j))
    done in
  Format.fprintf fmt "\n␣/";
  pp_row 0;
  Format.fprintf fmt "␣\\\n";
  for i = 1 to 2 do
    Format.fprintf fmt "␣|";
    pp_row i;
    Format.fprintf fmt "␣|\n"
  done;
  Format.fprintf fmt "␣\\";
  pp_row 3;
  Format.fprintf fmt "␣/\n"

open OUnit
```

```
let two = QC.make (Q.make 2 1) Q.null
let half = QC.make (Q.make 1 2) Q.null
let two_unit = times two unit

let ac_lhs mu nu =
  add (mul gamma.(mu) gamma.(nu)) (mul gamma.(nu) gamma.(mu))

let ac_rhs mu nu =
  if mu = nu then
    if mu = 0 then
      two_unit
    else
      neg two_unit
  else
    null

let test_ac mu nu =
  (ac_lhs mu nu) = (ac_rhs mu nu)

let ac_lhs_all =
  let lhs = Array.make_matrix 4 4 null in
  for mu = 0 to 3 do
    for nu = 0 to 3 do
      lhs.(mu).(nu) ← ac_lhs mu nu
    done
  done;
  lhs

let ac_rhs_all =
  let rhs = Array.make_matrix 4 4 null in
  for mu = 0 to 3 do
    for nu = 0 to 3 do
      rhs.(mu).(nu) ← ac_rhs mu nu
    done
  done;
  rhs

let dump2 lhs rhs =
  for i = 0 to 3 do
    for j = 0 to 3 do
      Printf.printf
        "␣␣␣i␣=␣%d,␣j␣=%d:␣%s␣+␣%s*I␣|␣%s␣+␣%s*I\n"
        i j
        (Q.to_string (QC.re lhs.(i).(j)))
        (Q.to_string (QC.im lhs.(i).(j)))
        (Q.to_string (QC.re rhs.(i).(j)))
        (Q.to_string (QC.im rhs.(i).(j)))
    done
  done

let dump2_all lhs rhs =
  for mu = 0 to 3 do
    for nu = 0 to 3 do
      Printf.printf "mu␣=␣%d,␣nu␣=%d:␣\n" mu nu;
      dump2 lhs.(mu).(nu) rhs.(mu).(nu)
    done
  done

let anticommute =
  "anticommutation␣relations" >::
    (fun () →
      assert_bool
        ""
        (if ac_lhs_all = ac_rhs_all then
           true
```

```
          else
            begin
              dump2_all ac_lhs_all ac_rhs_all;
              false
            end))

  let equal_or_dump2 lhs rhs =
    if lhs = rhs then
      true
    else
      begin
        dump2 lhs rhs;
        false
      end

  let gamma5_def =
    "gamma5" >::
      (fun () →
        assert_bool
          "definition"
          (equal_or_dump2
             gamma5
             (times i (product [gamma0; gamma1; gamma2; gamma3]))))

  let self_adjoint =
    "(anti)selfadjointness" >:::
      [ "gamma0" >::
          (fun () →
            assert_bool "self" (equal_or_dump2 gamma0 (adjoint gamma0)));
        "gamma1" >::
          (fun () →
            assert_bool "anti" (equal_or_dump2 gamma1 (neg (adjoint gamma1))));
        "gamma2" >::
          (fun () →
            assert_bool "anti" (equal_or_dump2 gamma2 (neg (adjoint gamma2))));
        "gamma3" >::
          (fun () →
            assert_bool "anti" (equal_or_dump2 gamma3 (neg (adjoint gamma3))));
        "gamma5" >::
          (fun () →
            assert_bool "self" (equal_or_dump2 gamma5 (adjoint gamma5))) ]
```

$C^2 = -\mathbf{1}$ is *not* true in all realizations, but we assume it at several points in *UFO_Lorentz*. Therefore we must test it here for all realizations that are implemented.

```
  let cc_inv = neg cc
```

Verify that $\Gamma^T = -C\Gamma C^{-1}$ using the actual matrix transpose:

```
  let cc_gamma g =
    equal_or_dump2 (neg (transpose g)) (product [cc; g; cc_inv])
```

Of course, $C = \mathrm{i}\gamma^2\gamma^0$ is also not true in *all* realizations. But it is true in the chiral representation used here and we can test it.

```
  let charge_conjugation =
    "charge␣conjugation" >:::
      [ "inverse" >::
          (fun () →
            assert_bool "" (equal_or_dump2 (mul cc cc_inv) unit));

        "gamma0" >:: (fun () → assert_bool "" (cc_gamma gamma0));
        "gamma1" >:: (fun () → assert_bool "" (cc_gamma gamma1));
        "gamma2" >:: (fun () → assert_bool "" (cc_gamma gamma2));
        "gamma3" >:: (fun () → assert_bool "" (cc_gamma gamma3));

        "gamma5" >::
           (fun () →
              assert_bool "" (equal_or_dump2 (transpose gamma5)
                                              (product [cc; gamma5; cc_inv])));
         "=i*g2*g0" >::
           (fun () →
              skip_if (¬ R.cc_is_i_gamma2_gamma_0)
                "representation␣dependence";
              assert_bool "" (equal_or_dump2 cc (times i (mul gamma2 gamma0))))
       ]

    let test_suite =
      "Dirac␣Matrices" >:::
        [anticommute;
         gamma5_def;
         self_adjoint;
         charge_conjugation]

  end

module Chiral_R : R =
  struct

    type qc = complex_rational
    type t = matrix

    let gamma0 =
      [| [| zero; zero; one; zero |];
         [| zero; zero; zero; one |];
         [| one; zero; zero; zero |];
         [| zero; one; zero; zero |] |]

    let gamma1 =
      [| [| zero; zero; zero; one |];
         [| zero; zero; one; zero |];
         [| zero; minus_one; zero; zero |];
         [| minus_one; zero; zero; zero |] |]

    let gamma2 =
      [| [| zero; zero; zero; minus_i |];
         [| zero; zero; i; zero |];
         [| zero; i; zero; zero |];
         [| minus_i; zero; zero; zero |] |]

    let gamma3 =
      [| [| zero; zero; one; zero |];
         [| zero; zero; zero; minus_one |];
         [| minus_one; zero; zero; zero |];
         [| zero; one; zero; zero |] |]

    let gamma5 =
      [| [| minus_one; zero; zero; zero |];
         [| zero; minus_one; zero; zero |];
         [| zero; zero; one; zero |];
         [| zero; zero; zero; one |] |]

    let cc =
      [| [| zero; one; zero; zero |];
         [| minus_one; zero; zero; zero |];
         [| zero; zero; zero; minus_one |];
         [| zero; zero; one; zero |] |]

    let cc_is_i_gamma2_gamma_0 = true

  end

module Dirac_R : R =
  struct
```

```
    type qc = complex_rational
    type t = matrix

    let gamma0 =
      [| [| one; zero; zero; zero |];
         [| zero; one; zero; zero |];
         [| zero; zero; minus_one; zero |];
         [| zero; zero; zero; minus_one |] |]

    let gamma1 = Chiral_R.gamma1
    let gamma2 = Chiral_R.gamma2
    let gamma3 = Chiral_R.gamma3

    let gamma5 =
      [| [| zero; zero; one; zero |];
         [| zero; zero; zero; one |];
         [| one; zero; zero; zero |];
         [| zero; one; zero; zero |] |]

    let cc =
      [| [| zero; zero; zero; minus_one |];
         [| zero; zero; one; zero |];
         [| zero; minus_one; zero; zero |];
         [| one; zero; zero; zero |] |]

    let cc_is_i_gamma2_gamma_0 = true

  end

module Majorana_R : R =
  struct

    type qc = complex_rational
    type t = matrix

    let gamma0 =
      [| [| zero; zero; zero; minus_i |];
         [| zero; zero; i; zero |];
         [| zero; minus_i; zero; zero |];
         [| i; zero; zero; zero |] |]

    let gamma1 =
      [| [| i; zero; zero; zero |];
         [| zero; minus_i; zero; zero |];
         [| zero; zero; i; zero |];
         [| zero; zero; zero; minus_i |] |]

    let gamma2 =
      [| [| zero; zero; zero; i |];
         [| zero; zero; minus_i; zero |];
         [| zero; minus_i; zero; zero |];
         [| i; zero; zero; zero |] |]

    let gamma3 =
      [| [| zero; minus_i; zero; zero |];
         [| minus_i; zero; zero; zero |];
         [| zero; zero; zero; minus_i |];
         [| zero; zero; minus_i; zero |] |]

    let gamma5 =
      [| [| zero; minus_i; zero; zero |];
         [| i; zero; zero; zero |];
         [| zero; zero; zero; i |];
         [| zero; zero; minus_i; zero |] |]

    let cc =
      [| [| zero; zero; zero; minus_one |];
         [| zero; zero; one; zero |];
```

```
          [| zero; minus_one; zero; zero |];
          [| one; zero; zero; zero |] |]

    let cc_is_i_gamma2_gamma_0 = false

  end

module Chiral = Make (Chiral_R)
module Dirac = Make (Dirac_R)
module Majorana = Make (Majorana_R)
```

16.5 Interface of Target

```
module type T =
  sig
    type amplitudes

    val options : Options.t
    type diagnostic = All | Arguments | Momenta | Gauge
```

Format the amplitudes as a sequence of strings.

```
    val amplitudes_to_channel : string → out_channel →
      (diagnostic × bool) list → amplitudes → unit

    val parameters_to_channel : out_channel → unit

  end

module type Maker =
    functor (F : Fusion.Maker) →
      functor (P : Momentum.T) → functor (M : Model.T) →
        T with type amplitudes = Fusion.Multi(F)(P)(M).amplitudes
```

—17—
Conserved Quantum Numbers

17.1 Interface of Charges

17.1.1 Abstract Type

```
module type T =
  sig
```

The abstract type of the set of conserved charges or additive quantum numbers.

```
    type t
```

Add the quantum numbers of a pair or a list of particles.

```
    val add : t → t → t
    val sum : t list → t
```

Test the charge conservation.

```
    val is_null : t → bool
  end
```

17.1.2 Trivial Realisation

```
module Null : T with type t = unit
```

17.1.3 Nontrivial Realisations

Z

```
module Z : T with type t = int
```

$\mathbf{Z}\times\mathbf{Z}\times\cdots\times\mathbf{Z}$

```
module ZZ : T with type t = int list
```

Q

```
module Q : T with type t = Algebra.Small_Rational.t
```

$\mathbf{Q}\times\mathbf{Q}\times\cdots\times\mathbf{Q}$

```
module QQ : T with type t = Algebra.Small_Rational.t list
```

17.2 Implementation of Charges

```
module type T =
  sig
    type t
    val add : t → t → t
    val sum : t list → t
    val is_null : t → bool
  end

module Null : T with type t = unit =
  struct
    type t = unit
    let add () () = ()
    let sum _ = ()
    let is_null _ = true
  end

module Z : T with type t = int =
  struct
    type t = int
    let add = ( + )
    let sum = List.fold_left add 0
    let is_null n = (n = 0)
  end

module ZZ : T with type t = int list =
  struct
    type t = int list
    let add = List.map2 ( + )
    let sum = function
      | [] → []
      | [charges] → charges
      | charges :: rest → List.fold_left add charges rest
    let is_null = List.for_all (fun n → n = 0)
  end

module Rat = Algebra.Small_Rational

module Q : T with type t = Rat.t =
  struct
    type t = Rat.t
    let add = Rat.add
    let sum = List.fold_left Rat.add Rat.null
    let is_null = Rat.is_null
  end

module QQ : T with type t = Rat.t list =
  struct
    type t = Rat.t list
    let add = List.map2 Rat.add
    let sum = function
      | [] → []
      | [charges] → charges
      | charges :: rest → List.fold_left add charges rest
    let is_null = List.for_all Rat.is_null
  end
```

—18—
Processes

18.1 Interface of Process

```
module type T =
  sig

    type flavor
```

Eventually this should become an abstract type:

```
    type t = flavor list × flavor list

    val incoming : t → flavor list
    val outgoing : t → flavor list
```

parse_decay s decodes a decay description `"a␣->␣b␣c␣..."`, where each word is split into a bag of flavors separated by `':'`s.

```
    type decay
    val parse_decay : string → decay
    val expand_decays : decay list → t list
```

parse_scattering s decodes a scattering description `"a␣b␣->␣c␣d␣..."`, where each word is split into a bag of flavors separated by `':'`s.

```
    type scattering
    val parse_scattering : string → scattering
    val expand_scatterings : scattering list → t list
```

parse_process s decodes process descriptions

$$\texttt{"a b c d"} \Rightarrow \mathit{Any}\ [a;\ b;\ c;\ d] \tag{18.1a}$$
$$\texttt{"a -> b c d"} \Rightarrow \mathit{Decay}\ (a,\ [b;\ c;\ d]) \tag{18.1b}$$
$$\texttt{"a b -> c d"} \Rightarrow \mathit{Scattering}\ (a,\ b,\ [c;\ d]) \tag{18.1c}$$

where each word is split into a bag of flavors separated by `':'`s.

```
    type any
    type process = Any of any | Decay of decay | Scattering of scattering
    val parse_process : string → process
```

remove_duplicate_final_states partition processes removes duplicates from *processes*, which differ only by a permutation of final state particles. The permutation must respect the partitioning given by the offset 1 integers in *partition*.

```
    val remove_duplicate_final_states : int list list → t list → t list
```

diff set1 set2 returns the processes in *set1* with the processes in *set2* removed. *set2* does not need to be a subset of *set1*.

```
    val diff : t list → t list → t list
```

Not functional yet. Interface subject to change. Should be moved to *Fusion.Multi*, because we will want to cross *colored* matrix elements.

Factor amplitudes that are related by crossing symmetry.

```
    val crossing : t list → (flavor list × int list × t) list
  end
module Make (M : Model.T) : T with type flavor = M.flavor
```

18.2 Implementation of Process

```
module type T =
  sig
    type flavor
    type t = flavor list × flavor list
    val incoming : t → flavor list
    val outgoing : t → flavor list
    type decay
    val parse_decay : string → decay
    val expand_decays : decay list → t list
    type scattering
    val parse_scattering : string → scattering
    val expand_scatterings : scattering list → t list
    type any
    type process = Any of any | Decay of decay | Scattering of scattering
    val parse_process : string → process
    val remove_duplicate_final_states : int list list → t list → t list
    val diff : t list → t list → t list
    val crossing : t list → (flavor list × int list × t) list
  end
module Make (M : Model.T) =
  struct

    type flavor = M.flavor

    type t = flavor list × flavor list

    let incoming (fin, _ ) = fin
    let outgoing (_, fout) = fout
```

18.2.1 Select Charge Conserving Processes

```
    let allowed (fin, fout) =
      M.Ch.is_null (M.Ch.sum (List.map M.charges (List.map M.conjugate fin @ fout)))
```

18.2.2 Parsing Process Descriptions

```
    type α bag = α list

    type any = flavor bag list
    type decay = flavor bag × flavor bag list
    type scattering = flavor bag × flavor bag × flavor bag list

    type process =
      | Any of any
      | Decay of decay
      | Scattering of scattering

    let unique_flavors f_bags =
      List.for_all (function [f] → true | _ → false) f_bags

    let unique_final_state = function
      | Any fs → unique_flavors fs
      | Decay (_, fs) → unique_flavors fs
```

```
  | Scattering (_, _, fs) → unique_flavors fs

let parse_process process =
  let last = String.length process − 1
  and flavor off len = M.flavor_of_string (String.sub process off len) in

  let add_flavors flavors = function
    | Any l → Any (List.rev flavors :: l)
    | Decay (i, f) → Decay (i, List.rev flavors :: f)
    | Scattering (i1, i2, f) → Scattering (i1, i2, List.rev flavors :: f) in

  let rec scan_list so_far n =
    if n > last then
      so_far
    else
      let n′ = succ n in
      match process.[n] with
      | ' ' | '\n' → scan_list so_far n′
      | '-' → scan_gtr so_far n′
      | c → scan_flavors so_far [] n n′

  and scan_flavors so_far flavors w n =
    if n > last then
      add_flavors (flavor w (last − w + 1) :: flavors) so_far
    else
      let n′ = succ n in
      match process.[n] with
      | ' ' | '\n' →
          scan_list (add_flavors (flavor w (n − w) :: flavors) so_far) n′
      | ':' → scan_flavors so_far (flavor w (n − w) :: flavors) n′ n′
      | _ → scan_flavors so_far flavors w n′

  and scan_gtr so_far n =
    if n > last then
      invalid_arg "expecting␣‘>’"
    else
      let n′ = succ n in
      match process.[n] with
      | '>' →
          begin match so_far with
          | Any [i] → scan_list (Decay (i, [])) n′
          | Any [i2; i1] → scan_list (Scattering (i1, i2, [])) n′
          | Any _ → invalid_arg "only␣1␣or␣2␣particles␣in␣|in>"
          | _ → invalid_arg "too␣many␣‘->’s"
          end
      | _ → invalid_arg "expecting␣‘>’" in

  match scan_list (Any []) 0 with
  | Any l → Any (List.rev l)
  | Decay (i, f) → Decay (i, List.rev f)
  | Scattering (i1, i2, f) → Scattering (i1, i2, List.rev f)

let parse_decay process =
  match parse_process process with
  | Any (i :: f) →
      prerr_endline "missing␣‘->’␣in␣process␣description,␣assuming␣decay.";
      (i, f)
  | Decay (i, f) → (i, f)
  | _ → invalid_arg "expecting␣decay␣description:␣got␣scattering"

let parse_scattering process =
  match parse_process process with
  | Any (i1 :: i2 :: f) →
      prerr_endline "missing␣‘->’␣in␣process␣description,␣assuming␣scattering.";
      (i1, i2, f)
    | Scattering (i1, i2, f) → (i1, i2, f)
    | _ → invalid_arg "expecting␣scattering␣description:␣got␣decay"

let expand_scatterings scatterings =
  ThoList.flatmap
    (function (fin1, fin2, fout) →
      Product.fold
        (fun flist acc →
          match flist with
          | fin1′ :: fin2′ :: fout′ →
              let fin_fout′ = ([fin1′; fin2′], fout′) in
              if allowed fin_fout′ then
                fin_fout′ :: acc
              else
                acc
          | [_] | [] → failwith "Omega.expand_scatterings:␣can't␣happen")
        (fin1 :: fin2 :: fout) []) scatterings

let expand_decays decays =
  ThoList.flatmap
    (function (fin, fout) →
      Product.fold
        (fun flist acc →
          match flist with
          | fin′ :: fout′ →
              let fin_fout′ = ([fin′], fout′) in
              if allowed fin_fout′ then
                fin_fout′ :: acc
              else
                acc
          | [] → failwith "Omega.expand_decays:␣can't␣happen")
        (fin :: fout) []) decays
```

18.2.3 Remove Duplicate Final States

Test if all final states are the same. Identical to *ThoList.homogeneous* ∘ (*List.map snd*).

```
let rec homogeneous_final_state = function
  | [] | [_] → true
  | (_, fs1) :: ((_, fs2) :: _ as rest) →
      if fs1 ≠ fs2 then
        false
      else
        homogeneous_final_state rest

let by_color f1 f2 =
  let c = Color.compare (M.color f1) (M.color f2) in
  if c ≠ 0 then
    c
  else
    compare f1 f2

module Pre_Bundle =
  struct

    type elt = t
    type base = elt

    let compare_elt (fin1, fout1) (fin2, fout2) =
      let c = ThoList.compare ~cmp : by_color fin1 fin2 in
      if c ≠ 0 then
        c
      else
        ThoList.compare ~cmp : by_color fout1 fout2
    let compare_base b1 b2 = compare_elt b2 b1

  end

module Process_Bundle = Bundle.Dyn (Pre_Bundle)

let to_string (fin, fout) =
  String.concat "␣" (List.map M.flavor_to_string fin)
  ^ "␣->␣" ^ String.concat "␣" (List.map M.flavor_to_string fout)

let fiber_to_string (base, fiber) =
  (to_string base) ^ "␣->␣[" ^
  (String.concat ",␣" (List.map to_string fiber)) ^ "]"

let bundle_to_strings list =
  List.map fiber_to_string list
```

Subtract $n + 1$ from each element in *index_set* and drop all negative numbers from the result.

```
  let shift_left_pred' n index_set =
    List.fold_right
      (fun i acc → let i' = i − n − 1 in if i' < 0 then acc else i' :: acc)
      index_set []
```

Convert 1-based indices for initial and final state to 0-based indices for the final state only. (NB: *ThoList.partitioned_sort* expects 0-based indices.)

```
  let shift_left_pred fin index_sets =
    let n = match fin with [_] → 1 | [_; _] → 2 | _ → 0 in
    List.fold_right
      (fun iset acc →
        match shift_left_pred' n iset with
        | [] → acc
        | iset' → iset' :: acc)
      index_sets []

  module FSet = Set.Make (struct type t = flavor let compare = compare end)
```

Take a list of final states and return a list of sets of flavors appearing in each slot.

```
  let flavors = function
    | [] → []
    | fs :: fs_list →
        List.fold_right (List.map2 FSet.add) fs_list (List.map FSet.singleton fs)

  let flavor_sums flavor_sets =
    let _, result =
      List.fold_left
        (fun (n, acc) flavors →
          if FSet.cardinal flavors = 1 then
            (succ n, acc)
          else
            (succ n, (n, flavors) :: acc))
        (0, []) flavor_sets in
    List.rev result

  let overlapping s1 s2 =
    ¬ (FSet.is_empty (FSet.inter s1 s2))

  let rec merge_overlapping (n, flavors) = function
    | [] → [([n], flavors)]
    | (n_list, flavor_set) :: rest →
        if overlapping flavors flavor_set then
          (n :: n_list, FSet.union flavors flavor_set) :: rest
        else
          (n_list, flavor_set) :: merge_overlapping (n, flavors) rest

  let overlapping_flavor_sums flavor_sums =
    List.rev_map
```

```
      (fun (n_list, flavor_set) → (n_list, FSet.elements flavor_set))
      (List.fold_right merge_overlapping flavor_sums [])

let integer_range n1 n2 =
  let rec integer_range' acc n' =
    if n' < n1 then
      acc
    else
      integer_range' (Sets.Int.add n' acc) (pred n') in
  integer_range' Sets.Int.empty n2

let coarsest_partition = function
  | [] → invalid_arg "coarsest_partition:␣empty␣process␣list"
  | ((_, fs) :: _) as proc_list →
      let fs_list = List.map snd proc_list in
      let overlaps =
        List.map fst (overlapping_flavor_sums (flavor_sums (flavors fs_list))) in
      let singletons =
        Sets.Int.elements
          (List.fold_right Sets.Int.remove
             (List.concat overlaps) (integer_range 0 (pred (List.length fs)))) in
      List.map (fun n → [n]) singletons @ overlaps

module IPowSet =
  PowSet.Make (struct type t = int let compare = compare let to_string = string_of_int end)

let merge_partitions p_list =
  IPowSet.to_lists (IPowSet.basis (IPowSet.union (List.map IPowSet.of_lists p_list)))

let remove_duplicate_final_states cascade_partition = function
  | [] → []
  | [process] → [process]
  | list →
      if homogeneous_final_state list then
        list
      else
        let partition = coarsest_partition list in
        let pi (fin, fout) =
          let partition' =
            merge_partitions [partition; shift_left_pred fin cascade_partition] in
          (fin, ThoList.partitioned_sort by_color partition' fout) in
        Process_Bundle.base (Process_Bundle.of_list pi list)

type t' = t
module PSet = Set.Make (struct type t = t' let compare = compare end)

let set list =
  List.fold_right PSet.add list PSet.empty

let diff list1 list2 =
  PSet.elements (PSet.diff (set list1) (set list2))
```

Not functional yet.

```
module Crossing_Projection =
  struct

    type elt = t
    type base = flavor list × int list × t

    let compare_elt (fin1, fout1) (fin2, fout2) =
      let c = ThoList.compare ~cmp : by_color fin1 fin2 in
      if c ≠ 0 then
        c
      else
        ThoList.compare ~cmp : by_color fout1 fout2
      let compare_base (f1, _, _) (f2, _, _) =
        ThoList.compare ~cmp : by_color f1 f2

      let pi (fin, fout as process) =
        let flist, indices =
          ThoList.ariadne_sort ~cmp : by_color (List.map M.conjugate fin @ fout) in
        (flist, indices, process)

    end

  module Crossing_Bundle = Bundle.Make (Crossing_Projection)

  let crossing processes =
    List.map
      (fun (fin, fout as process) →
        (List.map M.conjugate fin @ fout, [], process))
      processes

end
```

—19—
UFO Models

19.1 Interface of UFOx_syntax

19.1.1 UFO Extensions

We accept the following extensions to the UFO format:

1. Young tableaux: they are representated as a list of lists of integers using "`,`" as separators. E. g.

$$\begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array} \tag{19.1}$$

   is written as `[[1,3],[2]]`. The contents of cells in a Young tableau for the representation of a particle must be consecutive positive integers starting with 1. The representation for the anti particle has all integers negated, e. g. `[[-1,-3],[-2]]`.

2. Young tableaux for particles and anti particles can appear in the *new* optional attribute `color_young`. If `color_young` is present, `color` should be set to the non-standard value 0.

3. Young tableaux for particles (but not for anti particles!) can also appear in the `color` attribute of vertices as the first argument of the new tensors `Delta` and `TY`, representing the Kronecker-δ and the generator T_a in the given representation. The gauge vertex in the above representation would be written

```
color = ['TY([[1,3],[2]],3,1,2)']
```

   where the gluon would be at position 3, the particle at position 1 and the anti particle at position 2. The numbers in the Young tableau and the numbers denoting the position of the particles are completely unrelated, of course.

Note that the cells in the Young tableaux used internally by O'Mega start from 0. Using this in the UFO files would have required to introduce even more special syntax for charge conjugation.

19.1.2 Abstract Syntax

```
exception Syntax_Error of string × Lexing.position × Lexing.position

type expr =
  | Integer of int
  | Float of float
  | Variable of string
  | Quoted of string
  | Young_Tableau of int Young.tableau
  | Sum of expr × expr
  | Difference of expr × expr
  | Product of expr × expr
  | Quotient of expr × expr
  | Power of expr × expr
  | Application of string × expr list

val integer : int → expr
val float : float → expr
val variable : string → expr
val quoted : string → expr
val young_tableau : int Young.tableau → expr
val add : expr → expr → expr
val subtract : expr → expr → expr
val multiply : expr → expr → expr
val divide : expr → expr → expr
val power : expr → expr → expr
val apply : string → expr list → expr
```

Return the sets of variable and function names referenced in the expression.

```
val variables : expr → Sets.String_Caseless.t
val functions : expr → Sets.String_Caseless.t
```

19.2 Implementation of *UFOx_syntax*

19.2.1 Abstract Syntax

```
exception Syntax_Error of string × Lexing.position × Lexing.position

type expr =
  | Integer of int
  | Float of float
  | Variable of string
  | Quoted of string
  | Young_Tableau of int Young.tableau
  | Sum of expr × expr
  | Difference of expr × expr
  | Product of expr × expr
  | Quotient of expr × expr
  | Power of expr × expr
  | Application of string × expr list

let integer i =
  Integer i

let float x =
  Float x

let variable s =
  Variable s

let quoted s =
  Quoted s

let young_tableau y =
  Young_Tableau y

let add e1 e2 =
  Sum (e1, e2)

let subtract e1 e2 =
  Difference (e1, e2)
```

This smart constructor is required since we parse negative numbers as unary minus applied to a positive number. *UFOx.Lorentz_Atom′.of_expr* and *UFOx.Color_Atom′.of_expr* expect negative numbers as summation indices and not expressions. Strictly speaking, we only need the case $e1 = Integer\ (-1)$ for this, but the rest is natural.

There used to be a special rule in the grammar, but this cause reduce/reduce conflicts, that harmless, but annoying.

```
let multiply e1 e2 =
  match e1, e2 with
  | Integer i1, Integer i2 → Integer (i1 × i2)
  | Integer i, Float x | Float x, Integer i → Float (float_of_int i *. x)
  | Float x1, Float x2 → Float (x1 *. x2)
  | e1, e2 → Product (e1, e2)
```

```
let divide e1 e2 =
  Quotient (e1, e2)

let power e p =
  Power (e, p)

let apply f args =
  Application (f, args)

module CSet = Sets.String_Caseless

let rec variables = function
  | Integer _ | Float _ | Quoted _ | Young_Tableau _ → CSet.empty
  | Variable name → CSet.singleton name
  | Sum (e1, e2) | Difference (e1, e2)
  | Product (e1, e2) | Quotient (e1, e2)
  | Power (e1, e2) → CSet.union (variables e1) (variables e2)
  | Application (_, elist) →
      List.fold_left CSet.union CSet.empty (List.map variables elist)

let rec functions = function
  | Integer _ | Float _ | Variable _ | Quoted _ | Young_Tableau _ → CSet.empty
  | Sum (e1, e2) | Difference (e1, e2)
  | Product (e1, e2) | Quotient (e1, e2)
  | Power (e1, e2) → CSet.union (functions e1) (functions e2)
  | Application (f, elist) →
      List.fold_left CSet.union (CSet.singleton f) (List.map functions elist)
```

19.3 Expression Lexer

```
{
open Lexing
open UFOx_parser

let string_of_char c =
  String.make 1 c

let init_position fname lexbuf =
  let curr_p = lexbuf.lex_curr_p in
  lexbuf.lex_curr_p ←
    { curr_p with
      pos_fname = fname;
      pos_lnum = 1;
      pos_bol = curr_p.pos_cnum };
  lexbuf

}

let digit = ['0'-'9']
let upper = ['A'-'Z']
let lower = ['a'-'z']
let char = upper | lower
let word = char | digit | '_'
let white = [' ' '\t' '\n']

rule token = parse
    white { token lexbuf } (* skip blanks *)
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | ',' { COMMA }
  | '*' '*' { POWER }
  | '*' { TIMES }
  | '/' { DIV }
  | '+' { PLUS }
  | '-' { MINUS }
  | ( digit+ as i ) ( '.' '0'* )?
                              { INT (int_of_string i) }
  | ( digit | digit* '.' digit+
            | digit+ '.' digit* ) ( ['E''e'] '-'? digit+ )? as x
                              { FLOAT (float_of_string x) }
  | '\'' (char word* as s) '\''
                              { QUOTED s }
  | char word* ('.' char word+ )? as s
                              { ID s }
  | '\\' '[' (word+ as stem) ']' (word* as suffix)
                              { ID (UFO_tools.mathematica_symbol stem suffix) }
  | _ as c { raise (UFO_tools.Lexical_Error
                                        ("invalid␣character␣`" ^ string_of_char c ^ "'",
                                         lexbuf.lex_start_p, lexbuf.lex_curr_p)) }
  | eof { END }
```

19.4 Expression Parser

Right recursion is more convenient for constructing the value. Since the lists will always be short, there is no performace or stack size reason for prefering left recursion.

Header

```
module X = UFOx_syntax

let parse_error msg =
  raise (UFOx_syntax.Syntax_Error
           (msg, symbol_start_pos (), symbol_end_pos ()))

let invalid_parameter_attr () =
  parse_error "invalid␣parameter␣attribute"
```

Token declarations

```
%token < int > INT
%token < float > FLOAT
%token < string > ID QUOTED
%token PLUS MINUS TIMES POWER DIV
%token LPAREN RPAREN LBRACKET RBRACKET COMMA

%token END

%left PLUS MINUS
%left TIMES DIV
%nonassoc UNARY
%right POWER

%start input
%type < UFOx_syntax.expr > input
```

Grammar rules

```
input ::=
 | expr END { $1 }
```

```
expr ::=
 | INT { X.integer $1 }
 | FLOAT { X.float $1 }
 | ID { X.variable $1 }
 | QUOTED { X.quoted $1 }
 | young_tableau { X.young_tableau $1 }
 | expr PLUS expr { X.add $1 $3 }
 | expr MINUS expr { X.subtract $1 $3 }
 | expr TIMES expr { X.multiply $1 $3 }
 | expr DIV expr { X.divide $1 $3 }
 | PLUS expr %prec UNARY { $2 }
 | MINUS expr %prec UNARY { X.multiply (X.integer (−1)) $2 }
 | expr POWER expr { X.power $1 $3 }
 | LPAREN expr RPAREN { $2 }
 | ID LPAREN RPAREN { X.apply $1 [] }
 | ID LPAREN args RPAREN { X.apply $1 $3 }

args ::=
 | expr { [$1] }
 | expr COMMA args { $1 :: $3 }

young_tableau ::=
 | LBRACKET RBRACKET { [] }
 | LBRACKET integer_lists RBRACKET { $2 }

integer_lists ::=
 | integer_list { [$1] }
 | integer_list COMMA integer_lists { $1 :: $3 }

integer_list ::=
 | LBRACKET RBRACKET { [] }
 | LBRACKET integers RBRACKET { $2 }

integers ::=
 | integer { [$1] }
 | integer COMMA integers { $1 :: $3 }

integer ::=
 | INT { $1 }
 | MINUS INT { − $2 }
```

19.5 Interface of UFOx

```
module Expr :
  sig
    type t
    val of_string : string → t
    val of_strings : string list → t
    val substitute : string → t → t → t
    val rename : (string × string) list → t → t
    val map_names : (string → string) → t → t
    val half : string → t
    val variables : t → Sets.String_Caseless.t
    val functions : t → Sets.String_Caseless.t
  end
```

```
module Value :
  sig
    type t
    val of_expr : Expr.t → t
    val to_string : t → string
    val to_coupling : (string → β) → t → β Coupling.expr
  end
```

UFO represents rank-2 indices (i, j) as $1000 \cdot j + i$. This should be replaced by a proper union type eventually. Unfortunately, this requires many changes in the *Atoms* in *UFOx*. Therefore, we try a quick'n'dirty proof of principle first.

```
module type Index =
  sig
    type t = int

    val position : t → int
    val factor : t → int
    val unpack : t → int × int
    val pack : int → int → t
    val map_position : (int → int) → t → t
    val to_string : t → string
    val list_to_string : t list → string
```

Indices are represented by a pair $int \times \rho$, where ρ denotes the representation the index belongs to.
free indices returns all free indices in the list *indices*, i. e. all positive indices.

```
    val free : (t × ρ) list → (t × ρ) list
```

summation indices returns all summation indices in the list *indices*, i. e. all negative indices.

```
    val summation : (t × ρ) list → (t × ρ) list

    val classes_to_string : (ρ → string) → (t × ρ) list → string
```

Generate summation indices, starting from -1001. TODO: check that there are no clashes with explicitely named indices.

```
    val fresh_summation : unit → t
    val named_summation : string → unit → t

  end

module Index : Index

module type Tensor =
  sig

    type atom
```

A tensor is a linear combination of products of *atoms* with rational coefficients. The following could be refined by introducing *scalar* atoms and restricting the denominators to (*scalar list* × *Algebra.QC.t*) *list*. At the moment, this restriction is implemented dynamically by *of_expr* and not statically in the type system. Polymorphic variants appear to be the right tool, either directly or as phantom types. However, this is certainly only *nice-to-have* and is not essential.

```
    type α linear = (α list × Algebra.QC.t) list
    type t =
      | Linear of atom linear
      | Ratios of (atom linear × atom linear) list
```

We might need to replace atoms if the syntax is not context free.

```
    val map_atoms : (atom → atom) → t → t
```

We need to rename indices to implement permutations ...

```
    val map_indices : (int → int) → t → t
```

... but in order to to clean up inconsistencies in the syntax of `lorentz.py` and `propagators.py` we also need to rename indices without touching the second argument of `P`, the argument of `Mass` etc.

```
    val rename_indices : (int → int) → t → t
```

We need scale coefficients.

```
    val map_coeff : (Algebra.QC.t → Algebra.QC.t) → t → t
```

Try to contract adjacent pairs of *atoms* as allowed but *Atom.contract_pair*. This is not exhaustive, but helps a lot with invariant squares of momenta in applications of *Lorentz*.

```
    val contract_pairs : t → t
```

The list of variable referenced in the tensor expression, that will need to be imported by the numerical code.

```
    val variables : t → string list
```

Parsing and unparsing. Lists of *string*s are interpreted as sums.

```
    val of_expr : UFOx_syntax.expr → t
    val of_string : string → t
    val of_strings : string list → t
    val to_string : t → string
```

The supported representations.

```
    type r
    val classify_indices : t → (int × r) list
    val rep_to_string : r → string
    val rep_to_string_whizard : r → string
    val rep_of_int : bool → int → r
    val rep_of_int_or_young_tableau : bool → int option → int Young.tableau option → r
    val rep_conjugate : r → r
    val rep_trivial : r → bool
```

There is not a 1-to-1 mapping between the representations in the model files and the representations used by O'Mega, e. g. in *Coupling.lorentz*. We might need to use heuristics.

```
    type r_omega
    val omega : r → r_omega

  end

module type Atom =
  sig
    type t
    val map_indices : (int → int) → t → t
    val rename_indices : (int → int) → t → t
    val contract_pair : t → t → t option
    val variable : t → string option
    val scalar : t → bool
    val is_unit : t → bool
    val invertible : t → bool
    val invert : t → t
    val of_expr : string → UFOx_syntax.expr list → t list
    val to_string : t → string
    type r
    val classify_indices : t list → (int × r) list
    val disambiguate_indices : t list → t list
    val rep_to_string : r → string
    val rep_to_string_whizard : r → string
    val rep_of_int : bool → int → r
    val rep_of_int_or_young_tableau : bool → int option → int Young.tableau option → r
    val rep_conjugate : r → r
    val rep_trivial : r → bool
    type r_omega
    val omega : r → r_omega
  end

module type Lorentz_Atom =
  sig
```

```
    type dirac = private
      | C of int × int
      | Gamma of int × int × int
      | Gamma5 of int × int
      | Identity of int × int
      | ProjP of int × int
      | ProjM of int × int
      | Sigma of int × int × int × int

    type vector = (* private *)
      | Epsilon of int × int × int × int
      | Metric of int × int
      | P of int × int

    type scalar = (* private *)
      | Mass of int
      | Width of int
      | P2 of int
      | P12 of int × int
      | Variable of string
      | Coeff of Value.t

    type t = (* private *)
      | Dirac of dirac
      | Vector of vector
      | Scalar of scalar
      | Inverse of scalar

    val map_indices_scalar : (int → int) → scalar → scalar
    val map_indices_vector : (int → int) → vector → vector
    val rename_indices_vector : (int → int) → vector → vector

  end

module Lorentz_Atom : Lorentz_Atom

module Lorentz : Tensor
  with type atom = Lorentz_Atom.t and type r_omega = Coupling.lorentz

module type Color_Atom =
  sig
    type t = (* private *)
      | Identity of int × int
      | Identity8 of int × int
      | Delta of int Young.tableau × int × int
      | T of int × int × int
      | TY of int Young.tableau × int × int × int
      | F of int × int × int
      | D of int × int × int
      | Epsilon of int × int × int
      | EpsilonBar of int × int × int
      | T6 of int × int × int
      | K6 of int × int × int
      | K6Bar of int × int × int
  end

module Color_Atom : Color_Atom

module Color : Tensor
  with type atom = Color_Atom.t and type r_omega = Color.t

module type Test =
  sig
    val suite : OUnit.test
  end
module Test : Test
```

19.6 Implementation of UFOx

```
let error_in_string text start_pos end_pos =
  let i = max 0 start_pos.Lexing.pos_cnum in
  let j = min (String.length text) (max (i + 1) end_pos.Lexing.pos_cnum) in
  String.sub text i (j - i)

let error_in_file name start_pos end_pos =
  Printf.sprintf
    "%s:%d.%d-%d.%d"
    name
    start_pos.Lexing.pos_lnum
    (start_pos.Lexing.pos_cnum - start_pos.Lexing.pos_bol)
    end_pos.Lexing.pos_lnum
    (end_pos.Lexing.pos_cnum - end_pos.Lexing.pos_bol)

module SMap = Map.Make(String)

module Expr =
  struct

    type t = UFOx_syntax.expr

    let of_string text =
      try
        UFOx_parser.input
          UFOx_lexer.token
          (UFOx_lexer.init_position "" (Lexing.from_string text))
      with
      | UFO_tools.Lexical_Error (msg, start_pos, end_pos) →
          invalid_arg (Printf.sprintf "lexical␣error␣(%s)␣at:␣‘%s’"
                         msg (error_in_string text start_pos end_pos))
      | UFOx_syntax.Syntax_Error (msg, start_pos, end_pos) →
          invalid_arg (Printf.sprintf "syntax␣error␣(%s)␣at:␣‘%s’"
                         msg (error_in_string text start_pos end_pos))
      | Parsing.Parse_error →
          invalid_arg ("parse␣error:␣" ^ text)

    let of_strings = function
      | [] → UFOx_syntax.integer 0
      | string :: strings →
          List.fold_right
            (fun s acc → UFOx_syntax.add (of_string s) acc)
            strings (of_string string)

    open UFOx_syntax

    let rec map f = function
      | Integer _ | Float _ | Quoted _ | Young_Tableau _ as e → e
      | Variable s as e →
          begin match f s with
          | Some value → value
          | None → e
          end
      | Sum (e1, e2) → Sum (map f e1, map f e2)
      | Difference (e1, e2) → Difference (map f e1, map f e2)
      | Product (e1, e2) → Product (map f e1, map f e2)
      | Quotient (e1, e2) → Quotient (map f e1, map f e2)
      | Power (e1, e2) → Power (map f e1, map f e2)
      | Application (s, el) → Application (s, List.map (map f) el)

    let substitute name value expr =
      map (fun s → if s = name then Some value else None) expr

    let rename1 name_map name =
      try Some (Variable (SMap.find name name_map)) with Not_found → None
    let rename alist_names value =
      let name_map =
        List.fold_left
          (fun acc (name, name') → SMap.add name name' acc)
          SMap.empty alist_names in
      map (rename1 name_map) value

    let map_name1 f name =
      Some (Variable (f name))

    let map_names f value =
      map (fun name → Some (Variable (f name))) value

    let half name =
      Quotient (Variable name, Integer 2)

    let variables = UFOx_syntax.variables
    let functions = UFOx_syntax.functions

  end
```

It might seem to be a hack to base the decision of whether a sign or parentheses are required on the textual representation of a term. However we control the textual representation, it's efficient and we can avoid duplicating quite a bit of code testing for terms that might produce minus signs.

```
let starts_with_a_sign s =
  String.length s > 0 ∧ let c = s.[0] in c = '-' ∨ c = '+'

let starts_with_a_plus s =
  String.length s > 0 ∧ s.[0] = '+'

let starts_with_a_minus s =
  String.length s > 0 ∧ s.[0] = '-'

let prepend_binary_plus s =
  if starts_with_a_sign s then
    s
  else
    "+" ^ s
```

The safe version that might produce terms like $-(-a)$.

```
let prepend_binary_minus s =
  if starts_with_a_sign s then
    "-(" ^ s ^ ")"
  else
    "-" ^ s
```

The version that produces fewer parentheses, but must assume that a leading minus sign always applies to the *whole* term!

```
let prepend_binary_minus s =
  if starts_with_a_plus s then
    "-" ^ String.sub s 1 (String.length s − 1)
  else if starts_with_a_minus s then
    "+" ^ String.sub s 1 (String.length s − 1)
  else
    "-" ^ s

module Value =
  struct

    module S = UFOx_syntax
    module Q = Algebra.Q

    type builtin =
      | Sqrt
      | Exp | Log | Log10
      | Sin | Asin
      | Cos | Acos
      | Tan | Atan
      | Sinh | Asinh
      | Cosh | Acosh
      | Tanh | Atanh
      | Sec | Asec
      | Csc | Acsc
      | Conj | Abs
    let builtin_to_string = function
      | Sqrt → "sqrt"
      | Exp → "exp"
      | Log → "log"
      | Log10 → "log10"
      | Sin → "sin"
      | Cos → "cos"
      | Tan → "tan"
      | Asin → "asin"
      | Acos → "acos"
      | Atan → "atan"
      | Sinh → "sinh"
      | Cosh → "cosh"
      | Tanh → "tanh"
      | Asinh → "asinh"
      | Acosh → "acosh"
      | Atanh → "atanh"
      | Sec → "sec"
      | Csc → "csc"
      | Asec → "asec"
      | Acsc → "acsc"
      | Conj → "conjg"
      | Abs → "abs"
    let builtin_of_string = function
      | "cmath.sqrt" → Sqrt
      | "cmath.exp" → Exp
      | "cmath.log" → Log
      | "cmath.log10" → Log10
      | "cmath.sin" → Sin
      | "cmath.cos" → Cos
      | "cmath.tan" → Tan
      | "cmath.asin" → Asin
      | "cmath.acos" → Acos
      | "cmath.atan" → Atan
      | "cmath.sinh" → Sinh
      | "cmath.cosh" → Cosh
      | "cmath.tanh" → Tanh
      | "cmath.asinh" → Asinh
      | "cmath.acosh" → Acosh
      | "cmath.atanh" → Atanh
      | "sec" → Sec
      | "csc" → Csc
      | "asec" → Asec
      | "acsc" → Acsc
      | "complexconjugate" → Conj
      | "abs" → Abs
      | name → failwith ("UFOx.Value:␣unsupported␣function:␣" ^ name)
    type t =
      | Integer of int
      | Rational of Q.t
      | Real of float
      | Complex of float × float
      | Variable of string
```

```
      | Sum of t list
      | Difference of t × t
      | Product of t list
      | Quotient of t × t
      | Power of t × t
      | Application of builtin × t list
```

At first sight, unparsing appears to be simpler than parsing. Nevertheless, it can become tricky and error prone if one wants to produce readable output that is not cluttered by too many parentheses.

```
let signed_string_of_float x =
  (if x < 0.0 then "-" else "+") ^ string_of_float (abs_float x)
```

Collect the numerical factors in a *Product* in order to reduce the number of parentheses required.

We could include *Rational*, but is it worth it?

```
let collect_factors elist =
  let rec collect_factors′ factor elist_rev elist =
    match factor, elist with
    | (Integer 1 | Real 1.), [] → List.rev elist_rev
    | _, [] → factor :: List.rev elist_rev
    | Integer i1, Integer i2 :: elist′ →
        collect_factors′ (Integer (i1 × i2)) elist_rev elist′
    | Integer i, Real x :: elist′ | Real x, Integer i :: elist′ →
        collect_factors′ (Real (float i ∗. x)) elist_rev elist′
    | Real x1, Real x2 :: elist′ →
        collect_factors′ (Real (x1 ∗. x2)) elist_rev elist′
    | _, e :: elist′ → collect_factors′ factor (e :: elist_rev) elist′ in
  collect_factors′ (Integer 1) [] elist

let rec to_string = function
  | Integer i → string_of_int i
  | Rational q → Q.to_string q
  | Real x → string_of_float x
  | Complex (0.0, 1.0) → "I"
  | Complex (0.0, i) → group_product (Product [Real i; Complex (0.0, 1.0)])
  | Complex (r, 0.0) → to_string (Real r)
  | Complex (r, i) → group_sum (Sum [Real r; Product [Real i; Complex (0.0, 1.0)]])
  | Variable s → s
  | Sum [] → "0"
  | Sum [e] → to_string e
  | Sum (e :: es) → to_string e ^ String.concat "" (List.map with_binary_plus es)
  | Difference (e1, e2) → to_string e1 ^ prepend_binary_minus (group_sum e2)
  | Product [] → "1"
  | Product es →
      begin match collect_factors es with
      | (Integer (−1) | Real (−1.)) :: es → "-" ^ to_string (Product es)
      | es → String.concat "*" (List.map group_sum es)
      end
  | Quotient (e1, e2) → group_numerator e1 ^ "/" ^ group_denominator e2
  | Power ((Power (_, _) as e1, (Power (_, _) as e2))) →
      "(" ^ group_product e1 ^ ")^(" ^ to_string e2 ^ ")"
  | Power ((Power (_, _) as e1, e2)) →
      "(" ^ group_product e1 ^ ")^" ^ to_string e2
  | Power (e1, (Power (_, _) as e2)) →
      group_product e1 ^ "^(" ^ to_string e2 ^ ")"
  | Power ((Integer i as e), Integer p) →
      if p < 0 then
        group_product (Real (float_of_int i)) ^ "^(" ^ string_of_int p ^ ")"
      else if p = 0 then
        "1"
      else if p ≤ 4 then
```

```
          group_product e ^ "^" ^ string_of_int p
        else
          group_product (Real (float_of_int i)) ^ "^" ^ string_of_int p
    | Power (e1, e2) → group_product e1 ^ "^" ^ group_product e2
    | Application (f, [Integer i]) → to_string (Application (f, [Real (float i)]))
    | Application (f, es) →
        builtin_to_string f ^ "(" ^ String.concat "," (List.map to_string es) ^ ")"
```

Expressions that appear as arguments of *Power*s must be enclosed in parentheses, unless they are singletons. In a denominator, we don't have to put function applications in parentheses.

Check this with `Whizard`'s parser, since this is the main (only?) consumer of our output.

```
  and group_product = function
    | Application (_, _) as e → "(" ^ to_string e ^ ")"
    | e → group_denominator e
```

In numerators, we must be careful not to leave an unprotected minus sign, since they can appear inside products.

```
  and group_numerator = function
    | Product (_ :: _ as es) →
        begin match collect_factors es with
        | (Integer (−1) | Real (−1.)) :: es → "(-" ^ to_string (Product es) ^ ")"
        | es → String.concat "*" (List.map group_sum es)
        end
    | e → group_denominator e

  and group_denominator = function
    | Sum [e] | Product [e] → group_product e
    | Sum ( _ :: _) | Difference (_, _)
    | Product ( _ :: _) | Quotient (_, _) as e → "(" ^ to_string e ^ ")"
    | e → to_string e
```

*Sum*s that appear in *Product*s must be enclosed in parentheses, unless they are singletons.

```
  and group_sum = function
    | Sum [e] | Product [e] → group_sum e
    | Sum ( _ :: _) | Difference (_, _) as e → "(" ^ to_string e ^ ")"
    | e → to_string e
```

Add a `'+'` at the front of a term iff if has no sign.

```
  and with_binary_plus e =
    prepend_binary_plus (to_string e)

  let rec to_coupling atom = function
    | Integer i → Coupling.Integer i
    | Rational q →
        let n, d = Q.to_ratio q in
        Coupling.Quot (Coupling.Integer n, Coupling.Integer d)
    | Real x → Coupling.Float x
    | Product es → Coupling.Prod (List.map (to_coupling atom) es)
    | Variable s → Coupling.Atom (atom s)
    | Complex (r, 0.0) → Coupling.Float r
    | Complex (0.0, 1.0) → Coupling.I
    | Complex (0.0, − 1.0) → Coupling.Prod [Coupling.I; Coupling.Integer (−1)]
    | Complex (0.0, i) → Coupling.Prod [Coupling.I; Coupling.Float i]
    | Complex (r, 1.0) →
        Coupling.Sum [Coupling.Float r; Coupling.I]
    | Complex (r, − 1.0) →
        Coupling.Diff (Coupling.Float r, Coupling.I)
    | Complex (r, i) →
        Coupling.Sum [Coupling.Float r;
                      Coupling.Prod [Coupling.I; Coupling.Float i]]
    | Sum es → Coupling.Sum (List.map (to_coupling atom) es)
    | Difference (e1, e2) →
```

```
      Coupling.Diff (to_coupling atom e1, to_coupling atom e2)
    | Quotient (e1, e2) →
      Coupling.Quot (to_coupling atom e1, to_coupling atom e2)
    | Power (e1, Integer e2) →
      Coupling.Pow (to_coupling atom e1, e2)
    | Power (e1, e2) →
      Coupling.PowX (to_coupling atom e1, to_coupling atom e2)
    | Application (f, [e]) → apply1 (to_coupling atom e) f
    | Application (f, []) →
      failwith
        ("UFOx.Value.to_coupling:␣␣" ^ builtin_to_string f ^
           ":␣empty␣argument␣list")
    | Application (f, _ :: _ :: _) →
      failwith
        ("UFOx.Value.to_coupling:␣" ^ builtin_to_string f ^
           ":␣more␣than␣one␣argument␣in␣list")

and apply1 e = function
    | Sqrt → Coupling.Sqrt e
    | Exp → Coupling.Exp e
    | Log → Coupling.Log e
    | Log10 → Coupling.Log10 e
    | Sin → Coupling.Sin e
    | Cos → Coupling.Cos e
    | Tan → Coupling.Tan e
    | Asin → Coupling.Asin e
    | Acos → Coupling.Acos e
    | Atan → Coupling.Atan e
    | Sinh → Coupling.Sinh e
    | Cosh → Coupling.Cosh e
    | Tanh → Coupling.Tanh e
    | Sec → Coupling.Quot (Coupling.Integer 1, Coupling.Cos e)
    | Csc → Coupling.Quot (Coupling.Integer 1, Coupling.Sin e)
    | Asec → Coupling.Acos (Coupling.Quot (Coupling.Integer 1, e))
    | Acsc → Coupling.Asin (Coupling.Quot (Coupling.Integer 1, e))
    | Conj → Coupling.Conj e
    | Abs → Coupling.Abs e
    | (Asinh | Acosh | Atanh as f) →
      failwith
        ("UFOx.Value.to_coupling:␣function␣‘"
         ^ builtin_to_string f ^ "’␣not␣supported␣yet!")
```

The constant propagation here is incomplete. *S.Quotient* and *S.Power* are not yet handled and in *S.Sum* and *S.Product* only adjacent constants are combined.

We could include *Rational*, but is it worth it?

```
let compress terms = terms

let rec of_expr e =
  compress (of_expr′ e)

and of_expr′ = function
    | S.Integer i → Integer i
    | S.Float x → Real x
    | S.Variable "cmath.pi" → Variable "pi"
    | S.Quoted name →
      invalid_arg ("UFOx.Value.of_expr:␣unexpected␣quoted␣variable␣’" ^
                     name ^ "’")
    | S.Young_Tableau y →
      invalid_arg ("UFOx.Value.of_expr:␣unexpected␣Young␣tableau␣’" ^
                    Young.tableau_to_string string_of_int y ^ "’")
```

```
| S.Variable name → Variable name
| S.Sum (e1, e2) →
    begin match of_expr e1, of_expr e2 with
    | Integer i1, Integer i2 → Integer (i1 + i2)
    | Integer i, Real x | Real x, Integer i → Real (float_of_int i +. x)
    | Real x1, Real x2 → Real (x1 +. x2)
    | (Integer 0 | Real 0.), e → e
    | e, (Integer 0 | Real 0.) → e
    | Sum e1, Sum e2 → Sum (e1 @ e2)
    | e1, Sum e2 → Sum (e1 :: e2)
    | Sum e1, e2 → Sum (e1 @ [e2])
    | e1, e2 → Sum [e1; e2]
    end
| S.Difference (e1, e2) →
    begin match of_expr e1, of_expr e2 with
    | Integer i1, Integer i2 → Integer (i1 − i2)
    | Integer i, Real x → Real (float_of_int i −. x)
    | Real x, Integer i → Real (x −. float_of_int i)
    | Real x1, Real x2 → Real (x1 −. x2)
    | e1, (Integer 0 | Real 0.) → e1
    | e1, e2 → Difference (e1, e2)
    end
| S.Product (e1, e2) →
    begin match of_expr e1, of_expr e2 with
    | Integer i1, Integer i2 → Integer (i1 × i2)
    | Integer i, Real x | Real x, Integer i → Real (float_of_int i ∗. x)
    | Real x1, Real x2 → Real (x1 ∗. x2)
    | (Integer 0 | Real 0.), _ → Integer 0
    | _, (Integer 0 | Real 0.) → Integer 0
    | (Integer 1 | Real 1.), e → e
    | e, (Integer 1 | Real 1.) → e
    | Product e1, Product e2 → Product (e1 @ e2)
    | e1, Product e2 → Product (e1 :: e2)
    | Product e1, e2 → Product (e1 @ [e2])
    | e1, e2 → Product [e1; e2]
    end
| S.Quotient (e1, e2) →
    begin match of_expr e1, of_expr e2 with
    | e1, (Integer 0 | Real 0.) →
        invalid_arg "UFOx.Value:␣divide␣by␣0"
    | e1, (Integer 1 | Real 1.) → e1
    | Integer i1, Integer i2 → Rational (Q.make i1 i2)
    | Real x, Integer i → Real (x /. float i)
    | Integer i, Real x → Real (float i /. x)
    | Real x1, Real x2 → Real (x1 /. x2)
    | e1, e2 → Quotient (e1, e2)
    end
| S.Power (e, p) →
    begin match of_expr e, of_expr p with
    | (Integer 0 | Real 0.), (Integer 0 | Real 0.) →
        invalid_arg "UFOx.Value:␣0^0"
    | _, (Integer 0 | Real 0.) → Integer 1
    | e, (Integer 1 | Real 1.) → e
    | Integer e, Integer p →
        if p < 0 then
          Power (Real (float_of_int e), Integer p)
        else if p = 0 then
          Integer 1
        else if p ≤ 4 then
          Power (Integer e, Integer p)
        else
```

```
              Power (Real (float_of_int e), Integer p)
          | e, p → Power (e, p)
          end
      | S.Application ("complex", [r; i]) →
          begin match of_expr r, of_expr i with
          | r, (Integer 0 | Real 0.0) → r
          | Real r, Real i → Complex (r, i)
          | Integer r, Real i → Complex (float_of_int r, i)
          | Real r, Integer i → Complex (r, float_of_int i)
          | Integer r, Integer i → Complex (float_of_int r, float_of_int i)
          | _ → invalid_arg "UFOx.Value:␣complex␣expects␣two␣numeric␣arguments"
          end
      | S.Application ("complex", _) →
          invalid_arg "UFOx.Value:␣complex␣expects␣two␣arguments"
      | S.Application ("complexconjugate", [e]) →
          Application (Conj, [of_expr e])
      | S.Application ("complexconjugate", _) →
          invalid_arg "UFOx.Value:␣complexconjugate␣expects␣single␣argument"
      | S.Application ("cmath.sqrt", [e]) →
          Application (Sqrt, [of_expr e])
      | S.Application ("cmath.sqrt", _) →
          invalid_arg "UFOx.Value:␣sqrt␣expects␣single␣argument"
      | S.Application (name, args) →
          Application (builtin_of_string name, List.map of_expr args)
  end

let positive integers =
  List.filter (fun (i, _) → i > 0) integers

let not_positive integers =
  List.filter (fun (i, _) → i ≤ 0) integers

module type Index =
  sig

    type t = int

    val position : t → int
    val factor : t → int
    val unpack : t → int × int
    val pack : int → int → t
    val map_position : (int → int) → t → t
    val to_string : t → string
    val list_to_string : t list → string

    val free : (t × ρ) list → (t × ρ) list
    val summation : (t × ρ) list → (t × ρ) list
    val classes_to_string : (ρ → string) → (t × ρ) list → string

    val fresh_summation : unit → t
    val named_summation : string → unit → t

  end

module Index : Index =
  struct

    type t = int

    let free i = positive i
    let summation i = not_positive i

    let position i =
      if i > 0 then
        i mod 1000
      else
        i
```

```
let factor i =
  if i > 0 then
    i / 1000
  else
    invalid_arg "UFOx.Index.factor:␣argument␣not␣positive"

let unpack i =
  if i > 0 then
    (position i, factor i)
  else
    (i, 0)

let pack i j =
  if j > 0 then
    if i > 0 then
      1000 × j + i
    else
      invalid_arg "UFOx.Index.pack:␣position␣not␣positive"
  else if j = 0 then
    i
  else
    invalid_arg "UFOx.Index.pack:␣factor␣negative"

let map_position f i =
  let pos, fac = unpack i in
  pack (f pos) fac

let to_string i =
  let pos, fac = unpack i in
  if fac = 0 then
    Printf.sprintf "%d" pos
  else
    Printf.sprintf "%d.%d" pos fac

let to_string' = string_of_int

let list_to_string is =
  "[" ^ String.concat ",␣" (List.map to_string is) ^ "]"

let classes_to_string rep_to_string index_classes =
  let reps =
    ThoList.uniq (List.sort compare (List.map snd index_classes)) in
  "[" ^
    String.concat ",␣"
    (List.map
       (fun r →
          (rep_to_string r) ^ "=" ^
            (list_to_string
               (List.map
                  fst
                  (List.filter (fun (_, r') → r = r') index_classes))))
       reps) ^ "]"

type factory =
  { mutable named : int SMap.t;
    mutable used : Sets.Int.t }

let factory =
  { named = SMap.empty;
    used = Sets.Int.empty }

let first_anonymous = −1001

let fresh_summation () =
  let next_anonymous =
    try
      pred (Sets.Int.min_elt factory.used)
```

```
        with
        | Not_found → first_anonymous in
      factory.used ← Sets.Int.add next_anonymous factory.used;
      next_anonymous

    let named_summation name () =
      try
        SMap.find name factory.named
      with
      | Not_found →
          begin
            let next_named = fresh_summation () in
            factory.named ← SMap.add name next_named factory.named;
            next_named
          end

  end

module type Atom =
  sig
    type t
    val map_indices : (int → int) → t → t
    val rename_indices : (int → int) → t → t
    val contract_pair : t → t → t option
    val variable : t → string option
    val scalar : t → bool
    val is_unit : t → bool
    val invertible : t → bool
    val invert : t → t
    val of_expr : string → UFOx_syntax.expr list → t list
    val to_string : t → string
    type r
    val classify_indices : t list → (Index.t × r) list
    val disambiguate_indices : t list → t list
    val rep_to_string : r → string
    val rep_to_string_whizard : r → string
    val rep_of_int : bool → int → r
    val rep_of_int_or_young_tableau : bool → int option → int Young.tableau option → r
    val rep_conjugate : r → r
    val rep_trivial : r → bool
    type r_omega
    val omega : r → r_omega
  end

module type Tensor =
  sig
    type atom
    type α linear = (α list × Algebra.QC.t) list
    type t =
      | Linear of atom linear
      | Ratios of (atom linear × atom linear) list
    val map_atoms : (atom → atom) → t → t
    val map_indices : (int → int) → t → t
    val rename_indices : (int → int) → t → t
    val map_coeff : (Algebra.QC.t → Algebra.QC.t) → t → t
    val contract_pairs : t → t
    val variables : t → string list
    val of_expr : UFOx_syntax.expr → t
    val of_string : string → t
    val of_strings : string list → t
    val to_string : t → string
    type r
    val classify_indices : t → (Index.t × r) list
    val rep_to_string : r → string
    val rep_to_string_whizard : r → string
    val rep_of_int : bool → int → r
    val rep_of_int_or_young_tableau : bool → int option → int Young.tableau option → r
    val rep_conjugate : r → r
    val rep_trivial : r → bool
    type r_omega
    val omega : r → r_omega
  end

module Tensor (A : Atom) : Tensor
  with type atom = A.t and type r = A.r and type r_omega = A.r_omega =
  struct

    module S = UFOx_syntax
    (* TODO: we have to switch to Algebra.QC to support complex coefficients, as used in custom propagators.
*)
    module Q = Algebra.Q
    module QC = Algebra.QC

    type atom = A.t
    type α linear = (α list × Algebra.QC.t) list
    type t =
      | Linear of atom linear
      | Ratios of (atom linear × atom linear) list

    let term_to_string (tensors, c) =
      if QC.is_null c then
        ""
      else
        match tensors with
        | [] → QC.to_string c
        | tensors →
            String.concat
              "*" ((if QC.is_unit c then [] else [QC.to_string c]) @
                     List.map A.to_string tensors)

    let linear_to_string = function
      | [] → ""
      | term :: terms →
          term_to_string term ^
            String.concat "" (List.map (fun t → prepend_binary_plus (term_to_string t)) terms)

    let to_string = function
      | Linear terms → linear_to_string terms
      | Ratios ratios →
          String.concat
            "␣+␣"
            (List.map
               (fun (n, d) →
                  Printf.sprintf "(%s)/(%s)"
                    (linear_to_string n) (linear_to_string d)) ratios)

    let variables_of_atoms atoms =
      List.fold_left
        (fun acc a →
           match A.variable a with
           | None → acc
           | Some name → Sets.String.add name acc)
        Sets.String.empty atoms

    let variables_of_linear linear =
      List.fold_left
        (fun acc (atoms, _) → Sets.String.union (variables_of_atoms atoms) acc)
        Sets.String.empty linear
```

```
let variables_set = function
  | Linear linear → variables_of_linear linear
  | Ratios ratios →
      List.fold_left
        (fun acc (numerator, denominator) →
          Sets.String.union
            (variables_of_linear numerator)
            (Sets.String.union (variables_of_linear denominator) acc))
        Sets.String.empty ratios

let variables t =
  Sets.String.elements (variables_set t)

let map_ratios f = function
  | Linear n → Linear (f n)
  | Ratios ratios → Ratios (List.map (fun (n, d) → (f n, f d)) ratios)

let map_summands f t =
  map_ratios (List.map f) t

let map_numerators f = function
  | Linear n → Linear (List.map f n)
  | Ratios ratios →
      Ratios (List.map (fun (n, d) → (List.map f n, d)) ratios)

let map_atoms f t =
  map_summands (fun (atoms, q) → (List.map f atoms, q)) t

let map_indices f t =
  map_atoms (A.map_indices f) t

let rename_indices f t =
  map_atoms (A.rename_indices f) t

let map_coeff f t =
  map_numerators (fun (atoms, q) → (atoms, f q)) t

type result =
  | Matched of atom list
  | Unmatched of atom list
```

contract_pair a rev_prefix suffix returns *Unmatched* (*a* :: *List.rev_append rev_prefix suffix* if there is no match (as defined by *A.contract_pair*) and *Matched* with the reduced list otherwise.

```
let rec contract_pair a rev_prefix = function
  | [] → Unmatched (a :: List.rev rev_prefix)
  | a′ :: suffix →
      begin match A.contract_pair a a′ with
      | None → contract_pair a (a′ :: rev_prefix) suffix
      | Some a″ →
          if A.is_unit a″ then
            Matched (List.rev_append rev_prefix suffix)
          else
            Matched (List.rev_append rev_prefix (a″ :: suffix))
      end
```

Use *contract_pair* to find all pairs that match according to *A.contract_pair*.

```
let rec contract_pairs1 = function
  | ([] | [_] as t) → t
  | a :: t →
      begin match contract_pair a [] t with
      | Unmatched ([]) → []
      | Unmatched (a′ :: t′) → a′ :: contract_pairs1 t′
      | Matched t′ → contract_pairs1 t′
      end

let contract_pairs t =
  map_summands (fun (t', c) → (contract_pairs1 t', c)) t

let add t1 t2 =
  match t1, t2 with
  | Linear l1, Linear l2 → Linear (l1 @ l2)
  | Ratios r, Linear l | Linear l, Ratios r →
      Ratios ((l, [([], QC.unit)]) :: r)
  | Ratios r1, Ratios r2 → Ratios (r1 @ r2)

let multiply1 (t1, c1) (t2, c2) =
  (List.sort compare (t1 @ t2), QC.mul c1 c2)

let multiply2 t1 t2 =
  Product.list2 multiply1 t1 t2

let multiply t1 t2 =
  match t1, t2 with
  | Linear l1, Linear l2 → Linear (multiply2 l1 l2)
  | Ratios r, Linear l | Linear l, Ratios r →
      Ratios (List.map (fun (n, d) → (multiply2 l n, d)) r)
  | Ratios r1, Ratios r2 →
      Ratios (Product.list2
                (fun (n1, d1) (n2, d2) →
                  (multiply2 n1 n2, multiply2 d1 d2))
                r1 r2)

let rec power n t =
  if n < 0 then
    invalid_arg "UFOx.Tensor.power:␣n␣<␣0"
  else if n = 0 then
    Linear [([], QC.unit)]
  else if n = 1 then
    t
  else
    multiply t (power (pred n) t)

let compress ratios =
  map_ratios
    (fun terms →
      List.map (fun (t, cs) → (t, QC.sum cs)) (ThoList.factorize terms))
    ratios

let rec of_expr e =
  contract_pairs (compress (of_expr' e))

and of_expr' = function
  | S.Integer i → Linear [([], QC.make (Q.make i 1) Q.null)]
  | S.Float _ → invalid_arg "UFOx.Tensor.of_expr:␣unexpected␣float"
  | S.Quoted name →
      invalid_arg ("UFOx.Tensor.of_expr:␣unexpected␣quoted␣variable␣'" ^
                     name ^ "'")
  | S.Young_Tableau y →
      invalid_arg ("UFOx.Tensor.of_expr:␣unexpected␣top␣level␣Young␣tableau␣'" ^
                    Young.tableau_to_string string_of_int y ^ "'")
  | S.Variable name →
      (* There should be a gatekeeper here or in A.of_expr: *)
      Linear [(A.of_expr name [], QC.unit)]
  | S.Application ("complex", [re; im]) →
      begin match of_expr re, of_expr im with
      | Linear [([], re)], Linear [([], im)] →
          if QC.is_real re ∧ QC.is_real im then
            Linear [([], QC.make (QC.re re) (QC.re im))]
          else
            invalid_arg ("UFOx.Tensor.of_expr:␣argument␣of␣complex␣is␣complex")
      | _ →
```

```
          invalid_arg "UFOx.Tensor.of_expr:␣unexpected␣argument␣of␣complex"
        end
    | S.Application (name, args) →
        Linear [(A.of_expr name args, QC.unit)]
    | S.Sum (e1, e2) → add (of_expr e1) (of_expr e2)
    | S.Difference (e1, e2) →
        add (of_expr e1) (of_expr (S.Product (S.Integer (−1), e2)))
    | S.Product (e1, e2) → multiply (of_expr e1) (of_expr e2)
    | S.Quotient (n, d) →
        begin match of_expr n, of_expr d with
        | n, Linear [] →
            invalid_arg "UFOx.Tensor.of_expr:␣zero␣denominator"
        | n, Linear [([], q)] → map_coeff (fun c → QC.div c q) n
        | n, Linear ([(invertibles, q)] as d) →
            if List.for_all A.invertible invertibles then
              let inverses = List.map A.invert invertibles in
              multiply (Linear [(inverses, QC.inv q)]) n
            else
              multiply (Ratios [[([], QC.unit)], d]) n
        | n, (Linear d as d′) →
            if List.for_all (fun (t, _) → List.for_all A.scalar t) d then
              multiply (Ratios [[([], QC.unit)], d]) n
            else
              invalid_arg ("UFOx.Tensor.of_expr:␣non␣scalar␣denominator:␣" ^
                           to_string d′)
        | n, (Ratios _ as d) →
            invalid_arg ("UFOx.Tensor.of_expr:␣illegal␣denominator:␣" ^
                         to_string d)
        end
    | S.Power (e, p) →
        begin match of_expr e, of_expr p with
        | Linear [([], q)], Linear [([], p)] →
            if QC.is_real p then
              let re_p = QC.re p in
              if Q.is_integer re_p then
                Linear [([], QC.pow q (Q.to_integer re_p))]
              else
                invalid_arg "UFOx.Tensor.of_expr:␣rational␣power␣of␣number"
            else
              invalid_arg "UFOx.Tensor.of_expr:␣complex␣power␣of␣number"
        | Linear [([], q)], _ →
            invalid_arg "UFOx.Tensor.of_expr:␣non-numeric␣power␣of␣number"
        | t, Linear [([], p)] →
            if QC.is_integer p then
              power (Q.to_integer (QC.re p)) t
            else
              invalid_arg "UFOx.Tensor.of_expr:␣non␣integer␣power␣of␣tensor"
        | _ → invalid_arg "UFOx.Tensor.of_expr:␣non␣numeric␣power␣of␣tensor"
        end

type r = A.r
let rep_to_string = A.rep_to_string
let rep_to_string_whizard = A.rep_to_string_whizard
let rep_of_int = A.rep_of_int
let rep_of_int_or_young_tableau = A.rep_of_int_or_young_tableau
let rep_conjugate = A.rep_conjugate
let rep_trivial = A.rep_trivial

let numerators = function
    | Linear tensors → tensors
    | Ratios ratios → ThoList.flatmap fst ratios
    let classify_indices' filter tensors =
          ThoList.uniq
            (List.sort compare
               (List.map
                  (fun (t, c) → filter (A.classify_indices t))
                  (numerators tensors)))
```

NB: the number of summation indices is not guarateed to be the same! Therefore it was foolish to try to check for uniqueness . . .

```
    let classify_indices tensors =
      match classify_indices' Index.free tensors with
      | [] →
         (* There's always at least an empty list! *)
         failwith "UFOx.Tensor.classify_indices:␣can't␣happen!"
      | [f] → f
      | _ →
         invalid_arg "UFOx.Tensor.classify_indices:␣incompatible␣free␣indices!"

    let disambiguate_indices1 (atoms, q) =
      (A.disambiguate_indices atoms, q)

    let disambiguate_indices tensors =
      map_ratios (List.map disambiguate_indices1) tensors

    let check_indices t =
      ignore (classify_indices t)

    let of_expr e =
      let t = disambiguate_indices (of_expr e) in
      check_indices t;
      t

    let of_string s =
      of_expr (Expr.of_string s)

    let of_strings s =
      of_expr (Expr.of_strings s)

    type r_omega = A.r_omega
    let omega = A.omega

  end

module type Lorentz_Atom =
  sig

    type dirac = private
      | C of int × int
      | Gamma of int × int × int
      | Gamma5 of int × int
      | Identity of int × int
      | ProjP of int × int
      | ProjM of int × int
      | Sigma of int × int × int × int

    type vector = (* private *)
      | Epsilon of int × int × int × int
      | Metric of int × int
      | P of int × int

    type scalar = (* private *)
      | Mass of int
      | Width of int
      | P2 of int
      | P12 of int × int
      | Variable of string
      | Coeff of Value.t
    type t = (* private *)
      | Dirac of dirac
      | Vector of vector
      | Scalar of scalar
      | Inverse of scalar

    val map_indices_scalar : (int → int) → scalar → scalar
    val map_indices_vector : (int → int) → vector → vector
    val rename_indices_vector : (int → int) → vector → vector

  end

module Lorentz_Atom =
  struct

    type dirac =
      | C of int × int
      | Gamma of int × int × int
      | Gamma5 of int × int
      | Identity of int × int
      | ProjP of int × int
      | ProjM of int × int
      | Sigma of int × int × int × int

    type vector =
      | Epsilon of int × int × int × int
      | Metric of int × int
      | P of int × int

    type scalar =
      | Mass of int
      | Width of int
      | P2 of int
      | P12 of int × int
      | Variable of string
      | Coeff of Value.t

    type t =
      | Dirac of dirac
      | Vector of vector
      | Scalar of scalar
      | Inverse of scalar

    let map_indices_scalar f = function
      | Mass i → Mass (f i)
      | Width i → Width (f i)
      | P2 i → P2 (f i)
      | P12 (i, j) → P12 (f i, f j)
      | (Variable _ | Coeff _ as s) → s

    let map_indices_vector f = function
      | Epsilon (mu, nu, ka, la) → Epsilon (f mu, f nu, f ka, f la)
      | Metric (mu, nu) → Metric (f mu, f nu)
      | P (mu, n) → P (f mu, f n)

    let rename_indices_vector f = function
      | Epsilon (mu, nu, ka, la) → Epsilon (f mu, f nu, f ka, f la)
      | Metric (mu, nu) → Metric (f mu, f nu)
      | P (mu, n) → P (f mu, n)

  end

module Lorentz_Atom' : Atom
  with type t = Lorentz_Atom.t and type r_omega = Coupling.lorentz =
  struct

    type t = Lorentz_Atom.t
```

```
open Lorentz_Atom

let map_indices_dirac f = function
  | C (i, j) → C (f i, f j)
  | Gamma (mu, i, j) → Gamma (f mu, f i, f j)
  | Gamma5 (i, j) → Gamma5 (f i, f j)
  | Identity (i, j) → Identity (f i, f j)
  | ProjP (i, j) → ProjP (f i, f j)
  | ProjM (i, j) → ProjM (f i, f j)
  | Sigma (mu, nu, i, j) → Sigma (f mu, f nu, f i, f j)

let rename_indices_dirac = map_indices_dirac

let map_indices_scalar f = function
  | Mass i → Mass (f i)
  | Width i → Width (f i)
  | P2 i → P2 (f i)
  | P12 (i, j) → P12 (f i, f j)
  | Variable s → Variable s
  | Coeff c → Coeff c

let map_indices f = function
  | Dirac d → Dirac (map_indices_dirac f d)
  | Vector v → Vector (map_indices_vector f v)
  | Scalar s → Scalar (map_indices_scalar f s)
  | Inverse s → Inverse (map_indices_scalar f s)

let rename_indices2 fd fv = function
  | Dirac d → Dirac (rename_indices_dirac fd d)
  | Vector v → Vector (rename_indices_vector fv v)
  | Scalar s → Scalar s
  | Inverse s → Inverse s

let rename_indices f atom =
  rename_indices2 f f atom

let contract_pair a1 a2 =
  match a1, a2 with
  | Vector (P (mu1, i1)), Vector (P (mu2, i2)) →
     if mu1 ≤ 0 ∧ mu1 = mu2 then
       if i1 = i2 then
         Some (Scalar (P2 i1))
       else
         Some (Scalar (P12 (i1, i2)))
     else
       None
  | Scalar s, Inverse s′ | Inverse s, Scalar s′ →
     if s = s′ then
       Some (Scalar (Coeff (Value.Integer 1)))
     else
       None
  | _ → None

let variable = function
  | Scalar (Variable s) | Inverse (Variable s) → Some s
  | _ → None

let scalar = function
  | Dirac _ | Vector _ → false
  | Scalar _ | Inverse _ → true

let is_unit = function
  | Scalar (Coeff c) | Inverse (Coeff c) →
     begin match c with
     | Value.Integer 1 → true
     | Value.Rational q → Algebra.Q.is_unit q
```

```
      | _ → false
      end
  | _ → false

let invertible = scalar

let invert = function
  | Dirac _ → invalid_arg "UFOx.Lorentz_Atom.invert␣Dirac"
  | Vector _ → invalid_arg "UFOx.Lorentz_Atom.invert␣Vector"
  | Scalar s → Inverse s
  | Inverse s → Scalar s

let i2s = Index.to_string

let dirac_to_string = function
  | C (i, j) →
      Printf.sprintf "C(%s,%s)" (i2s i) (i2s j)
  | Gamma (mu, i, j) →
      Printf.sprintf "Gamma(%s,%s,%s)" (i2s mu) (i2s i) (i2s j)
  | Gamma5 (i, j) →
      Printf.sprintf "Gamma5(%s,%s)" (i2s i) (i2s j)
  | Identity (i, j) →
      Printf.sprintf "Identity(%s,%s)" (i2s i) (i2s j)
  | ProjP (i, j) →
      Printf.sprintf "ProjP(%s,%s)" (i2s i) (i2s j)
  | ProjM (i, j) →
      Printf.sprintf "ProjM(%s,%s)" (i2s i) (i2s j)
  | Sigma (mu, nu, i, j) →
      Printf.sprintf "Sigma(%s,%s,%s,%s)" (i2s mu) (i2s nu) (i2s i) (i2s j)

let vector_to_string = function
  | Epsilon (mu, nu, ka, la) →
      Printf.sprintf "Epsilon(%s,%s,%s,%s)" (i2s mu) (i2s nu) (i2s ka) (i2s la)
  | Metric (mu, nu) →
      Printf.sprintf "Metric(%s,%s)" (i2s mu) (i2s nu)
  | P (mu, n) →
      Printf.sprintf "P(%s,%d)" (i2s mu) n

let scalar_to_string = function
  | Mass id → Printf.sprintf "Mass(%d)" id
  | Width id → Printf.sprintf "Width(%d)" id
  | P2 id → Printf.sprintf "P(%d)**2" id
  | P12 (id1, id2) → Printf.sprintf "P(%d)*P(%d)" id1 id2
  | Variable s → s
  | Coeff c → Value.to_string c

let to_string = function
  | Dirac d → dirac_to_string d
  | Vector v → vector_to_string v
  | Scalar s → scalar_to_string s
  | Inverse s → "1/" ^ scalar_to_string s

module S = UFOx_syntax
```

Here we handle some special cases in order to be able to parse propagators. This needs to be made more general, but unfortunately the syntax for the propagator extension is not well documented and appears to be a bit chaotic!

```
let quoted_index s =
  Index.named_summation s ()

let integer_or_id = function
  | S.Integer n → n
  | S.Variable "id" → 1
  | _ → failwith "UFOx.Lorentz_Atom.integer_or_id:␣impossible"
```

```
let vector_index = function
  | S.Integer n → n
  | S.Quoted mu → quoted_index mu
  | S.Variable id →
     let l = String.length id in
     if l > 1 then
       if id.[0] = 'l' then
         int_of_string (String.sub id 1 (pred l))
       else
         invalid_arg ("UFOx.Lorentz_Atom.vector_index:␣" ^ id)
     else
       invalid_arg "UFOx.Lorentz_Atom.vector_index:␣empty␣variable"
  | _ → invalid_arg "UFOx.Lorentz_Atom.vector_index"

let spinor_index = function
  | S.Integer n → n
  | S.Variable id →
     let l = String.length id in
     if l > 1 then
       if id.[0] = 's' then
         int_of_string (String.sub id 1 (pred l))
       else
         invalid_arg ("UFOx.Lorentz_Atom.spinor_index:␣" ^ id)
     else
       invalid_arg "UFOx.Lorentz_Atom.spinor_index:␣empty␣variable"
  | _ → invalid_arg "UFOx.Lorentz_Atom.spinor_index"

let of_expr name args =
  match name, args with
  | "C", [i; j] → [Dirac (C (spinor_index i, spinor_index j))]
  | "C", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣C()"
  | "Epsilon", [mu; nu; ka; la] →
     [Vector (Epsilon (vector_index mu, vector_index nu,
                       vector_index ka, vector_index la))]
  | "Epsilon", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Epsilon()"
  | "Gamma", [mu; i; j] →
     [Dirac (Gamma (vector_index mu, spinor_index i, spinor_index j))]
  | "Gamma", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Gamma()"
  | "Gamma5", [i; j] → [Dirac (Gamma5 (spinor_index i, spinor_index j))]
  | "Gamma5", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Gamma5()"
  | "Identity", [i; j] → [Dirac (Identity (spinor_index i, spinor_index j))]
  | "Identity", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Identity()"
  | "Metric", [mu; nu] → [Vector (Metric (vector_index mu, vector_index nu))]
  | "Metric", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Metric()"
  | "P", [mu; id] → [Vector (P (vector_index mu, integer_or_id id))]
  | "P", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣P()"
  | "ProjP", [i; j] → [Dirac (ProjP (spinor_index i, spinor_index j))]
  | "ProjP", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣ProjP()"
  | "ProjM", [i; j] → [Dirac (ProjM (spinor_index i, spinor_index j))]
  | "ProjM", _ →
     invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣ProjM()"
  | "Sigma", [mu; nu; i; j] →
     if mu ≠ nu then
       [Dirac (Sigma (vector_index mu, vector_index nu,
                                spinor_index i, spinor_index j))]
        else
          invalid_arg "UFOx.Lorentz.of_expr:␣implausible␣arguments␣to␣Sigma()"
    | "Sigma", _ →
        invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Sigma()"
    | "PSlash", [i; j; id] →
        let mu = Index.fresh_summation () in
        [Dirac (Gamma (mu, spinor_index i, spinor_index j));
          Vector (P (mu, integer_or_id id))]
    | "PSlash", _ →
        invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣PSlash()"
    | "Mass", [id] → [Scalar (Mass (integer_or_id id))]
    | "Mass", _ →
        invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Mass()"
    | "Width", [id] → [Scalar (Width (integer_or_id id))]
    | "Width", _ →
        invalid_arg "UFOx.Lorentz.of_expr:␣invalid␣arguments␣to␣Width()"
    | name, [] →
        [Scalar (Variable name)]
    | name, _ →
        invalid_arg ("UFOx.Lorentz.of_expr:␣invalid␣tensor␣'" ^ name ^ "'")

type r = S | V | T | Sp | CSp | Maj | VSp | CVSp | VMaj | Ghost

let rep_trivial = function
  | S | Ghost → true
  | V | T | Sp | CSp | Maj | VSp | CVSp | VMaj → false

let rep_to_string = function
  | S → "0"
  | V → "1"
  | T → "2"
  | Sp → "1/2"
  | CSp → "1/2bar"
  | Maj → "1/2M"
  | VSp → "3/2"
  | CVSp → "3/2bar"
  | VMaj → "3/2M"
  | Ghost → "Ghost"

let rep_to_string_whizard = function
  | S → "0"
  | V → "1"
  | T → "2"
  | Sp | CSp | Maj → "1/2"
  | VSp | CVSp | VMaj → "3/2"
  | Ghost → "Ghost"

let rep_of_int neutral = function
  | −1 → Ghost
  | 1 → S
  | 2 → if neutral then Maj else Sp
  | −2 → if neutral then Maj else CSp (* used by UFO.Particle.force_conjspinor *)
  | 3 → V
  | 4 → if neutral then VMaj else VSp
  | −4 → if neutral then VMaj else CVSp (* used by UFO.Particle.force_conjspinor *)
  | 5 → T
  | s when s > 0 →
      failwith "UFOx.Lorentz:␣spin␣>␣2␣not␣supported!"
  | _ →
      invalid_arg "UFOx.Lorentz:␣invalid␣non-positive␣spin␣value"

let rep_of_int_or_young_tableau neutral i yt =
  match i, yt with
  | Some i, None → rep_of_int neutral i
  | None, None → S
  | _, Some _ → invalid_arg "UFOx.Lorentz:␣Young␣tableau␣not␣supported"

let rep_conjugate = function
  | S → S
  | V → V
  | T → T
  | Sp → CSp (* ??? *)
  | CSp → Sp (* ??? *)
  | Maj → Maj
  | VSp → CVSp
  | CVSp → VSp
  | VMaj → VMaj
  | Ghost → Ghost

let classify_vector_indices1 = function
  | Epsilon (mu, nu, ka, la) → [(mu, V); (nu, V); (ka, V); (la, V)]
  | Metric (mu, nu) → [(mu, V); (nu, V)]
  | P (mu, n) → [(mu, V)]

let classify_dirac_indices1 = function
  | C (i, j) → [(i, CSp); (j, Sp)] (* ??? *)
  | Gamma5 (i, j) | Identity (i, j)
  | ProjP (i, j) | ProjM (i, j) → [(i, CSp); (j, Sp)]
  | Gamma (mu, i, j) → [(mu, V); (i, CSp); (j, Sp)]
  | Sigma (mu, nu, i, j) → [(mu, V); (nu, V); (i, CSp); (j, Sp)]

let classify_indices1 = function
  | Dirac d → classify_dirac_indices1 d
  | Vector v → classify_vector_indices1 v
  | Scalar _ | Inverse _ → []

module IMap = Map.Make(Int)

exception Incompatible_factors of r × r

let product rep1 rep2 =
  match rep1, rep2 with
  | V, V → T
  | V, Sp → VSp
  | V, CSp → CVSp
  | V, Maj → VMaj
  | Sp, V → VSp
  | CSp, V → CVSp
  | Maj, V → VMaj
  | _, _ → raise (Incompatible_factors (rep1, rep2))

let combine_or_add_index (i, rep) map =
  let pos, fac = Index.unpack i in
  try
    let fac', rep' = IMap.find pos map in
    if pos < 0 then
      IMap.add pos (fac, rep) map
    else if fac ≠ fac' then
      IMap.add pos (0, product rep rep') map
    else if rep ≠ rep' then (* Can be disambiguated! *)
      IMap.add pos (0, product rep rep') map
    else
      invalid_arg (Printf.sprintf "UFO:␣duplicate␣subindex␣%d" pos)
  with
  | Not_found → IMap.add pos (fac, rep) map
  | Incompatible_factors (rep1, rep2) →
     invalid_arg
       (Printf.sprintf
              "UFO:␣incompatible␣factors␣(%s,%s)␣at␣%d"
              (rep_to_string rep1) (rep_to_string rep2) pos)

    let combine_or_add_indices atom map =
      List.fold_right combine_or_add_index (classify_indices1 atom) map

    let project_factors (pos, (fac, rep)) =
      if fac = 0 then
        (pos, rep)
      else
        invalid_arg (Printf.sprintf "UFO:␣leftover␣subindex␣%d.%d" pos fac)

    let classify_indices atoms =
      List.map
        project_factors
        (IMap.bindings (List.fold_right combine_or_add_indices atoms IMap.empty))

    let add_factor fac indices pos =
      if pos > 0 then
        if Sets.Int.mem pos indices then
          Index.pack pos fac
        else
          pos
      else
        pos

    let disambiguate_indices1 indices atom =
      rename_indices2 (add_factor 1 indices) (add_factor 2 indices) atom

    let vectorspinors atoms =
      List.fold_left
        (fun acc (i, r) →
          match r with
          | S | V | T | Sp | CSp | Maj | Ghost → acc
          | VSp | CVSp | VMaj → Sets.Int.add i acc)
        Sets.Int.empty (classify_indices atoms)

    let disambiguate_indices atoms =
      let vectorspinor_indices = vectorspinors atoms in
      List.map (disambiguate_indices1 vectorspinor_indices) atoms

    type r_omega = Coupling.lorentz
    let omega = function
      | S → Coupling.Scalar
      | V → Coupling.Vector
      | T → Coupling.Tensor_2
      | Sp → Coupling.Spinor
      | CSp → Coupling.ConjSpinor
      | Maj → Coupling.Majorana
      | VSp → Coupling.Vectorspinor
      | CVSp → Coupling.Vectorspinor (* TODO: not really! *)
      | VMaj → Coupling.Vectorspinor (* TODO: not really! *)
      | Ghost → Coupling.Scalar

  end

module Lorentz = Tensor(Lorentz_Atom′)

module type Color_Atom =
  sig
    type t = (* private *)
      | Identity of int × int
      | Identity8 of int × int
      | Delta of int Young.tableau × int × int
      | T of int × int × int
      | TY of int Young.tableau × int × int × int
      | F of int × int × int
      | D of int × int × int
      | Epsilon of int × int × int
      | EpsilonBar of int × int × int
      | T6 of int × int × int
      | K6 of int × int × int
      | K6Bar of int × int × int
  end

module Color_Atom =
  struct
    type t =
      | Identity of int × int
      | Identity8 of int × int
      | Delta of int Young.tableau × int × int
      | T of int × int × int
      | TY of int Young.tableau × int × int × int
      | F of int × int × int
      | D of int × int × int
      | Epsilon of int × int × int
      | EpsilonBar of int × int × int
      | T6 of int × int × int
      | K6 of int × int × int
      | K6Bar of int × int × int
  end

module Color_Atom′ : Atom
  with type t = Color_Atom.t and type r_omega = Color.t =
  struct

    type t = Color_Atom.t

    module S = UFOx_syntax

    open Color_Atom

    let map_indices f = function
      | Identity (i, j) → Identity (f i, f j)
      | Identity8 (a, b) → Identity8 (f a, f b)
      | Delta (y, a, b) → Delta (y, f a, f b)
      | T (a, i, j) → T (f a, f i, f j)
      | TY (y, a, i, j) → TY (y, f a, f i, f j)
      | F (a, i, j) → F (f a, f i, f j)
      | D (a, i, j) → D (f a, f i, f j)
      | Epsilon (i, j, k) → Epsilon (f i, f j, f k)
      | EpsilonBar (i, j, k) → EpsilonBar (f i, f j, f k)
      | T6 (a, i′, j′) → T6 (f a, f i′, f j′)
      | K6 (i′, j, k) → K6 (f i′, f j, f k)
      | K6Bar (i′, j, k) → K6Bar (f i′, f j, f k)

    let rename_indices = map_indices

    let contract_pair _ _ = None
    let variable _ = None
    let scalar _ = false
    let invertible _ = false
    let is_unit _ = false

    let invert _ =
      invalid_arg "UFOx.Color_Atom.invert"

    let young_tableau_valid_particle y =
      Young.standard_tableau ~offset : 1 y

    let of_expr1 name args =
      match name, args with
      | "Identity", [S.Integer i; S.Integer j] → Identity (i, j)
      | "Identity", _ →
```

```
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣Identity()"
    | "Delta", [S.Young_Tableau y; S.Integer i; S.Integer j] →
        if young_tableau_valid_particle y then
            Delta (y, i, j)
        else
          invalid_arg ("UFOx.Color.of_expr:␣invalid␣Young␣tableau␣in␣Delta:␣" ^
                         Young.tableau_to_string string_of_int y)
    | "Delta", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣Identity()"
    | "T", [S.Integer a; S.Integer i; S.Integer j] → T (a, i, j)
    | "T", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣T()"
    | "TY", [S.Young_Tableau y; S.Integer a; S.Integer i; S.Integer j] →
        if young_tableau_valid_particle y then
          TY (y, a, i, j)
        else
          invalid_arg ("UFOx.Color.of_expr:␣invalid␣Young␣tableau␣in␣TY:␣" ^
                         Young.tableau_to_string string_of_int y)
    | "TY", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣TY()"
    | "f", [S.Integer a; S.Integer b; S.Integer c] → F (a, b, c)
    | "f", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣f()"
    | "d", [S.Integer a; S.Integer b; S.Integer c] → D (a, b, c)
    | "d", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣d()"
    | "Epsilon", [S.Integer i; S.Integer j; S.Integer k] →
        Epsilon (i, j, k)
    | "Epsilon", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣Epsilon()"
    | "EpsilonBar", [S.Integer i; S.Integer j; S.Integer k] →
        EpsilonBar (i, j, k)
    | "EpsilonBar", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣EpsilonBar()"
    | "T6", [S.Integer a; S.Integer i'; S.Integer j'] → T6 (a, i', j')
    | "T6", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣T6()"
    | "K6", [S.Integer i'; S.Integer j; S.Integer k] → K6 (i', j, k)
    | "K6", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣K6()"
    | "K6Bar", [S.Integer i'; S.Integer j; S.Integer k] → K6Bar (i', j, k)
    | "K6Bar", _ →
        invalid_arg "UFOx.Color.of_expr:␣invalid␣arguments␣to␣K6Bar()"
    | name, _ →
        invalid_arg ("UFOx.Color.of_expr:␣invalid␣tensor␣'" ^ name ^ "'")

let of_expr name args =
  [of_expr1 name args]

let to_string = function
    | Identity (i, j) → Printf.sprintf "Identity(%d,%d)" i j
    | Identity8 (a, b) → Printf.sprintf "Identity8(%d,%d)" a b
    | Delta (y, a, b) → Printf.sprintf "Delta(%s,%d,%d)" (Young.tableau_to_string string_of_int y) a b
    | T (a, i, j) → Printf.sprintf "T(%d,%d,%d)" a i j
    | TY (y, a, i, j) → Printf.sprintf "TY(%s,%d,%d,%d)" (Young.tableau_to_string string_of_int y) a i j
    | F (a, b, c) → Printf.sprintf "f(%d,%d,%d)" a b c
    | D (a, b, c) → Printf.sprintf "d(%d,%d,%d)" a b c
    | Epsilon (i, j, k) → Printf.sprintf "Epsilon(%d,%d,%d)" i j k
    | EpsilonBar (i, j, k) → Printf.sprintf "EpsilonBar(%d,%d,%d)" i j k
    | T6 (a, i', j') → Printf.sprintf "T6(%d,%d,%d)" a i' j'
    | K6 (i', j, k) → Printf.sprintf "K6(%d,%d,%d)" i' j k
    | K6Bar (i', j, k) → Printf.sprintf "K6Bar(%d,%d,%d)" i' j k
```

```
type r = S | F | C | A | YT of int Young.tableau

let conjugate_tableau y =
  Young.map (−) y

let young_tableau_valid_UFO y =
  young_tableau_valid_particle y ∨
    young_tableau_valid_particle (conjugate_tableau y)

let young_to_string y =
  ThoList.to_string (ThoList.to_string string_of_int) y

let rep_trivial = function
  | S | YT [] | YT [[]] → true
  | F | C | A | YT _ → false

let rep_to_string = function
  | S → "1"
  | F → "3"
  | C → "3bar"
  | A → "8"
  | YT y → young_to_string y

let rep_to_string_whizard = function
  | S → "1"
  | F → "3"
  | C → "-3"
  | A → "8"
  | YT y → young_to_string y

let rep_of_int neutral = function
  | 1 → S
  | 3 → F
  | − 3 → C
  | 8 → A
  | 6 → YT [[1; 2]]
  | − 6 → YT [[−1; −2]]
  | 10 → YT [[1; 2; 3]]
  | − 10 → YT [[−1; −2; −3]]
  | n →
     invalid_arg
       (Printf.sprintf
          "UFOx.Color:␣impossible␣representation␣color␣=␣%d!" n)

let simplify_young_tableau = function
  | [] | [[]] → S
  | [[i]] →
     if i < 0 then
       C
     else
       F
  | y → YT y

let rep_of_int_or_young_tableau neutral i = function
  | None →
     begin match i with
     | Some i → rep_of_int neutral i
     | None →
        Printf.eprintf "UFO:␣warning:␣missing␣required␣attribute␣color!\n";
        S
     end
  | Some y →
     if young_tableau_valid_UFO y then
       begin match i with
       | None | Some 0 → YT y
       | Some i →
```

```
              let ri = rep_of_int neutral i in
              if ri = simplify_young_tableau y then
                 ri
              else
                 invalid_arg
                    (Printf.sprintf
                       "UFOx.Color.rep_of_int_or_young_tableau:␣color␣=␣%d␣!=␣color_young␣=␣%s"
                       i (young_to_string y))
           end
        else
           invalid_arg
             ("UFOx.Color.rep_of_int_or_young_tableau:␣not␣a␣standard␣tableau:␣" ^ young_to_string y)

    let rep_conjugate = function
      | S → S
      | C → F
      | F → C
      | A → A
      | YT y → YT (conjugate_tableau y)
```

Check the particle/anti-particle assignments for the sextets!

```
    let classify_indices1 = function
      | Identity (i, j) → [(i, C); (j, F)]
      | Identity8 (a, b) → [(a, A); (b, A)]
      | Delta (y, a, b) → [(a, YT (conjugate_tableau y)); (b, YT y)]
      | T (a, i, j) → [(i, F); (j, C); (a, A)]
      | TY (y, a, i, j) → [(i, YT y); (j, YT (conjugate_tableau y)); (a, A)]
      | Color_Atom.F (a, b, c) | D (a, b, c) → [(a, A); (b, A); (c, A)]
      | Epsilon (i, j, k) → [(i, F); (j, F); (k, F)]
      | EpsilonBar (i, j, k) → [(i, C); (j, C); (k, C)]
      | T6 (a, i, j) → [(a, A); (i, YT [[1; 2]]); (j, YT [[−1; −2]])]
      | K6 (i, j, k) → [(i, YT [[−1; −2]]); (j, F); (k, F)]
      | K6Bar (i, j, k) → [(i, YT [[1; 2]]); (j, C); (k, C)]

    let classify_indices tensors =
      List.sort compare
        (List.fold_right
           (fun v acc → classify_indices1 v @ acc)
           tensors [ ])

    let disambiguate_indices atoms =
      atoms

    type r_omega = Color.t
```

Our encoding of charge conjugation only works if the indices start from 1. In *SU3*, we use tableau with indices that start from 0.
FIXME: $N_C = 3$ should not be hardcoded!

```
    let omega = function
      | S → Color.Singlet
      | F → Color.SUN (3)
      | C → Color.SUN (−3)
      | A → Color.AdjSUN (3)
      | YT [ ] | YT [[ ]] → Color.Singlet
      | YT ([ ] :: _ as y) → failwith ("UFOx.Color.omega:␣invalid␣tableau:␣" ^ young_to_string y)
      | YT ((i0 :: _) :: _ as y) →
         let y = Young.map (fun i → abs i − 1) y in
         if i0 < 0 then
           Color.YTC y
         else
           Color.YT y
```

```
  end
module Color = Tensor(Color_Atom')
module type Test =
  sig
    val suite : OUnit.test
  end
module Test : Test =
  struct
    open OUnit
    let parse_unparse s =
      Value.to_string (Value.of_expr (Expr.of_string s))
    let apup unparsed expr =
      assert_equal ~printer : (fun s → s) unparsed (parse_unparse expr)
    let apup_id expr =
      apup expr expr
    let suite_arithmetic =
      "arithmetic" >:::
        [ "1␣+␣2" >:: (fun () → apup "3" "1+2");
          "1␣-␣2" >:: (fun () → apup "-1" "1-2");
          "3␣*␣2" >:: (fun () → apup "6" "3*2");
          "3␣*␣(-2)" >:: (fun () → apup "-6" "3*(-2)");
          "3␣/␣2" >:: (fun () → apup "(3/2)" "3/2");
          "4␣/␣12" >:: (fun () → apup "(1/3)" "4/12");
          "4␣/␣(-6)" >:: (fun () → apup "(-2/3)" "4/(-6)");
          "3␣*␣(6␣/␣12)" >:: (fun () → apup "3*(1/2)" "3*(6/12)");
          "(3␣*␣6)␣/␣12)" >:: (fun () → apup "(3/2)" "(3*6)/12") ]
    let suite_complex =
      "complex" >:::
        [ "1+I" >:: (fun () → apup "1+I" "1+complex(0,1)");
          "1-I" >:: (fun () → apup "1-I" "1-complex(0,1)");
          "1-I'" >:: (fun () → apup "1+(-I)" "1+complex(0,-1)");
          "1+I'" >:: (fun () → apup "1-(-I)" "1-complex(0,-1)");
          "1+1.+I" >:: (fun () → apup "1+(1.+I)" "1+complex(1,1)");
          "1+1.-I" >:: (fun () → apup "1+(1.-I)" "1+complex(1,-1)");
          "1-1.-I" >:: (fun () → apup "1-(1.+I)" "1-complex(1,1)");
          "1-1.+I" >:: (fun () → apup "1-(1.-I)" "1-complex(1,-1)");
          "2-I" >:: (fun () → apup "1-(1.+I)" "1-complex(1,1)");
          "-I+1" >:: (fun () → apup "-I+1" "-complex(0,1)+1");
          "1.-I+1" >:: (fun () → apup "(1.-I)+1" "complex(1,-1)+1");
          "1/I" >:: (fun () → apup "1/I" "1/complex(0,1)");
          "1/1" >:: (fun () → apup "1" "1/complex(1,0)");
          "1/(-1)" >:: (fun () → apup "-1" "1/complex(-1,0)");
          "1/(-I)" >:: (fun () → apup "1/(-I)" "1/complex(0,-1)");
          "1/(2*I)" >:: (fun () → apup "1/(2.*I)" "1/complex(0,2)");
          "1/(1+I)" >:: (fun () → apup "1/(1.+I)" "1/complex(1,1)");
          "1/(1-I)" >:: (fun () → apup "1/(1.-I)" "1/complex(1,-1)");
          "I/2" >:: (fun () → apup "I/2" "complex(0,1)/2");
          "1/2" >:: (fun () → apup "(1/2)" "complex(1,0)/2");
          "-1/2" >:: (fun () → apup "(-1/2)" "complex(-1,0)/2");
          "-I/2" >:: (fun () → apup "(-I)/2" "complex(0,-1)/2");
          "(2␣*␣I)␣/␣2" >:: (fun () → apup "(2.*I)/2" "complex(0,2)/2");
          "(1␣+␣I)␣/␣2" >:: (fun () → apup "(1.+I)/2" "complex(1,1)/2");
          "(1␣-␣I)␣/␣2" >:: (fun () → apup "(1.-I)/2" "complex(1,-1)/2") ]
    let suite_product =
      "product" >:::
        [ "(-a)␣*␣(-b)" >:: (fun () → apup "a*b" "(-a)*(-b)");
          "a␣*␣(-2*b)" >:: (fun () → apup "-2*a*b" "a*(-2*b)");
          "a␣*␣(-2/3*b)" >:: (fun () → apup "a*(-2/3)*b" "a*(-2/3*b)");
          "(-2*a)␣*␣(-2*b)" >:: (fun () → apup "4*a*b" "(-2*a)*(-2*b)") ]

    let suite_power =
      "power" >:::
        [ "a^b^c^d" >:: (fun () → apup "a^(b^(c^d))" "a**b**c**d");
          "(a^b)^c^d" >:: (fun () → apup "(a^b)^(c^d)" "(a**b)**c**d");
          "(a^b)^(c^d)" >:: (fun () → apup "(a^b)^(c^d)" "(a**b)**(c**d)");
          "((a^b)^c)^d" >:: (fun () → apup "((a^b)^c)^d" "((a**b)**c)**d") ]

    let suite_apply =
      "apply" >:::
        [ "sin(x)␣*␣cos(x)**2" >:: (fun () → apup "sin(x)*(cos(x))^2" "cmath.sin(x)*cmath.cos(x)**2");
          "sin(x)␣/␣cos(x)**2" >:: (fun () → apup "sin(x)/(cos(x))^2" "cmath.sin(x)/cmath.cos(x)**2");
          "(sin(x)␣/␣cos(x))**2" >:: (fun () → apup "(sin(x)/cos(x))^2" "(cmath.sin(x)/cmath.cos(x))**2") ]

    let suite_expr =
      "unparse/parse" >:::
        [ "a␣+␣b" >:: (fun () → apup_id "a+b");
          "a␣-␣b" >:: (fun () → apup_id "a-b");
          "a␣+␣b␣-␣c" >:: (fun () → apup_id "a+b-c");
          "a␣-␣b␣-␣c" >:: (fun () → apup_id "a-b-c");
          "-a␣+␣b␣-␣c" >:: (fun () → apup_id "-a+b-c");
          "-a␣-␣b␣-␣c" >:: (fun () → apup_id "-a-b-c");
          "(a␣-␣b)␣/␣c" >:: (fun () → apup_id "(a-b)/c");
          "(a␣-␣b)␣/␣(c␣+␣d)" >:: (fun () → apup_id "(a-b)/(c+d)");
          "(a␣+␣b␣-␣c)␣/␣d" >:: (fun () → apup_id "(a+b-c)/d");
          "a^b␣/␣c" >:: (fun () → apup "a^b/c" "a**b/c");
          "(a␣*␣b)^c␣/␣d" >:: (fun () → apup "(a*b)^c/d" "(a*b)**c/d");
          "(a␣*␣b)^(c/d)" >:: (fun () → apup "(a*b)^(c/d)" "(a*b)**(c/d)");
          "(a␣/␣b)^c␣/␣d" >:: (fun () → apup "(a/b)^c/d" "(a/b)**c/d");
          "(a␣+␣b)^c␣/␣d" >:: (fun () → apup "(a+b)^c/d" "(a+b)**c/d");
          "(a␣-␣b)^c␣/␣d" >:: (fun () → apup "(a-b)^c/d" "(a-b)**c/d");
          "-a^2" >:: (fun () → apup "-a^2" "-a**2");
          "(-a)^2" >:: (fun () → apup "(-a)^2" "(-a)**2");
          "a-b^2" >:: (fun () → apup "a-b^2" "a-b**2");
          "-a^2␣+␣b␣+␣c" >:: (fun () → apup "-a^2+b+c" "-a**2+b+c");
          "a␣-␣b^2␣+␣c" >:: (fun () → apup "a-b^2+c" "a-b**2+c") ]

    let suite_bugreports =
      "bug␣reports" >:::
        [ "S2HDMIV:lam1" >::
            (fun () →
              apup
                "(Mh1^2*RA1x1^2+Mh2^2*RA2x1^2+Mh3^2*RA3x1^2-musq*SB^2)/(CB^2*vH^2)"
                "(Mh1**2*RA1x1**2␣+␣Mh2**2*RA2x1**2␣+␣Mh3**2*RA3x1**2␣-␣musq*SB**2)/(CB**2*vH**2)");
          "loop_sm:AxialZUp" >::
            (fun () → apup "(3/2)*(-ee*sw)/(6*cw)-(1/2)*cw*ee/(2*sw)" "(3.0/2.0)*(-(ee*sw)/(6.*cw))-(1.0/2
          "loop_sm:AxialZUp'" >:: (fun () → apup "(3/2)*(-ee*sw)/(6*cw)" "(3.0/2.0)*(-(ee*sw)/(6.*cw))");
          "loop_sm:AxialZUp''" >:: (fun () → apup "(3/2)*(-ee)/2" "(3.0/2.0)*(-ee/2)") ]

    let suite =
      "UFOx" >:::
        [suite_arithmetic;
         suite_complex;
         suite_product;
         suite_power;
         suite_apply;
         suite_expr;
         suite_bugreports]

  end
```

19.7 Interface of UFO_syntax

19.7.1 Abstract Syntax

```
exception Syntax_Error of string × Lexing.position × Lexing.position

type name = string list

type string_atom =
  | Macro of name
  | Literal of string

type value =
  | Name of name
  | Integer of int
  | Float of float
  | Fraction of int × int
  | String of string
  | String_Expr of string_atom list
  | Empty_List
  | Name_List of name list
  | Integer_List of int list
  | String_List of string list
  | Young_Tableau of int Young.tableau
  | Order_Dictionary of (string × int) list
  | Coupling_Dictionary of (int × int × name) list
  | Decay_Dictionary of (name list × string) list

type attrib =
  { a_name : string;
    a_value : value }

type declaration =
  { name : string;
    kind : name;
    attribs : attrib list }

type t = declaration list
```

A macro expansion is encoded as a special *declaration*, with *kind* = "$" and a single attribute. There should not never be the risk of a name clash.

```
val macro : string → value → declaration

val to_strings : t → string list
```

19.8 Implementation of UFO_syntax

19.8.1 Abstract Syntax

```
exception Syntax_Error of string × Lexing.position × Lexing.position

type name = string list

type string_atom =
  | Macro of name
  | Literal of string

type value =
  | Name of name
  | Integer of int
  | Float of float
  | Fraction of int × int
  | String of string
  | String_Expr of string_atom list
  | Empty_List
  | Name_List of name list
  | Integer_List of int list
  | String_List of string list
  | Young_Tableau of int Young.tableau
  | Order_Dictionary of (string × int) list
  | Coupling_Dictionary of (int × int × name) list
  | Decay_Dictionary of (name list × string) list

type attrib =
  { a_name : string;
    a_value : value }

type declaration =
  { name : string;
    kind : name;
    attribs : attrib list }

type t = declaration list

let macro name expansion =
  { name;
    kind = ["$"];
    attribs = [ { a_name = name; a_value = expansion } ] }

let to_strings declarations =
  []
```

19.9 Lexer

```
{
open Lexing
open UFO_parser

let string_of_char c =
  String.make 1 c

let init_position fname lexbuf =
  let curr_p = lexbuf.lex_curr_p in
  lexbuf.lex_curr_p ←
    { curr_p with
      pos_fname = fname;
      pos_lnum = 1;
      pos_bol = curr_p.pos_cnum };
  lexbuf

}

let digit = ['0'–'9']
let upper = ['A'–'Z']
let lower = ['a'–'z']
let char = upper | lower
let word = char | digit | '_'
let white = [' ' '\t']
let esc = ['\'' '"' '\\']
let crlf = ['\r' '\n']
let not_crlf = [^'\r' '\n']

rule token = parse
    white { token lexbuf } (* skip blanks *)
  | '#' not_crlf* { token lexbuf } (* skip comments *)
  | crlf { new_line lexbuf; token lexbuf }
  | "from" not_crlf* { token lexbuf } (* skip imports *)
  | "import" not_crlf* { token lexbuf } (* skip imports (for now) *)
  | "try:" not_crlf* { token lexbuf } (* skip imports (for now) *)
  | "except" not_crlf* { token lexbuf } (* skip imports (for now) *)
  | "pass" { token lexbuf } (* skip imports (for now) *)
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '{' { LBRACE }
  | '}' { RBRACE }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | '=' { EQUAL }
  | '+' { PLUS }
  | '-' { MINUS }
  | '/' { DIV }
  | '.' { DOT }
  | ',' { COMMA }
  | ':' { COLON }
  | '-'? ( digit+ '.' digit* | digit* '.' digit+ )
        ( ['E''e'] '-'? digit+ )? as x
                      { FLOAT (float_of_string x) }
  | '-'? digit+ as i { INT (int_of_string i) }
  | char word* as s { ID s }
  | '\\' '[' (word+ as stem) ']' (word* as suffix)
                      { ID (UFO_tools.mathematica_symbol stem suffix) }
  | '\'' { let sbuf = Buffer.create 20 in
                        STRING (string1 sbuf lexbuf) }
  | '"' { let sbuf = Buffer.create 20 in
                        STRING (string2 sbuf lexbuf) }
  | _ as c { raise (UFO_tools.Lexical_Error
                              ("invalid␣character␣`" ^ string_of_char c ^ "'",
                               lexbuf.lex_start_p, lexbuf.lex_curr_p)) }
  | eof { END }
and string1 sbuf = parse
    '\'' { Buffer.contents sbuf }
  | '\\' (esc as c) { Buffer.add_char sbuf c; string1 sbuf lexbuf }
  | eof { raise End_of_file }
  | '\\' '[' (word+ as stem) ']' (word* as suffix)
                      { Buffer.add_string
                          sbuf (UFO_tools.mathematica_symbol stem suffix);
                        string1 sbuf lexbuf }
  | _ as c { Buffer.add_char sbuf c; string1 sbuf lexbuf }
and string2 sbuf = parse
    '"' { Buffer.contents sbuf }
  | '\\' (esc as c) { Buffer.add_char sbuf c; string2 sbuf lexbuf }
  | eof { raise End_of_file }
  | '\\' '[' (word+ as stem) ']' (word* as suffix)
                      { Buffer.add_string
                          sbuf (UFO_tools.mathematica_symbol stem suffix);
                        string2 sbuf lexbuf }
  | _ as c { Buffer.add_char sbuf c; string2 sbuf lexbuf }
```

19.10 Parser

Right recursion is more convenient for constructing the value. Since the lists will always be short, there is no performace or stack size reason for prefering left recursion.

Header

```
module U = UFO_syntax

let parse_error msg =
  raise (UFO_syntax.Syntax_Error
            (msg, symbol_start_pos (), symbol_end_pos ()))
let invalid_parameter_attr () =
  parse_error "invalid␣parameter␣attribute"
```

Token declarations

```
%token < int > INT
%token < float > FLOAT
%token < string > STRING ID
%token DOT COMMA COLON
%token EQUAL PLUS MINUS DIV
%token LPAREN RPAREN
%token LBRACE RBRACE
%token LBRACKET RBRACKET

%token END

%start file
%type < UFO_syntax.t > file
```

Grammar rules

```
file ::=
 | declarations END { $1 }


declarations ::=
 | { [] }
 | declaration declarations { $1 :: $2 }


declaration ::=
 | ID EQUAL name LPAREN RPAREN { { U.name = $1;
                                    U.kind = $3;
                                    U.attribs = [] } }
 | ID EQUAL name LPAREN attributes RPAREN { { U.name = $1;
                                    U.kind = $3;
                                    U.attribs = $5 } }
 | ID EQUAL STRING { U.macro $1 (U.String $3) }
 | ID EQUAL string_expr { U.macro $1 (U.String_Expr $3) }


name ::=
 | ID { [$1] }
 | name DOT ID { $3 :: $1 }


attributes ::=
 | attribute { [$1] }
 | attribute COMMA attributes { $1 :: $3 }


attribute ::=
 | ID EQUAL value { { U.a_name = $1; U.a_value = $3 } }
 | ID EQUAL list { { U.a_name = $1; U.a_value = $3 } }
 | ID EQUAL dictionary { { U.a_name = $1; U.a_value = $3 } }


value ::=
 | INT { U.Integer $1 }
```

```
  | INT DIV INT { U.Fraction ($1, $3) }
  | FLOAT { U.Float $1 }
  | string { U.String $1 }
  | string_expr { U.String_Expr $1 }
  | name { U.Name $1 }
```

```
list ::=
  | LBRACKET RBRACKET { U.Empty_List }
  | LBRACKET names RBRACKET { U.Name_List $2 }
  | LBRACKET strings RBRACKET { U.String_List $2 }
  | LBRACKET integers RBRACKET { U.Integer_List $2 }
  | LBRACKET integer_lists RBRACKET { U.Young_Tableau $2 }
```

```
integer_list ::=
  | LBRACKET RBRACKET { [] }
  | LBRACKET integers RBRACKET { $2 }
```

```
dictionary ::=
  | LBRACE orders RBRACE { U.Order_Dictionary $2 }
  | LBRACE couplings RBRACE { U.Coupling_Dictionary $2 }
  | LBRACE decays RBRACE { U.Decay_Dictionary $2 }
```

```
names ::=
  | name { [$1] }
  | name COMMA names { $1 :: $3 }
```

```
integers ::=
  | INT { [$1] }
  | INT COMMA integers { $1 :: $3 }
```

```
integer_lists ::=
  | integer_list { [$1] }
  | integer_list COMMA integer_lists { $1 :: $3 }
```

We demand that a *U.String_Expr* contains no adjacent literal strings. Instead, they are concatenated already in the parser. Note that a *U.String_Expr* must have at least two elements: singletons are parsed as *U.Name* or *U.String* instead.

```
string_expr ::=
  | literal_string_expr { $1 }
  | macro_string_expr { $1 }
```

```
literal_string_expr ::=
  | string PLUS name { [U.Literal $1; U.Macro $3] }
  | string PLUS macro_string_expr { U.Literal $1 :: $3 }
```

```
macro_string_expr ::=
  | name PLUS string { [U.Macro $1; U.Literal $3] }
  | name PLUS string_expr { U.Macro $1 :: $3 }
```

```
strings ::=
  | string { [$1] }
  | string COMMA strings { $1 :: $3 }
```

```
string ::=
  | STRING { $1 }
```

| *string PLUS STRING* { $1 ^ $3 }

orders ::=
| *order* { [$1] }
| *order COMMA orders* { $1 :: $3 }

order ::=
| *STRING COLON INT* { ($1, $3) }

couplings ::=
| *coupling* { [$1] }
| *coupling COMMA couplings* { $1 :: $3 }

coupling ::=
| *LPAREN INT COMMA INT RPAREN COLON name* { ($2, $4, $7) }

decays ::=
| *decay* { [$1] }
| *decay COMMA decays* { $1 :: $3 }

decay ::=
| *LPAREN names RPAREN COLON STRING* { ($2, $5) }

19.11 Interface of *UFO_Lorentz*

19.11.1 Processed UFO Lorentz Structures

Just like *UFOx.Lorentz_Atom.dirac*, but without the Dirac matrix indices.

```
type dirac = (* private *)
  | Gamma5
  | ProjM
  | ProjP
  | Gamma of int
  | Sigma of int × int
  | C
  | Minus
```

A sandwich of a string of γ-matrices. *bra* and *ket* are positions of fields in the vertex, *not* spinor indices.

```
type dirac_string = (* private *)
  { bra : int;
    ket : int;
    conjugated : bool;
    gammas : dirac list }
```

In the case of Majorana spinors, we have to insert charge conjugation matrices.
$\Gamma \to -\Gamma$:

```
val minus : dirac_string → dirac_string
```

$\Gamma \to C\Gamma$:

```
val cc_times : dirac_string → dirac_string
```

$\Gamma \to -\Gamma C$:

```
val times_minus_cc : dirac_string → dirac_string
```

$\Gamma \to \Gamma^T$:

```
val transpose : dirac_string → dirac_string
```

$\Gamma \to C\Gamma C^{-1}$:

```
val conjugate : dirac_string → dirac_string
```

$\Gamma \to C\Gamma^T C^{-1}$, i. e. the composition of *conjugate* and *transpose*:

```
val conjugate_transpose : dirac_string → dirac_string
```

The Lorentz indices appearing in a term are either negative internal summation indices or positive external polarization indices. Note that the external indices are not really indices, but denote the position of the particle in the vertex.

```
type α term = (* private *)
  { indices : int list;
    atom : α }
```

Split the list of indices into summation and polarization indices.

```
val classify_indices : int list → int list × int list
```

Replace the atom keeping the associated indices.

```
val map_atom : (α → β) → α term → β term
```

A contraction consists of a (possibly empty) product of Dirac strings and a (possibly empty) product of Lorentz tensors with a rational coefficient. The *denominator* is required for the poorly documented propagator extensions. The type *atom linear* is a *list* and an empty list is interpreted as 1.

The *denominator* is a *contraction list* to allow code reuse, though a (*A.scalar list* × *A.scalar list* × *QC.t*) *list* would suffice.

```
type contraction = (* private *)
  { coeff : Algebra.QC.t;
    dirac : dirac_string term list;
    vector : UFOx.Lorentz_Atom.vector term list;
    scalar : UFOx.Lorentz_Atom.scalar list;
    inverse : UFOx.Lorentz_Atom.scalar list;
    denominator : contraction list }
```

A sum of *contraction*s.

```
type t = contraction list
```

Fermion line connections.

```
val fermion_lines : t → Coupling.fermion_lines
```

$\Gamma \to C\Gamma C^{-1}$

```
val charge_conjugate : int × int → t → t
```

parse spins lorentz uses the *spins* to parse the UFO *lorentz* structure as a list of *contraction*s.

```
val parse : ?allow_denominator :bool → Coupling.lorentz list → UFOx.Lorentz.t → t
```

map_indices f lorentz applies the map *f* to the free indices in *lorentz*.

```
val map_indices : (int → int) → t → t
val map_fermion_lines :
  (int → int) → Coupling.fermion_lines → Coupling.fermion_lines
```

Create a readable representation for debugging and documenting generated code.

```
val to_string : t → string
val fermion_lines_to_string : Coupling.fermion_lines → string
```

Punting ...

```
val dummy : t
```

More debugging and documenting.

```
val dirac_string_to_string : dirac_string → string
```

dirac_string_to_matrix substitute ds take a string of γ-matrices *ds*, applies *substitute* to the indices and returns the product as a matrix.

```
val dirac_string_to_matrix : (int → int) → dirac_string → Dirac.Chiral.t

module type Test =
  sig
    val suite : OUnit.test
  end

module Test : Test
```

19.12 Implementation of UFO_Lorentz

19.12.1 Processed UFO Lorentz Structures

```
module Q = Algebra.Q
module QC = Algebra.QC
module A = UFOx.Lorentz_Atom
module D = Dirac.Chiral
```

Take a *A.t list* and return the corresponding pair *A.dirac list* × *A.vector list* × *A.scalar list* × *A.scalar list*, without preserving the order (currently, the order is reversed).

```
let split_atoms atoms =
  List.fold_left
    (fun (d, v, s, i) → function
      | A.Vector v' → (d, v' :: v, s, i)
      | A.Dirac d' → (d' :: d, v, s, i)
      | A.Scalar s' → (d, v, s' :: s, i)
      | A.Inverse i' → (d, v, s, i' :: i))
    ([], [], [], []) atoms
```

Just like *UFOx.Lorentz_Atom.dirac*, but without the Dirac matrix indices.

```
type dirac =
  | Gamma5
  | ProjM
  | ProjP
  | Gamma of int
  | Sigma of int × int
  | C
  | Minus

let map_indices_gamma f = function
  | (Gamma5 | ProjM | ProjP | C | Minus as g) → g
  | Gamma mu → Gamma (f mu)
  | Sigma (mu, nu) → Sigma (f mu, f nu)
```

A sandwich of a string of γ-matrices. *bra* and *ket* are positions of fields in the vertex.

```
type dirac_string =
  { bra : int;
    ket : int;
    conjugated : bool;
    gammas : dirac list }

let map_indices_dirac f d =
  { bra = f d.bra;
    ket = f d.ket;
    conjugated = d.conjugated;
    gammas = List.map (map_indices_gamma f) d.gammas }

let toggle_conjugated ds =
  { ds with conjugated = ¬ ds.conjugated }

let flip_bra_ket ds =
  { ds with bra = ds.ket; ket = ds.bra }
```

The implementation of couplings for Dirac spinors in `omega_spinors` uses `conjspinor_spinor` which is a straightforward positive inner product

$$\texttt{psibar0 * psi1} = \bar\psi_0\psi_1 = \sum_\alpha \bar\psi_{0,\alpha}\psi_{1,\alpha}\,. \tag{19.2}$$

Note that the row spinor $\bar\psi_0$ is the actual argument, it is *not* conjugated and multplied by γ_0! In contrast, JRR's implementation of couplings for Majorana spinors uses `spinor_product` in `omega_bispinors`

$$\texttt{chi0 * chi1} = \chi_0^T C\chi_1 \tag{19.3}$$

with a charge antisymmetric and unitary conjugation matrix: $C^{-1} = C^\dagger$ and $C^T = -C$. This product is obviously antisymmetric:

$$\texttt{chi0 * chi1} = \chi_0^T C\chi_1 = \chi_1^T C^T\chi_0 = -\chi_1^T C\chi_0 = \texttt{- chi1 * chi0}\,. \tag{19.4}$$

In the following, we assume to be in a realization with $C^{-1} = -C$, i. e. $C^2 = -\mathbf{1}$:

```
let inv_C = [Minus; C]
```

In JRR's implementation of Majorana fermions (see page 412), *all* fermion-boson fusions are realized with the `f_`ϕ`f(g,phi,chi)` functions, where $\phi \in \{\mathtt{v}, \mathtt{a}, \ldots\}$. This is different from the original Dirac implementation, where *both* `f_`ϕ`f(g,phi,psi)` and `f_f`ϕ`(g,psibar,phi)` are used. However, the latter plays nicer with the permutations in the UFO version of *fuse*. Therefore, we can attempt to automatically map `f_`ϕ`f(g,phi,chi)` to `f_f`ϕ`(g,chi,phi)` by an appropriate transformation of the γ-matrices involved.
Starting from

$$\texttt{f\_}\phi\texttt{f(g,phi,chi)} = \Gamma_\phi^\mu\chi \tag{19.5}$$

where Γ_ϕ is the contraction of the bosonic field ϕ with the appropriate product of γ-matrices, we obtain a condition on the corresponding matrix $\tilde\Gamma_\phi$ that appears in `f_f`ϕ:

$$\texttt{f\_f}\phi\texttt{(g,chi,phi)} = \chi^T\tilde\Gamma_\phi^\mu = \left((\tilde\Gamma_\phi)^T\chi\right)^T \overset{!}{=} (\Gamma_\phi\chi)^T\,. \tag{19.6}$$

This amounts to requiring $\tilde\Gamma = \Gamma^T$, as one might have expected. Below we will see that this is *not* the correct approach.
In any case, we can use the standard charge conjugation matrix relations

$$\mathbf{1}^T = \mathbf{1} \tag{19.7a}$$
$$\gamma_\mu^T = -C\gamma_\mu C^{-1} \tag{19.7b}$$
$$\sigma_{\mu\nu}^T = C\sigma_{\nu\mu}C^{-1} = -C\sigma_{\mu\nu}C^{-1} \tag{19.7c}$$
$$(\gamma_5\gamma_\mu)^T = \gamma_\mu^T\gamma_5^T = -C\gamma_\mu\gamma_5 C^{-1} = C\gamma_5\gamma_\mu C^{-1} \tag{19.7d}$$
$$\gamma_5^T = C\gamma_5 C^{-1} \tag{19.7e}$$

to perform the transpositions symbolically. For the chiral projectors

$$\gamma_\pm = \mathbf{1} \pm \gamma_5 \tag{19.8}$$

this means[1]

$$\gamma_\pm^T = (\mathbf{1}\pm\gamma_5)^T = C(\mathbf{1}\pm\gamma_5)C^{-1} = C\gamma_\pm C^{-1} \tag{19.9a}$$
$$(\gamma_\mu\gamma_\pm)^T = \gamma_\pm^T\gamma_\mu^T = -C\gamma_\pm\gamma_\mu C^{-1} = -C\gamma_\mu\gamma_\mp C^{-1} \tag{19.9b}$$
$$(\gamma_\mu \pm \gamma_\mu\gamma_5)^T = -C(\gamma_\mu \mp \gamma_\mu\gamma_5)C^{-1} \tag{19.9c}$$

and of course

$$C^T = -C\,. \tag{19.10}$$

The implementation starts from transposing a single factor using (19.7) and (19.9):

```
let transpose1 = function
  | (Gamma5 | ProjM | ProjP as g) → [C; g] @ inv_C
  | (Gamma _ | Sigma (_, _) as g) → [Minus] @ [C; g] @ inv_C
```

[1]The final two equations are two different ways to obtain the same result, of course.

```
  | C → [Minus; C]
  | Minus → [Minus]
```

In general, this will leave more than one *Minus* in the result and we can pull these out:

```
let rec collect_signs_rev (negative, acc) = function
  | [] → (negative, acc)
  | Minus :: g_list → collect_signs_rev (¬ negative, acc) g_list
  | g :: g_list → collect_signs_rev (negative, g :: acc) g_list
```

Also, there will be products CC inside the result, these can be canceled, since we assume $C^2 = -\mathbf{1}$:

```
let rec compress_ccs_rev (negative, acc) = function
  | [] → (negative, acc)
  | C :: C :: g_list → compress_ccs_rev (¬ negative, acc) g_list
  | g :: g_list → compress_ccs_rev (negative, g :: acc) g_list
```

Compose *collect_signs_rev* and *compress_ccs_rev*. The two list reversals will cancel.

```
let compress_signs g_list =
  let negative, g_list_rev = collect_signs_rev (false, []) g_list in
  match compress_ccs_rev (negative, []) g_list_rev with
  | true, g_list → Minus :: g_list
  | false, g_list → g_list
```

Transpose all factors in reverse order and clean up:

```
let transpose d =
  { d with
    gammas = compress_signs (ThoList.rev_flatmap transpose1 d.gammas) }
```

We can also easily flip the sign:

```
let minus d =
  { d with gammas = compress_signs (Minus :: d.gammas) }
```

Also in `omega_spinors`

$$\phi\texttt{\_ff(g,psibar1,psi2)} = \bar\psi_1\Gamma_\phi\psi_2\,, \tag{19.11}$$

while in `omega_bispinors`

$$\phi\texttt{\_ff(g,chi1,chi2)} = \chi_1^T C\Gamma_\phi\chi_2\,. \tag{19.12}$$

The latter has mixed symmetry, depending on the γ-matrices in Γ_ϕ according to (19.7) and (19.9)

$$\phi\texttt{\_ff(g,chi2,chi1)} = \chi_2^T C\Gamma_\phi\chi_1 = \chi_1^T\Gamma_\phi^T C^T\chi_2 = -\chi_1^T\Gamma_\phi^T C\chi_2 = \pm\chi_1^T C\Gamma_\phi C^{-1}C\chi_2 = \pm\chi_1^T C\Gamma_\phi\chi_2\,. \tag{19.13}$$

19.12.2 Testing for Self-Consistency Numerically

In the tests `keystones_omegalib` and `keystones_UFO`, we check that the vertex $\bar\psi_0\Gamma_{\phi_1}\psi_2$ can be expressed in three ways, which must all agree. In the case of `keystones_omegalib`, the equivalences are

$$\texttt{psibar0 * f\_}\phi\texttt{f(g,phi1,psi2)} = \bar\psi_0\Gamma_{\phi_1}\psi_2 \tag{19.14a}$$
$$\texttt{f\_f}\phi\texttt{(g,psibar0,phi1) * psi2} = \bar\psi_0\Gamma_{\phi_1}\psi_2 \tag{19.14b}$$
$$\texttt{phi1 * }\phi\texttt{\_ff(g,psibar0,psi2)} = \bar\psi_0\Gamma_{\phi_1}\psi_2\,. \tag{19.14c}$$

In the case of `keystones_UFO`, we use cyclic permutations to match the use in *UFO_targets*, as described in the table following (19.26)

$$\texttt{psibar0 * f}\phi\texttt{f\_p012(g,phi1,psi2)} = \bar\psi_0\Gamma_{\phi_1}\psi_2 \tag{19.15a}$$
$$\texttt{f}\phi\texttt{f\_p201(g,psibar0,phi1) * psi2} = \bar\psi_0\Gamma_{\phi_1}\psi_2 \tag{19.15b}$$
$$\texttt{phi1 * f}\phi\texttt{f\_p120(g,psi2,psibar0)} = \mathrm{tr}\left(\Gamma_{\phi_1}\psi_2\otimes\bar\psi_0\right) = \bar\psi_0\Gamma_{\phi_1}\psi_2\,. \tag{19.15c}$$

In both cases, there is no ambiguity regarding the position of spinors and conjugate spinors, since the inner product `conjspinor_spinor` is not symmetrical.

Note that, from the point of view of permutations, the notation $\mathrm{tr}(\Gamma\psi'\otimes\bar\psi)$ is more natural than the equivalent $\bar\psi\Gamma\psi'$ that inspired the ϕ`_ff` functions in the `omegalib` more than 20 years ago.

We would like to perform the same tests in `keystones_omegalib_bispinors` and `keystones_UFO_bispinors`, but now we have to be more careful in positioning the Majorana spinors, because we can not rely on the Fortran type system to catch cofusions of `spinor` and `conjspinor` fields. In addition, we must make sure to insert charge conjugation matrices in the proper places [7].

Regarding the tests in `keystones_omegalib_bispinors`, we observe

$$\texttt{chi0 * f\_}\phi\texttt{f(g,phi1,chi2)} = \chi_0^T C\Gamma_{\phi_1}\chi_2 \tag{19.16a}$$

$$\texttt{phi1 * }\phi\texttt{\_ff(g,chi0,chi2)} = \chi_0^T C\Gamma_{\phi_1}\chi_2 \tag{19.16b}$$

and

$$\texttt{chi2 * f\_f}\phi\texttt{(g,chi0,phi1)} = \chi_2^T C(\chi_0^T\tilde\Gamma^\mu_{\phi_1})^T = \chi_2^T C(\tilde\Gamma^\mu_{\phi_1})^T\chi_0 = \chi_2^T C\Gamma_{\phi_1}\chi_0 \tag{19.17a}$$

$$\texttt{phi1 * }\phi\texttt{\_ff(g,chi2,chi0)} = \chi_2^T C\Gamma_{\phi_1}\chi_0\,, \tag{19.17b}$$

while

$$\texttt{f\_f}\phi\texttt{(g,chi0,phi1) * chi2} = \chi_0^T\tilde\Gamma_{\phi_1}C\chi_2 = \chi_0^T\Gamma_{\phi_1}^T C\chi_2 = (\Gamma_{\phi_1}\chi_0)^T C\chi_2 \tag{19.18}$$

is different. JRR solved this problem by abandoning `f_f`ϕ altogether and using ϕ`_ff` only in the form ϕ`_ff(g,chi0,chi2)`. Turning to the tests in `keystones_UFO_bispinors`, it would be convenient to be able to use

$$\texttt{chi0 * f}\phi\texttt{f\_p012(g,phi1,chi2)} = \chi_0^T C\Gamma^{012}_{\phi_1}\chi_2 \tag{19.19a}$$

$$\texttt{f}\phi\texttt{f\_p201(g,chi0,phi1) * chi2} = \chi_0^T\Gamma^{201}_{\phi_1}C\chi_2 \tag{19.19b}$$

$$\texttt{phi1 * f}\phi\texttt{f\_p120(g,chi2,chi0)} = \operatorname{tr}\left(\Gamma^{120}_{\phi_1}\chi_2\otimes\chi_0^T\right) = \chi_0^T\Gamma^{120}_{\phi_1}\chi_2 = \chi_2^T(\Gamma^{120}_{\phi_1})^T\chi_0\,, \tag{19.19c}$$

where $\Gamma^{012} = \Gamma$ is the string of γ-matrices as written in the Lagrangian. Obviously, we should require

$$\Gamma^{120} = C\Gamma^{012} = C\Gamma \tag{19.20}$$

as expected from `omega_bispinors`.

```
let cc_times d =
  { d with gammas = compress_signs (C :: d.gammas) }
```

For Γ^{201} we must require[2]

$$\Gamma^{201}C = C\Gamma^{012} = C\Gamma \tag{19.21}$$

i. e.

$$\Gamma^{201} = C\Gamma C^{-1} \neq \Gamma^T\,. \tag{19.22}$$

```
let conjugate d =
  { d with gammas = compress_signs (C :: d.gammas @ inv_C) }

let conjugate_transpose d =
  conjugate (transpose d)

let times_minus_cc d =
  { d with gammas = compress_signs (d.gammas @ [Minus; C]) }
```

19.12.3 From Dirac Strings to 4 × 4 Matrices

dirac_string bind ds applies the mapping *bind* to the indices of γ_μ and $\sigma_{\mu\nu}$ and multiplies the resulting matrices in order using complex rational arithmetic.

```
module type To_Matrix =
  sig
```

[2]Note that we don't get anything new, if we reverse the scalar product

$$\texttt{chi2 * f}\phi\texttt{f\_p201(g,chi0,phi1)} = \chi_2^T C(\chi_0^T\Gamma^{201}_{\phi_1})^T = \chi_0^T\Gamma^{201}_{\phi_1}C^T\chi_2\,.$$

We would find the condition

$$-\Gamma^{201}C = \Gamma^{201}C^T = C\Gamma$$

i. e. only a sign

$$\Gamma^{201} = -C\Gamma C^{-1} \neq \Gamma^T\,,$$

as was to be expected from the antisymmetry of `spinor_product`, of course.

```
    val dirac_string : (int → int) → dirac_string → D.t
  end

module To_Matrix : To_Matrix =
  struct

    let half = QC.make (Q.make 1 2) Q.null
    let half_i = QC.make Q.null (Q.make 1 2)

    let gamma_L = D.times half (D.sub D.unit D.gamma5)
    let gamma_R = D.times half (D.add D.unit D.gamma5)

    let sigma = Array.make_matrix 4 4 D.null
    let () =
      for mu = 0 to 3 do
        for nu = 0 to 3 do
          sigma.(mu).(nu) ←
            D.times
              half_i
              (D.sub
                 (D.mul D.gamma.(mu) D.gamma.(nu))
                 (D.mul D.gamma.(nu) D.gamma.(mu)))
        done
      done

    let dirac bind_indices = function
      | Gamma5 → D.gamma5
      | ProjM → gamma_L
      | ProjP → gamma_R
      | Gamma (mu) → D.gamma.(bind_indices mu)
      | Sigma (mu, nu) → sigma.(bind_indices mu).(bind_indices nu)
      | C → D.cc
      | Minus → D.neg D.unit

    let dirac_string bind_indices ds =
      D.product (List.map (dirac bind_indices) ds.gammas)

  end

let dirac_string_to_matrix = To_Matrix.dirac_string
```

The Lorentz indices appearing in a term are either negative internal summation indices or positive external polarization indices. Note that the external indices are not really indices, but denote the position of the particle in the vertex.

```
type α term =
  { indices : int list;
    atom : α }

let map_atom f term =
  { term with atom = f term.atom }

let map_term f_index f_atom term =
  { indices = List.map f_index term.indices;
    atom = f_atom term.atom }
```

Return a pair of lists: first the (negative) summation indices, second the (positive) external indices.

```
let classify_indices ilist =
  List.partition
    (fun i →
      if i < 0 then
        true
      else if i > 0 then
        false
      else
        invalid_arg "classify_indices")
    ilist
```

Recursions on this type only stop when we come across an empty *denominator*. In practice, this is no problem (we never construct values that recurse more than once), but it would be cleaner to use polymorphic variants as suggested for *UFOx.Tensor.t*.

```
type contraction =
  { coeff : QC.t;
    dirac : dirac_string term list;
    vector : A.vector term list;
    scalar : A.scalar list;
    inverse : A.scalar list;
    denominator : contraction list }

let fermion_lines_of_contraction contraction =
  List.sort
    compare
    (List.map (fun term → (term.atom.ket, term.atom.bra)) contraction.dirac)

let rec map_indices_contraction f c =
  { coeff = c.coeff;
    dirac = List.map (map_term f (map_indices_dirac f)) c.dirac;
    vector = List.map (map_term f (A.map_indices_vector f)) c.vector;
    scalar = List.map (A.map_indices_scalar f) c.scalar;
    inverse = List.map (A.map_indices_scalar f) c.inverse;
    denominator = List.map (map_indices_contraction f) c.denominator }

type t = contraction list

let dummy =
  []

let rec charge_conjugate_dirac (ket, bra as fermion_line) = function
  | [] → []
  | dirac :: dirac_list →
     if dirac.atom.bra = bra ∧ dirac.atom.ket = ket then
       map_atom toggle_conjugated dirac :: dirac_list
     else
       dirac :: charge_conjugate_dirac fermion_line dirac_list

let charge_conjugate_contraction fermion_line c =
  { c with dirac = charge_conjugate_dirac fermion_line c.dirac }

let charge_conjugate fermion_line l =
  List.map (charge_conjugate_contraction fermion_line) l

let fermion_lines contractions =
  let pairs = List.map fermion_lines_of_contraction contractions in
  match ThoList.uniq (List.sort compare pairs) with
  | [] → invalid_arg "UFO_Lorentz.fermion_lines:␣impossible"
  | [pairs] → pairs
  | _ → invalid_arg "UFO_Lorentz.fermion_lines:␣ambiguous"

let map_indices f contractions =
  List.map (map_indices_contraction f) contractions

let map_fermion_lines f pairs =
  List.map (fun (i, j) → (f i, f j)) pairs

let dirac_of_atom = function
  | A.Identity (_, _) → []
  | A.C (_, _) → [C]
  | A.Gamma5 (_, _) → [Gamma5]
  | A.ProjP (_, _) → [ProjP]
  | A.ProjM (_, _) → [ProjM]
  | A.Gamma (mu, _, _) → [Gamma mu]
  | A.Sigma (mu, nu, _, _) → [Sigma (mu, nu)]

let dirac_indices = function
  | A.Identity (i, j) | A.C (i, j)
  | A.Gamma5 (i, j) | A.ProjP (i, j) | A.ProjM (i, j)
  | A.Gamma (_, i, j) | A.Sigma (_, _, i, j) → (i, j)

let rec scan_for_dirac_string stack = function

  | [] →
      (* We're done with this pass. There must be no leftover atoms on the stack of spinor atoms, but we'll
check this in the calling function. *)
      (None, List.rev stack)

  | atom :: atoms →
      let i, j = dirac_indices atom in
      if i > 0 then
        if j > 0 then
          (* That's an atomic Dirac string. Collect all atoms for further processing. *)
          (Some { bra = i; ket = j; conjugated = false;
                  gammas = dirac_of_atom atom },
           List.rev_append stack atoms)
        else
          (* That's the start of a new Dirac string. Search for the remaining elements, not forgetting matrices
that we might pushed on the stack earlier. *)
          collect_dirac_string
            i j (dirac_of_atom atom) [] (List.rev_append stack atoms)
      else
        (* The interior of a Dirac string. Push it on the stack until we find the start. *)
        scan_for_dirac_string (atom :: stack) atoms
```

Complete the string starting with i and the current summation index j.

```
and collect_dirac_string i j rev_ds stack = function

  | [] →
      (* We have consumed all atoms without finding the end of the string. *)
      invalid_arg "collect_dirac_string:␣open␣string"

  | atom :: atoms →
      let i', j' = dirac_indices atom in
      if i' = j then
        if j' > 0 then
          (* Found the conclusion. Collect all atoms on the stack for further processing. *)
          (Some { bra = i; ket = j'; conjugated = false;
                  gammas = List.rev_append rev_ds (dirac_of_atom atom)},
           List.rev_append stack atoms)
        else
          (* Found the continuation. Pop the stack of open indices, since we're looking for a new one. *)
          collect_dirac_string
            i j' (dirac_of_atom atom @ rev_ds) [] (List.rev_append stack atoms)
      else
        (* Either the start of another Dirac string or a non-matching continuation. Push it on the stack until
we're done with the current one. *)
        collect_dirac_string i j rev_ds (atom :: stack) atoms

let dirac_string_of_dirac_atoms atoms =
  scan_for_dirac_string [] atoms

let rec dirac_strings_of_dirac_atoms' rev_ds atoms =
  match dirac_string_of_dirac_atoms atoms with
  | (None, []) → List.rev rev_ds
  | (None, _) → invalid_arg "dirac_string_of_dirac_atoms:␣leftover␣atoms"
  | (Some ds, atoms) → dirac_strings_of_dirac_atoms' (ds :: rev_ds) atoms

let dirac_strings_of_dirac_atoms atoms =
  dirac_strings_of_dirac_atoms' [] atoms

let indices_of_vector = function
  | A.Epsilon (mu1, mu2, mu3, mu4) → [mu1; mu2; mu3; mu4]
  | A.Metric (mu1, mu2) → [mu1; mu2]
  | A.P (mu, n) →
      if n > 0 then
        [mu]
      else
        invalid_arg "indices_of_vector:␣invalid␣momentum"

let classify_vector atom =
  { indices = indices_of_vector atom;
    atom }

let indices_of_dirac = function
  | Gamma5 | ProjM | ProjP | C | Minus → []
  | Gamma (mu) → [mu]
  | Sigma (mu, nu) → [mu; nu]

let indices_of_dirac_string ds =
  ThoList.flatmap indices_of_dirac ds.gammas

let classify_dirac atom =
  { indices = indices_of_dirac_string atom;
    atom }

let contraction_of_lorentz_atoms denominator (atoms, coeff) =
  let dirac_atoms, vector_atoms, scalar, inverse = split_atoms atoms in
  let dirac =
    List.map classify_dirac (dirac_strings_of_dirac_atoms dirac_atoms)
  and vector =
    List.map classify_vector vector_atoms in
  { coeff; dirac; vector; scalar; inverse; denominator }

type redundancy =
  | Trace of int
  | Replace of int × int

let rec redundant_metric′ rev_atoms = function
  | [] → (None, List.rev rev_atoms)
  | { atom = A.Metric (mu, nu) } as atom :: atoms →
      if mu < 1 then
        if nu = mu then
          (Some (Trace mu), List.rev_append rev_atoms atoms)
        else
          (Some (Replace (mu, nu)), List.rev_append rev_atoms atoms)
      else if nu < 0 then
        (Some (Replace (nu, mu)), List.rev_append rev_atoms atoms)
      else
        redundant_metric′ (atom :: rev_atoms) atoms
  | { atom = (A.Epsilon (_, _, _, _ ) | A.P (_, _) ) } as atom :: atoms →
      redundant_metric′ (atom :: rev_atoms) atoms

let redundant_metric atoms =
  redundant_metric′ [] atoms
```

Substitude any occurance of the index *mu* by the index *nu*:

```
let substitute_index_vector1 mu nu = function
  | A.Epsilon (mu1, mu2, mu3, mu4) as eps →
      if mu = mu1 then
        A.Epsilon (nu, mu2, mu3, mu4)
      else if mu = mu2 then
        A.Epsilon (mu1, nu, mu3, mu4)
      else if mu = mu3 then
        A.Epsilon (mu1, mu2, nu, mu4)
      else if mu = mu4 then
        A.Epsilon (mu1, mu2, mu3, nu)
      else
        eps
  | A.Metric (mu1, mu2) as g →
      if mu = mu1 then
        A.Metric (nu, mu2)
      else if mu = mu2 then
        A.Metric (mu1, nu)
      else
        g
  | A.P (mu1, n) as p →
      if mu = mu1 then
        A.P (nu, n)
      else
        p
```

```
let remove a alist =
  List.filter ((≠) a) alist
```

```
let substitute_index1 mu nu mu1 =
  if mu = mu1 then
    nu
  else
    mu1
```

```
let substitute_index mu nu indices =
  List.map (substitute_index1 mu nu) indices
```

This assumes that *mu* is a summation index and *nu* is a polarization index.

```
let substitute_index_vector mu nu vectors =
  List.map
    (fun v →
      { indices = substitute_index mu nu v.indices;
        atom = substitute_index_vector1 mu nu v.atom })
    vectors
```

Substitude any occurance of the index *mu* by the index *nu*:

```
let substitute_index_dirac1 mu nu = function
  | (Gamma5 | ProjM | ProjP | C | Minus) as g → g
  | Gamma (mu1) as g →
      if mu = mu1 then
        Gamma (nu)
      else
        g
  | Sigma (mu1, mu2) as g →
      if mu = mu1 then
        Sigma (nu, mu2)
      else if mu = mu2 then
        Sigma (mu1, nu)
      else
        g
```

This assumes that *mu* is a summation index and *nu* is a polarization index.

```
let substitute_index_dirac mu nu dirac_strings =
  List.map
    (fun ds →
      { indices = substitute_index mu nu ds.indices;
        atom = { ds.atom with
                 gammas =
                   List.map
                     (substitute_index_dirac1 mu nu)
                     ds.atom.gammas } } )
    dirac_strings
```

```
let trace_metric = QC.make (Q.make 4 1) Q.null
```

FIXME: can this be made typesafe by mapping to a type that *only* contains *P* and *Epsilon*?

```
let rec compress_metrics c =
  match redundant_metric c.vector with
  | None, _ → c
  | Some (Trace mu), vector′ →
     compress_metrics
       { coeff = QC.mul trace_metric c.coeff;
         dirac = c.dirac;
         vector = vector′;
         scalar = c.scalar;
         inverse = c.inverse;
         denominator = c.denominator }
  | Some (Replace (mu, nu)), vector′ →
     compress_metrics
       { coeff = c.coeff;
         dirac = substitute_index_dirac mu nu c.dirac;
         vector = substitute_index_vector mu nu vector′;
         scalar = c.scalar;
         inverse = c.inverse;
         denominator = c.denominator }

let compress_denominator = function
  | [([], q)] as denominator → if QC.is_unit q then [] else denominator
  | denominator → denominator

let parse1 spins denominator atom =
  compress_metrics (contraction_of_lorentz_atoms denominator atom)

let parse ?(allow_denominator =false) spins = function
  | UFOx.Lorentz.Linear l → List.map (parse1 spins []) l
  | UFOx.Lorentz.Ratios r →
     ThoList.flatmap
       (fun (numerator, denominator) →
         match compress_denominator denominator with
         | [] → List.map (parse1 spins []) numerator
         | d →
            if allow_denominator then
              let parsed_denominator =
                List.map
                  (parse1 [Coupling.Scalar; Coupling.Scalar] [])
                  denominator in
              List.map (parse1 spins parsed_denominator) numerator
            else
              invalid_arg
                (Printf.sprintf
                   "UFO_Lorentz.parse:␣denominator␣%s␣in␣%s␣not␣allowed␣here!"
                    (UFOx.Lorentz.to_string (UFOx.Lorentz.Linear d))
                    (UFOx.Lorentz.to_string (UFOx.Lorentz.Ratios r))))
       r

let i2s = UFOx.Index.to_string

let vector_to_string = function
  | A.Epsilon (mu, nu, ka, la) →
     Printf.sprintf "Epsilon(%s,%s,%s,%s)" (i2s mu) (i2s nu) (i2s ka) (i2s la)
  | A.Metric (mu, nu) →
     Printf.sprintf "Metric(%s,%s)" (i2s mu) (i2s nu)
  | A.P (mu, n) →
     Printf.sprintf "P(%s,%d)" (i2s mu) n

let dirac_to_string = function
  | Gamma5 → "g5"
  | ProjM → "(1-g5)/2"
  | ProjP → "(1+g5)/2"
  | Gamma (mu) → Printf.sprintf "g(%s)" (i2s mu)
  | Sigma (mu, nu) → Printf.sprintf "s(%s,%s)" (i2s mu) (i2s nu)
  | C → "C"
  | Minus → "-1"

let dirac_string_to_string ds =
  match ds.gammas with
  | [] → Printf.sprintf "<%s|%s>" (i2s ds.bra) (i2s ds.ket)
  | gammas →
      Printf.sprintf
        "<%s|%s|%s>"
        (i2s ds.bra)
        (String.concat "*" (List.map dirac_to_string gammas))
        (i2s ds.ket)

let scalar_to_string = function
  | A.Mass _ → "m"
  | A.Width _ → "w"
  | A.P2 i → Printf.sprintf "p%d**2" i
  | A.P12 (i, j) → Printf.sprintf "p%d*p%d" i j
  | A.Variable s → s
  | A.Coeff c → UFOx.Value.to_string c

let rec contraction_to_string c =
  String.concat
    "␣*␣"
    (List.concat
        [if QC.is_unit c.coeff then
            []
         else
            [QC.to_string c.coeff];
         List.map (fun ds → dirac_string_to_string ds.atom) c.dirac;
         List.map (fun v → vector_to_string v.atom) c.vector;
         List.map scalar_to_string c.scalar]) ^
    (match c.inverse with
     | [] → ""
     | inverse →
         "␣/␣(" ^ String.concat "*" (List.map scalar_to_string inverse) ^ ")") ^
    (match c.denominator with
     | [] → ""
     | denominator → "␣/␣(" ^ to_string denominator ^ ")")

and to_string contractions =
  String.concat "␣+␣" (List.map contraction_to_string contractions)

let fermion_lines_to_string fermion_lines =
  ThoList.to_string
    (fun (ket, bra) → Printf.sprintf "%s->%s" (i2s ket) (i2s bra))
    fermion_lines

module type Test =
  sig
    val suite : OUnit.test
  end

module Test : Test =
  struct

    open OUnit

    let braket gammas =
      { bra = 11; ket = 22; conjugated = false; gammas }

    let assert_transpose gt g =
      assert_equal ~printer : dirac_string_to_string
        (braket gt) (transpose (braket g))

    let assert_conjugate_transpose gct g =
    assert_equal ~printer : dirac_string_to_string
      (braket gct) (conjugate_transpose (braket g))

let suite_transpose =
  "transpose" >:::

    [ "identity" >::
        (fun () →
          assert_transpose [] []);

      "gamma_mu" >::
        (fun () →
          assert_transpose [C; Gamma 1; C] [Gamma 1]);

      "sigma_munu" >::
        (fun () →
          assert_transpose [C; Sigma (1, 2); C] [Sigma (1, 2)]);

      "gamma_5*gamma_mu" >::
        (fun () →
          assert_transpose
            [C; Gamma 1; Gamma5; C]
            [Gamma5; Gamma 1]);

      "gamma5" >::
        (fun () →
          assert_transpose [Minus; C; Gamma5; C] [Gamma5]);

      "gamma+" >::
        (fun () →
          assert_transpose [Minus; C; ProjP; C] [ProjP]);

      "gamma-" >::
        (fun () →
          assert_transpose [Minus; C; ProjM; C] [ProjM]);

      "gamma_mu*gamma_nu" >::
        (fun () →
          assert_transpose
            [Minus; C; Gamma 2; Gamma 1; C]
            [Gamma 1; Gamma 2]);

      "gamma_mu*gamma_nu*gamma_la" >::
        (fun () →
          assert_transpose
            [C; Gamma 3; Gamma 2; Gamma 1; C]
            [Gamma 1; Gamma 2; Gamma 3]);

      "gamma_mu*gamma+" >::
        (fun () →
          assert_transpose
            [C; ProjP; Gamma 1; C]
            [Gamma 1; ProjP]);

      "gamma_mu*gamma-" >::
        (fun () →
          assert_transpose
            [C; ProjM; Gamma 1; C]
            [Gamma 1; ProjM]) ]

let suite_conjugate_transpose =
  "conjugate_transpose" >:::

    [ "identity" >::
        (fun () →
          assert_conjugate_transpose [] []);

      "gamma_mu" >::
        (fun () →
          assert_conjugate_transpose [Minus; Gamma 1] [Gamma 1]);

      "sigma_munu" >::
        (fun () →
          assert_conjugate_transpose [Minus; Sigma (1, 2)] [Sigma (1,2)]);

      "gamma_mu*gamma5" >::
        (fun () →
          assert_conjugate_transpose
            [Minus; Gamma5; Gamma 1] [Gamma 1; Gamma5]);

      "gamma5" >::
        (fun () →
          assert_conjugate_transpose [Gamma5] [Gamma5]) ]

  let suite =
    "UFO_Lorentz" >:::
      [suite_transpose;
       suite_conjugate_transpose]

end
```

19.13 Interface of UFO

```
val parse_string : string → UFO_syntax.t
val parse_file : string → UFO_syntax.t
```

These are the contents of the Python files after lexical analysis as context-free variable declarations, before any semantic interpretation.

```
module type Files =
  sig

    type t = private
      { particles : UFO_syntax.t;
        couplings : UFO_syntax.t;
        coupling_orders : UFO_syntax.t;
        vertices : UFO_syntax.t;
        lorentz : UFO_syntax.t;
        parameters : UFO_syntax.t;
        propagators : UFO_syntax.t;
        decays : UFO_syntax.t }

    val parse_directory : string → t

  end

type t

exception Unhandled of string
```

If we want we can switch the implementation from type *init* = *string* × *string list* to type *init* = *string* × *flag list* with a structured *flag* type.

```
module Model : Model.Mutable with type init = string × string list

val parse_directory : string → t

module type Fortran_Target =
  sig
```

fuse c v s fl g wfs ps fusion fuses the wavefunctions named *wfs* with momenta named *ps* using the vertex named *v* with legs reordered according to *fusion*. The overall coupling constant named *g* is multiplied by the rational coefficient *c*. The list of spins *s* and the fermion lines *fl* are used for selecting the appropriately transformed version of the vertex *v*.

```
    val fuse :
      Algebra.QC.t → string →
      Coupling.lorentzn → Coupling.fermion_lines →
      string → string list → string list → Coupling.fusen → unit

    val lorentz_module :
      ?only : Sets.String.t → ?name :string →
      ?fortran_module :string → ?parameter_module :string →
      Format_Fortran.formatter → unit → unit

  end

module Targets :
  sig
    module Fortran : Fortran_Target
  end
```

Export some functions for testing:

```
module Propagator_UFO :
  sig
    type t = (* private *)
      { name : string;
        numerator : UFOx.Lorentz.t;
        denominator : UFOx.Lorentz.t }
  end

module Propagator :
  sig
    type t = (* private *)
      { name : string;
        spins : Coupling.lorentz × Coupling.lorentz;
        numerator : UFO_Lorentz.t;
        denominator : UFO_Lorentz.t;
        variables : string list }
    val of_propagator_UFO : ?majorana :bool → Propagator_UFO.t → t
    val transpose : t → t
  end

module type Test =
  sig
    val suite : OUnit.test
  end

module Test : Test
```

19.14 Implementation of UFO

Unfortunately, `ocamlweb` will not typeset all multi character operators nicely. E. g. `f @< g` comes out as f @ $<$ g.

```
let (< * >) f g x =
 f (g x)

let (< ** >) f g x y =
  f (g x y)

module SMap = Map.Make(String)
module SSet = Sets.String

module CMap =
  Map.Make
    (struct
      type t = string
      let compare = ThoString.compare_caseless
    end)
module CSet = Sets.String_Caseless

let error_in_string text start_pos end_pos =
  let i = start_pos.Lexing.pos_cnum
  and j = end_pos.Lexing.pos_cnum in
  String.sub text i (j - i)

let error_in_file name start_pos end_pos =
  Printf.sprintf
    "%s:%d.%d-%d.%d"
    name
    start_pos.Lexing.pos_lnum
    (start_pos.Lexing.pos_cnum - start_pos.Lexing.pos_bol)
    end_pos.Lexing.pos_lnum
    (end_pos.Lexing.pos_cnum - end_pos.Lexing.pos_bol)

let parse_string text =
  try
    UFO_parser.file
      UFO_lexer.token
      (UFO_lexer.init_position "" (Lexing.from_string text))
  with
  | UFO_tools.Lexical_Error (msg, start_pos, end_pos) →
     invalid_arg (Printf.sprintf "lexical␣error␣(%s)␣at:␣‘%s’"
                    msg (error_in_string text start_pos end_pos))
  | UFO_syntax.Syntax_Error (msg, start_pos, end_pos) →
     invalid_arg (Printf.sprintf "syntax␣error␣(%s)␣at:␣‘%s’"
                    msg (error_in_string text start_pos end_pos))
  | Parsing.Parse_error →
     invalid_arg ("parse␣error:␣" ^ text)

exception File_missing of string

let parse_file name =
  let ic =
    try open_in name with
    | Sys_error msg as exc →
       if msg = name ^ ":␣No␣such␣file␣or␣directory" then
         raise (File_missing name)
       else
         raise exc in
  let result =
    begin
      try
        UFO_parser.file
          UFO_lexer.token
          (UFO_lexer.init_position name (Lexing.from_channel ic))
      with
      | UFO_tools.Lexical_Error (msg, start_pos, end_pos) →
         begin
           close_in ic;
           invalid_arg (Printf.sprintf
                          "%s:␣lexical␣error␣(%s)"
                          (error_in_file name start_pos end_pos) msg)
         end
      | UFO_syntax.Syntax_Error (msg, start_pos, end_pos) →
         begin
           close_in ic;
           invalid_arg (Printf.sprintf
                          "%s:␣syntax␣error␣(%s)"
                          (error_in_file name start_pos end_pos) msg)
         end
      | Parsing.Parse_error →
         begin
           close_in ic;
           invalid_arg ("parse␣error:␣" ^ name)
         end
    end in
  close_in ic;
  result
```

These are the contents of the Python files after lexical analysis as context-free variable declarations, before any semantic interpretation.

```
module type Files =
  sig

    type t = private
      { particles : UFO_syntax.t;
        couplings : UFO_syntax.t;
        coupling_orders : UFO_syntax.t;
        vertices : UFO_syntax.t;
        lorentz : UFO_syntax.t;
        parameters : UFO_syntax.t;
        propagators : UFO_syntax.t;
        decays : UFO_syntax.t }

    val parse_directory : string → t

  end

module Files : Files =
  struct

    type t =
      { particles : UFO_syntax.t;
        couplings : UFO_syntax.t;
        coupling_orders : UFO_syntax.t;
        vertices : UFO_syntax.t;
        lorentz : UFO_syntax.t;
        parameters : UFO_syntax.t;
        propagators : UFO_syntax.t;
        decays : UFO_syntax.t }

    let parse_directory dir =
      let filename stem = Filename.concat dir (stem ^ ".py") in
      let parse stem = parse_file (filename stem) in
      let parse_optional stem =
        try parse stem with File_missing _ → [] in
      { particles = parse "particles";
        couplings = parse "couplings";
        coupling_orders = parse_optional "coupling_orders";
        vertices = parse "vertices";
        lorentz = parse "lorentz";
        parameters = parse "parameters";
        propagators = parse_optional "propagators";
        decays = parse_optional "decays" }

  end

let dump_file pfx f =
  List.iter
    (fun s → print_endline (pfx ^ ":␣" ^ s))
    (UFO_syntax.to_strings f)

type charge =
  | Q_Integer of int
  | Q_Fraction of int × int

let charge_to_string = function
  | Q_Integer i → Printf.sprintf "%d" i
  | Q_Fraction (n, d) → Printf.sprintf "%d/%d" n d

module S = UFO_syntax
```

```
let find_attrib name attribs =
  try
    (List.find (fun a → name = a.S.a_name) attribs).S.a_value
  with
  | Not_found → failwith ("UFO.find_attrib:␣\"" ^ name ^ "\"␣not␣found")

let find_attrib name attribs =
  (List.find (fun a → name = a.S.a_name) attribs).S.a_value

let name_to_string ?strip name =
  let stripped =
    begin match strip, List.rev name with
    | Some pfx, head :: tail →
       if pfx = head then
         tail
       else
         failwith ("UFO.name_to_string:␣expected␣prefix␣'" ^ pfx ^
                     "',␣got␣'" ^ head ^ "'")
    | _, name → name
    end in
  String.concat "." stripped

let name_attrib ?strip name attribs =
  match find_attrib name attribs with
  | S.Name n → name_to_string ?strip n
  | _ → invalid_arg ("UFO.name_attrib:␣" ^ name)

let integer_attrib name attribs =
  match find_attrib name attribs with
  | S.Integer i → i
  | _ → invalid_arg ("UFO.integer_attrib:␣" ^ name)

let charge_attrib name attribs =
  match find_attrib name attribs with
  | S.Integer i → Q_Integer i
  | S.Fraction (n, d) → Q_Fraction (n, d)
  | _ → invalid_arg ("UFO.charge_attrib:␣" ^ name)

let string_attrib name attribs =
  match find_attrib name attribs with
  | S.String s → s
  | _ → invalid_arg ("UFO.string_attrib:␣" ^ name)

let string_expr_attrib name attribs =
  match find_attrib name attribs with
  | S.Name n → [S.Macro n]
  | S.String s → [S.Literal s]
  | S.String_Expr e → e
  | _ → invalid_arg ("UFO.string_expr_attrib:␣" ^ name)

let young_tableau_attrib name attribs =
  match find_attrib name attribs with
  | S.Young_Tableau y → y
  | _ → invalid_arg ("UFO.young_tableau_attrib:␣" ^ name)

let boolean_attrib name attribs =
  try
    match ThoString.lowercase (name_attrib name attribs) with
    | "true" → true
    | "false" → false
    | _ → invalid_arg ("UFO.boolean_attrib:␣" ^ name)
  with
  | Not_found → false

type value =
  | Integer of int
  | Fraction of int × int
  | Float of float
  | Expr of UFOx.Expr.t
  | Name of string list

let map_expr f default = function
  | Integer _ | Fraction (_, _) | Float _ | Name _ → default
  | Expr e → f e

let variables = map_expr UFOx.Expr.variables CSet.empty
let functions = map_expr UFOx.Expr.functions CSet.empty

let add_to_set_in_map key element map =
  let set = try CMap.find key map with Not_found → CSet.empty in
  CMap.add key (CSet.add element set) map
```

Add all variables in *value* to the *map* from variables to the names in which they appear, indicating that *name* depends on these variables.

```
let dependency name value map =
  CSet.fold
    (fun variable acc → add_to_set_in_map variable name acc)
    (variables value)
    map

let dependencies name_value_list =
  List.fold_left
    (fun acc (name, value) → dependency name value acc)
    CMap.empty
    name_value_list

let dependency_to_string (variable, appearences) =
  Printf.sprintf
    "%s␣->␣{%s}"
    variable (String.concat ",␣" (CSet.elements appearences))

let dependencies_to_strings map =
  List.map dependency_to_string (CMap.bindings map)

let expr_to_string =
  UFOx.Value.to_string <*> UFOx.Value.of_expr

let value_to_string = function
  | Integer i → Printf.sprintf "%d" i
  | Fraction (n, d) → Printf.sprintf "%d/%d" n d
  | Float x → string_of_float x
  | Expr e → "'" ^ expr_to_string e ^ "'"
  | Name n → name_to_string n

let value_to_expr substitutions = function
  | Integer i → Printf.sprintf "%d" i
  | Fraction (n, d) → Printf.sprintf "%d/%d" n d
  | Float x → string_of_float x
  | Expr e → expr_to_string (substitutions e)
  | Name n → name_to_string n

let value_to_coupling substitutions atom = function
  | Integer i → Coupling.Integer i
  | Fraction (n, d) → Coupling.Quot (Coupling.Integer n, Coupling.Integer d)
  | Float x → Coupling.Float x
  | Expr e →
      UFOx.Value.to_coupling atom (UFOx.Value.of_expr (substitutions e))
  | Name n → failwith "UFO.value_to_coupling:␣Name␣not␣supported␣yet!"

let value_to_numeric = function
  | Integer i → Printf.sprintf "%d" i
  | Fraction (n, d) → Printf.sprintf "%g" (float n /. float d)
  | Float x → Printf.sprintf "%g" x
  | Expr e → invalid_arg ("UFO.value_to_numeric:␣expr␣=␣" ^ (expr_to_string e))
  | Name n → invalid_arg ("UFO.value_to_numeric:␣name␣=␣" ^ name_to_string n)

let value_to_float = function
  | Integer i → float i
  | Fraction (n, d) → float n /. float d
  | Float x → x
  | Expr e → invalid_arg ("UFO.value_to_float:␣string␣=␣" ^ (expr_to_string e))
  | Name n → invalid_arg ("UFO.value_to_float:␣name␣=␣" ^ name_to_string n)

let value_attrib name attribs =
  match find_attrib name attribs with
  | S.Integer i → Integer i
  | S.Fraction (n, d) → Fraction (n, d)
  | S.Float x → Float x
  | S.String s → Expr (UFOx.Expr.of_string s)
  | S.Name n → Name n
  | _ → invalid_arg ("UFO.value_attrib:␣" ^ name)

let string_list_attrib name attribs =
  match find_attrib name attribs with
  | S.String_List l → l
  | _ → invalid_arg ("UFO.string_list_attrib:␣" ^ name)

let name_list_attrib ~strip name attribs =
  match find_attrib name attribs with
  | S.Name_List l → List.map (name_to_string ~strip) l
  | _ → invalid_arg ("UFO.name_list_attrib:␣" ^ name)

let integer_list_attrib name attribs =
  match find_attrib name attribs with
  | S.Integer_List l → l
  | _ → invalid_arg ("UFO.integer_list_attrib:␣" ^ name)

let order_dictionary_attrib name attribs =
  match find_attrib name attribs with
  | S.Order_Dictionary d → d
  | _ → invalid_arg ("UFO.order_dictionary_attrib:␣" ^ name)

let coupling_dictionary_attrib ~strip name attribs =
  match find_attrib name attribs with
  | S.Coupling_Dictionary d →
      List.map (fun (i, j, c) → (i, j, name_to_string ~strip c)) d
  | _ → invalid_arg ("UFO.coupling_dictionary_attrib:␣" ^ name)

let decay_dictionary_attrib name attribs =
  match find_attrib name attribs with
  | S.Decay_Dictionary d →
      List.map (fun (p, w) → (List.map List.hd p, w)) d
  | _ → invalid_arg ("UFO.decay_dictionary_attrib:␣" ^ name)

let required_handler kind symbol attribs query name =
  try
    query name attribs
  with
  | Not_found →
      invalid_arg
        (Printf.sprintf
           "fatal␣UFO␣error:␣mandatory␣attribute␣'%s'␣missing␣for␣%s␣'%s'!"
           name kind symbol)

let optional_handler attribs query name default =
  try
    query name attribs
  with
  | Not_found → default
```

The UFO paper [18] is not clear on the question whether the `name` attribute of an instance must match its Python name. While the examples appear to imply this, there are examples of UFO files in the wild that violate this constraint.

```
let warn_symbol_name file symbol name =
  if name ≠ symbol then
    Printf.eprintf
      "UFO:␣warning:␣symbol␣'%s'␣<>␣name␣'%s'␣in␣%s.py:␣\
 ␣␣␣␣␣␣␣while␣legal␣in␣UFO,␣it␣is␣unusual␣and␣can␣cause␣problems!\n"
      symbol name file

let valid_fortran_id kind name =
  if ¬ (ThoString.valid_fortran_id name) then
    invalid_arg
      (Printf.sprintf
         "fatal␣UFO␣error:␣the␣%s␣'%s'␣is␣not␣a␣valid␣fortran␣id!"
         kind name)

let map_to_alist map =
  SMap.fold (fun key value acc → (key, value) :: acc) map []

let keys map =
  SMap.fold (fun key _ acc → key :: acc) map []

let keys_caseless map =
  CMap.fold (fun key _ acc → key :: acc) map []

let values map =
  SMap.fold (fun _ value acc → value :: acc) map []

module SKey =
  struct
    type t = string
    let hash = Hashtbl.hash
    let equal = (=)
  end
module SHash = Hashtbl.Make (SKey)

module type Particle =
  sig

    type t = private
      { pdg_code : int;
        name : string;
        antiname : string;
        spin : UFOx.Lorentz.r;
        color : UFOx.Color.r;
        mass : string;
        width : string;
        propagator : string option;
        texname : string;
        antitexname : string;
        charge : charge;
        ghost_number : int;
        lepton_number : int;
        y : charge;
        goldstone : bool;
        propagating : bool; (* NOT HANDLED YET! *)
        line : string option; (* NOT HANDLED YET! *)
        is_anti : bool }

    val of_file : S.t → t SMap.t
    val to_string : string → t → string
    val conjugate : t → t
    val map_mass_and_width : (string → string) → t → t
    val force_spinor : t → t
    val force_conjspinor : t → t
    val force_majorana : t → t
    val is_majorana : t → bool
    val is_ghost : t → bool
    val is_goldstone : t → bool
    val is_physical : t → bool
    val filter : (t → bool) → t SMap.t → t SMap.t

  end

module Particle : Particle =
  struct

    type t =
      { pdg_code : int;
        name : string;
        antiname : string;
        spin : UFOx.Lorentz.r;
        color : UFOx.Color.r;
        mass : string;
        width : string;
        propagator : string option;
        texname : string;
        antitexname : string;
        charge : charge;
        ghost_number : int;
        lepton_number : int;
        y : charge;
        goldstone : bool;
        propagating : bool; (* NOT HANDLED YET! *)
        line : string option; (* NOT HANDLED YET! *)
        is_anti : bool }

    let to_string symbol p =
      Printf.sprintf
        "particle:␣%s␣=>␣[pdg␣=␣%d,␣name␣=␣'%s'/'%s',␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣spin␣=␣%s,␣color␣=␣%s,␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣mass␣=␣%s,␣width␣=␣%s,%s␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Q␣=␣%s,␣G␣=␣%d,␣L␣=␣%d,␣Y␣=␣%s,␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣TeX␣=␣'%s'/'%s'%s]"
        symbol p.pdg_code p.name p.antiname
        (UFOx.Lorentz.rep_to_string p.spin)
        (UFOx.Color.rep_to_string p.color)
        p.mass p.width
        (match p.propagator with
         | None → ""
         | Some p → "␣propagator␣=␣" ^ p ^ ",")
        (charge_to_string p.charge)
        p.ghost_number p.lepton_number
        (charge_to_string p.y)
        p.texname p.antitexname
        (if p.goldstone then ",␣GB" else "")

    let conjugate_charge = function
      | Q_Integer i → Q_Integer (−i)
      | Q_Fraction (n, d) → Q_Fraction (−n, d)

    let is_neutral p =
      (p.name = p.antiname)
```

We *must not* mess with *pdg_code* and *color* if the particle is neutral!

```
    let conjugate p =
      if is_neutral p then
        p
    else
      { pdg_code = − p.pdg_code;
        name = p.antiname;
        antiname = p.name;
        spin = UFOx.Lorentz.rep_conjugate p.spin;
        color = UFOx.Color.rep_conjugate p.color;
        mass = p.mass;
        width = p.width;
        propagator = p.propagator;
        texname = p.antitexname;
        antitexname = p.texname;
        charge = conjugate_charge p.charge;
        ghost_number = − p.ghost_number;
        lepton_number = − p.lepton_number;
        y = conjugate_charge p.y;
        goldstone = p.goldstone;
        propagating = p.propagating;
        line = p.line;
        is_anti = ¬ p.is_anti }

  let map_mass_and_width f p =
    { p with mass = f p.mass; width = f p.width }

  let of_file1 map d =
    let symbol = d.S.name in
    match d.S.kind, d.S.attribs with
    | [ "Particle" ], attribs →
        let required query name =
          required_handler "particle" symbol attribs query name
        and optional query name default =
          optional_handler attribs query name default in
        let name = required string_attrib "name"
        and antiname = required string_attrib "antiname" in
        let neutral = (name = antiname) in
        let pdg_code = required integer_attrib "pdg_code" in
        SMap.add symbol
          { (* The required attributes per UFO docs. *)
            pdg_code;
            name; antiname;
            spin =
              UFOx.Lorentz.rep_of_int neutral (required integer_attrib "spin");
            color =
              UFOx.Color.rep_of_int_or_young_tableau neutral
                (try Some (integer_attrib "color" attribs) with _ → None)
                (try Some (young_tableau_attrib "color_young" attribs) with _ → None);
            mass = required (name_attrib ~strip :"Param") "mass";
            width = required (name_attrib ~strip :"Param") "width";
            texname = required string_attrib "texname";
            antitexname = required string_attrib "antitexname";
            charge = required charge_attrib "charge";
            (* The optional attributes per UFO docs. *)
            ghost_number = optional integer_attrib "GhostNumber" 0;
            lepton_number = optional integer_attrib "LeptonNumber" 0;
            y = optional charge_attrib "Y" (Q_Integer 0);
            goldstone = optional boolean_attrib "goldstone" false;
            propagating = optional boolean_attrib "propagating" true;
            line =
              (try Some (name_attrib "line" attribs) with _ → None);
            (* Undocumented extensions. *)
            propagator =
              (try Some (name_attrib ~strip :"Prop" "propagator" attribs) with _ → None);
            (* O'Mega extensions. *)
            (* Instead of “first come is particle” rely on a negative PDG code to identify antiparticles. *)
            is_anti = pdg_code < 0 } map
    | [ "anti"; p ], [] →
        begin
          try
            SMap.add symbol (conjugate (SMap.find p map)) map
          with
          | Not_found →
              invalid_arg
                ("Particle.of_file:␣" ^ p ^ ".anti()␣not␣yet␣defined!")
        end
    | _ → invalid_arg ("Particle.of_file:␣" ^ name_to_string d.S.kind)

  let of_file particles =
    List.fold_left of_file1 SMap.empty particles

  let is_spinor p =
    match UFOx.Lorentz.omega p.spin with
    | Coupling.Spinor | Coupling.ConjSpinor | Coupling.Majorana → true
    | _ → false
```

TODO: this is a bit of a hack: try to expose the type *UFOx.Lorentz_Atom′.r* instead.

```
  let force_spinor p =
    if is_spinor p then
      { p with spin = UFOx.Lorentz.rep_of_int false 2 }
    else
      p

  let force_conjspinor p =
    if is_spinor p then
      { p with spin = UFOx.Lorentz.rep_of_int false (−2) }
    else
      p

  let force_majorana p =
    if is_spinor p then
      { p with spin = UFOx.Lorentz.rep_of_int true 2 }
    else
      p

  let is_majorana p =
    match UFOx.Lorentz.omega p.spin with
    | Coupling.Majorana | Coupling.Vectorspinor | Coupling.Maj_Ghost → true
    | _ → false

  let is_ghost p =
    p.ghost_number ≠ 0

  let is_goldstone p =
    p.goldstone

  let is_physical p =
    ¬ (is_ghost p ∨ is_goldstone p)

  let filter predicate map =
    SMap.filter (fun symbol p → predicate p) map

end

module type UFO_Coupling =
  sig

    type t = private
      { name : string;
        value : UFOx.Expr.t;
        order : (string × int) list }
  val of_file : S.t → t SMap.t
  val to_string : string → t → string

end

module UFO_Coupling : UFO_Coupling =
struct

  type t =
    { name : string;
      value : UFOx.Expr.t;
      order : (string × int) list }

  let order_to_string orders =
    String.concat ",␣"
      (List.map (fun (s, i) → Printf.sprintf "'%s':%d" s i) orders)

  let to_string symbol c =
    Printf.sprintf
      "coupling:␣%s␣=>␣[name␣=␣'%s',␣value␣=␣'%s',␣order␣=␣[%s]]"
      symbol c.name (expr_to_string c.value) (order_to_string c.order)

  let of_file1 map d =
    let symbol = d.S.name in
    match d.S.kind, d.S.attribs with
    | [ "Coupling" ], attribs →
        let required query name =
          required_handler "coupling" symbol attribs query name in
        let name = required string_attrib "name" in
        warn_symbol_name "couplings" symbol name;
        valid_fortran_id "coupling" name;
        SMap.add symbol
          { name;
            value = UFOx.Expr.of_string (required string_attrib "value");
            order = required order_dictionary_attrib "order" } map
    | _ → invalid_arg ("UFO_Coupling.of_file:␣" ^ name_to_string d.S.kind)

  let of_file couplings =
    List.fold_left of_file1 SMap.empty couplings

end

module type Coupling_Order =
sig

  type t = private
    { name : string;
      expansion_order : int;
      hierarchy : int }

  val of_file : S.t → t SMap.t
  val to_string : string → t → string

end

module Coupling_Order : Coupling_Order =
struct

  type t =
    { name : string;
      expansion_order : int;
      hierarchy : int }

  let to_string symbol c =
    Printf.sprintf
      "coupling_order:␣%s␣=>␣[name␣=␣'%s',␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣expansion_order␣=␣'%d',␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣hierarchy␣=␣%d]"
      symbol c.name c.expansion_order c.hierarchy
    let of_file1 map d =
      let symbol = d.S.name in
      match d.S.kind, d.S.attribs with
      | [ "CouplingOrder" ], attribs →
         let required query name =
           required_handler "coupling␣order" symbol attribs query name in
         let name = required string_attrib "name" in
         warn_symbol_name "coupling_orders" symbol name;
         SMap.add symbol
           { name;
             expansion_order = required integer_attrib "expansion_order";
             hierarchy = required integer_attrib "hierarchy" } map
      | _ → invalid_arg ("Coupling_order.of_file:␣" ^ name_to_string d.S.kind)

    let of_file coupling_orders =
      List.fold_left of_file1 SMap.empty coupling_orders
  end

module type Lorentz_UFO =
  sig
```

If the `name` attribute of a `Lorentz` object does *not* match the the name of the object, we need the latter for weeding out unused Lorentz structures (see *Vertex.contains* below). Therefore, we keep it around.

```
    type t = private
      { name : string;
        symbol : string;
        spins : int list;
        structure : UFOx.Lorentz.t }

    val of_file : S.t → t SMap.t
    val to_string : string → t → string

  end

module Lorentz_UFO : Lorentz_UFO =
  struct

    type t =
      { name : string;
        symbol : string;
        spins : int list;
        structure : UFOx.Lorentz.t }

    let to_string symbol l =
      Printf.sprintf
        "lorentz:␣%s␣=>␣[name␣=␣'%s',␣spins␣=␣[%s],␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣structure␣=␣%s]"
        symbol l.name
        (String.concat ",␣" (List.map string_of_int l.spins))
        (UFOx.Lorentz.to_string l.structure)

    let of_file1 map d =
      let symbol = d.S.name in
      match d.S.kind, d.S.attribs with
      | [ "Lorentz" ], attribs →
         let required query name =
           required_handler "lorentz" symbol attribs query name in
         let name = required string_attrib "name" in
         warn_symbol_name "lorentz" symbol name;
         valid_fortran_id "lorentz" symbol;
         SMap.add symbol
           { name;
             symbol;
             spins = required integer_list_attrib "spins";
             structure =
              UFOx.Lorentz.of_string (required string_attrib "structure") } map
      | _ → invalid_arg ("Lorentz.of_file: " ^ name_to_string d.S.kind)

    let of_file lorentz =
      List.fold_left of_file1 SMap.empty lorentz

  end

module type Vertex =
  sig

    type lcc = private (* Lorentz-color-coupling *)
      { lorentz : string;
        color : UFOx.Color.t;
        coupling : string }

    type t = private
      { name : string;
        particles : string array;
        lcc : lcc list }

    val of_file : Particle.t SMap.t → S.t → t SMap.t
    val to_string : string → t → string
    val to_string_expanded :
      Lorentz_UFO.t SMap.t → UFO_Coupling.t SMap.t → t → string
    val contains : Particle.t SMap.t → (Particle.t → bool) → t → bool
    val filter : (t → bool) → t SMap.t → t SMap.t

  end

module Vertex : Vertex =
  struct

    type lcc =
      { lorentz : string;
        color : UFOx.Color.t;
        coupling : string }

    type t =
      { name : string;
        particles : string array;
        lcc : lcc list }

    let to_string symbol c =
      Printf.sprintf
        "vertex: %s => [name = '%s', particles = [%s], \
                          lorentz-color-couplings = [%s]"
        symbol c.name
        (String.concat
           ", " (Array.to_list c.particles))
        (String.concat
           ", "
           (List.map
              (fun lcc →
                 Printf.sprintf
                   "%s * %s * %s"
                   lcc.coupling lcc.lorentz
                   (UFOx.Color.to_string lcc.color))
              c.lcc))

    let to_string_expanded lorentz couplings c =
      let expand_lorentz s =
        try
          UFOx.Lorentz.to_string (SMap.find s lorentz).Lorentz_UFO.structure
        with
        | Not_found → "?" in
      Printf.sprintf
```

```
        "expanded:␣[%s]␣->␣{␣lorentz-color-couplings␣=␣[%s]␣}"
        (String.concat ",␣" (Array.to_list c.particles))
        (String.concat
           ",␣"
           (List.map
              (fun lcc →
                 Printf.sprintf
                   "%s␣*␣%s␣*␣%s"
                   lcc.coupling (expand_lorentz lcc.lorentz)
                   (UFOx.Color.to_string lcc.color))
              c.lcc))

  let contains particles predicate v =
    let p = v.particles in
    let rec contains' i =
      if i < 0 then
        false
      else if predicate (SMap.find p.(i) particles) then
        true
      else
        contains' (pred i) in
    contains' (Array.length p − 1)

  let force_adj_identity1 adj_indices = function
    | UFOx.Color_Atom.Identity (a, b) as atom →
       begin match List.mem a adj_indices, List.mem b adj_indices with
       | true, true → UFOx.Color_Atom.Identity8 (a, b)
       | false, false → atom
       | true, false | false, true →
          invalid_arg "force_adj_identity:␣mixed␣representations!"
       end
    | atom → atom

  let force_adj_identity adj_indices tensor =
    UFOx.Color.map_atoms (force_adj_identity1 adj_indices) tensor

  let find_adj_indices map particles =
    let adj_indices = ref [] in
    Array.iteri
      (fun i p →
         (* We must pattern match against the O'Mega representation, because UFOx.Color.r is abstract.
*)
         match UFOx.Color.omega (SMap.find p map).Particle.color with
         | Color.AdjSUN _ → adj_indices := succ i :: !adj_indices
         | _ → ())
      particles;
    !adj_indices

  let classify_color_indices map particles =
    let fund_indices = ref []
    and conj_indices = ref []
    and adj_indices = ref [] in
    Array.iteri
      (fun i p →
         (* We must pattern match against the O'Mega representation, because UFOx.Color.r is abstract.
*)
         match UFOx.Color.omega (SMap.find p map).Particle.color with
         | Color.SUN n →
            if n > 0 then
              fund_indices := succ i :: !fund_indices
            else if n < 0 then
              conj_indices := succ i :: !conj_indices
            else
              failwith "classify_color_indices:␣SU(0)"
        | Color.AdjSUN n →
            if n ≠ 0 then
              adj_indices := succ i :: !adj_indices
            else
              failwith "classify_color_indices:␣SU(0)"
        | _ → ())
      particles;
    (!fund_indices, !conj_indices, !adj_indices)
```

FIXME: would have expected the opposite order ...

```
  let force_identity1 (fund_indices, conj_indices, adj_indices) = function
    | UFOx.Color_Atom.Identity (a, b) as atom →
        if List.mem a fund_indices then
          begin
            if List.mem b conj_indices then
              UFOx.Color_Atom.Identity (b, a)
            else
              invalid_arg "force_adj_identity:␣mixed␣representations!"
          end
        else if List.mem a conj_indices then
          begin
            if List.mem b fund_indices then
              UFOx.Color_Atom.Identity (a, b)
            else
              invalid_arg "force_adj_identity:␣mixed␣representations!"
          end else if List.mem a adj_indices then begin
            if List.mem b adj_indices then
              UFOx.Color_Atom.Identity8 (a, b)
            else
              invalid_arg "force_adj_identity:␣mixed␣representations!"
          end
        else
          atom
    | atom → atom

  let force_identity indices tensor =
    UFOx.Color.map_atoms (force_identity1 indices) tensor
```

Here we don't have the Lorentz structures available yet. Thus we set *fermion_lines* = [] for now and correct this later.

```
  let of_file1 particle_map map d =
    let symbol = d.S.name in
    match d.S.kind, d.S.attribs with
    | [ "Vertex" ], attribs →
        let required query name =
          required_handler "vertex" symbol attribs query name in
        let name = required string_attrib "name" in
        warn_symbol_name "vertices" symbol name;
        let particles =
          Array.of_list (required (name_list_attrib ~strip:"P") "particles") in
        let color =
          let indices = classify_color_indices particle_map particles in
          Array.of_list
            (List.map
               (force_identity indices <*> UFOx.Color.of_string)
               (required string_list_attrib "color"))
        and lorentz =
          Array.of_list (required (name_list_attrib ~strip:"L") "lorentz")
        and couplings_alist =
          required (coupling_dictionary_attrib ~strip:"C") "couplings" in
        let lcc =
          List.map
```

```
              (fun (i, j, c) →
                 { lorentz = lorentz.(j);
                   color = color.(i);
                   coupling = c })
              couplings_alist in
          SMap.add symbol { name; particles; lcc } map
      | _ → invalid_arg ("Vertex.of_file:␣" ^ name_to_string d.S.kind)

    let of_file particles vertices =
      List.fold_left (of_file1 particles) SMap.empty vertices

    let filter predicate map =
      SMap.filter (fun symbol p → predicate p) map

  end

module type Parameter =
  sig

    type nature = private Internal | External
    type ptype = private Real | Complex

    type t = private
      { name : string;
        nature : nature;
        ptype : ptype;
        value : value;
        texname : string;
        lhablock : string option;
        lhacode : int list option;
        sequence : int }

    val of_file : S.t → t SMap.t
    val to_string : string → t → string

    val missing : string → t

    val map_names : (string → string) → t → t

  end

module Parameter : Parameter =
  struct

    type nature = Internal | External

    let nature_to_string = function
      | Internal → "internal"
      | External → "external"

    let nature_of_string = function
      | "internal" → Internal
      | "external" → External
      | s → invalid_arg ("Parameter.nature_of_string:␣" ^ s)

    type ptype = Real | Complex

    let ptype_to_string = function
      | Real → "real"
      | Complex → "complex"

    let ptype_of_string = function
      | "real" → Real
      | "complex" → Complex
      | s → invalid_arg ("Parameter.ptype_of_string:␣" ^ s)

    type t =
      { name : string;
        nature : nature;
        ptype : ptype;
        value : value;
        texname : string;
        lhablock : string option;
        lhacode : int list option;
        sequence : int }

  let to_string symbol p =
    Printf.sprintf
      "parameter:␣%s␣=>␣[#%d,␣name␣=␣'%s',␣nature␣=␣%s,␣type␣=␣%s,␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣value␣=␣%s,␣texname␣=␣'%s',␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣lhablock␣=␣%s,␣lhacode␣=␣[%s]]"
      symbol p.sequence p.name
      (nature_to_string p.nature)
      (ptype_to_string p.ptype)
      (value_to_string p.value) p.texname
      (match p.lhablock with None → "???" | Some s → s)
      (match p.lhacode with
      | None → ""
      | Some c → String.concat ",␣" (List.map string_of_int c))

  let of_file1 (map, n) d =
    let symbol = d.S.name in
    match d.S.kind, d.S.attribs with
    | [ "Parameter" ], attribs →
      let required query name =
        required_handler "particle" symbol attribs query name in
      let name = required string_attrib "name" in
      warn_symbol_name "parameters" symbol name;
      valid_fortran_id "parameter" name;
      (SMap.add symbol
        { name;
          nature = nature_of_string (required string_attrib "nature");
          ptype = ptype_of_string (required string_attrib "type");
          value = required value_attrib "value";
          texname = required string_attrib "texname";
          lhablock =
            (try Some (string_attrib "lhablock" attribs) with
              Not_found → None);
          lhacode =
            (try Some (integer_list_attrib "lhacode" attribs) with
              Not_found → None);
          sequence = n } map, succ n)
    | _ → invalid_arg ("Parameter.of_file:␣" ^ name_to_string d.S.kind)

  let of_file parameters =
    let map, _ = List.fold_left of_file1 (SMap.empty, 0) parameters in
    map

  let missing name =
    { name;
      nature = External;
      ptype = Real;
      value = Integer 0;
      texname = Printf.sprintf "\\texttt{%s}" name;
      lhablock = None;
      lhacode = None;
      sequence = 0 }
```

If the *Name* has a prefix, apply f only to the last component.

```
  let map_value f = function
    | (Integer _ | Fraction (_, _) | Float _ as v) → v
    | Name n →
      begin match List.rev n with
```

```
          | [] → Name []
          | stem :: prefix → Name (List.rev (f stem :: prefix))
          end
        | Expr e → Expr (UFOx.Expr.map_names f e)

      let map_names f p =
        { p with name = f p.name; value = map_value f p.value }

    end
```

Macros are encoded as a special *S.declaration* with *S.kind* = `"$"`. This is slightly hackish, but general enough and the overhead of a special union type is probably not worth the effort.

```
module type Macro =
  sig
    type t
    val empty : t
```

The domains and codomains are still a bit too much ad hoc, but it does the job.

```
    val define : t → string → S.value → t
    val expand_string : t → string → S.value
    val expand_expr : t → S.string_atom list → string
```

Only for documentation:

```
    val expand_atom : t → S.string_atom → string
  end

module Macro : Macro =
  struct

    type t = S.value SMap.t

    let empty = SMap.empty

    let define macros name expansion =
      SMap.add name expansion macros

    let expand_string macros name =
      SMap.find name macros

    let rec expand_atom macros = function
      | S.Literal s → s
      | S.Macro [name] →
          begin
            try
              begin match SMap.find name macros with
              | S.String s → s
              | S.String_Expr expr → expand_expr macros expr
              | _ → invalid_arg ("expand_atom:␣not␣a␣string:␣" ^ name)
              end
            with
            | Not_found → invalid_arg ("expand_atom:␣not␣found:␣" ^ name)
          end
      | S.Macro [] → invalid_arg "expand_atom:␣empty"
      | S.Macro name →
          invalid_arg ("expand_atom:␣compound␣name:␣" ^ String.concat "." name)

    and expand_expr macros expr =
      String.concat "" (List.map (expand_atom macros) expr)

  end

module type Propagator_UFO =
  sig

    type t = (* private *)
      { name : string;
        numerator : UFOx.Lorentz.t;
        denominator : UFOx.Lorentz.t }
    val of_file : S.t → t SMap.t
    val to_string : string → t → string
  end
module Propagator_UFO : Propagator_UFO =
  struct
    type t =
      { name : string;
        numerator : UFOx.Lorentz.t;
        denominator : UFOx.Lorentz.t }
    let to_string symbol p =
      Printf.sprintf
        "propagator:␣%s␣=>␣[name␣=␣'%s',␣numerator␣=␣'%s',␣\
   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣denominator␣=␣'%s']"
        symbol p.name
        (UFOx.Lorentz.to_string p.numerator)
        (UFOx.Lorentz.to_string p.denominator)
```

The `denominator` attribute is optional and there is a default (cf. `arXiv:1308.1668`)

```
    let default_denominator =
      "P('mu',␣id)␣*␣P('mu',␣id)␣\
   ␣␣␣␣␣␣␣-␣Mass(id)␣*␣Mass(id)␣\
   ␣␣␣␣␣␣␣+␣complex(0,1)␣*␣Mass(id)␣*␣Width(id)"
    let of_string_with_error_correction symbol num_or_den s =
      try
        UFOx.Lorentz.of_string s
      with
      | Invalid_argument msg →
          begin
            let fixed = s ^ ")" in
            try
              let tensor = UFOx.Lorentz.of_string fixed in
              Printf.eprintf
                "UFO.Propagator.of_string:␣added␣missing␣closing␣parenthesis␣\
   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in␣%s␣of␣%s:␣\"%s\"\n"
                num_or_den symbol s;
              tensor
            with
            | Invalid_argument _ →
                invalid_arg
                  (Printf.sprintf
                     "UFO.Propagator.of_string:␣%s␣of␣%s:␣%s␣in␣\"%s\"\n"
                     num_or_den symbol msg fixed)
          end
    let of_file1 (macros, map) d =
      let symbol = d.S.name in
      match d.S.kind, d.S.attribs with
      | [ "Propagator" ], attribs →
          let required query name =
            required_handler "particle" symbol attribs query name
          and optional query name default =
            optional_handler attribs query name default in
         let name = required string_attrib "name" in
          warn_symbol_name "propagators" symbol name;
          let num_string_expr = required string_expr_attrib "numerator"
          and den_string =
            begin match optional find_attrib "denominator"
                                      (S.String default_denominator) with
            | S.String s  →  s
            | S.Name [n]  →
                begin match Macro.expand_string macros n with
                | S.String s  →  s
                | _  →  invalid_arg "Propagator.denominator"
                end
            | _  →  invalid_arg "Propagator.denominator:␣"
            end in
          let num_string  =  Macro.expand_expr macros num_string_expr in
          let numerator  =
            of_string_with_error_correction symbol "numerator" num_string
          and denominator  =
            of_string_with_error_correction symbol "denominator" den_string in
          (macros, SMap.add symbol { name; numerator; denominator } map)
      | [ "$" ], [ macro ]  →
          begin match macro.S.a_value with
          | S.String _ as s  →
              (Macro.define macros symbol s, map);
          | S.String_Expr expr  →
              let expanded  =  S.String (Macro.expand_expr macros expr) in
              (Macro.define macros symbol expanded, map)
          | _  →  invalid_arg ("Propagator:of_file:␣not␣a␣string␣" ^ symbol)
          end
      | [ "$" ], []  →
          invalid_arg ("Propagator:of_file:␣empty␣declaration␣" ^ symbol)
      | [ "$" ], _  →
          invalid_arg ("Propagator:of_file:␣multiple␣declaration␣" ^ symbol)
      | _  →  invalid_arg ("Propagator:of_file:␣" ^ name_to_string d.S.kind)

    let of_file propagators  =
      let _, propagators′  =
        List.fold_left of_file1 (Macro.empty, SMap.empty) propagators in
      propagators′

  end

module type Decay  =
  sig

    type t  =  private
      { name  :  string;
        particle  :  string;
        widths  :  (string list × string) list }

    val of_file  :  S.t  →  t SMap.t
    val to_string  :  string →  t  →  string

  end

module Decay  :  Decay  =
  struct

    type t  =
      { name  :  string;
        particle  :  string;
        widths  :  (string list × string) list }

    let width_to_string ws  =
      String.concat ",␣"
        (List.map
            (fun (ps, w)  →
              "(" ^ String.concat ",␣" ps ^ ")␣->␣'" ^ w ^ "'")
            ws)

    let to_string symbol d  =
      Printf.sprintf
```

```
        "decay:␣%s␣=>␣[name␣=␣'%s',␣particle␣=␣'%s',␣widths␣=␣[%s]]"
        symbol d.name d.particle (width_to_string d.widths)

    let of_file1 map d =
      let symbol = d.S.name in
      match d.S.kind, d.S.attribs with
      | [ "Decay" ], attribs →
          let required query name =
            required_handler "particle" symbol attribs query name in
          let name = required string_attrib "name" in
          warn_symbol_name "decays" symbol name;
          SMap.add symbol
            { name;
              particle = required (name_attrib ~strip:"P") "particle";
              widths = required decay_dictionary_attrib "partial_widths" } map
      | _ → invalid_arg ("Decay.of_file:␣" ^ name_to_string d.S.kind)

    let of_file decays =
      List.fold_left of_file1 SMap.empty decays

  end
```

We can read the spinor representations off the vertices to check for consistency.

Note that we have to conjugate the representations!

```
let collect_spinor_reps_of_vertex particles lorentz v sets =
  List.fold_left
    (fun sets' lcc →
      let l = (SMap.find lcc.Vertex.lorentz lorentz).Lorentz_UFO.structure in
      List.fold_left
        (fun (spinors, conj_spinors as sets'') (i, rep) →
          let p = v.Vertex.particles.(pred i) in
          match UFOx.Lorentz.omega rep with
          | Coupling.ConjSpinor → (SSet.add p spinors, conj_spinors)
          | Coupling.Spinor → (spinors, SSet.add p conj_spinors)
          | _ → sets'')
        sets' (UFOx.Lorentz.classify_indices l))
    sets v.Vertex.lcc

let collect_spinor_reps_of_vertices particles lorentz vertices =
  SMap.fold
    (fun _ v → collect_spinor_reps_of_vertex particles lorentz v)
    vertices (SSet.empty, SSet.empty)

let lorentz_reps_of_vertex particles v =
  ThoList.alist_of_list ~predicate:(¬ <∗> UFOx.Lorentz.rep_trivial) ~offset:1
    (List.map
       (fun p →
         (∗ Why do we need to conjugate??? ∗)
         UFOx.Lorentz.rep_conjugate
           (SMap.find p particles).Particle.spin)
       (Array.to_list v.Vertex.particles))

let rep_compatible rep_vertex rep_particle =
  let open UFOx.Lorentz in
  let open Coupling in
  match omega rep_vertex, omega rep_particle with
  | (Spinor | ConjSpinor), Majorana → true
  | r1, r2 → r1 = r2

let reps_compatible reps_vertex reps_particles =
  List.for_all2
    (fun (iv, rv) (ip, rp) → iv = ip ∧ rep_compatible rv rp)
    reps_vertex reps_particles
```

```
let check_lorentz_reps_of_vertex particles lorentz v =
  let reps_particles =
    List.sort compare (lorentz_reps_of_vertex particles v) in
  List.iter
    (fun lcc →
      let l = (SMap.find lcc.Vertex.lorentz lorentz).Lorentz_UFO.structure in
      let reps_vertex = List.sort compare (UFOx.Lorentz.classify_indices l) in
      if ¬ (reps_compatible reps_vertex reps_particles) then begin
        Printf.eprintf "%s␣<>␣%s␣[%s]\n"
          (UFOx.Index.classes_to_string
              UFOx.Lorentz.rep_to_string reps_particles)
          (UFOx.Index.classes_to_string
              UFOx.Lorentz.rep_to_string reps_vertex)
          v.Vertex.name (* (Vertex.to_string v.Vertex.name v) *);
        (* invalid_arg "check_lorentz_reps_of_vertex" *) ()
      end)
    v.Vertex.lcc

let color_reps_of_vertex particles v =
  ThoList.alist_of_list ~predicate : (¬ <*> UFOx.Color.rep_trivial) ~offset : 1
    (List.map
       (fun p → (SMap.find p particles).Particle.color)
       (Array.to_list v.Vertex.particles))

let check_color_reps_of_vertex particles v =
  let reps_particles =
    List.sort compare (color_reps_of_vertex particles v) in
  List.iter
    (fun lcc →
      let reps_vertex =
        List.sort compare (UFOx.Color.classify_indices lcc.Vertex.color) in
      if reps_vertex ≠ reps_particles then begin
        Printf.eprintf "particles:␣%s\n<>␣vertex:␣%s\n"
          (UFOx.Index.classes_to_string UFOx.Color.rep_to_string reps_particles)
          (UFOx.Index.classes_to_string UFOx.Color.rep_to_string reps_vertex);
        invalid_arg "check_color_reps_of_vertex"
     end)
    v.Vertex.lcc

module P = Permutation.Default

module type Lorentz =
  sig

    type spins = private
      | Unused
      | Unique of Coupling.lorentz array
      | Ambiguous of Coupling.lorentz array SMap.t

    type t = private
      { name : string;
        n : int;
        spins : spins;
        structure : UFO_Lorentz.t;
        fermion_lines : Coupling.fermion_lines;
        variables : string list }

    val required_charge_conjugates : t → t list
    val permute : P.t → t → t

    val of_lorentz_UFO :
      Particle.t SMap.t → Vertex.t SMap.t →
      Lorentz_UFO.t SMap.t → t SMap.t

    val lorentz_to_string : Coupling.lorentz → string
    val to_string : string → t → string
  end

module Lorentz : Lorentz =
  struct

    let rec lorentz_to_string = function
      | Coupling.Scalar → "Scalar"
      | Coupling.Spinor → "Spinor"
      | Coupling.ConjSpinor → "ConjSpinor"
      | Coupling.Majorana → "Majorana"
      | Coupling.Maj_Ghost → "Maj_Ghost"
      | Coupling.Vector → "Vector"
      | Coupling.Massive_Vector → "Massive_Vector"
      | Coupling.Vectorspinor → "Vectorspinor"
      | Coupling.Tensor_1 → "Tensor_1"
      | Coupling.Tensor_2 → "Tensor_2"
      | Coupling.BRS l → "BRS(" ^ lorentz_to_string l ^ ")"
```

Unlike UFO, O'Mega distinguishes bewteen spinors and conjugate spinors. However, we can inspect the particles in the vertices in which a Lorentz structure is used to determine the correct quantum numbers.

Most model files in the real world contain unused Lorentz structures. This is not a problem, we can just ignore them.

```
    type spins =
      | Unused
      | Unique of Coupling.lorentz array
      | Ambiguous of Coupling.lorentz array SMap.t
```

Use *UFO_targets.Fortran.fusion_name* below in order to avoid communication problems. Or even move away from strings alltogether.

```
    type t =
      { name : string;
        n : int;
        spins : spins;
        structure : UFO_Lorentz.t;
        fermion_lines : Coupling.fermion_lines;
        variables : string list }
```

Add one charge conjugated fermion lines.

```
    let charge_conjugate1 l (ket, bra as fermion_line) =
      { name = l.name ^ Printf.sprintf "_c%x%x" ket bra;
        n = l.n;
        spins = l.spins;
        structure = UFO_Lorentz.charge_conjugate fermion_line l.structure;
        fermion_lines = l.fermion_lines;
        variables = l.variables }
```

Add several charge conjugated fermion lines.

```
    let charge_conjugate l fermion_lines =
      List.fold_left charge_conjugate1 l fermion_lines
```

Add all combinations of charge conjugated fermion lines that don't leave the fusion.

```
    let required_charge_conjugates l =
      let saturated_fermion_lines =
        List.filter
          (fun (ket, bra) → ket ≢ 1 ∧ bra ≢ 1)
          l.fermion_lines in
      List.map (charge_conjugate l) (ThoList.power saturated_fermion_lines)

    let permute_spins p = function
      | Unused → Unused
      | Unique s → Unique (P.array p s)
    | Ambiguous map → Ambiguous (SMap.map (P.array p) map)
```

Note that we apply the *inverse* permutation to the indices in order to match the permutation of the particles/spins.

```
  let permute_structure n p (l, f) =
    let permuted = P.array (P.inverse p) (Array.init n succ) in
    let permute_index i =
      if i > 0 then
        UFOx.Index.map_position (fun pos → permuted.(pred pos)) i
      else
        i in
    (UFO_Lorentz.map_indices permute_index l,
     UFO_Lorentz.map_fermion_lines permute_index f)

  let permute p l =
    let structure, fermion_lines =
      permute_structure l.n p (l.structure, l.fermion_lines) in
    { name = l.name ^ "_p" ^ P.to_string (P.inverse p);
      n = l.n;
      spins = permute_spins p l.spins;
      structure;
      fermion_lines;
      variables = l.variables }

  let omega_lorentz_reps n alist =
    let reps = Array.make n Coupling.Scalar in
    List.iter
      (fun (i, rep) → reps.(pred i) ← UFOx.Lorentz.omega rep)
      alist;
    reps

  let contained lorentz vertex =
    List.exists
      (fun lcc1 → lcc1.Vertex.lorentz = lorentz.Lorentz_UFO.symbol)
      vertex.Vertex.lcc
```

Find all vertices in with the Lorentz structure *lorentz* is used and build a map from those vertices to the O'Mega Lorentz representations inferred from UFO's Lorentz structure and the *particles* involved. Then scan the bindings and check that we have inferred the same Lorentz representation from all vertices.

```
  let lorentz_reps_of_structure particles vertices lorentz =
    let uses =
      SMap.fold
        (fun name v acc →
          if contained lorentz v then
            SMap.add
              name
              (omega_lorentz_reps
                 (Array.length v.Vertex.particles)
                 (lorentz_reps_of_vertex particles v)) acc
          else
            acc) vertices SMap.empty in
    let variants =
      ThoList.uniq (List.sort compare (List.map snd (SMap.bindings uses))) in
    match variants with
    | [] → Unused
    | [s] → Unique s
    | _ →
       Printf.eprintf "UFO.Lorentz.lorentz_reps_of_structure:␣AMBIGUOUS!\n";
       List.iter
         (fun variant →
           Printf.eprintf
             "UFO.Lorentz.lorentz_reps_of_structure:␣%s\n"
             (ThoList.to_string lorentz_to_string (Array.to_list variant)))
            variants;
          Ambiguous uses

    let of_lorentz_tensor spins lorentz =
      match spins with
      | Unique s →
          begin
            try
              Some (UFO_Lorentz.parse (Array.to_list s) lorentz)
            with
            | Failure msg →
                begin
                  prerr_endline msg;
                  Some (UFO_Lorentz.dummy)
                end
          end
      | Unused →
          Printf.eprintf
            "UFO.Lorentz:␣stripping␣unused␣structure␣%s\n"
            (UFOx.Lorentz.to_string lorentz);
          None
      | Ambiguous _ → invalid_arg "UFO.Lorentz.of_lorentz_tensor:␣Ambiguous"
```

NB: if the `name` attribute of a `Lorentz` object does *not* match the the name of the object, the former has a better chance to correspond to a valid Fortran name. Therefore we use it.

```
    let of_lorentz_UFO particles vertices lorentz_UFO =
      SMap.fold
        (fun name l acc →
          let spins = lorentz_reps_of_structure particles vertices l in
          match of_lorentz_tensor spins l.Lorentz_UFO.structure with
          | None → acc
          | Some structure →
              SMap.add
                name
                { name = l.Lorentz_UFO.symbol;
                  n = List.length l.Lorentz_UFO.spins;
                  spins;
                  structure;
                  fermion_lines = UFO_Lorentz.fermion_lines structure;
                  variables = UFOx.Lorentz.variables l.Lorentz_UFO.structure }
                acc)
        lorentz_UFO SMap.empty

    let to_string symbol l =
      Printf.sprintf
        "lorentz:␣%s␣=>␣[name␣=␣'%s',␣spins␣=␣%s,␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣structure␣=␣%s,␣fermion_lines␣=␣%s]"
        symbol l.name
        (match l.spins with
         | Unique s →
             "[" ^ String.concat
                     ",␣" (List.map lorentz_to_string (Array.to_list s)) ^ "]"
         | Ambiguous _ → "AMBIGUOUS!"
         | Unused → "UNUSED!")
        (UFO_Lorentz.to_string l.structure)
        (UFO_Lorentz.fermion_lines_to_string l.fermion_lines)

  end
```

According to arxiv:1308:1668, there should not be a factor of i in the numerators of propagators, but the (unused) `propagators.py` in most models violate this rule!

```
let divide_propagators_by_i = ref false
```

```
module type Propagator =
  sig

    type t = (* private *)
      { name : string;
        spins : Coupling.lorentz × Coupling.lorentz;
        numerator : UFO_Lorentz.t;
        denominator : UFO_Lorentz.t;
        variables : string list }

    val of_propagator_UFO : ?majorana :bool → Propagator_UFO.t → t
    val of_propagators_UFO : ?majorana :bool → Propagator_UFO.t SMap.t → t SMap.t

    val transpose : t → t

    val to_string : string → t → string

  end

module Propagator : Propagator =
  struct

    type t = (* private *)
      { name : string;
        spins : Coupling.lorentz × Coupling.lorentz;
        numerator : UFO_Lorentz.t;
        denominator : UFO_Lorentz.t;
        variables : string list }

    let lorentz_rep_at rep_classes i =
      try
        UFOx.Lorentz.omega (List.assoc i rep_classes)
      with
      | Not_found → Coupling.Scalar

    let imaginary = Algebra.QC.make Algebra.Q.null Algebra.Q.unit
    let scalars = [Coupling.Scalar; Coupling.Scalar]
```

If 51 and 52 show up as indices, we must map $(1, 51) \to (1001, 2001)$ and $(2, 52) \to (1002, 2002)$, as per the UFO conventions for Lorentz structures.

This does not work yet, because *UFOx.Lorentz.map_indices* affects also the position argument of *P*, *Mass* and *Width*.

```
    let contains_51_52 tensor =
      List.exists
        (fun (i, _) → i = 51 ∨ i = 52)
        (UFOx.Lorentz.classify_indices tensor)

    let remap_51_52 = function
      | 1 → 1001 | 51 → 2001
      | 2 → 1002 | 52 → 2002
      | i → i

    let canonicalize_51_52 tensor =
      if contains_51_52 tensor then
        UFOx.Lorentz.rename_indices remap_51_52 tensor
      else
        tensor

    let force_majorana = function
      | Coupling.Spinor | Coupling.ConjSpinor → Coupling.Majorana
      | s → s

    let string_list_union l1 l2 =
      Sets.String.elements
        (Sets.String.union
           (Sets.String.of_list l1)
```

```
          (Sets.String.of_list l2))
```

In the current conventions, the factor of i is not included:

```
  let of_propagator_UFO ?(majorana =false) p =
    let numerator = canonicalize_51_52 p.Propagator_UFO.numerator in
    let lorentz_reps = UFOx.Lorentz.classify_indices numerator in
    let spin1 = lorentz_rep_at lorentz_reps 1
    and spin2 = lorentz_rep_at lorentz_reps 2 in
    let numerator_sans_i =
      if !divide_propagators_by_i then
        UFOx.Lorentz.map_coeff (fun q → Algebra.QC.div q imaginary) numerator
      else
        numerator in
    { name = p.Propagator_UFO.name;
      spins =
        if majorana then
          (force_majorana spin1, force_majorana spin2)
        else
          (spin1, spin2);
      numerator =
        UFO_Lorentz.parse ~allow_denominator :true [spin1; spin2] numerator_sans_i;
      denominator = UFO_Lorentz.parse scalars p.Propagator_UFO.denominator;
      variables =
        string_list_union
          (UFOx.Lorentz.variables p.Propagator_UFO.denominator)
          (UFOx.Lorentz.variables numerator_sans_i) }

  let of_propagators_UFO ?majorana propagators_UFO =
    SMap.fold
      (fun name p acc → SMap.add name (of_propagator_UFO ?majorana p) acc)
      propagators_UFO SMap.empty

  let permute12 = function
    | 1 → 2
    | 2 → 1
    | n → n

  let transpose_positions t =
    UFOx.Index.map_position permute12 t

  let transpose p =
    { name = p.name;
      spins = (snd p.spins, fst p.spins);
      numerator = UFO_Lorentz.map_indices transpose_positions p.numerator;
      denominator = p.denominator;
      variables = p.variables }

  let to_string symbol p =
    Printf.sprintf
      "propagator:␣%s␣=>␣[name␣=␣'%s',␣spin␣=␣'(%s,␣%s)',␣numerator/I␣=␣'%s',␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣denominator␣=␣'%s']"
      symbol p.name
      (Lorentz.lorentz_to_string (fst p.spins))
      (Lorentz.lorentz_to_string (snd p.spins))
      (UFO_Lorentz.to_string p.numerator)
      (UFO_Lorentz.to_string p.denominator)

end

type t =
  { particles : Particle.t SMap.t;
    particle_array : Particle.t array; (* for diagnostics *)
    couplings : UFO_Coupling.t SMap.t;
    coupling_orders : Coupling_Order.t SMap.t;
    vertices : Vertex.t SMap.t;
    lorentz_UFO : Lorentz_UFO.t SMap.t;
    lorentz : Lorentz.t SMap.t;
    parameters : Parameter.t SMap.t;
    propagators_UFO : Propagator_UFO.t SMap.t;
    propagators : Propagator.t SMap.t;
    decays : Decay.t SMap.t;
    nc : int }

let use_majorana_spinors = ref false

let fallback_to_majorana_if_necessary particles vertices lorentz_UFO =
  let majoranas =
    SMap.fold
      (fun p particle acc →
        if Particle.is_majorana particle then
          SSet.add p acc
        else
          acc)
      particles SSet.empty in
  let spinors, conj_spinors =
    collect_spinor_reps_of_vertices particles lorentz_UFO vertices in
  let ambiguous =
    SSet.diff (SSet.inter spinors conj_spinors) majoranas in
  let no_majoranas = SSet.is_empty majoranas
  and no_ambiguities = SSet.is_empty ambiguous in
  if no_majoranas ∧ no_ambiguities ∧ ¬ !use_majorana_spinors then
    (SMap.mapi
       (fun p particle →
         if SSet.mem p spinors then
           Particle.force_spinor particle
         else if SSet.mem p conj_spinors then
           Particle.force_conjspinor particle
         else
           particle)
       particles,
     false)
  else
    begin
      if !use_majorana_spinors then
        Printf.eprintf "O'Mega:␣Majorana␣fermions␣requested.\n";
      if ¬ no_majoranas then
        Printf.eprintf "O'Mega:␣found␣Majorana␣fermions!\n";
      if ¬ no_ambiguities then
        Printf.eprintf
          "O'Mega:␣found␣ambiguous␣spinor␣representations␣for␣%s!\n"
          (String.concat ",␣" (SSet.elements ambiguous));
      Printf.eprintf
        "O'Mega:␣falling␣back␣to␣the␣Majorana␣representation␣for␣all␣fermions.\n";
      (SMap.map Particle.force_majorana particles,
       true)
    end

let nc_of_particles particles =
  let nc_set =
    List.fold_left
      (fun nc_set (_, p) →
        match UFOx.Color.omega p.Particle.color with
        | Color.Singlet | Color.YT _ | Color.YTC _ → nc_set
        | Color.SUN nc → Sets.Int.add (abs nc) nc_set
        | Color.AdjSUN nc → Sets.Int.add (abs nc) nc_set)
      Sets.Int.empty (SMap.bindings particles) in
  match Sets.Int.elements nc_set with
  | [] → 0
  | [n] → n
  | nc_list →
      invalid_arg
        ("UFO.Model:␣more␣than␣one␣value␣of␣N_C:␣" ^
           String.concat ",␣" (List.map string_of_int nc_list))

let of_file u =
  let particles = Particle.of_file u.Files.particles in
  let vertices = Vertex.of_file particles u.Files.vertices
  and lorentz_UFO = Lorentz_UFO.of_file u.Files.lorentz
  and propagators_UFO = Propagator_UFO.of_file u.Files.propagators in
  let particles, majorana =
    fallback_to_majorana_if_necessary particles vertices lorentz_UFO in
  let particle_array = Array.of_list (values particles)
  and lorentz = Lorentz.of_lorentz_UFO particles vertices lorentz_UFO
  and propagators = Propagator.of_propagators_UFO ~majorana propagators_UFO in
  let model =
    { particles;
      particle_array;
      couplings = UFO_Coupling.of_file u.Files.couplings;
      coupling_orders = Coupling_Order.of_file u.Files.coupling_orders;
      vertices;
      lorentz_UFO;
      lorentz;
      parameters = Parameter.of_file u.Files.parameters;
      propagators_UFO;
      propagators;
      decays = Decay.of_file u.Files.decays;
      nc = nc_of_particles particles } in
  SMap.iter
    (fun _ v →
      check_color_reps_of_vertex model.particles v;
      check_lorentz_reps_of_vertex model.particles model.lorentz_UFO v)
    model.vertices;
  model

let map_parameter_names f m =
  { m with
    particles = SMap.map (Particle.map_mass_and_width f) m.particles;
    particle_array = Array.map (Particle.map_mass_and_width f) m.particle_array;
    parameters = SMap.map (Parameter.map_names f) m.parameters }

let parse_directory dir =
  of_file (Files.parse_directory dir)

let dump model =
  Printf.printf "NC␣=␣%d\n" model.nc;
  SMap.iter (print_endline <**> Particle.to_string) model.particles;
  SMap.iter (print_endline <**> UFO_Coupling.to_string) model.couplings;
  SMap.iter (print_endline <**> Coupling_Order.to_string) model.coupling_orders;
  (* SMap.iter (print_endline <**> Vertex.to_string) model.vertices; *)
  SMap.iter
    (fun symbol v →
      (print_endline <**> Vertex.to_string) symbol v;
      print_endline
        (Vertex.to_string_expanded model.lorentz_UFO model.couplings v))
    model.vertices;
  SMap.iter (print_endline <**> Lorentz_UFO.to_string) model.lorentz_UFO;
  SMap.iter (print_endline <**> Lorentz.to_string) model.lorentz;
  SMap.iter (print_endline <**> Parameter.to_string) model.parameters;
  SMap.iter (print_endline <**> Propagator_UFO.to_string) model.propagators_UFO;
  SMap.iter (print_endline <**> Propagator.to_string) model.propagators;
  SMap.iter (print_endline <**> Decay.to_string) model.decays;
  SMap.iter
    (fun symbol d →
      List.iter (fun (_, w) → ignore (UFOx.Expr.of_string w)) d.Decay.widths)
    model.decays
exception Unhandled of string
let unhandled s = raise (Unhandled s)
module Model =
  struct
```

NB: we could use type *flavor* = *Particle.t*, but that would be very inefficient, because we will use *flavor* as a key for maps below.

```
    type flavor = int
    type constant = string
    type coupling_order = string
    type gauge = unit
    module M =
      Modeltools.Mutable (struct type f = flavor type g = gauge type c = constant type co = string end)
    let setup = M.setup
    let flavors = M.flavors
    let external_flavors = M.external_flavors
    let lorentz = M.lorentz
    let all_coupling_orders = M.all_coupling_orders
    let coupling_orders = M.coupling_orders
    let coupling_order_to_string co = co
    let color = M.color
    let nc = M.nc
    let propagator = M.propagator
    let width = M.width
    let goldstone = M.goldstone
    let conjugate = M.conjugate
    let fermion = M.fermion
    let vertices = M.vertices
    let fuse2 = M.fuse2
    let fuse3 = M.fuse3
    let fuse = M.fuse
    let max_degree = M.max_degree
    let parameters = M.parameters
    let flavor_of_string = M.flavor_of_string
    let flavor_to_string = M.flavor_to_string
    let flavor_to_TeX = M.flavor_to_TeX
    let flavor_symbol = M.flavor_symbol
    let gauge_symbol = M.gauge_symbol
    let pdg = M.pdg
    let mass_symbol = M.mass_symbol
    let width_symbol = M.width_symbol
    let constant_symbol = M.constant_symbol
    module Ch = M.Ch
    let charges = M.charges
    let rec fermion_of_lorentz = function
      | Coupling.Spinor → 1
      | Coupling.ConjSpinor → − 1
      | Coupling.Majorana → 2
      | Coupling.Maj_Ghost → 2
      | Coupling.Vectorspinor → 1
      | Coupling.Vector | Coupling.Massive_Vector → 0
      | Coupling.Scalar | Coupling.Tensor_1 | Coupling.Tensor_2 → 0
      | Coupling.BRS f → fermion_of_lorentz f
    module Q = Algebra.Q
```

```
module QC = Algebra.QC

let dummy_tensor3 = Coupling.Scalar_Scalar_Scalar 1
let dummy_tensor4 = Coupling.Scalar4 1

let triplet p = (p.(0), p.(1), p.(2))
let quartet p = (p.(0), p.(1), p.(2), p.(3))

let half_times q1 q2 =
  Q.mul (Q.make 1 2) (Q.mul q1 q2)

let name g =
  g.UFO_Coupling.name

let fractional_coupling g r =
  let g = name g in
  match Q.to_ratio r with
  | 0, _ → "0.0_default"
  | 1, 1 → g
  | −1, 1 → Printf.sprintf "(-%s)" g
  | n, 1 → Printf.sprintf "(%d*%s)" n g
  | 1, d → Printf.sprintf "(%s/%d)" g d
  | −1, d → Printf.sprintf "(-%s/%d)" g d
  | n, d → Printf.sprintf "(%d*%s/%d)" n g d

let lorentz_of_symbol model symbol =
  try
    SMap.find symbol model.lorentz
  with
  | Not_found → invalid_arg ("lorentz_of_symbol:␣" ^ symbol)

let lorentz_UFO_of_symbol model symbol =
  try
    SMap.find symbol model.lorentz_UFO
  with
  | Not_found → invalid_arg ("lorentz_UFO_of_symbol:␣" ^ symbol)

let coupling_of_symbol model symbol =
  try
    SMap.find symbol model.couplings
  with
  | Not_found → invalid_arg ("coupling_of_symbol:␣" ^ symbol)

let spin_triplet model name =
  match (lorentz_of_symbol model name).Lorentz.spins with
  | Lorentz.Unique [|s0; s1; s2|] → (s0, s1, s2)
  | Lorentz.Unique _ → invalid_arg "spin_triplet:␣wrong␣number␣of␣spins"
  | Lorentz.Unused → invalid_arg "spin_triplet:␣Unused"
  | Lorentz.Ambiguous _ → invalid_arg "spin_triplet:␣Ambiguous"

let spin_quartet model name =
  match (lorentz_of_symbol model name).Lorentz.spins with
  | Lorentz.Unique [|s0; s1; s2; s3|] → (s0, s1, s2, s3)
  | Lorentz.Unique _ → invalid_arg "spin_quartet:␣wrong␣number␣of␣spins"
  | Lorentz.Unused → invalid_arg "spin_quartet:␣Unused"
  | Lorentz.Ambiguous _ → invalid_arg "spin_quartet:␣Ambiguous"

let spin_multiplet model name =
  match (lorentz_of_symbol model name).Lorentz.spins with
  | Lorentz.Unique sarray → sarray
  | Lorentz.Unused → invalid_arg "spin_multiplet:␣Unused"
  | Lorentz.Ambiguous _ → invalid_arg "spin_multiplet:␣Ambiguous"
```

If we have reason to belive that a δ_{ab}-vertex is an effective $\mathrm{tr}(T_aT_b)$-vertex generated at loop level, like $gg \to H \ldots$ in the SM, we should interpret it as such and use the expression (6.2) from [17].
AFAIK, there is no way to distinguish these cases directly in a UFO file. Instead we rely in a heuristic, in which each massless color octet vector particle or ghost is a gluon and colorless scalars are potential Higgses.

```
let is_massless p =
  match ThoString.uppercase p.Particle.mass with
  | "ZERO" → true
  | _ → false

let is_gluon model f =
  let p = model.particle_array.(f) in
  match UFOx.Color.omega p.Particle.color,
        UFOx.Lorentz.omega p.Particle.spin with
  | Color.AdjSUN _, Coupling.Vector → is_massless p
  | Color.AdjSUN _, Coupling.Scalar →
     if p.Particle.ghost_number ≠ 0 then
       is_massless p
     else
       false
  | _ → false

let is_color_singlet model f =
  let p = model.particle_array.(f) in
  match UFOx.Color.omega p.Particle.color with
  | Color.Singlet → true
  | _ → false

let is_higgs_gluon_vertex model p adjoints =
  if Array.length p > List.length adjoints then
    List.for_all
      (fun (i, p) →
        if List.mem i adjoints then
          is_gluon model p
        else
          is_color_singlet model p)
      (ThoList.enumerate 1 (Array.to_list p))
  else
    false

let delta8_heuristics model p a b =
  if is_higgs_gluon_vertex model p [a; b] then
    Color.Vertex.delta8_loop a b
  else
    Color.Vertex.delta8 a b

let verbatim_higgs_glue = ref false

let yt_to_omega y =
  Young.map pred y

let translate_color_atom model p = function
  | UFOx.Color_Atom.Identity (i, j) → Color.Vertex.delta3 j i
  | UFOx.Color_Atom.Identity8 (a, b) →
     if !verbatim_higgs_glue then
       Color.Vertex.delta8 a b
     else
       delta8_heuristics model p a b
  | UFOx.Color_Atom.Delta (y, a, b) → Color.Vertex.delta_of_tableau (yt_to_omega y) a b
  | UFOx.Color_Atom.T (a, i, j) → Color.Vertex.t a i j
  | UFOx.Color_Atom.TY (y, a, i, j) → Color.Vertex.t_of_tableau (yt_to_omega y) a i j
  | UFOx.Color_Atom.F (a, b, c) → Color.Vertex.f a b c
  | UFOx.Color_Atom.D (a, b, c) → Color.Vertex.d a b c
  | UFOx.Color_Atom.Epsilon (i, j, k) → Color.Vertex.epsilon [i; j; k]
  | UFOx.Color_Atom.EpsilonBar (i, j, k) → Color.Vertex.epsilon_bar [i; j; k]
  | UFOx.Color_Atom.T6 (a, i, j) → Color.Vertex.t6 a i j
  | UFOx.Color_Atom.K6 (i, j, k) → Color.Vertex.k6 i j k
  | UFOx.Color_Atom.K6Bar (i, j, k) → Color.Vertex.k6bar i j k

let translate_color_term model p = function
```

```
  | [], q → Birdtracks.scale q Birdtracks.one
  | [atom], q → Birdtracks.scale q (translate_color_atom model p atom)
  | atoms, q →
      let atoms = List.map (translate_color_atom model p) atoms in
      Birdtracks.scale q (Birdtracks.multiply atoms)

let translate_color model p terms =
  match terms with
  | [] → invalid_arg "translate_color:␣empty"
  | [ term ] → translate_color_term model p term
  | terms → Birdtracks.sum (List.map (translate_color_term model p) terms)

let translate_coupling_1 model p lcc =
  let l = lcc.Vertex.lorentz in
  let s = Array.to_list (spin_multiplet model l)
  and fl = (SMap.find l model.lorentz).Lorentz.fermion_lines
  and c = name (coupling_of_symbol model lcc.Vertex.coupling) in
  match lcc.Vertex.color with
  | UFOx.Color.Linear color →
      let col = translate_color model p color in
      (Array.to_list p, Coupling.UFO (QC.unit, l, s, fl, col), c)
  | UFOx.Color.Ratios _ as color →
      invalid_arg
        ("UFO.Model.translate_coupling:␣invalid␣color␣structure" ^
           UFOx.Color.to_string color)

let translate_coupling model p lcc =
  List.map (translate_coupling_1 model p) lcc

let long_flavors = ref false

module type Lookup =
  sig
    type f = private
      { flavors : flavor list;
        flavor_of_string : string → flavor;
        flavor_of_symbol : string → flavor;
        particle : flavor → Particle.t;
        flavor_symbol : flavor → string;
        conjugate : flavor → flavor }
    type flavor_format =
      | Long
      | Decimal
      | Hexadecimal
    val flavor_format : flavor_format ref
    val of_model : t → f
  end

module Lookup : Lookup =
  struct

    type f =
      { flavors : flavor list;
        flavor_of_string : string → flavor;
        flavor_of_symbol : string → flavor;
        particle : flavor → Particle.t;
        flavor_symbol : flavor → string;
        conjugate : flavor → flavor }

    type flavor_format =
      | Long
      | Decimal
      | Hexadecimal

    let flavor_format = ref Hexadecimal
    let conjugate_of_particle_array particles =
      Array.init
        (Array.length particles)
        (fun i →
          let f' = Particle.conjugate particles.(i) in
          match ThoArray.match_all f' particles with
          | [i'] → i'
          | [] →
             invalid_arg ("no␣charge␣conjugate:␣" ^ f'.Particle.name)
          | _ →
             invalid_arg ("multiple␣charge␣conjugates:␣" ^ f'.Particle.name))

    let invert_flavor_array a =
      let table = SHash.create 37 in
      Array.iteri (fun i s → SHash.add table s i) a;
      (fun name →
        try
          SHash.find table name
        with
        | Not_found → invalid_arg ("not␣found:␣" ^ name))

    let digits base n =
      let rec digits' acc n =
        if n < 1 then
          acc
        else
          digits' (succ acc) (n / base) in
      if n < 0 then
        digits' 1 (−n)
      else if n = 0 then
        1
      else
        digits' 0 n

    let of_model model =
      let particle_array = Array.of_list (values model.particles) in
      let conjugate_array = conjugate_of_particle_array particle_array
      and name_array = Array.map (fun f → f.Particle.name) particle_array
      and symbol_array = Array.of_list (keys model.particles) in
      let flavor_symbol f =
        begin match !flavor_format with
        | Long → symbol_array.(f)
        | Decimal →
           let w = digits 10 (Array.length particle_array − 1) in
           Printf.sprintf "%0*d" w f
        | Hexadecimal →
           let w = digits 16 (Array.length particle_array − 1) in
           Printf.sprintf "%0*X" w f
        end in
      { flavors = ThoList.range 0 (Array.length particle_array − 1);
        flavor_of_string = invert_flavor_array name_array;
        flavor_of_symbol = invert_flavor_array symbol_array;
        particle = Array.get particle_array;
        flavor_symbol = flavor_symbol;
        conjugate = Array.get conjugate_array }

  end
```

We appear to need to conjugate all flavors. Why???

```
let translate_vertices model tables =
  let vn =
    List.fold_left
```

```
        (fun acc v →
          let p = Array.map tables.Lookup.flavor_of_symbol v.Vertex.particles
          and lcc = v.Vertex.lcc in
          let p = Array.map conjugate p in (* FIXME: why? *)
          translate_coupling model p lcc @ acc)
        [] (values model.vertices) in
    ([], [], vn)

let propagator_of_lorentz = function
  | Coupling.Scalar → Coupling.Prop_Scalar
  | Coupling.Spinor → Coupling.Prop_Spinor
  | Coupling.ConjSpinor → Coupling.Prop_ConjSpinor
  | Coupling.Majorana → Coupling.Prop_Majorana
  | Coupling.Maj_Ghost → invalid_arg
      "UFO.Model.propagator_of_lorentz:␣SUSY␣ghosts␣do␣not␣propagate"
  | Coupling.Vector → Coupling.Prop_Feynman
  | Coupling.Massive_Vector → Coupling.Prop_Unitarity
  | Coupling.Tensor_2 → Coupling.Prop_Tensor_2
  | Coupling.Vectorspinor → invalid_arg
      "UFO.Model.propagator_of_lorentz:␣Vectorspinor"
  | Coupling.Tensor_1 → invalid_arg
      "UFO.Model.propagator_of_lorentz:␣Tensor_1"
  | Coupling.BRS _ → invalid_arg
      "UFO.Model.propagator_of_lorentz:␣no␣BRST"

let filter_unphysical model =
  let physical_particles =
    Particle.filter Particle.is_physical model.particles in
  let physical_particle_array =
    Array.of_list (values physical_particles) in
  let physical_vertices =
    Vertex.filter
      (¬ <*> (Vertex.contains model.particles (¬ <*> Particle.is_physical)))
      model.vertices in
  { model with
    particles = physical_particles;
    particle_array = physical_particle_array;
    vertices = physical_vertices }

let whizard_constants =
  SSet.of_list
    [ "ZERO" ]

let filter_constants parameters =
  List.filter
    (fun p →
      ¬ (SSet.mem (ThoString.uppercase p.Parameter.name) whizard_constants))
    parameters

let add_name set parameter =
  CSet.add parameter.Parameter.name set

let hardcoded_parameters =
  CSet.of_list
    ["cmath.pi"]

let missing_parameters input derived couplings =
  let input_parameters =
    List.fold_left add_name hardcoded_parameters input in
  let all_parameters =
    List.fold_left add_name input_parameters derived in
  let derived_dependencies =
    dependencies
      (List.map
          (fun p → (p.Parameter.name, p.Parameter.value))
```

```
             derived) in
    let coupling_dependencies =
      dependencies
        (List.map
           (fun p → (p.UFO_Coupling.name, Expr p.UFO_Coupling.value))
           (values couplings)) in
    let missing_input =
      CMap.filter
        (fun parameter derived_parameters →
          ¬ (CSet.mem parameter all_parameters))
        derived_dependencies
    and missing =
      CMap.filter
        (fun parameter couplings →
          ¬ (CSet.mem parameter all_parameters))
        coupling_dependencies in
    CMap.iter
      (fun parameter derived_parameters →
        Printf.eprintf
          "UFO␣warning:␣undefined␣input␣parameter␣%s␣appears␣in␣derived␣\
␣␣␣␣␣␣␣␣␣␣␣␣parameters␣{%s}:␣will␣be␣added␣to␣the␣list␣of␣input␣parameters!\n"
          parameter (String.concat ";␣" (CSet.elements derived_parameters)))
      missing_input;
    CMap.iter
      (fun parameter couplings →
        Printf.eprintf
          "UFO␣warning:␣undefined␣parameter␣%s␣appears␣in␣couplings␣{%s}:␣\
␣␣␣␣␣␣␣␣␣␣␣␣will␣be␣added␣to␣the␣list␣of␣input␣parameters!\n"
          parameter (String.concat ";␣" (CSet.elements couplings)))
      missing;
    keys_caseless missing_input @ keys_caseless missing

  let classify_parameters model =
    let compare_parameters p1 p2 =
      compare p1.Parameter.sequence p2.Parameter.sequence in
    let input, derived =
      List.fold_left
        (fun (input, derived) p →
          match p.Parameter.nature with
          | Parameter.Internal → (input, p :: derived)
          | Parameter.External →
             begin match p.Parameter.ptype with
             | Parameter.Real → ()
             | Parameter.Complex →
                Printf.eprintf
                  "UFO␣warning:␣invalid␣complex␣declaration␣of␣input␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣parameter␣‘%s’␣ignored!\n"
                  p.Parameter.name
             end;
             (p :: input, derived))
        ([], []) (filter_constants (values model.parameters)) in
    let additional = missing_parameters input derived model.couplings in
    (List.sort compare_parameters input @ List.map Parameter.missing additional,
     List.sort compare_parameters derived)

  let translate_input p =
    (p.Parameter.name, value_to_float p.Parameter.value)

  let alpha_s_half e =
    UFOx.Expr.substitute "aS" (UFOx.Expr.half "aS") e

  let translate_derived p =
    let make_atom s = s in
    let c = make_atom p.Parameter.name
    and v = value_to_coupling alpha_s_half make_atom p.Parameter.value in
    match p.Parameter.ptype with
    | Parameter.Real → (Coupling.Real c, v)
    | Parameter.Complex → (Coupling.Complex c, v)

  let translate_coupling_constant c =
    let make_atom s = s in
    (Coupling.Complex c.UFO_Coupling.name,
     Coupling.Quot (value_to_coupling alpha_s_half make_atom (Expr c.UFO_Coupling.value), Coupling.I))

  module Lowercase_Parameters =
    struct
      type elt = string
      type base = string
      let compare_elt = compare
      let compare_base = compare
      let pi = ThoString.lowercase
    end

  module Lowercase_Bundle = Bundle.Make (Lowercase_Parameters)

  let coupling_names model =
    SMap.fold
      (fun _ c acc → c.UFO_Coupling.name :: acc)
      model.couplings []

  let parameter_names model =
    SMap.fold
      (fun _ c acc → c.Parameter.name :: acc)
      model.parameters []

  let ambiguous_parameters model =
    let all_names =
      List.rev_append (coupling_names model) (parameter_names model) in
    let lc_bundle = Lowercase_Bundle.of_list all_names in
    let lc_set =
      List.fold_left
        (fun acc s → SSet.add s acc)
        SSet.empty (Lowercase_Bundle.base lc_bundle)
    and ambiguities =
      List.filter
        (fun (_, names) → List.length names > 1)
        (Lowercase_Bundle.fibers lc_bundle) in
    (lc_set, ambiguities)

  let disambiguate1 lc_set name =
    let rec disambiguate1' i =
      let name' = Printf.sprintf "%s_%d" name i in
      let lc_name' = ThoString.lowercase name' in
      if SSet.mem lc_name' lc_set then
        disambiguate1' (succ i)
      else
        (SSet.add lc_name' lc_set, name') in
    disambiguate1' 1

  let disambiguate lc_set names =
    let _, replacements =
      List.fold_left
        (fun (lc_set', acc) name →
          let lc_set'', name' = disambiguate1 lc_set' name in
          (lc_set'', SMap.add name name' acc))
        (lc_set, SMap.empty) names in
    replacements

  let omegalib_names =
    ["u"; "ubar"; "v"; "vbar"; "eps"]

  let replacement_map model =
    let lc_set, ambiguities = ambiguous_parameters model in
    let replacement_list =
      disambiguate lc_set (ThoList.flatmap snd ambiguities) in
    SMap.iter
      (Printf.eprintf
         "UFO␣warning:␣case␣sensitive␣parameter␣names:␣renaming␣'%s'␣->␣'%s'\n")
      replacement_list;
    List.fold_left
      (fun acc name → SMap.add name ("UFO_" ^ name) acc)
      replacement_list omegalib_names

  let translated_parameters model =
    let input_parameters, derived_parameters = classify_parameters model
    and couplings = values model.couplings in
    { Coupling.input = List.map translate_input input_parameters;
      Coupling.derived =
        List.map translate_derived derived_parameters @
          List.map translate_coupling_constant couplings;
      Coupling.derived_arrays = [] }
```

UFO requires us to look up the mass parameter to distinguish between massless and massive vectors.
TODO: this is a candidate for another lookup table.

```
  let lorentz_of_particle p =
    match UFOx.Lorentz.omega p.Particle.spin with
    | Coupling.Vector →
       begin match ThoString.uppercase p.Particle.mass with
       | "ZERO" → Coupling.Vector
       | _ → Coupling.Massive_Vector
       end
    | s → s

  type state =
    { directory : string;
      model : t }

  let initialized = ref None

  let is_initialized_from dir =
    match !initialized with
    | None → false
    | Some state → dir = state.directory

  let dump_raw = ref false
```

Using *translated_parameters* only to extract the parameters, without affecting the corresponding changes in the model tables couldn't work! (Cf. https://answers.launchpad.net/whizard/+question/706815 and https://gitlab.tp.nt.uni-siegen.de/whizard/development/-/issues/450)

```
  let map_names map name =
    match SMap.find_opt name map with
    | None → name
    | Some name → name

  type init = string × string list

  let init (dir, flags) =
    if List.mem "dump" flags then
      dump_raw := true;
    let model = filter_unphysical (parse_directory dir) in
    if !dump_raw then
      dump model;
    let replacements = replacement_map model in
    let model = map_parameter_names (map_names replacements) model in
    let parameters = translated_parameters model in
    let tables = Lookup.of_model model in
    let vertices () = translate_vertices model tables in
    let particle f = tables.Lookup.particle f in
    let lorentz f = lorentz_of_particle (particle f) in
    let propagator f =
      let p = particle f in
      match p.Particle.propagator with
      | None → propagator_of_lorentz (lorentz_of_particle p)
      | Some s → Coupling.Prop_UFO s in
    let gauge_symbol () = "?GAUGE?" in
    let constant_symbol s = s in
    let all_coupling_orders () =
      List.map fst (SMap.bindings model.coupling_orders)
    and coupling_orders c =
      (coupling_of_symbol model c).UFO_Coupling.order
    and coupling_order_to_string co = co in
    M.setup
      ~color : (fun f → UFOx.Color.omega (particle f).Particle.color)
      ~nc : (fun () → model.nc)
      ~pdg : (fun f → (particle f).Particle.pdg_code)
      ~lorentz
      ~propagator
      ~width : (fun f → Coupling.Constant)
      ~goldstone : (fun f → None)
      ~conjugate : tables.Lookup.conjugate
      ~fermion : (fun f → fermion_of_lorentz (lorentz f))
      ~vertices
      ~flavors : [("All␣Flavors", tables.Lookup.flavors)]
      ~parameters : (fun () → parameters)
      ~flavor_of_string : tables.Lookup.flavor_of_string
      ~flavor_to_string : (fun f → (particle f).Particle.name)
      ~flavor_to_TeX : (fun f → (particle f).Particle.texname)
      ~flavor_symbol : tables.Lookup.flavor_symbol
      ~gauge_symbol
      ~mass_symbol : (fun f → (particle f).Particle.mass)
      ~width_symbol : (fun f → (particle f).Particle.width)
      ~constant_symbol
      ~all_coupling_orders
      ~coupling_orders
      ~coupling_order_to_string;
    initialized := Some { directory = dir; model = model }

  let ufo_directory = ref Config.default_UFO_dir

  let load () =
    if is_initialized_from !ufo_directory then
      ()
    else
      init (!ufo_directory, [])

  let include_all_fusions = ref false
```

In case of Majorana spinors, also generate all combinations of charge conjugated fermion lines. The naming convention is to append _cnm if the γ-matrices of the fermion line $n \to m$ has been charge conjugated (this could become impractical for too many fermions at a vertex, but shouldn't matter in real life).
Here we alway generate *all* charge conjugations, because we treat *all* fermions as Majorana fermion, if there is at least one Majorana fermion in the model!

```
  let is_majorana = function
    | Coupling.Majorana | Coupling.Vectorspinor | Coupling.Maj_Ghost → true
    | _ → false

  let name_spins_structure spins l =
    (l.Lorentz.name, spins, l.Lorentz.structure)

  let fusions_of_model ?only model =
    let include_fusion =
      match !include_all_fusions, only with
      | true, _
      | false, None → (fun name → true)
      | false, Some names → (fun name → SSet.mem name names)
    in
    SMap.fold
      (fun name l acc →
        if include_fusion name then
          List.fold_left
            (fun acc p →
              let l' = Lorentz.permute p l in
              match l'.Lorentz.spins with
              | Lorentz.Unused → acc
              | Lorentz.Unique spins →
                 if Array.exists is_majorana spins then
                   List.map
                     (name_spins_structure spins)
                     (Lorentz.required_charge_conjugates l')
                   @ acc
                 else
                   name_spins_structure spins l' :: acc
              | Lorentz.Ambiguous _ → failwith "fusions:␣Lorentz.Ambiguous")
            [] (Permutation.Default.cyclic l.Lorentz.n) @ acc
        else
          acc)
      model.lorentz []

  let fusions ?only () =
    match !initialized with
    | None → []
    | Some { model = model } → fusions_of_model ?only model

  let propagators_of_model ?only model =
    let include_propagator =
      match !include_all_fusions, only with
      | true, _
      | false, None → (fun name → true)
      | false, Some names → (fun name → SSet.mem name names)
    in
    SMap.fold
      (fun name p acc →
        if include_propagator name then
          (name, p) :: acc
        else
          acc)
      model.propagators []

  let propagators ?only () =
    match !initialized with
    | None → []
    | Some { model = model } → propagators_of_model ?only model

  let include_hadrons = ref true

  let caveats () = []

  module Whizard : sig val write : out_channel → unit end =
    struct

      let write_header oc dir =
        let open Printf in
        fprintf oc "#␣WHIZARD␣Model␣file␣derived␣from␣UFO␣directory\n";
        fprintf oc "#␣␣␣␣'%s'\n\n" dir;
        List.iter (fun s → fprintf oc "#␣%s\n" s) (M.caveats ());
        fprintf oc "model␣\"%s\"\n\n" (Filename.basename dir)

      let write_input_parameters oc parameters =
        let open Printf in
        let open Parameter in
        fprintf oc "#␣Independent␣(input)␣Parameters\n";
        List.iter
          (fun p →
            fprintf oc
              "parameter␣%s␣=␣%s"
              p.name (value_to_numeric p.value);
            begin match p.lhablock, p.lhacode with
            | None, None → ()
            | Some name, Some (index :: indices) →
                fprintf oc "␣slha_entry␣%s␣%d" name index;
                List.iter (fun i → fprintf oc "␣%d" i) indices
            | Some name, None →
                eprintf "UFO:␣parameter␣%s:␣slhablock␣%s␣without␣slhacode\n" p.name name
            | Some name, Some [] →
                eprintf "UFO:␣parameter␣%s:␣slhablock␣%s␣with␣empty␣slhacode\n" p.name name
            | None, Some _ →
                eprintf "UFO:␣parameter␣%s:␣slhacode␣without␣slhablock\n" p.name
            end;
            fprintf oc "\n")
          parameters;
        fprintf oc "\n"

      let write_derived_parameters oc parameters =
        let open Printf in
        let open Parameter in
        fprintf oc "#␣Dependent␣(derived)␣Parameters\n";
        List.iter
          (fun p →
            fprintf oc
              "derived␣%s␣=␣%s\n"
              p.name (value_to_expr alpha_s_half p.value))
          parameters

      let write_particles oc particles =
        let open Printf in
        let open Particle in
        fprintf oc "#␣Particles\n";
        fprintf oc "#␣NB:␣hypercharge␣assignments␣appear␣to␣be␣unreliable\n";
        fprintf oc "#␣␣␣␣␣therefore␣we␣can't␣infer␣the␣isospin\n";
        fprintf oc "#␣NB:␣parton-,␣gauge-␣&␣handedness␣are␣unavailable\n";
        List.iter
          (fun p →
            if ¬ p.is_anti then begin
                fprintf oc
                  "particle␣\"%s\"␣%d␣###␣parton?␣gauge?␣left?\n"
                  p.name p.pdg_code;
                fprintf oc
                  "␣␣spin␣%s␣charge␣%s␣color␣%s␣###␣isospin?\n"
                  (UFOx.Lorentz.rep_to_string_whizard p.spin)
                  (charge_to_string p.charge)
                  (UFOx.Color.rep_to_string_whizard p.color);
                fprintf oc "␣␣name␣\"%s\"\n" p.name;
                if p.antiname ≠ p.name then
                  fprintf oc "␣␣anti␣\"%s\"\n" p.antiname;
                fprintf oc "␣␣tex_name␣\"%s\"\n" p.texname;
                if p.antiname ≠ p.name then
                  fprintf oc "␣␣tex_anti␣\"%s\"\n" p.antitexname;
                fprintf oc "␣␣mass␣%s␣width␣%s\n\n" p.mass p.width
              end)
          (values particles);
        fprintf oc "\n"

      let write_hadrons oc =
        let open Printf in
        fprintf oc "#␣Hadrons␣(protons␣and␣beam␣remnants)\n";
        fprintf oc "#␣NB:␣these␣are␣NOT␣part␣of␣the␣UFO␣model\n";
        fprintf oc "#␣␣␣␣␣but␣added␣for␣WHIZARD's␣convenience!\n";
        fprintf oc "particle␣PROTON␣2212\n";
        fprintf oc "␣␣spin␣1/2␣␣charge␣1\n";
        fprintf oc "␣␣name␣p␣\"p+\"\n";
        fprintf oc "␣␣anti␣pbar␣\"p-\"\n";
        fprintf oc "particle␣HADRON_REMNANT␣90\n";
        fprintf oc "␣␣name␣hr\n";
        fprintf oc "␣␣tex_name␣\"had_r\"\n";
        fprintf oc "particle␣HADRON_REMNANT_SINGLET␣91\n";
        fprintf oc "␣␣name␣hr1\n";
        fprintf oc "␣␣tex_name␣\"had_r^{(1)}\"\n";
        fprintf oc "particle␣HADRON_REMNANT_TRIPLET␣92\n";
        fprintf oc "␣␣color␣3\n";
        fprintf oc "␣␣name␣hr3\n";
        fprintf oc "␣␣tex_name␣\"had_r^{(3)}\"\n";
        fprintf oc "␣␣anti␣hr3bar\n";
        fprintf oc "␣␣tex_anti␣\"had_r^{(\\bar␣3)}\"\n";
        fprintf oc "particle␣HADRON_REMNANT_OCTET␣93\n";
        fprintf oc "␣␣color␣8\n";
        fprintf oc "␣␣name␣hr8\n";
        fprintf oc "␣␣tex_name␣\"had_r^{(8)}\"\n";
        fprintf oc "\n"

      let vertex_to_string model v =
        String.concat
          "␣"
          (List.map
             (fun s →
                "\"" ^ (SMap.find s model.particles).Particle.name ^ "\"")
             (Array.to_list v.Vertex.particles))

      let write_vertices3 oc model vertices =
        let open Printf in
        fprintf oc "#␣Vertices␣(for␣phasespace␣generation␣only)\n";
        fprintf oc "#␣NB:␣particles␣should␣be␣sorted␣increasing␣in␣mass.\n";
        fprintf oc "#␣␣␣␣␣This␣is␣NOT␣implemented␣yet!\n";
        List.iter
          (fun v →
            if Array.length v.Vertex.particles = 3 then
              fprintf oc "vertex␣%s\n" (vertex_to_string model v))
          (values vertices);
        fprintf oc "\n"

      let write_vertices_higher oc model vertices =
        let open Printf in
        fprintf oc
          "#␣Higher␣Order␣Vertices␣(ignored␣by␣phasespace␣generation)\n";
        List.iter
          (fun v →
            if Array.length v.Vertex.particles ≠ 3 then
              fprintf oc "#␣vertex␣%s\n" (vertex_to_string model v))
          (values vertices);
```

```
        fprintf oc "\n"

      let write_vertices oc model vertices =
        write_vertices3 oc model vertices;
        write_vertices_higher oc model vertices

      let write oc =
        match !initialized with
        | None → failwith "UFO.Whizard.write:␣UFO␣model␣not␣initialized"
        | Some { directory = dir; model = model } →
            let input_parameters, derived_parameters =
              classify_parameters model in
            write_header oc dir;
            write_input_parameters oc input_parameters;
            write_derived_parameters oc derived_parameters;
            write_particles oc model.particles;
            if !include_hadrons then
              write_hadrons oc;
            write_vertices oc model model.vertices;
            exit 0

    end

  let write_whizard = Whizard.write

  let coupling_order_option co =
    let s = M.coupling_order_to_string co in
    ("-order:" ^ s,
     Arg.Int
       (fun n →
         Printf.eprintf "coupling_order(%s)␣=␣%d\n" s n;
         flush stderr (*; M.set_coupling_order co n *) ),
     Printf.sprintf "n␣set␣%s␣coupling␣order␣n␣[>=0]␣(still␣ignored)" s)

  let coupling_order_options () =
    Arg.align (List.map coupling_order_option (all_coupling_orders ()))

  let flavor_list_to_string f_list =
    String.concat "|" (List.map flavor_to_string f_list)

  let all_flavors () =
    try
      ThoList.flatmap snd (external_flavors ())
    with
    | Modeltools.Uninitialized _ → []

  let load_and_update_cmdline () =
    load () (* ; Options.global := !Options.global @ (coupling_order_options ()); Options.usage := "usage:␣" ^ Sys.arg
"␣[options]␣[-scatter|-decay]␣process␣{flavors:␣" ^ flavor_list_to_string (all_flavors ()) ^ "}" *)

 let options =
    Options.create
      [ ("UFO_dir", Arg.String (fun name → ufo_directory := name),
         "dir␣UFO␣model␣directory␣(default:␣" ^ !ufo_directory ^ ")");
        ("Majorana", Arg.Set use_majorana_spinors,
         "␣use␣Majorana␣spinors␣(must␣come␣-before␣exec!)");
        ("divide_propagators_by_i", Arg.Set divide_propagators_by_i,
         "␣divide␣propagators␣by␣I␣(pre␣2013␣FeynRules␣convention)");
        ("verbatim_Hg", Arg.Set verbatim_higgs_glue,
         "␣don't␣correct␣the␣color␣flows␣for␣effective␣Higgs␣Gluon␣couplings");
        ("write_WHIZARD", Arg.Unit (fun () → Whizard.write stdout),
         "␣write␣the␣WHIZARD␣model␣file␣(required␣once␣per␣model)");
        ("long_flavors",
         Arg.Unit (fun () → Lookup.flavor_format := Lookup.Long),
         "␣write␣use␣the␣UFO␣flavor␣names␣instead␣of␣integers");
        ("dump", Arg.Set dump_raw,
         "␣dump␣UFO␣model␣for␣debugging␣the␣parser␣(must␣come␣-before-␣exec!)");
        ("all_fusions", Arg.Set include_all_fusions,
         "␣include␣all␣fusions␣in␣the␣fortran␣module");
        ("no_hadrons", Arg.Clear include_hadrons,
         "␣don't␣add␣any␣particle␣not␣in␣the␣UFO␣file");
        ("add_hadrons", Arg.Set include_hadrons,
         "␣add␣protons␣and␣beam␣remants␣for␣WHIZARD");
        ("exec", Arg.Unit load_and_update_cmdline,
         "␣load␣the␣UFO␣model␣files␣(required␣-before-␣using␣particles␣names)");
        ("help", Arg.Unit (fun () → prerr_endline "..."),
         "␣print␣information␣on␣the␣model")]

 end

module type Fortran_Target =
 sig

  val fuse :
    Algebra.QC.t → string →
    Coupling.lorentzn → Coupling.fermion_lines →
    string → string list → string list → Coupling.fusen → unit

  val lorentz_module :
    ?only : SSet.t → ?name :string →
    ?fortran_module :string → ?parameter_module :string →
    Format_Fortran.formatter → unit → unit

 end

module Targets =
 struct

  module Fortran : Fortran_Target =
    struct

      open Format_Fortran

      let fuse = UFO_targets.Fortran.fuse

      let lorentz_functions ff fusions () =
        List.iter
          (fun (name, s, l) →
            UFO_targets.Fortran.lorentz ff name s l)
          fusions

      let propagator_functions ff parameter_module propagators () =
        List.iter
          (fun (name, p) →
            UFO_targets.Fortran.propagator
              ff name
              parameter_module p.Propagator.variables
              p.Propagator.spins
              p.Propagator.numerator p.Propagator.denominator)
          propagators

      let lorentz_module
            ?only ?(name ="omega_amplitude_ufo")
            ?(fortran_module ="omega95")
            ?(parameter_module ="parameter_module") ff () =
        let printf fmt = fprintf ff fmt
        and nl = pp_newline ff in
        printf "module␣%s" name; nl ();
        printf "␣␣use␣kinds"; nl ();
        printf "␣␣use␣%s" fortran_module; nl ();
        printf "␣␣implicit␣none"; nl ();
        printf "␣␣private"; nl ();
        let fusions = Model.fusions ?only ()
        and propagators = Model.propagators () in
        List.iter
          (fun (name, _, _) → printf "␣␣public␣::␣%s" name; nl ())
          fusions;
        List.iter
          (fun (name, _) → printf "␣␣public␣::␣pr_U_%s" name; nl ())
          propagators;
        UFO_targets.Fortran.eps4_g4_g44_decl ff ();
        UFO_targets.Fortran.eps4_g4_g44_init ff ();
        printf "contains"; nl ();
        UFO_targets.Fortran.inner_product_functions ff ();
        lorentz_functions ff fusions ();
        propagator_functions ff parameter_module propagators ();
        printf "end␣module␣%s" name; nl ();
        pp_flush ff ()

    end

 end

module type Test =
 sig
  val suite : OUnit.test
 end

module Test : Test =
 struct

  open OUnit

  let lexer s =
    UFO_lexer.token (UFO_lexer.init_position "" (Lexing.from_string s))

  let suite_lexer_escapes =
    "escapes" >:::

      [ "single-quote" >::
          (fun () →
            assert_equal (UFO_parser.STRING "a'b'c") (lexer "'a\\'b\\'c'"));

        "unterminated" >::
          (fun () →
            assert_raises End_of_file (fun () → lexer "'a\\'b\\'c")) ]

  let suite_lexer =
    "lexer" >:::
      [suite_lexer_escapes]

  let suite =
    "UFO" >:::
      [suite_lexer]

 end
```

19.15 Targets

19.16 Interface of UFO_targets

19.16.1 Generating Code for UFO Lorentz Structures

```
module type T =
  sig
```

lorentz ff name spins lorentz writes the Fortran code implementing the fusion corresponding to the Lorentz structure *lorentz* to *ff*. NB: The *spins* : *int list* element of *UFO.Lorentz.t* from the UFO file is *not* sufficient to determine the domain and codomain of the function. We had to inspect the flavors, where the Lorentz structure is referenced to heuristically compute the *spins* as a *Coupling.lorentz array* .

```
    val lorentz :
      Format_Fortran.formatter → string →
      Coupling.lorentz array → UFO_Lorentz.t → unit

    val propagator :
      Format_Fortran.formatter → string → string → string list →
      Coupling.lorentz × Coupling.lorentz →
      UFO_Lorentz.t → UFO_Lorentz.t → unit
```

fusion_name name perm cc_list forms a name for the fusion *name* with the permutations *perm* and charge conjugations applied to the fermion lines *cc_list*.

```
    val fusion_name :
      string → Permutation.Default.t → Coupling.fermion_lines → string
```

fuse c v s fl g wfs ps fusion fuses the wavefunctions named *wfs* with momenta named *ps* using the vertex named *v* with legs reordered according to *fusion*. The overall coupling constant named *g* is multiplied by the rational coefficient *c*. The list of spins *s* and the fermion lines *fl* are used for selecting the appropriately transformed version of the vertex *v*.

```
    val fuse :
      Algebra.QC.t → string →
      Coupling.lorentzn → Coupling.fermion_lines →
      string → string list → string list → Coupling.fusen → unit

    val eps4_g4_g44_decl : Format_Fortran.formatter → unit → unit
    val eps4_g4_g44_init : Format_Fortran.formatter → unit → unit
    val inner_product_functions : Format_Fortran.formatter → unit → unit

    module type Test =
      sig
        val suite : OUnit.test
      end

    module Test : Test

  end

module Fortran : T
```

19.17 Implementation of UFO_targets

19.17.1 Generating Code for UFO Lorentz Structures

O'Caml before 4.02 had a module typing bug that forced us to put these definitions outside of *Lorentz_Fusion*. Since then, they might have appeared in more places. Investigate, if it is worthwhile to encapsulate them again.

```
module Q = Algebra.Q
module QC = Algebra.QC

module type T =
  sig
```

lorentz formatter name spins v writes a representation of the Lorentz structure *v* of particles with the Lorentz representations *spins* as a (Fortran) function *name* to *formatter*.

```
    val lorentz :
      Format_Fortran.formatter → string →
      Coupling.lorentz array → UFO_Lorentz.t → unit

    val propagator :
      Format_Fortran.formatter → string → string → string list →
      Coupling.lorentz × Coupling.lorentz →
      UFO_Lorentz.t → UFO_Lorentz.t → unit

    val fusion_name :
      string → Permutation.Default.t → Coupling.fermion_lines → string
    val fuse :
      Algebra.QC.t → string →
      Coupling.lorentzn → Coupling.fermion_lines →
      string → string list → string list → Coupling.fusen → unit

    val eps4_g4_g44_decl : Format_Fortran.formatter → unit → unit
    val eps4_g4_g44_init : Format_Fortran.formatter → unit → unit
    val inner_product_functions : Format_Fortran.formatter → unit → unit

    module type Test =
      sig
        val suite : OUnit.test
      end

    module Test : Test

  end

module Fortran : T =
  struct

    open Format_Fortran

    let pp_divide ?(indent = 0) ff () =
      fprintf ff "%*s!␣%s" indent "" (String.make (70 − indent) '-');
      pp_newline ff ()

    let conjugate = function
      | Coupling.Spinor → Coupling.ConjSpinor
      | Coupling.ConjSpinor → Coupling.Spinor
      | r → r

    let spin_mnemonic = function
      | Coupling.Scalar → "phi"
      | Coupling.Spinor → "psi"
      | Coupling.ConjSpinor → "psibar"
      | Coupling.Majorana → "chi"
      | Coupling.Maj_Ghost →
          invalid_arg "UFO_targets:␣Maj_Ghost"
      | Coupling.Vector → "a"
      | Coupling.Massive_Vector → "v"
      | Coupling.Vectorspinor → "grav" (* itino *)
      | Coupling.Tensor_1 →
          invalid_arg "UFO_targets:␣Tensor_1"
      | Coupling.Tensor_2 → "h"
      | Coupling.BRS l →
          invalid_arg "UFO_targets:␣BRS"

    let fortran_type = function
      | Coupling.Scalar → "complex(kind=default)"
      | Coupling.Spinor → "type(spinor)"
      | Coupling.ConjSpinor → "type(conjspinor)"
      | Coupling.Majorana → "type(bispinor)"
      | Coupling.Maj_Ghost →
          invalid_arg "UFO_targets:␣Maj_Ghost"
      | Coupling.Vector → "type(vector)"
      | Coupling.Massive_Vector → "type(vector)"
      | Coupling.Vectorspinor → "type(vectorspinor)"
      | Coupling.Tensor_1 →
          invalid_arg "UFO_targets:␣Tensor_1"
      | Coupling.Tensor_2 → "type(tensor)"
      | Coupling.BRS l →
          invalid_arg "UFO_targets:␣BRS"
```

The `omegalib` separates time from space. Maybe not a good idea after all. Mend it locally …

```
    type wf =
  { pos : int;
    spin : Coupling.lorentz;
    name : string;
    local_array : string option;
    momentum : string;
    momentum_array : string;
    fortran_type : string }

let wf_table spins =
  Array.mapi
    (fun i s →
      let spin =
        if i = 0 then
          conjugate s
        else
          s in
      let pos = succ i in
      let i = string_of_int pos in
      let name = spin_mnemonic s ^ i in
      let local_array =
        begin match spin with
        | Coupling.Vector | Coupling.Massive_Vector → Some (name ^ "a")
        | _ → None
        end in
      { pos;
        spin;
        name;
        local_array;
        momentum = "k" ^ i;
        momentum_array = "p" ^ i;
        fortran_type = fortran_type spin } )
    spins

module L = UFO_Lorentz
```

Format rational ($Q.t$) and complex rational ($QC.t$) numbers as fortran values.

```
let format_rational q =
  if Q.is_integer q then
    string_of_int (Q.to_integer q)
  else
    let n, d = Q.to_ratio q in
    Printf.sprintf "%d.0_default/%d" n d

let format_complex_rational cq =
  let real = QC.re cq
  and imag = QC.im cq in
  if Q.is_null imag then
    begin
      if Q.is_negative real then
        "(" ^ format_rational real ^ ")"
      else
        format_rational real
    end
  else if Q.is_integer real ∧ Q.is_integer imag then
    Printf.sprintf "(%d,%d)" (Q.to_integer real) (Q.to_integer imag)
  else
    Printf.sprintf
      "cmplx(%s,%s,kind=default)"
      (format_rational real) (format_rational imag)
```

Optimize the representation if used as a prefactor of a summand in a sum.

```
let format_rational_factor q =
  if Q.is_unit q then
```

```
        "+␣"
      else if Q.is_unit (Q.neg q) then
        "-␣"
      else if Q.is_negative q then
        "-␣" ^ format_rational (Q.neg q) ^ "*"
      else
        "+␣" ^ format_rational q ^ "*"

  let format_complex_rational_factor cq =
    let real = QC.re cq
    and imag = QC.im cq in
    if Q.is_null imag then
      begin
        if Q.is_unit real then
          "+␣"
        else if Q.is_unit (Q.neg real) then
          "-␣"
        else if Q.is_negative real then
          "-␣" ^ format_rational (Q.neg real) ^ "*"
        else
          "+␣" ^ format_rational real ^ "*"
      end
    else if Q.is_integer real ∧ Q.is_integer imag then
      Printf.sprintf "+␣(%d,%d)*" (Q.to_integer real) (Q.to_integer imag)
    else
      Printf.sprintf
        "+␣cmplx(%s,%s,kind=default)*"
        (format_rational real) (format_rational imag)
```

Append a formatted list of indices to *name*.

```
  let append_indices name = function
    | [] → name
    | indices →
        name ^ "(" ^ String.concat "," (List.map string_of_int indices) ^ ")"
```

Dirac string variables and their names.

```
  type dsv =
    | Ket of int
    | Bra of int
    | Braket of int

  let dsv_name = function
    | Ket n → Printf.sprintf "ket%02d" n
    | Bra n → Printf.sprintf "bra%02d" n
    | Braket n → Printf.sprintf "bkt%02d" n

  let dirac_dimension dsv indices =
    let tail ilist =
      String.concat "," (List.map (fun _ → "0:3") ilist) ^ ")" in
    match dsv, indices with
    | Braket _, [] → ""
    | (Ket _ | Bra _), [] → ",␣dimension(1:4)"
    | Braket _, indices → ",␣dimension(" ^ tail indices
    | (Ket _ | Bra _), indices → ",␣dimension(1:4," ^ tail indices
```

Write Fortran code to *decl* and *eval*: apply the Dirac matrix *gamma* with complex rational entries to the spinor *ket* from the left. *ket* must be the name of a scalar variable and cannot be an array element. The result is stored in *dsv_name* (*Ket n*) which can have additional *indices*. Return *Ket n* for further processing.

```
  let dirac_ket_to_fortran_decl ff n indices =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Ket n in
    printf
```

```
      "␣␣␣␣@[<2>complex(kind=default)%s␣::@␣%s@]"
      (dirac_dimension dsv indices) (dsv_name dsv);
    nl ()

  let dirac_ket_to_fortran_eval ff n indices gamma ket =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Ket n in
    for i = 0 to 3 do
      let name = append_indices (dsv_name dsv) (succ i :: indices) in
      printf "␣␣␣␣@[<%d>%s␣=␣0" (String.length name + 4) name;
      for j = 0 to 3 do
        if ¬ (QC.is_null gamma.(i).(j)) then
          printf
            "@␣%s%s%%a(%d)"
            (format_complex_rational_factor gamma.(i).(j))
            ket.name (succ j)
      done;
      printf "@]";
      nl ()
    done;
    dsv
```

The same as *dirac_ket_to_fortran*, but apply the Dirac matrix *gamma* to *bra* from the right and return *Bra n*.

```
  let dirac_bra_to_fortran_decl ff n indices =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Bra n in
    printf
      "␣␣␣␣@[<2>complex(kind=default)%s␣::@␣%s@]"
      (dirac_dimension dsv indices) (dsv_name dsv);
    nl ()

  let dirac_bra_to_fortran_eval ff n indices bra gamma =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Bra n in
    for j = 0 to 3 do
      let name = append_indices (dsv_name dsv) (succ j :: indices) in
      printf "␣␣␣␣@[<%d>%s␣=␣0" (String.length name + 4) name;
      for i = 0 to 3 do
        if ¬ (QC.is_null gamma.(i).(j)) then
          printf
            "@␣%s%s%%a(%d)"
            (format_complex_rational_factor gamma.(i).(j))
            bra.name (succ i)
      done;
      printf "@]";
      nl ()
    done;
    dsv
```

More of the same, but evaluating a spinor sandwich and returning *Braket n*.

```
  let dirac_braket_to_fortran_decl ff n indices =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Braket n in
    printf
      "␣␣␣␣@[<2>complex(kind=default)%s␣::@␣%s@]"
      (dirac_dimension dsv indices) (dsv_name dsv);
    nl ()

  let dirac_braket_to_fortran_eval ff n indices bra gamma ket =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let dsv = Braket n in
    let name = append_indices (dsv_name dsv) indices in
    printf "␣␣␣␣@[<%d>%s␣=␣0" (String.length name + 4) name;
    for i = 0 to 3 do
      for j = 0 to 3 do
        if ¬ (QC.is_null gamma.(i).(j)) then
          printf
            "@␣%s%s%%a(%d)*%s%%a(%d)"
            (format_complex_rational_factor gamma.(i).(j))
            bra.name (succ i) ket.name (succ j)
      done
    done;
    printf "@]";
    nl ();
    dsv
```

Choose among the previous functions according to the position of *bra* and *ket* among the wavefunctions. If any is in the first position evaluate the spinor expression with the corresponding spinor removed, otherwise evaluate the spinir sandwich.

```
  let dirac_bra_or_ket_to_fortran_decl ff n indices bra ket =
    if bra = 1 then
      dirac_ket_to_fortran_decl ff n indices
    else if ket = 1 then
      dirac_bra_to_fortran_decl ff n indices
    else
      dirac_braket_to_fortran_decl ff n indices

  let dirac_bra_or_ket_to_fortran_eval ff n indices wfs bra gamma ket =
    if bra = 1 then
      dirac_ket_to_fortran_eval ff n indices gamma wfs.(pred ket)
    else if ket = 1 then
      dirac_bra_to_fortran_eval ff n indices wfs.(pred bra) gamma
    else
      dirac_braket_to_fortran_eval
        ff n indices wfs.(pred bra) gamma wfs.(pred ket)
```

UFO summation indices are negative integers. Derive a valid Fortran variable name.

```
  let prefix_summation = "mu"
  let prefix_polarization = "nu"
  let index_spinor = "alpha"
  let index_tensor = "nu"

  let index_variable mu =
    if mu < 0 then
      Printf.sprintf "%s%d" prefix_summation (− mu)
    else if mu ≡ 0 then
     prefix_polarization
    else
      Printf.sprintf "%s%d" prefix_polarization mu

  let format_indices indices =
    String.concat "," (List.map index_variable indices)

  module IntPM =
    Partial.Make (struct type t = int let compare = compare end)

  type tensor =
    | DS of dsv
    | V of string
    | T of UFOx.Lorentz_Atom.vector
    | S of UFOx.Lorentz_Atom.scalar
    | Inv of UFOx.Lorentz_Atom.scalar
```

Transform the Dirac strings if we have Majorana fermions involved, in order to implement the algorithm from JRR's thesis. NB: The following is for reference only, to better understand what JRR was doing...
If the vertex is (suppressing the Lorentz indices of ϕ_2 and Γ)

$$\bar\psi\Gamma\phi\psi = \Gamma_{\alpha\beta}\bar\psi_\alpha\phi\psi_\beta \tag{19.23}$$

(cf. *Coupling.FBF* in the hardcoded O'Mega models), then this is the version implemented by *fuse* below.

```
let tho_print_dirac_current f c wf1 wf2 fusion =
  match fusion with
  | [1; 3] -> printf "%s_ff(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta}$ *)
  | [3; 1] -> printf "%s_ff(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta}$ *)
  | [2; 3] -> printf "f_%sf(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta}$ *)
  | [3; 2] -> printf "f_%sf(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta}$ *)
  | [1; 2] -> printf "f_f%s(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2$ *)
  | [2; 1] -> printf "f_f%s(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2$ *)
  | _ -> ()
```

The corresponding UFO *fuse* exchanges the arguments in the case of two fermions. This is the natural choice for cyclic permutations.

```
let tho_print_FBF_current f c wf1 wf2 fusion =
  match fusion with
  | [3; 1] -> printf "f%sf_p120(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha}$ *)
  | [1; 3] -> printf "f%sf_p120(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha}$ *)
  | [2; 3] -> printf "f%sf_p012(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta}$ *)
  | [3; 2] -> printf "f%sf_p012(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta}$ *)
  | [1; 2] -> printf "f%sf_p201(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2$ *)
  | [2; 1] -> printf "f%sf_p201(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2$ *)
  | _ -> ()
```

This is how JRR implemented (see subsection AB.26.1) the Dirac matrices that don't change sign under $C\Gamma^TC^{-1} = \Gamma$, i. e. $\mathbf{1}$, γ_5 and $\gamma_5\gamma_\mu$ (see *Targets.Fortran_Majorana_Fermions.print_fermion_current*)

- In the case of two fermions, the second wave function *wf2* is always put into the second slot, as described in JRR's thesis.
- In the case of a boson and a fermion, there is no need for both `"f_%sf"` and `"f_f%s"`, since the latter can be obtained by exchanging arguments.

```
let jrr_print_majorana_current_S_P_A f c wf1 wf2 fusion =
  match fusion with
  | [1; 3] -> printf "%s_ff(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta} \cong C\Gamma$ *)
  | [3; 1] -> printf "%s_ff(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\psi_{1,\alpha}\bar\psi_{2,\beta} \cong C\Gamma = C\,C\Gamma^TC^{-1}$ *)
  | [2; 3] -> printf "f_%sf(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [3; 2] -> printf "f_%sf(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 2] -> printf "f_%sf(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong \Gamma = C\Gamma^TC^{-1}$ *)
  | [2; 1] -> printf "f_%sf(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong \Gamma = C\Gamma^TC^{-1}$ *)
  | _ -> ()
```

This is how JRR implemented the Dirac matrices that do change sign under $C\Gamma^TC^{-1} = -\Gamma$, i. e. γ_μ and $\sigma_{\mu\nu}$ (see *Targets.Fortran_Majorana_Fermions.print_fermion_current_vector*).

```
let jrr_print_majorana_current_V f c wf1 wf2 fusion =
  match fusion with
  | [1; 3] -> printf "%s_ff(␣%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta} \cong C\Gamma$ *)
  | [3; 1] -> printf "%s_ff(-%s,%s,%s)" f c wf1 wf2 (* $-(C\Gamma)_{\alpha\beta}\psi_{1,\alpha}\bar\psi_{2,\beta} \cong -C\Gamma = C\,C\Gamma^TC^{-1}$ *)
  | [2; 3] -> printf "f_%sf(␣%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [3; 2] -> printf "f_%sf(␣%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 2] -> printf "f_%sf(-%s,%s,%s)" f c wf2 wf1 (* $-\Gamma_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong -\Gamma = C\Gamma^TC^{-1}$ *)
  | [2; 1] -> printf "f_%sf(-%s,%s,%s)" f c wf1 wf2 (* $-\Gamma_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong -\Gamma = C\Gamma^TC^{-1}$ *)
  | _ -> ()
```

These two can be unified, if the `_c` functions implement $\Gamma' = C\Gamma^TC^{-1}$, but we *must* make sure that the multiplication with C from the left happens *after* the transformation $\Gamma \to \Gamma'$.

```
let jrr_print_majorana_current f c wf1 wf2 fusion =
  match fusion with
  | [1; 3] → printf "%s_ff␣␣(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta} \cong C\Gamma$ *)
  | [3; 1] → printf "%s_ff_c(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma')_{\alpha\beta}\psi_{1,\alpha}\bar\psi_{2,\beta} \cong C\Gamma' = C\,C\Gamma^T C^{-1}$ *)
  | [2; 3] → printf "f_%sf␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [3; 2] → printf "f_%sf␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 2] → printf "f_%sf_c(%s,%s,%s)" f c wf2 wf1 (* $\Gamma'_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong \Gamma' = C\Gamma^T C^{-1}$ *)
  | [2; 1] → printf "f_%sf_c(%s,%s,%s)" f c wf1 wf2 (* $\Gamma'_{\alpha\beta}\phi_1\bar\psi_{2,\beta} \cong \Gamma' = C\Gamma^T C^{-1}$ *)
  | _ → ()
```

Since we may assume $C^{-1} = -C = C^T$, this can be rewritten if the `_c` functions implement

$$\Gamma'^T = \left(C\Gamma^T C^{-1}\right)^T = \left(C^{-1}\right)^T \Gamma C^T = C\Gamma C^{-1} \tag{19.24}$$

instead.

```
let jrr_print_majorana_current_transposing f c wf1 wf2 fusion =
  match fusion with
  | [1; 3] → printf "%s_ff␣␣(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta} \cong C\Gamma$ *)
  | [3; 1] → printf "%s_ff_c(%s,%s,%s)" f c wf2 wf1 (* $(C\Gamma')^T_{\alpha\beta}\bar\psi_{1,\alpha}\psi_{2,\beta} \cong (C\Gamma')^T = -C\Gamma$ *)
  | [2; 3] → printf "f_%sf␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [3; 2] → printf "f_%sf␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 2] → printf "f_f%s_c(%s,%s,%s)" f c wf1 wf2 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [2; 1] → printf "f_f%s_c(%s,%s,%s)" f c wf2 wf1 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | _ → ()
```

where we have used

$$(C\Gamma')^T = \Gamma'^{,T} C^T = C\Gamma C^{-1} C^T = C\Gamma C^{-1}(-C) = -C\Gamma\,. \tag{19.25}$$

This puts the arguments in the same slots as *tho_print_dirac_current* above and can be implemented by *fuse*, iff we inject the proper transformations in *dennerize* below. We notice that we do *not* need the conjugated version for all combinations, but only for the case of two fermions. In the two cases of one column spinor ψ, only the original version appears and in the two cases of one row spinor $\bar\psi$, only the conjugated version appears. Before we continue, we must however generalize from the assumption (19.23) that the fields in the vertex are always ordered as in *Coupling.FBF*. First, even in this case the slots of the fermions must be exchanged to accomodate the cyclic permutations. Therefore we exchange the arguments of the [1; 3] and [3; 1] fusions.

```
let jrr_print_majorana_FBF f c wf1 wf2 fusion =
  match fusion with (* fline = (3, 1) *)
  | [3; 1] → printf "f%sf_p120_c(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma')^T_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong (C\Gamma')^T = -C\Gamma$ *)
  | [1; 3] → printf "f%sf_p120␣␣(%s,%s,%s)" f c wf2 wf1 (* $(C\Gamma)_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong C\Gamma$ *)
  | [2; 3] → printf "f%sf_p012␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [3; 2] → printf "f%sf_p012␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 2] → printf "f%sf_p201␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [2; 1] → printf "f%sf_p201␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | _ → ()
```

The other two permutations:

```
let jrr_print_majorana_FFB f c wf1 wf2 fusion =
  match fusion with (* fline = (1, 2) *)
  | [3; 1] → printf "ff%s_p120␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [1; 3] → printf "ff%s_p120␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [2; 3] → printf "ff%s_p012␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [3; 2] → printf "ff%s_p012␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [1; 2] → printf "ff%s_p201␣␣(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong C\Gamma$ *)
  | [2; 1] → printf "ff%s_p201_c(%s,%s,%s)" f c wf2 wf1 (* $(C\Gamma')^T_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong (C\Gamma')^T = -C\Gamma$ *)
  | _ → ()
```

```
let jrr_print_majorana_BFF f c wf1 wf2 fusion =
  match fusion with (* fline = (2, 3) *)
  | [3; 1] → printf "%sff_p120␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [1; 3] → printf "%sff_p120␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma'^T_{\alpha\beta}\bar\psi_{1,\alpha}\phi_2 \cong \Gamma'^T = C\Gamma C^{-1}$ *)
  | [2; 3] → printf "%sff_p012␣␣(%s,%s,%s)" f c wf1 wf2 (* $(C\Gamma)_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong C\Gamma$ *)
  | [3; 2] → printf "%sff_p012_c(%s,%s,%s)" f c wf2 wf1 (* $(C\Gamma')^T_{\alpha\beta}\psi_{1,\beta}\bar\psi_{2,\alpha} \cong (C\Gamma')^T = -C\Gamma$ *)
  | [1; 2] → printf "%sff_p201␣␣(%s,%s,%s)" f c wf1 wf2 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | [2; 1] → printf "%sff_p201␣␣(%s,%s,%s)" f c wf2 wf1 (* $\Gamma_{\alpha\beta}\phi_1\psi_{2,\beta} \cong \Gamma$ *)
  | _ → ()
```

In the model, the necessary information is provided as *Coupling.fermion_lines*, encoded as $(right, left)$ in the usual direction of the lines. E. g. the case of (19.23) is $(3, 1)$. Equivalent information is available as $(ket,\ bra)$ in *UFO_Lorentz.dirac_string*.

```
let is_majorana = function
  | Coupling.Majorana | Coupling.Vectorspinor | Coupling.Maj_Ghost → true
  | _ → false

let is_dirac = function
  | Coupling.Spinor | Coupling.ConjSpinor → true
  | _ → false

let dennerize ~eval wfs atom =
  let printf fmt = fprintf eval fmt
  and nl = pp_newline eval in
  if is_majorana wfs.(pred atom.L.bra).spin ∨
       is_majorana wfs.(pred atom.L.ket).spin then
    if atom.L.bra = 1 then
      (* Fusing one or more bosons with a ket like fermion: $\chi \leftarrow \Gamma\chi$. *)
      (* Don't do anything, as per subsection AB.26.1. *)
      atom
    else if atom.L.ket = 1 then
      (* We fuse one or more bosons with a bra like fermion: $\bar\chi \leftarrow \bar\chi\Gamma$. *)
      (* $\Gamma \to C\Gamma C^{-1}$. *)
      begin
        let atom = L.conjugate atom in
        printf "␣␣␣␣!␣conjugated␣for␣Majorana"; nl ();
        printf "␣␣␣␣!␣%s" (L.dirac_string_to_string atom); nl ();
        atom
      end
    else if ¬ atom.L.conjugated then
      (* We fuse zero or more bosons with a sandwich of fermions. $\phi \leftarrow \bar\chi\gamma\chi$.*)
      (* Multiply by $C$ from the left, as per subsection AB.26.1. *)
      begin
        let atom = L.cc_times atom in
        printf "␣␣␣␣!␣multiplied␣by␣CC␣for␣Majorana"; nl ();
        printf "␣␣␣␣!␣%s" (L.dirac_string_to_string atom); nl ();
        atom
      end
    else
      (* Transposed: multiply by $-C$ from the left. *)
      begin
        let atom = L.minus (L.cc_times atom) in
        printf "␣␣␣␣!␣multiplied␣by␣-CC␣for␣Majorana"; nl ();
        printf "␣␣␣␣!␣%s" (L.dirac_string_to_string atom); nl ();
        atom
      end
  else
    atom
```

Write the ith Dirac string *ds* as Fortran code to *eval*, including a shorthand representation as a comment. Return *ds* with *ds.L.atom* replaced by the dirac string variable, i, e. *DS dsv* annotated with the internal and external indices. In addition write the declaration to *decl*.

```
let dirac_string_to_fortran ~decl ~eval i wfs ds =
  let printf fmt = fprintf eval fmt
  and nl = pp_newline eval in
  let bra = ds.L.atom.L.bra
  and ket = ds.L.atom.L.ket in
```

```
    pp_divide ~indent : 4 eval ();
    printf "␣␣␣␣!␣%s" (L.dirac_string_to_string ds.L.atom); nl ();
    let atom = dennerize ~eval wfs ds.L.atom in
    begin match ds.L.indices with
    | [] →
       let gamma = L.dirac_string_to_matrix (fun _ → 0) atom in
       dirac_bra_or_ket_to_fortran_decl decl i [] bra ket;
       let dsv =
          dirac_bra_or_ket_to_fortran_eval eval i [] wfs bra gamma ket in
       L.map_atom (fun _ → DS dsv) ds
    | indices →
       dirac_bra_or_ket_to_fortran_decl decl i indices bra ket;
       let combinations = Product.power (List.length indices) [0; 1; 2; 3] in
       let dsv =
          List.map
             (fun combination →
                let substitution = IntPM.of_lists indices combination in
                let substitute = IntPM.apply substitution in
                let indices = List.map substitute indices in
                let gamma = L.dirac_string_to_matrix substitute atom in
                dirac_bra_or_ket_to_fortran_eval eval i indices wfs bra gamma ket)
             combinations in
       begin match ThoList.uniq (List.sort compare dsv) with
       | [dsv] → L.map_atom (fun _ → DS dsv) ds
       | _ → failwith "dirac_string_to_fortran:␣impossible"
       end
    end
```

Write the Dirac strings in the list *ds_list* as Fortran code to *eval*, including shorthand representations as comments. Return the list of variables and corresponding indices to be contracted.

```
  let dirac_strings_to_fortran ~decl ~eval wfs last ds_list =
    List.fold_left
      (fun (i, acc) ds →
        let i = succ i in
        (i, dirac_string_to_fortran ~decl ~eval i wfs ds :: acc))
      (last, []) ds_list
```

Perform a nested sum of terms, as printed by *print_term* (which takes the number of spaces to indent as only argument) of the cartesian product of *indices* running from 0 to 3.

```
  let nested_sums ~decl ~eval initial_indent indices print_term =
    let rec nested_sums' indent = function
      | [] → print_term indent
      | index :: indices →
         let var = index_variable index in
         fprintf eval "%*s@[<2>do␣%s␣=␣0,␣3@]" indent "" var;
         pp_newline eval ();
         nested_sums' (indent + 2) indices; pp_newline eval ();
         fprintf eval "%*s@[<2>end␣do@]" indent "" in
    nested_sums' (initial_indent + 2) indices
```

Polarization indices also need to be summed over, but they appear only once.

```
  let indices_of_contractions contractions =
    let index_pairs, polarizations =
      L.classify_indices
        (ThoList.flatmap (fun ds → ds.L.indices) contractions) in
    try
      ThoList.pairs index_pairs @ ThoList.uniq (List.sort compare polarizations)
    with
    | Invalid_argument s →
       invalid_arg
         ("indices_of_contractions:␣" ^
            ThoList.to_string string_of_int index_pairs)

  let format_dsv dsv indices =
    match dsv, indices with
    | Braket _, [] → dsv_name dsv
    | Braket _, ilist →
       Printf.sprintf "%s(%s)" (dsv_name dsv) (format_indices indices)
    | (Bra _ | Ket _), [] →
       Printf.sprintf "%s(%s)" (dsv_name dsv) index_spinor
    | (Bra _ | Ket _), ilist →
       Printf.sprintf
         "%s(%s,%s)" (dsv_name dsv) index_spinor (format_indices indices)

  let denominator_name = "denom_"
  let mass_name = "m_"
  let width_name = "w_"

  let format_tensor t =
    let indices = t.L.indices in
    match t.L.atom with
    | DS dsv → format_dsv dsv indices
    | V vector → Printf.sprintf "%s(%s)" vector (format_indices indices)
    | T UFOx.Lorentz_Atom.P (mu, n) →
       Printf.sprintf "p%d(%s)" n (index_variable mu)
    | T UFOx.Lorentz_Atom.Epsilon (mu1, mu2, mu3, mu4) →
       Printf.sprintf "eps4_(%s)" (format_indices [mu1; mu2; mu3; mu4])
    | T UFOx.Lorentz_Atom.Metric (mu1, mu2) →
       if mu1 > 0 ∧ mu2 > 0 then
         Printf.sprintf "g44_(%s)" (format_indices [mu1; mu2])
       else
         failwith "format_tensor:␣compress_metrics␣has␣failed!"
    | S (UFOx.Lorentz_Atom.Mass _) → mass_name
    | S (UFOx.Lorentz_Atom.Width _) → width_name
    | S (UFOx.Lorentz_Atom.P2 i) → Printf.sprintf "g2_(p%d)" i
    | S (UFOx.Lorentz_Atom.P12 (i, j)) → Printf.sprintf "g12_(p%d,p%d)" i j
    | Inv (UFOx.Lorentz_Atom.Mass _) → "1/" ^ mass_name
    | Inv (UFOx.Lorentz_Atom.Width _) → "1/" ^ width_name
    | Inv (UFOx.Lorentz_Atom.P2 i) → Printf.sprintf "1/g2_(p%d)" i
    | Inv (UFOx.Lorentz_Atom.P12 (i, j)) →
       Printf.sprintf "1/g12_(p%d,p%d)" i j
    | S (UFOx.Lorentz_Atom.Variable s) → s
    | Inv (UFOx.Lorentz_Atom.Variable s) → "1/" ^ s
    | S (UFOx.Lorentz_Atom.Coeff c) → UFOx.Value.to_string c
    | Inv (UFOx.Lorentz_Atom.Coeff c) → "1/(" ^ UFOx.Value.to_string c ^ ")"

  let rec multiply_tensors ~decl ~eval = function
    | [] → fprintf eval "1";
    | [t] → fprintf eval "%s" (format_tensor t)
    | t :: tensors →
       fprintf eval "%s@,*" (format_tensor t);
       multiply_tensors ~decl ~eval tensors

  let pseudo_wfs_for_denominator =
    Array.init
      2
      (fun i →
        let ii = string_of_int i in
        { pos = i;
         spin = Coupling.Scalar;
         name = denominator_name;
         local_array = None;
         momentum = "k" ^ ii;
         momentum_array = "p" ^ ii;
         fortran_type = fortran_type Coupling.Scalar })
```

```
let contract_indices ~decl ~eval indent wf_indices wfs (fusion, contractees) =
  let printf fmt = fprintf eval fmt
  and nl = pp_newline eval in
  let sum_var =
    begin match wf_indices with
    | [] → wfs.(0).name
    | ilist →
        let indices = String.concat "," ilist in
        begin match wfs.(0).local_array with
        | None →
            let component =
              begin match wfs.(0).spin with
              | Coupling.Spinor | Coupling.ConjSpinor | Coupling.Majorana → "a"
              | Coupling.Tensor_2 → "t"
              | Coupling.Vector | Coupling.Massive_Vector →
                 failwith "contract_indices:␣expected␣local_array␣for␣vectors"
              | _ → failwith "contract_indices:␣unexpected␣spin"
              end in
            Printf.sprintf "%s%%%s(%s)" wfs.(0).name component indices
        | Some a → Printf.sprintf "%s(%s)" a indices
        end
    end in
  let indices =
    List.filter
      (fun i → UFOx.Index.position i ≠ 1)
      (indices_of_contractions contractees) in
  nested_sums
    ~decl ~eval
    indent indices
    (fun indent →
      printf "%*s@[<2>%s␣=␣%s" indent "" sum_var sum_var;
      printf "@␣%s" (format_complex_rational_factor fusion.L.coeff);
      List.iter (fun i → printf "@,g4_(%s)*" (index_variable i)) indices;
      printf "@,(";
      multiply_tensors ~decl ~eval contractees;
      printf ")";
      begin match fusion.L.denominator with
      | [] → ()
      | d → printf "␣/␣%s" denominator_name
      end;
      printf "@]");
  printf "@]";
  nl ()

let scalar_expression1 ~decl ~eval fusion =
  let printf fmt = fprintf eval fmt in
  match fusion.L.dirac, fusion.L.vector with
  | [], [] →
     let scalars =
       List.map (fun t → { L.atom = S t; L.indices = [] }) fusion.L.scalar
     and inverses =
       List.map (fun t → { L.atom = Inv t; L.indices = [] }) fusion.L.inverse in
     let contractees = scalars @ inverses in
     printf "@␣%s" (format_complex_rational_factor fusion.L.coeff);
     multiply_tensors ~decl ~eval contractees
  | _, [] →
     invalid_arg
       "UFO_targets.Fortran.scalar_expression1:␣unexpected␣spinor␣indices"
  | [], _ →
     invalid_arg
       "UFO_targets.Fortran.scalar_expression1:␣unexpected␣vector␣indices"
  | _, _ →
      invalid_arg
        "UFO_targets.Fortran.scalar_expression1:␣unexpected␣indices"

let scalar_expression ~decl ~eval indent name fusions =
  let printf fmt = fprintf eval fmt
  and nl = pp_newline eval in
  let sum_var = name in
  printf "%*s@[<2>%s␣=" indent "" sum_var;
  List.iter (scalar_expression1 ~decl ~eval) fusions;
  printf "@]";
  nl ()

let local_vector_copies ~decl ~eval wfs =
  begin match wfs.(0).local_array with
  | None → ()
  | Some a →
      fprintf
        decl "␣␣␣␣@[<2>complex(kind=default),@␣dimension(0:3)␣::@␣%s@]" a;
      pp_newline decl ()
  end;
  let n = Array.length wfs in
  for i = 1 to n − 1 do
    match wfs.(i).local_array with
    | None → ()
    | Some a →
        fprintf
          decl "␣␣␣␣@[<2>complex(kind=default),@␣dimension(0:3)␣::@␣%s@]" a;
        pp_newline decl ();
        fprintf eval "␣␣␣␣@[<2>%s(0)␣=␣%s%%t@]" a wfs.(i).name;
        pp_newline eval ();
        fprintf eval "␣␣␣␣@[<2>%s(1:3)␣=␣%s%%x@]" a wfs.(i).name;
        pp_newline eval ()
  done

let return_vector ff wfs =
  let printf fmt = fprintf ff fmt
  and nl = pp_newline ff in
  match wfs.(0).local_array with
  | None → ()
  | Some a →
      pp_divide ~indent : 4 ff ();
      printf "␣␣␣␣@[<2>%s%%t␣=␣%s(0)@]" wfs.(0).name a; nl ();
      printf "␣␣␣␣@[<2>%s%%x␣=␣%s(1:3)@]" wfs.(0).name a; nl ()

let multiply_coupling_and_scalars ff g_opt wfs =
  let printf fmt = fprintf ff fmt
  and nl = pp_newline ff in
  pp_divide ~indent : 4 ff ();
  let g =
    match g_opt with
    | None → ""
    | Some g → g ^ "*" in
  let wfs0name =
    match wfs.(0).local_array with
    | None → wfs.(0).name
    | Some a → a in
  printf "␣␣␣␣@[<2>%s␣=␣%s%s" wfs0name g wfs0name;
  for i = 1 to Array.length wfs − 1 do
    match wfs.(i).spin with
    | Coupling.Scalar → printf "@,*%s" wfs.(i).name
    | _ → ()
  done;
  printf "@]"; nl ()

let local_momentum_copies ~decl ~eval wfs =
  let n = Array.length wfs in
  fprintf
    decl "␣␣␣␣@[<2>real(kind=default),@␣dimension(0:3)␣::@␣%s"
    wfs.(0).momentum_array;
  for i = 1 to n - 1 do
    fprintf decl ",@␣%s" wfs.(i).momentum_array;
    fprintf
      eval "␣␣␣␣@[<2>%s(0)␣=␣%s%%t@]"
      wfs.(i).momentum_array wfs.(i).momentum;
    pp_newline eval ();
    fprintf
      eval "␣␣␣␣@[<2>%s(1:3)␣=␣%s%%x@]"
      wfs.(i).momentum_array wfs.(i).momentum;
    pp_newline eval ()
  done;
  fprintf eval "␣␣␣␣@[<2>%s␣=" wfs.(0).momentum_array;
  for i = 1 to n - 1 do
    fprintf eval "@␣-␣%s" wfs.(i).momentum_array
  done;
  fprintf decl "@]";
  pp_newline decl ();
  fprintf eval "@]";
  pp_newline eval ()

let contractees_of_fusion
      ~decl ~eval wfs (max_dsv, indices_seen, contractees) fusion =
  let max_dsv', dirac_strings =
    dirac_strings_to_fortran ~decl ~eval wfs max_dsv fusion.L.dirac
  and vectors =
    List.fold_left
      (fun acc wf →
        match wf.spin, wf.local_array with
        | Coupling.Tensor_2, None →
           { L.atom =
                V (Printf.sprintf "%s%d%%t" (spin_mnemonic wf.spin) wf.pos);
             L.indices = [UFOx.Index.pack wf.pos 1;
                          UFOx.Index.pack wf.pos 2] } :: acc
        | _, None → acc
        | _, Some a → { L.atom = V a; L.indices = [wf.pos] } :: acc)
      [] (List.tl (Array.to_list wfs))
  and tensors =
    List.map (L.map_atom (fun t → T t)) fusion.L.vector
  and scalars =
    List.map (fun t → { L.atom = S t; L.indices = [] }) fusion.L.scalar
  and inverses =
    List.map (fun t → { L.atom = Inv t; L.indices = [] }) fusion.L.inverse in
  let contractees' = dirac_strings @ vectors @ tensors @ scalars @ inverses in
  let indices_seen' =
    Sets.Int.of_list (indices_of_contractions contractees') in
  (max_dsv',
   Sets.Int.union indices_seen indices_seen',
   (fusion, contractees') :: contractees)

let local_name wf =
  match wf.local_array with
  | Some a → a
  | None →
     match wf.spin with
     | Coupling.Spinor | Coupling.ConjSpinor | Coupling.Majorana →
        wf.name ^ "%a"
      | Coupling.Scalar → wf.name
      | Coupling.Tensor_2 → wf.name ^ "%t"
      | Coupling.Vector | Coupling.Massive_Vector →
         failwith "UFO_targets.Fortran.local_name:␣unexpected␣spin␣1"
      | _ →
         failwith "UFO_targets.Fortran.local_name:␣unhandled␣spin"

let external_wf_loop ~decl ~eval ~indent wfs (fusion, _ as contractees) =
  pp_divide ~indent eval ();
  fprintf eval "%*s!␣%s" indent "" (L.to_string [fusion]); pp_newline eval ();
  pp_divide ~indent eval ();
  begin match fusion.L.denominator with
  | [] → ()
  | denominator →
     scalar_expression ~decl ~eval 4 denominator_name denominator
  end;
  match wfs.(0).spin with
  | Coupling.Scalar →
     contract_indices ~decl ~eval 2 [] wfs contractees
  | Coupling.Spinor | Coupling.ConjSpinor | Coupling.Majorana →
     let idx = index_spinor in
     fprintf eval "%*s@[<2>do␣%s␣=␣1,␣4@]" indent "" idx; pp_newline eval ();
     contract_indices ~decl ~eval 4 [idx] wfs contractees;
     fprintf eval "%*send␣do@]" indent ""; pp_newline eval ()
  | Coupling.Vector | Coupling.Massive_Vector →
     let idx = index_variable 1 in
     fprintf eval "%*s@[<2>do␣%s␣=␣0,␣3@]" indent "" idx; pp_newline eval ();
     contract_indices ~decl ~eval 4 [idx] wfs contractees;
     fprintf eval "%*send␣do@]" indent ""; pp_newline eval ()
  | Coupling.Tensor_2 →
     let idx1 = index_variable (UFOx.Index.pack 1 1)
     and idx2 = index_variable (UFOx.Index.pack 1 2) in
     fprintf eval "%*s@[<2>do␣%s␣=␣0,␣3@]" indent "" idx1;
     pp_newline eval ();
     fprintf eval "%*s@[<2>do␣%s␣=␣0,␣3@]" (indent + 2) "" idx2;
     pp_newline eval ();
     contract_indices ~decl ~eval 6 [idx1; idx2] wfs contractees;
     fprintf eval "%*send␣do@]" (indent + 2) ""; pp_newline eval ();
     fprintf eval "%*send␣do@]" indent ""; pp_newline eval ()
  | Coupling.Vectorspinor →
     failwith "external_wf_loop:␣Vectorspinor␣not␣supported␣yet!"
  | Coupling.Maj_Ghost →
     failwith "external_wf_loop:␣unexpected␣Maj_Ghost"
  | Coupling.Tensor_1 →
     failwith "external_wf_loop:␣unexpected␣Tensor_1"
  | Coupling.BRS _ →
     failwith "external_wf_loop:␣unexpected␣BRS"

let fusions_to_fortran ~decl ~eval wfs ?(denominator = []) ?coupling fusions =
  local_vector_copies ~decl ~eval wfs;
  local_momentum_copies ~decl ~eval wfs;
  begin match denominator with
  | [] → ()
  | _ →
     fprintf decl "␣␣␣␣@[<2>complex(kind=default)␣::␣%s@]" denominator_name;
     pp_newline decl ()
  end;
  let max_dsv, indices_used, contractions =
    List.fold_left
      (contractees_of_fusion ~decl ~eval wfs)
      (0, Sets.Int.empty, [])
      fusions in
  Sets.Int.iter
    (fun index →
      fprintf decl "␣␣␣␣@[<2>integer␣::@␣%s@]" (index_variable index);
      pp_newline decl ())
    indices_used;
  begin match wfs.(0).spin with
  | Coupling.Spinor | Coupling.ConjSpinor | Coupling.Majorana →
     fprintf decl "␣␣␣␣@[<2>integer␣::@␣%s@]" index_spinor;
     pp_newline decl ()
  | _ → ()
  end;
  pp_divide ~indent : 4 eval ();
  let wfs0name = local_name wfs.(0) in
  fprintf eval "␣␣␣␣%s␣=␣0" wfs0name;
  pp_newline eval ();
  List.iter (external_wf_loop ~decl ~eval ~indent : 4 wfs) contractions;
  multiply_coupling_and_scalars eval coupling wfs;
  begin match denominator with
  | [] → ()
  | denominator →
     pp_divide ~indent : 4 eval ();
     fprintf eval "%*s!␣%s" 4 "" (L.to_string denominator);
     pp_newline eval ();
     scalar_expression ~decl ~eval 4 denominator_name denominator;
     fprintf eval
       "␣␣␣␣@[<2>%s␣=@␣%s␣/␣%s@]" wfs0name wfs0name denominator_name;
     pp_newline eval ()
  end;
  return_vector eval wfs
```

TODO: eventually, we should include the momentum among the arguments only if required. But this can wait for another day.

```
  let lorentz ff name spins lorentz =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    let wfs = wf_table spins in
    let n = Array.length wfs in
    printf "␣␣@[<4>pure␣function␣%s@␣(g,@␣" name;
    for i = 1 to n − 2 do
      printf "%s,@␣%s,@␣" wfs.(i).name wfs.(i).momentum
    done;
    printf "%s,@␣%s" wfs.(n − 1).name wfs.(n − 1).momentum;
    printf ")@␣result␣(%s)@]" wfs.(0).name; nl ();
    printf "␣␣␣␣@[<2>%s␣::@␣%s@]" wfs.(0).fortran_type wfs.(0).name; nl();
    printf "␣␣␣␣@[<2>complex(kind=default),@␣intent(in)␣::@␣g@]"; nl();
    for i = 1 to n − 1 do
      printf
        "␣␣␣␣@[<2>%s,␣intent(in)␣::␣%s@]"
        wfs.(i).fortran_type wfs.(i).name; nl();
    done;
    printf "␣␣␣␣@[<2>type(momentum),␣intent(in)␣::@␣%s" wfs.(1).momentum;
    for i = 2 to n − 1 do
      printf ",@␣%s" wfs.(i).momentum
    done;
    printf "@]";
    nl ();
    let width = 80 in (* get this from the default formatter instead! *)
    let decl_buf = Buffer.create 1024
    and eval_buf = Buffer.create 1024 in
    let decl = formatter_of_buffer ~width decl_buf
    and eval = formatter_of_buffer ~width eval_buf in
    fusions_to_fortran ~decl ~eval ~coupling :"g" wfs lorentz;
    pp_flush decl ();
    pp_flush eval ();
    pp_divide ~indent : 4 ff ();
    printf "%s" (Buffer.contents decl_buf);
    pp_divide ~indent : 4 ff ();
    printf "␣␣␣␣if␣(g␣==␣0)␣then"; nl ();
    printf "␣␣␣␣␣␣call␣set_zero␣(%s)" wfs.(0).name; nl ();
    printf "␣␣␣␣␣␣return"; nl ();
    printf "␣␣␣␣end␣if"; nl ();
    pp_divide ~indent : 4 ff ();
    printf "%s" (Buffer.contents eval_buf);
    printf "␣␣end␣function␣%s@]" name; nl ();
    Buffer.reset decl_buf;
    Buffer.reset eval_buf;
    ()

let use_variables ff parameter_module variables =
  let printf fmt = fprintf ff fmt
  and nl = pp_newline ff in
  match variables with
  | [] → ()
  | v :: v_list →
      printf "␣␣␣␣@[<2>use␣%s,␣only:␣%s" parameter_module v;
      List.iter (fun s → printf ",␣%s" s) v_list;
      printf "@]"; nl ()

let propagator ff name parameter_module variables
      (bra_spin, ket_spin) numerator denominator =
  let printf fmt = fprintf ff fmt
  and nl = pp_newline ff in
  let width = 80 in (* get this from the default formatter instead! *)
  let wf_name = spin_mnemonic ket_spin
  and wf_type = fortran_type ket_spin in
  let wfs = wf_table [| ket_spin; ket_spin |] in
  printf
    "␣␣@[<4>pure␣function␣pr_U_%s@␣(k2,␣%s,␣%s,␣%s2)"
    name mass_name width_name wf_name;
  printf "␣result␣(%s1)@]" wf_name; nl ();
  use_variables ff parameter_module variables;
  printf "␣␣␣␣%s␣::␣%s1" wf_type wf_name; nl ();
  printf "␣␣␣␣type(momentum),␣intent(in)␣::␣k2"; nl ();
  printf
    "␣␣␣␣real(kind=default),␣intent(in)␣::␣%s,␣%s"
    mass_name width_name; nl ();
  printf "␣␣␣␣%s,␣intent(in)␣::␣%s2" wf_type wf_name; nl ();
  let decl_buf = Buffer.create 1024
  and eval_buf = Buffer.create 1024 in
  let decl = formatter_of_buffer ~width decl_buf
  and eval = formatter_of_buffer ~width eval_buf in
  fusions_to_fortran ~decl ~eval wfs ~denominator numerator;
  pp_flush decl ();
  pp_flush eval ();
  pp_divide ~indent : 4 ff ();
  printf "%s" (Buffer.contents decl_buf);
  pp_divide ~indent : 4 ff ();
  printf "%s" (Buffer.contents eval_buf);
  printf "␣␣end␣function␣pr_U_%s@]" name; nl ();
  Buffer.reset decl_buf;
  Buffer.reset eval_buf;
  ()
```

```
let scale_coupling c g =
  if c = 1 then
    g
  else if c = -1 then
    "-" ^ g
  else
    Printf.sprintf "%d*%s" c g

let scale_coupling z g =
  format_complex_rational_factor z ^ g
```

As a prototypical example consider the vertex

$$\bar\psi \not{A} \psi = \operatorname{tr}\left(\psi \otimes \bar\psi \not{A}\right) \tag{19.26a}$$

encoded as `FFV` in the SM UFO file. This example is useful, because all three fields have different type and we can use the Fortran compiler to check our implementation.

In this case we need to generate the following function calls with the arguments in the following order

F12:	$\psi_1\bar\psi_2 \to A$	`FFV_p201(g,psi1,p1,psibar2,p2)`
F21:	$\bar\psi_1\psi_2 \to A$	`FFV_p201(g,psi2,p2,psibar1,p1)`
F23:	$\bar\psi_1 A_2 \to \bar\psi$	`FFV_p012(g,psibar1,p1,A2,p2)`
F32:	$A_1\bar\psi_2 \to \bar\psi$	`FFV_p012(g,psibar2,p2,A1,p1)`
F31:	$A_1\psi_2 \to \psi$	`FFV_p120(g,A1,p1,psi2,p2)`
F13:	$\psi_1 A_2 \to \psi$	`FFV_p120(g,A2,p2,psi1,p1)`

Fortunately, all Fermi signs have been taken care of by *Fusions* and we can concentrate on injecting the wave functions into the correct slots.
The other possible cases are

$$\bar\psi \not{A} \psi \tag{19.26b}$$

which would be encoded as `FVF` in a UFO file

F12:	$\bar\psi_1 A_2 \to \bar\psi$	`FVF_p201(g,psibar1,p1,A2,p2)`
F21:	$A_1\bar\psi_2 \to \bar\psi$	`FVF_p201(g,psibar2,p2,A1,p1)`
F23:	$A_1\psi_2 \to \psi$	`FVF_p012(g,A1,p1,psi2,p2)`
F32:	$\psi_1 A_2 \to \psi$	`FVF_p012(g,A2,p2,psi1,p1)`
F31:	$\psi_1\bar\psi_2 \to A$	`FVF_p120(g,psi1,p1,psibar2,p2)`
F13:	$\bar\psi_1\psi_2 \to A$	`FVF_p120(g,psi2,p2,psibar1,p1)`

and

$$\bar\psi \not{A} \psi = \operatorname{tr}\left(\not{A}\psi \otimes \bar\psi\right) , \tag{19.26c}$$

corresponding to `VFF`

F12:	$A_1\psi_2 \to \psi$	`VFF_p201(g,A1,p1,psi2,p2)`
F21:	$\psi_1 A_2 \to \psi$	`VFF_p201(g,A2,p2,psi1,p1)`
F23:	$\psi_1\bar\psi_2 \to A$	`VFF_p012(g,psi1,p1,psibar2,p2)`
F32:	$\bar\psi_1\psi_2 \to A$	`VFF_p012(g,psi2,p2,psibar1,p1)`
F31:	$\bar\psi_1 A_2 \to \bar\psi$	`VFF_p120(g,psibar1,p1,A2,p2)`
F13:	$A_1\bar\psi_2 \to \bar\psi$	`VFF_p120(g,psibar2,p2,A1,p1)`

Once the Majorana code generation is fully debugged, we should replace the lists by reverted lists everywhere in order to become a bit more efficient.

```
module P = Permutation.Default

let factor_cyclic f12__n =
  let f12__, fn = ThoList.split_last f12__n in
  let cyclic = ThoList.cycle_until fn (List.sort compare f12__n) in
  (P.of_list (List.map pred cyclic),
   P.of_lists (List.tl cyclic) f12__)

let ccs_to_string ccs =
  String.concat "" (List.map (fun (f, i) → Printf.sprintf "_c%x%x" i f) ccs)

let fusion_name v perm ccs =
  Printf.sprintf "%s_p%s%s" v (P.to_string perm) (ccs_to_string ccs)

let fuse_dirac c v s fl g wfs ps fusion =
  let g = scale_coupling c g
  and cyclic, factor = factor_cyclic fusion in
  let wfs_ps = List.map2 (fun wf p → (wf, p)) wfs ps in
  let args = P.list (P.inverse factor) wfs_ps in
  printf "@[<2>%s(@,%s" (fusion_name v cyclic []) g;
  List.iter (fun (wf, p) → printf ",@,%s,@,%s" wf p) args;
  printf ")@]"
```

We need to look at the permuted fermion lines in order to decide wether to apply charge conjugations.
It is not enough to look at the cyclic permutation used to move the fields into the correct arguments of the fusions . . .

```
  let map_indices perm unit =
    let pmap = IntPM.of_lists unit (P.list perm unit) in
    IntPM.apply pmap
```

. . . we also need to inspect the full permutation of the fields.

```
  let map_indices2 perm unit =
    let pmap =
      IntPM.of_lists unit (1 :: P.list (P.inverse perm) (List.tl unit)) in
    IntPM.apply pmap
```

This is a more direct implementation of the composition of *map_indices2* and *map_indices*, that is used in the unit tests.

```
  let map_indices_raw fusion =
    let unit = ThoList.range 1 (List.length fusion) in
    let f12__, fn = ThoList.split_last fusion in
    let fusion = fn :: f12__ in
    let map_index = IntPM.of_lists fusion unit in
    IntPM.apply map_index
```

Map the fermion line indices in *fl* according to *map_index*.

```
  let map_fermion_lines map_index fl =
    List.map (fun (i, f) → (map_index i, map_index f)) fl
```

Map the fermion line indices in *fl* according to *map_index*, but keep a copy of the original.

```
  let map_fermion_lines2 map_index fl =
    List.map (fun (i, f) → ((i, f), (map_index i, map_index f))) fl

  let permute_fermion_lines cyclic unit fl =
    map_fermion_lines (map_indices cyclic unit) fl

  let permute_fermion_lines2 cyclic factor unit fl =
    map_fermion_lines2
      (map_indices2 factor unit)
      (map_fermion_lines (map_indices cyclic unit) fl)
```

TODO: this needs more more work for the fully general case with 4-fermion operators involving Majoranas.

```
  let charge_conjugations fl2 =
    ThoList.filtermap
      (fun ((i, f), (i', f')) →
        match (i, f), (i', f') with
        | (1, 2), _ | (2, 1), _ → Some (f, i) (* $\chi^T\Gamma'$ *)
        | _, (2, 3) → Some (f, i) (* $\chi^T(C\Gamma')\chi$ *)
        | _ → None)
      fl2

  let charge_conjugations fl2 =
    ThoList.filtermap
      (fun ((i, f), (i', f')) →
          match (i, f), (i', f') with
        | _, (2, 3) → Some (f, i)
        | _ → None)
      fl2

  let fuse_majorana c v s fl g wfs ps fusion =
    let g = scale_coupling c g
    and cyclic, factor = factor_cyclic fusion in
    let wfs_ps = List.map2 (fun wf p → (wf, p)) wfs ps in
    let args = P.list (P.inverse factor) wfs_ps in
    let unit = ThoList.range 1 (List.length fusion) in
    let ccs =
      charge_conjugations (permute_fermion_lines2 cyclic factor unit fl) in
    printf "@[<2>%s(%s" (fusion_name v cyclic ccs) g;
    List.iter (fun (wf, p) → printf ",@,%s,@,%s" wf p) args;
    printf ")@]"

  let fuse c v s fl g wfs ps fusion =
    if List.exists is_majorana s then
      fuse_majorana c v s fl g wfs ps fusion
    else
      fuse_dirac c v s fl g wfs ps fusion

  let eps4_g4_g44_decl ff () =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    printf "␣␣@[<2>integer,@␣dimension(0:3)";
    printf ",@␣save,@␣private␣::@␣g4_@]"; nl ();
    printf "␣␣@[<2>integer,@␣dimension(0:3,0:3)";
    printf ",@␣save,@␣private␣::@␣g44_@]"; nl ();
    printf "␣␣@[<2>integer,@␣dimension(0:3,0:3,0:3,0:3)";
    printf ",@␣save,@␣private␣::@␣eps4_@]"; nl ()

  let eps4_g4_g44_init ff () =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    printf "␣␣@[<2>data␣g4_@␣␣␣␣␣␣␣␣␣␣␣␣/@␣␣1,␣-1,␣-1,␣-1␣/@]"; nl ();
    printf "␣␣@[<2>data␣g44_(0,:)@␣␣␣␣␣␣/@␣␣1,␣␣0,␣␣0,␣␣0␣/@]"; nl ();
    printf "␣␣@[<2>data␣g44_(1,:)@␣␣␣␣␣␣/@␣␣0,␣-1,␣␣0,␣␣0␣/@]"; nl ();
    printf "␣␣@[<2>data␣g44_(2,:)@␣␣␣␣␣␣/@␣␣0,␣␣0,␣-1,␣␣0␣/@]"; nl ();
    printf "␣␣@[<2>data␣g44_(3,:)@␣␣␣␣␣␣/@␣␣0,␣␣0,␣␣0,␣-1␣/@]"; nl ();
    for mu1 = 0 to 3 do
      for mu2 = 0 to 3 do
        for mu3 = 0 to 3 do
        printf "␣␣@[<2>data␣eps4_(%d,%d,%d,:)@␣/@␣" mu1 mu2 mu3;
        for mu4 = 0 to 3 do
          if mu4 ≠ 0 then
            printf ",@␣";
          let mus = [mu1; mu2; mu3; mu4] in
          if List.sort compare mus = [0; 1; 2; 3] then
            printf "%2d" (Combinatorics.sign mus)
          else
            printf "%2d" 0;
        done;
        printf "␣/@]";
        nl ()
        done
      done
    done

  let inner_product_functions ff () =
    let printf fmt = fprintf ff fmt
    and nl = pp_newline ff in
    printf "␣␣pure␣function␣g2_␣(p)␣result␣(p2)"; nl();
```

```
    printf "␣␣␣␣real(kind=default),␣dimension(0:3),␣intent(in)␣::␣p"; nl();
    printf "␣␣␣␣real(kind=default)␣::␣p2"; nl();
    printf "␣␣␣␣p2␣=␣p(0)*p(0)␣-␣p(1)*p(1)␣-␣p(2)*p(2)␣-␣p(3)*p(3)"; nl();
    printf "␣␣end␣function␣g2_"; nl();
    printf "␣␣pure␣function␣g12_␣(p1,␣p2)␣result␣(p12)"; nl();
    printf "␣␣␣␣real(kind=default),␣dimension(0:3),␣intent(in)␣::␣p1,␣p2"; nl();
    printf "␣␣␣␣real(kind=default)␣::␣p12"; nl();
    printf "␣␣␣␣p12␣=␣p1(0)*p2(0)␣-␣p1(1)*p2(1)␣-␣p1(2)*p2(2)␣-␣p1(3)*p2(3)"; nl();
    printf "␣␣end␣function␣g12_"; nl()

  module type Test =
    sig
      val suite : OUnit.test
    end

  module Test : Test =
    struct

      open OUnit

      let assert_mappings fusion =
        let unit = ThoList.range 1 (List.length fusion) in
        let cyclic, factor = factor_cyclic fusion in
        let raw = map_indices_raw fusion
        and map1 = map_indices cyclic unit
        and map2 = map_indices2 factor unit in
        let map i = map2 (map1 i) in
        assert_equal ~printer : (ThoList.to_string string_of_int)
          (List.map raw unit) (List.map map unit)

      let suite_mappings =
        "mappings" >:::

          [ "1<-2" >::
              (fun () →
                List.iter assert_mappings (Combinatorics.permute [1; 2; 3]));

            "1<-3" >::
              (fun () →
                List.iter assert_mappings (Combinatorics.permute [1; 2; 3; 4])) ]

      let suite =
        "UFO_targets" >:::
          [suite_mappings]

    end
end
```

—20— Hardcoded Targets

The following modules used to be submodules of [Targets], but this as become unwieldy over time.

20.1 Interface of Format_Fortran

Mimic parts of the *Format* API with support for Fortran style line continuation.

```
type formatter
```

```
val std_formatter : formatter
```

```
val fprintf : formatter → (α, Format.formatter, unit) format → α
val printf : (α, Format.formatter, unit) format → α
```

Start a new line, *not* a continuation!

```
val pp_newline : formatter → unit → unit
val newline : unit → unit
```

```
val pp_flush : formatter → unit → unit
val flush : unit → unit
```

```
val formatter_of_out_channel : ?width :int → out_channel → formatter
val formatter_of_buffer : ?width :int → Buffer.t → formatter
```

```
val pp_set_formatter_out_channel : formatter → ?width :int → out_channel → unit
val set_formatter_out_channel : ?width :int → out_channel → unit
```

This must be exposed for the benefit of *Targets.Make_Fortran().print_interface*, because somebody decided to use it for the K-matrix support. Is this really necessary?

```
val pp_switch_line_continuation : formatter → bool → unit
val switch_line_continuation : bool → unit
```

```
module Test : sig val suite : OUnit.test end
```

20.2 Implementation of Format_Fortran

```
let default_width = 80
```

```
let max_clines = ref (−1) (* 255 *)
exception Continuation_Lines of int
```

Fortran style line continuation:

```
type formatter =
  { formatter : Format.formatter;
    mutable current_cline : int;
    mutable width : int }
```

```
let formatter_of_formatter ?(width = default_width) ff =
  { formatter = ff;
    current_cline = 1;
    width = width }
```

Default function to output new lines.

```
let pp_output_function ff =
  fst (Format.pp_get_formatter_output_functions ff.formatter ())
```

Default function to output spaces (copied from `format.ml`).

```
let blank_line = String.make 80 ' '
let rec pp_display_blanks ff n =
  if n > 0 then
    if n ≤ 80 then
      pp_output_function ff blank_line 0 n
    else begin
        pp_output_function ff blank_line 0 80;
        pp_display_blanks ff (n − 80)
      end

let pp_display_newline ff =
  pp_output_function ff "\n" 0 1
```

ff.current_cline

- ≤ 0: not continuing: print a straight newline,
- > 0: continuing: append "␣&" until we run up to !*max_clines*. NB: !*max_clines* < 0 means *unlimited* continuation lines.

```
let pp_switch_line_continuation ff = function
  | false → ff.current_cline ← 0
  | true → ff.current_cline ← 1

let pp_fortran_newline ff () =
  if ff.current_cline > 0 then
    begin
      if !max_clines ≥ 0 ∧ ff.current_cline > !max_clines then
        raise (Continuation_Lines ff.current_cline)
      else
        begin
          pp_output_function ff "␣&" 0 2;
          ff.current_cline ← succ ff.current_cline
        end
    end;
  pp_display_newline ff

let pp_newline ff () =
  pp_switch_line_continuation ff false;
  Format.pp_print_newline ff.formatter ();
  pp_switch_line_continuation ff true
```

Make a formatter with default functions to output spaces and new lines.

```
let pp_setup ff =
  let formatter_out_functions =
    Format.pp_get_formatter_out_functions ff.formatter () in
  Format.pp_set_formatter_out_functions
    ff.formatter
    { formatter_out_functions with
      Format.out_newline = pp_fortran_newline ff;
      Format.out_spaces = pp_display_blanks ff };
  Format.pp_set_margin ff.formatter (ff.width − 2)

let std_formatter =
  let ff = formatter_of_formatter Format.std_formatter in
  pp_setup ff;
  ff

let formatter_of_out_channel ?(width = default_width) oc =
  let ff = formatter_of_formatter ~width (Format.formatter_of_out_channel oc) in
  pp_setup ff;
  ff

let formatter_of_buffer ?(width = default_width) b =
  let ff =
    { formatter = Format.formatter_of_buffer b;
      current_cline = 1;
      width = width } in
  pp_setup ff;
  ff

let pp_set_formatter_out_channel ff ?(width = default_width) oc =
  Format.pp_set_formatter_out_channel ff.formatter oc;
  ff.width ← width;
  pp_setup ff

let set_formatter_out_channel ?(width = default_width) oc =
  Format.pp_set_formatter_out_channel std_formatter.formatter oc;
  std_formatter.width ← width;
  pp_setup std_formatter

let fprintf ff fmt = Format.fprintf ff.formatter fmt
let pp_flush ff = Format.pp_print_flush ff.formatter

let printf fmt = fprintf std_formatter fmt
let newline = pp_newline std_formatter
let flush = pp_flush std_formatter
let switch_line_continuation = pp_switch_line_continuation std_formatter

module Test =
  struct

    open OUnit

    let input_line_opt ic =
      try
        Some (input_line ic)
      with
      | End_of_file → None

    let read_lines ic =
      let rec read_lines' acc =
        match input_line_opt ic with
        | Some line → read_lines' (line :: acc)
        | None → List.rev acc
      in
      read_lines' [ ]

    let lines_of_file filename =
      let ic = open_in filename in
      let lines = read_lines ic in
      close_in ic;
      lines

    let equal_or_dump_lines lhs rhs =
      if lhs = rhs then
        true
      else
        begin
          Printf.printf "Unexpected␣output:\n";
          List.iter (Printf.printf "<␣%s\n") lhs;
          List.iter (Printf.printf ">␣%s\n") rhs;
          false
        end

    let format_and_compare f expected () =
      bracket_tmpfile
        ~prefix :"omega-" ~suffix :".f90"
        (fun (name, oc) →
          (* There can be something left in the queue from OUnit! *)
          Format.print_flush ();
          f oc;
          close_out oc;
          (* OUnit uses Format.printf! *)
          Format.set_formatter_out_channel stdout;
          assert_bool "" (equal_or_dump_lines expected (lines_of_file name)))
        ()

    let suite =
      "Format_Fortran" >:::
        [ "formatter_of_out_channel" >::
          format_and_compare
            (fun oc →
              let ff = formatter_of_out_channel ~width : 20 oc in
              let nl = pp_newline ff in
              List.iter
                (fprintf ff)
                ["@[<2>lhs␣=␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"];
              nl ())
            [ "lhs␣=␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs" ];

          "formatter_of_buffer" >::
          format_and_compare
            (fun oc →
              let buffer = Buffer.create 1024 in
              let ff = formatter_of_buffer ~width : 20 buffer in
              let nl = pp_newline ff in
              List.iter
                (fprintf ff)
                ["␣␣@[<2>lhs␣=␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"];
              nl ();
              pp_flush ff ();
              let ff′ = formatter_of_out_channel ~width : 20 oc in
              fprintf ff′ "do␣mu␣=␣0,␣3"; pp_newline ff′ ();
              fprintf ff′ "%s" (Buffer.contents buffer);
              fprintf ff′ "end␣do";
              pp_newline ff′ ())
            [ "do␣mu␣=␣0,␣3";
              "␣␣lhs␣=␣rhs␣+␣rhs␣&";
              "␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣+␣rhs";
              "end␣do" ];

          "formatter_of_out_channel+indentation" >::
          format_and_compare
            (fun oc →
              let ff = formatter_of_out_channel ~width : 20 oc in
              let nl = pp_newline ff in
              List.iter
```

```
                (fprintf ff)
                ["␣␣@[<4>lhs␣=␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"];
              nl ())
            [ "␣␣lhs␣=␣rhs␣+␣rhs␣&";
              "␣␣␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣␣␣+␣rhs␣+␣rhs␣&";
              "␣␣␣␣␣␣+␣rhs" ];

        "set_formatter_out_channel" >::
          format_and_compare
            (fun oc →
              let nl = newline in
              set_formatter_out_channel ~width : 20 oc;
              List.iter
                printf
                ["@[<2>lhs␣=␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs";
                 "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"; "@␣+␣rhs"];
              nl ())
            [ "lhs␣=␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs␣+␣rhs␣&";
              "␣␣+␣rhs" ]; ]

  end
```

20.3 Interface of *Target_Fortran_Names*

These are the names of Fortran types, wave function variables and propagator functions. This must by synchronized among the `omegalib`, modern and vintage Fortran *Target* implementations.

```
module type T =
  sig
    val psi_type : string
    val psibar_type : string
    val chi_type : string
    val grav_type : string
    val psi_incoming : string
    val brs_psi_incoming : string
    val psibar_incoming : string
    val brs_psibar_incoming : string
    val chi_incoming : string
    val brs_chi_incoming : string
    val grav_incoming : string
    val psi_outgoing : string
    val brs_psi_outgoing : string
    val psibar_outgoing : string
    val brs_psibar_outgoing : string
    val chi_outgoing : string
    val brs_chi_outgoing : string
    val grav_outgoing : string
    val psi_propagator : string
    val psibar_propagator : string
    val chi_propagator : string
    val grav_propagator : string
    val psi_projector : string
    val psibar_projector : string
    val chi_projector : string
    val grav_projector : string
    val psi_gauss : string
    val psibar_gauss : string
    val chi_gauss : string
    val grav_gauss : string
    val use_module : string
    val require_library : string list
  end

module Dirac : T
module Majorana : T
```

20.4 Implementation of *Target_Fortran_Names*

```
module type T =
  sig
    val psi_type : string
    val psibar_type : string
    val chi_type : string
    val grav_type : string
    val psi_incoming : string
    val brs_psi_incoming : string
    val psibar_incoming : string
    val brs_psibar_incoming : string
    val chi_incoming : string
    val brs_chi_incoming : string
    val grav_incoming : string
    val psi_outgoing : string
    val brs_psi_outgoing : string
    val psibar_outgoing : string
    val brs_psibar_outgoing : string
    val chi_outgoing : string
    val brs_chi_outgoing : string
    val grav_outgoing : string
    val psi_propagator : string
    val psibar_propagator : string
    val chi_propagator : string
    val grav_propagator : string
    val psi_projector : string
    val psibar_projector : string
    val chi_projector : string
    val grav_projector : string
    val psi_gauss : string
    val psibar_gauss : string
    val chi_gauss : string
    val grav_gauss : string
    val use_module : string
    val require_library : string list
  end

module Dirac : T =
  struct

    let psi_type = "spinor"
    let psibar_type = "conjspinor"
    let chi_type = "???"
    let grav_type = "???"

    let psi_incoming = "u"
    let brs_psi_incoming = "brs_u"
    let psibar_incoming = "vbar"
    let brs_psibar_incoming = "brs_vbar"
    let chi_incoming = "???"
    let brs_chi_incoming = "???"
    let grav_incoming = "???"
    let psi_outgoing = "v"
    let brs_psi_outgoing = "brs_v"
    let psibar_outgoing = "ubar"
    let brs_psibar_outgoing = "brs_ubar"
    let chi_outgoing = "???"
    let brs_chi_outgoing = "???"
    let grav_outgoing = "???"

    let psi_propagator = "pr_psi"
    let psibar_propagator = "pr_psibar"
    let chi_propagator = "???"
    let grav_propagator = "???"

    let psi_projector = "pj_psi"
    let psibar_projector = "pj_psibar"
    let chi_projector = "???"
    let grav_projector = "???"

    let psi_gauss = "pg_psi"
    let psibar_gauss = "pg_psibar"
    let chi_gauss = "???"
    let grav_gauss = "???"

    let use_module = "omega95"
    let require_library =
      ["omega_spinors_2010_01_A"; "omega_spinor_cpls_2010_01_A"]
  end

module Majorana : T =
  struct

    let psi_type = "bispinor"
    let psibar_type = "bispinor"
    let chi_type = "bispinor"
    let grav_type = "vectorspinor"
```

JR sez' (regarding the Majorana Feynman rules): Because of our rules for fermions we are going to give all incoming fermions a u spinor and all outgoing fermions a v spinor, no matter whether they are Dirac fermions, antifermions or Majorana fermions. *(JR's probably right, but I need to check myself ...)*

```
    let psi_incoming = "u"
    let brs_psi_incoming = "brs_u"
    let psibar_incoming = "u"
    let brs_psibar_incoming = "brs_u"
    let chi_incoming = "u"
    let brs_chi_incoming = "brs_u"
    let grav_incoming = "ueps"

    let psi_outgoing = "v"
    let brs_psi_outgoing = "brs_v"
    let psibar_outgoing = "v"
    let brs_psibar_outgoing = "brs_v"
    let chi_outgoing = "v"
    let brs_chi_outgoing = "brs_v"
    let grav_outgoing = "veps"

    let psi_propagator = "pr_psi"
    let psibar_propagator = "pr_psi"
    let chi_propagator = "pr_psi"
    let grav_propagator = "pr_grav"
    let psi_projector = "pj_psi"
    let psibar_projector = "pj_psi"
    let chi_projector = "pj_psi"
    let grav_projector = "pj_grav"

    let psi_gauss = "pg_psi"
    let psibar_gauss = "pg_psi"
    let chi_gauss = "pg_psi"
    let grav_gauss = "pg_grav"

    let use_module = "omega95_bispinors"
    let require_library =
      ["omega_bispinors_2010_01_A"; "omega_bispinor_cpls_2010_01_A"]
  end
```

20.5 Interface of Target_Fortran

```
module Make : Target.Maker
module Make_Majorana : Target.Maker
```

20.6 Implementation of Target_Fortran

```
module Make_Fortran (Names : Target_Fortran_Names.T)
    (Vintage_Fermions : Targets_vintage.Fermion_Maker)
    (Fusion_Maker : Fusion.Maker) (P : Momentum.T) (M : Model.T) =
  struct

    let require_library =
      Names.require_library @
      [ "omega_vectors_2010_01_A"; "omega_polarizations_2010_01_A";
        "omega_couplings_2010_01_A"; "omega_color_2010_01_A";
        "omega_utils_2010_01_A" ]

    module Fermions = Vintage_Fermions(Names)

    module CM = Colorize.It(M)
    module SCM = Orders.Slice(Colorize.It(M))
    module F = Fusion_Maker(P)(M)

    module CF = Fusion.Multi(Fusion_Maker)(P)(M)
    type amplitudes = CF.amplitudes

    open Coupling
    open Format

    type output_mode =
      | Single_Function
      | Single_Module of int
      | Single_File of int
      | Multi_File of int

    let line_length = ref 80
    let continuation_lines = ref (−1) (* 255 *)
    let kind = ref "default"
    let fortran95 = ref true
    let module_name = ref "omega_amplitude"
    let output_mode = ref (Single_Module 10)
    let use_modules = ref [ ]
    let whizard = ref false
    let amp_triv = ref false
    let parameter_module = ref ""
    let md5sum = ref None
    let no_write = ref false
    let km_write = ref false
    let km_pure = ref false
    let km_2_write = ref false
    let km_2_pure = ref false
    let openmp = ref false
    let pure_unless_openmp = false

    let options = Options.create
      [ "90", Arg.Clear fortran95, "␣use␣only␣Fortran90␣features";
        "kind", Arg.String (fun s → kind := s),
        "kind␣real␣and␣complex␣kind␣(default:␣'" ^ !kind ^ "')";
        "width", Arg.Int (fun w → line_length := w), "n␣maximum␣line␣length";
        "continuation", Arg.Int (fun l → continuation_lines := l),
        "n␣maximum␣#␣of␣continuation␣lines";
        "module", Arg.String (fun s → module_name := s), "name␣module␣name";
        "single_function", Arg.Unit (fun () → output_mode := Single_Function),
        "␣compute␣the␣matrix␣element␣in␣one␣function";
        "split_function", Arg.Int (fun n → output_mode := Single_Module n),
        "size␣split␣the␣matrix␣element␣into␣small␣functions";
        "split_module", Arg.Int (fun n → output_mode := Single_File n),
        "size␣split␣the␣matrix␣element␣into␣small␣modules";
        "split_file", Arg.Int (fun n → output_mode := Multi_File n),
        "size␣split␣the␣matrix␣element␣into␣small␣files";
        "use", Arg.String (fun s → use_modules := s :: !use_modules),
        "name␣use␣module";
        "parameter_module", Arg.String (fun s → parameter_module := s),
        "name␣parameter_module";
        "md5sum", Arg.String (fun s → md5sum := Some s),
        "sum␣transfer␣MD5␣checksum";
        "whizard", Arg.Set whizard, "␣include␣WHIZARD␣interface";
        "amp_triv", Arg.Set amp_triv, "␣only␣print␣trivial␣amplitude";
        "no_write", Arg.Set no_write, "␣no␣'write'␣statements";
        "kmatrix_write", Arg.Set km_2_write, "␣write␣K␣matrix␣functions";
        "kmatrix_2_write", Arg.Set km_write, "␣write␣K␣matrix␣2␣functions";
        "kmatrix_write_pure", Arg.Set km_pure, "␣write␣K␣matrix␣pure␣functions";
        "kmatrix_2_write_pure", Arg.Set km_2_pure, "␣write␣Kmatrix2pure␣functions";
        "openmp", Arg.Set openmp, "␣activate␣OpenMP␣support␣in␣generated␣code"]
```

Fortran style line continuation:

```
let nl = Format_Fortran.newline

let print_list = function
  | [] → ()
  | a :: rest →
      print_string a;
      List.iter (fun s → printf ",@␣%s" s) rest
```

Variables and Declarations

"NC" is already used up in the module "constants":

```
let nc_parameter = "N_"
let omega_color_factor_abbrev = "OCF"
let openmp_tld_type = "thread_local_data"
let openmp_tld = "tld"

let flavors_symbol ?(decl = false) ?orders flavors =
  let flavors_all_orders = List.map SCM.flavor_all_orders flavors in
  let orders_tag =
    match orders with
    | None → ""
    | Some orders → SCM.orders_symbol orders in
  (if !openmp ∧ ¬ decl then openmp_tld ^ "%" else "" ) ^
```

```
    "oks_" ^ String.concat "_" (List.map CM.flavor_symbol flavors_all_orders) ^ orders_tag

  let p2s p =
    if p ≥ 0 ∧ p ≤ 9 then
      string_of_int p
    else if p ≤ 36 then
      String.make 1 (Char.chr (Char.code 'A' + p − 10))
    else
      "_"
```

There many similar functions for formatting momenta. This is grown historically and should be cleaned up!

Prefix with a "p" to make a variable name holding a four momentum.

```
  let format_momentum : int list → string =
    fun p →
    "p" ^ String.concat "" (List.map p2s p)
```

No prefix, to be used as part of a variable name holding a wavefunction.

```
  let format_p : F.wf → string =
    fun wf →
    String.concat "" (List.map p2s (F.momentum_list wf))

  let ext_momentum wf =
    match F.momentum_list wf with
    | [n] → n
    | _ → invalid_arg "Targets.Fortran.ext_momentum"

  module PSet = Set.Make (struct type t = int list let compare = compare end)
  module WFSet = Set.Make (struct type t = F.wf let compare = compare end)

  let variable ?(decl = false) wf =
    (if !openmp ∧ ¬ decl then openmp_tld ^ "%" else "")
    ^ "owf_" ^ SCM.flavor_symbol (F.flavor wf) ^ "_p" ^ format_p wf

  let momentum wf = "p" ^ format_p wf
  let spin wf = "s(" ^ string_of_int (ext_momentum wf) ^ ")"

  let format_multiple_variable ?(decl = false) wf i =
    variable ˜decl wf ^ "_X" ^ string_of_int i

  let multiple_variable ?(decl = false) amplitude dictionary wf =
    try
      format_multiple_variable ˜decl wf (dictionary amplitude wf)
    with
    | Not_found → variable wf

  let multiple_variables ?(decl = false) multiplicity wf =
    try
      List.map
        (format_multiple_variable ˜decl wf)
        (ThoList.range 1 (multiplicity wf))
    with
    | Not_found → [variable ˜decl wf]

  let declaration_chunk_size = 64

  let declare_list_chunk multiplicity t = function
    | [] → ()
    | wfs →
        printf "␣␣␣␣@[<2>%s␣::␣" t;
        print_list (ThoList.flatmap (multiple_variables ˜decl :true multiplicity) wfs); nl ()

  let declare_list multiplicity t = function
    | [] → ()
    | wfs →
        List.iter
```

```
          (declare_list_chunk multiplicity t)
          (ThoList.chopn declaration_chunk_size wfs)
type declarations =
    { scalars : F.wf list;
      spinors : F.wf list;
      conjspinors : F.wf list;
      realspinors : F.wf list;
      ghostspinors : F.wf list;
      vectorspinors : F.wf list;
      vectors : F.wf list;
      ward_vectors : F.wf list;
      massive_vectors : F.wf list;
      tensors_1 : F.wf list;
      tensors_2 : F.wf list;
      brs_scalars : F.wf list;
      brs_spinors : F.wf list;
      brs_conjspinors : F.wf list;
      brs_realspinors : F.wf list;
      brs_vectorspinors : F.wf list;
      brs_vectors : F.wf list;
      brs_massive_vectors : F.wf list }
let rec classify_wfs' acc = function
  | [] → acc
  | wf :: rest →
      classify_wfs'
        (match SCM.lorentz (F.flavor wf) with
        | Scalar → {acc with scalars = wf :: acc.scalars}
        | Spinor → {acc with spinors = wf :: acc.spinors}
        | ConjSpinor → {acc with conjspinors = wf :: acc.conjspinors}
        | Majorana → {acc with realspinors = wf :: acc.realspinors}
        | Maj_Ghost → {acc with ghostspinors = wf :: acc.ghostspinors}
        | Vectorspinor →
            {acc with vectorspinors = wf :: acc.vectorspinors}
        | Vector → {acc with vectors = wf :: acc.vectors}
        | Massive_Vector →
            {acc with massive_vectors = wf :: acc.massive_vectors}
        | Tensor_1 → {acc with tensors_1 = wf :: acc.tensors_1}
        | Tensor_2 → {acc with tensors_2 = wf :: acc.tensors_2}
        | BRS Scalar → {acc with brs_scalars = wf :: acc.brs_scalars}
        | BRS Spinor → {acc with brs_spinors = wf :: acc.brs_spinors}
        | BRS ConjSpinor → {acc with brs_conjspinors =
                                wf :: acc.brs_conjspinors}
        | BRS Majorana → {acc with brs_realspinors =
                              wf :: acc.brs_realspinors}
        | BRS Vectorspinor → {acc with brs_vectorspinors =
                                  wf :: acc.brs_vectorspinors}
        | BRS Vector → {acc with brs_vectors = wf :: acc.brs_vectors}
        | BRS Massive_Vector → {acc with brs_massive_vectors =
                                    wf :: acc.brs_massive_vectors}
        | BRS _ → invalid_arg "Targets.wfs_classify':␣not␣needed␣here")
        rest
let classify_wfs wfs = classify_wfs'
    { scalars = []; spinors = []; conjspinors = []; realspinors = [];
      ghostspinors = []; vectorspinors = []; vectors = [];
      ward_vectors = [];
      massive_vectors = []; tensors_1 = []; tensors_2 = [];
      brs_scalars = [] ; brs_spinors = []; brs_conjspinors = [];
      brs_realspinors = []; brs_vectorspinors = [];
      brs_vectors = []; brs_massive_vectors = []}
    wfs
```

Parameters

```
type α parameters =
    { real_singles : α list;
      real_arrays : (α × int) list;
      complex_singles : α list;
      complex_arrays : (α × int) list }

let rec classify_singles acc = function
  | [] → acc
  | Real p :: rest → classify_singles
        { acc with real_singles = p :: acc.real_singles } rest
  | Complex p :: rest → classify_singles
        { acc with complex_singles = p :: acc.complex_singles } rest

let rec classify_arrays acc = function
  | [] → acc
  | (Real_Array p, rhs) :: rest → classify_arrays
        { acc with real_arrays =
          (p, List.length rhs) :: acc.real_arrays } rest
  | (Complex_Array p, rhs) :: rest → classify_arrays
        { acc with complex_arrays =
          (p, List.length rhs) :: acc.complex_arrays } rest

let classify_parameters params =
  classify_arrays
    (classify_singles
       { real_singles = [];
         real_arrays = [];
         complex_singles = [];
         complex_arrays = [] }
       (List.map fst params.derived)) params.derived_arrays

let schisma = ThoList.chopn

let schisma_num i n l =
  ThoList.enumerate i (schisma n l)

let declare_parameters' t = function
  | [] → ()
  | plist →
      printf "␣␣@[<2>%s(kind=%s),␣public,␣save␣::␣" t !kind;
      print_list (List.map SCM.constant_symbol plist); nl ()

let declare_parameters t plist =
  List.iter (declare_parameters' t) plist

let declare_parameter_array t (p, n) =
  printf "␣␣@[<2>%s(kind=%s),␣dimension(%d),␣public,␣save␣::␣%s"
    t !kind n (SCM.constant_symbol p); nl ()
```

NB: we use *string_of_float* to make sure that a decimal point is included to make Fortran compilers happy.

```
let default_parameter (x, v) =
  printf "@␣%s␣=␣%s_%s" (SCM.constant_symbol x) (string_of_float v) !kind

let declare_default_parameters t = function
  | [] → ()
  | p :: plist →
      printf "␣␣@[<2>%s(kind=%s),␣public,␣save␣::" t !kind;
      default_parameter p;
      List.iter (fun p' → printf ","; default_parameter p') plist;
      nl ()

let format_constant = function
  | I → "(0,1)"
  | Integer c →
```

```
        if c < 0 then
          sprintf "(%d.0_%s)" c !kind
        else
          sprintf "%d.0_%s" c !kind
    | Float x →
        if x < 0. then
          "(" ^ string_of_float x ^ "_" ^ !kind ^ ")"
        else
          string_of_float x ^ "_" ^ !kind
    | _ → invalid_arg "format_constant"

let rec eval_parameter' = function
  | (I | Integer _ | Float _) as c →
      printf "%s" (format_constant c)
  | Atom x → printf "%s" (SCM.constant_symbol x)
  | Sum [] → printf "0.0_%s" !kind
  | Sum [x] → eval_parameter' x
  | Sum (x :: xs) →
       printf "@,("; eval_parameter' x;
       List.iter (fun x → printf "@,␣+␣"; eval_parameter' x) xs;
       printf ")"
  | Diff (x, y) →
       printf "@,("; eval_parameter' x;
       printf "␣-␣"; eval_parameter' y; printf ")"
  | Neg x → printf "@,(␣-␣"; eval_parameter' x; printf ")"
  | Prod [] → printf "1.0_%s" !kind
  | Prod [x] → eval_parameter' x
  | Prod (x :: xs) →
       printf "@,("; eval_parameter' x;
       List.iter (fun x → printf "␣*␣"; eval_parameter' x) xs;
       printf ")"
  | Quot (x, y) →
       printf "@,("; eval_parameter' x;
       printf "␣/␣"; eval_parameter' y; printf ")"
  | Rec x →
       printf "@,␣(1.0_%s␣/␣" !kind; eval_parameter' x; printf ")"
  | Pow (x, n) →
      printf "@,("; eval_parameter' x;
      if n < 0 then
        printf "**(%d)" n
      else
        printf "**%d" n;
      printf ")"
  | PowX (x, y) →
       printf "@,("; eval_parameter' x;
        printf "**"; eval_parameter' y; printf ")"
  | Sqrt x → printf "@,sqrt␣("; eval_parameter' x; printf ")"
  | Sin x → printf "@,sin␣("; eval_parameter' x; printf ")"
  | Cos x → printf "@,cos␣("; eval_parameter' x; printf ")"
  | Tan x → printf "@,tan␣("; eval_parameter' x; printf ")"
  | Cot x → printf "@,cot␣("; eval_parameter' x; printf ")"
  | Asin x → printf "@,asin␣("; eval_parameter' x; printf ")"
  | Acos x → printf "@,acos␣("; eval_parameter' x; printf ")"
  | Atan x → printf "@,atan␣("; eval_parameter' x; printf ")"
  | Atan2 (y, x) → printf "@,atan2␣("; eval_parameter' y;
       printf ",@␣"; eval_parameter' x; printf ")"
  | Sinh x → printf "@,sinh␣("; eval_parameter' x; printf ")"
  | Cosh x → printf "@,cosh␣("; eval_parameter' x; printf ")"
  | Tanh x → printf "@,tanh␣("; eval_parameter' x; printf ")"
  | Exp x → printf "@,exp␣("; eval_parameter' x; printf ")"
  | Log x → printf "@,log␣("; eval_parameter' x; printf ")"
```

```
    | Log10 x → printf "@,log10␣("; eval_parameter' x; printf ")"
    | Conj (Integer _ | Float _ as x) → eval_parameter' x
    | Conj x → printf "@,cconjg␣("; eval_parameter' x; printf ")"
    | Abs x → printf "@,abs␣("; eval_parameter' x; printf ")"

let strip_single_tag = function
    | Real x → x
    | Complex x → x

let strip_array_tag = function
    | Real_Array x → x
    | Complex_Array x → x

let eval_parameter (lhs, rhs) =
    let x = SCM.constant_symbol (strip_single_tag lhs) in
    printf "␣␣␣␣@[<2>%s␣=␣" x; eval_parameter' rhs; nl ()

let eval_para_list n l =
    printf "␣␣subroutine␣setup_parameters_%03d␣()" n; nl ();
    List.iter eval_parameter l;
    printf "␣␣end␣subroutine␣setup_parameters_%03d" n; nl ()

let eval_parameter_pair (lhs, rhs) =
    let x = SCM.constant_symbol (strip_array_tag lhs) in
    let _ = List.fold_left (fun i rhs' →
        printf "␣␣␣␣@[<2>%s(%d)␣=␣" x i; eval_parameter' rhs'; nl ();
        succ i) 1 rhs in
    ()

let eval_para_pair_list n l =
    printf "␣␣subroutine␣setup_parameters_%03d␣()" n; nl ();
    List.iter eval_parameter_pair l;
    printf "␣␣end␣subroutine␣setup_parameters_%03d" n; nl ()

let print_echo fmt p =
    let s = CM.constant_symbol p in
    printf "␣␣␣␣write␣(unit␣=␣*,␣fmt␣=␣fmt_%s)␣\"%s\",␣%s"
        fmt s s; nl ()

let print_echo_array fmt (p, n) =
    let s = CM.constant_symbol p in
    for i = 1 to n do
        printf "␣␣␣␣write␣(unit␣=␣*,␣fmt␣=␣fmt_%s_array)␣" fmt ;
        printf "\"%s\",␣%d,␣%s(%d)" s i s i; nl ()
    done

let contains params couplings =
    List.exists
        (fun (name, _) → List.mem (SCM.constant_symbol name) params)
        couplings.input

let rec depends_on params = function
    | I | Integer _ | Float _ → false
    | Atom name → List.mem (SCM.constant_symbol name) params
    | Sum es | Prod es →
        List.exists (depends_on params) es
    | Diff (e1, e2) | Quot (e1, e2) | PowX (e1, e2) →
        depends_on params e1 ∨ depends_on params e2
    | Neg e | Rec e | Pow (e, _) →
        depends_on params e
    | Sqrt e | Exp e | Log e | Log10 e
    | Sin e | Cos e | Tan e | Cot e
    | Asin e | Acos e | Atan e
    | Sinh e | Cosh e | Tanh e
    | Conj e | Abs e →
        depends_on params e
```

```
  | Atan2 (e1, e2) →
     depends_on params e1 ∨ depends_on params e2
let dependencies params couplings =
  if contains params couplings then
    List.rev
      (fst (List.fold_left
              (fun (deps, plist) (param, v) →
                match param with
                | Real name | Complex name →
                   if depends_on plist v then
                     ((param, v) :: deps, CM.constant_symbol name :: plist)
                   else
                     (deps, plist))
              ([], params) couplings.derived))
  else
    []
let dependencies_arrays params couplings =
  if contains params couplings then
    List.rev
      (fst (List.fold_left
              (fun (deps, plist) (param, vlist) →
                match param with
                | Real_Array name | Complex_Array name →
                   if List.exists (depends_on plist) vlist then
                     ((param, vlist) :: deps,
                      CM.constant_symbol name :: plist)
                   else
                     (deps, plist))
              ([], params) couplings.derived_arrays))
  else
    []
let parameters_to_fortran oc params =
  Format_Fortran.set_formatter_out_channel ~width :!line_length oc;
  let declarations = classify_parameters params in
  printf "module␣%s" !parameter_module; nl ();
  printf "␣␣use␣kinds"; nl ();
  printf "␣␣use␣constants"; nl ();
  printf "␣␣implicit␣none"; nl ();
  printf "␣␣private"; nl ();
  printf "␣␣@[<2>public␣::␣setup_parameters";
  printf ",@␣import_from_whizard";
  printf ",@␣model_update_alpha_s";
  if !no_write then begin
    printf "!␣No␣print_parameters";
  end else begin
    printf ",@␣print_parameters";
  end; nl ();
  declare_default_parameters "real" params.input;
  declare_parameters "real" (schisma 69 declarations.real_singles);
  List.iter (declare_parameter_array "real") declarations.real_arrays;
  declare_parameters "complex" (schisma 69 declarations.complex_singles);
  List.iter (declare_parameter_array "complex") declarations.complex_arrays;
  printf "␣␣interface␣cconjg"; nl ();
  printf "␣␣␣␣module␣procedure␣cconjg_real,␣cconjg_complex"; nl ();
  printf "␣␣end␣interface"; nl ();
  printf "␣␣private␣::␣cconjg_real,␣cconjg_complex"; nl ();
  printf "contains"; nl ();
  printf "␣␣function␣cconjg_real␣(x)␣result␣(xc)"; nl ();
  printf "␣␣␣␣real(kind=default),␣intent(in)␣::␣x"; nl ();
  printf "␣␣␣␣real(kind=default)␣::␣xc"; nl ();
  printf "␣␣␣␣xc␣=␣x"; nl ();
  printf "␣␣end␣function␣cconjg_real"; nl ();
  printf "␣␣function␣cconjg_complex␣(z)␣result␣(zc)"; nl ();
  printf "␣␣␣␣complex(kind=default),␣intent(in)␣::␣z"; nl ();
  printf "␣␣␣␣complex(kind=default)␣::␣zc"; nl ();
  printf "␣␣␣␣zc␣=␣conjg␣(z)"; nl ();
  printf "␣␣end␣function␣cconjg_complex"; nl ();
  printf "␣␣!␣derived␣parameters:"; nl ();
  let shredded = schisma_num 1 120 params.derived in
  let shredded_arrays = schisma_num 1 120 params.derived_arrays in
  let num_sub = List.length shredded in
  let num_sub_arrays = List.length shredded_arrays in
  List.iter (fun (i, l) → eval_para_list i l) shredded;
  List.iter (fun (i, l) → eval_para_pair_list (num_sub + i) l)
    shredded_arrays;
  printf "␣␣subroutine␣setup_parameters␣()"; nl ();
  for i = 1 to num_sub + num_sub_arrays do
    printf "␣␣␣␣call␣setup_parameters_%03d␣()" i; nl ();
  done;
  printf "␣␣end␣subroutine␣setup_parameters"; nl ();
  printf "␣␣subroutine␣import_from_whizard␣(par_array,␣scheme)"; nl ();
  printf
    "␣␣␣␣real(%s),␣dimension(%d),␣intent(in)␣::␣par_array"
    !kind (List.length params.input); nl ();
  printf "␣␣␣␣integer,␣intent(in)␣::␣scheme"; nl ();
  let i = ref 1 in
  List.iter
    (fun (p, _) →
      printf "␣␣␣␣%s␣=␣par_array(%d)" (SCM.constant_symbol p) !i; nl ();
      incr i)
    params.input;
  printf "␣␣␣␣call␣setup_parameters␣()"; nl ();
  printf "␣␣end␣subroutine␣import_from_whizard"; nl ();
  printf "␣␣subroutine␣model_update_alpha_s␣(alpha_s)"; nl ();
  printf "␣␣␣␣real(%s),␣intent(in)␣::␣alpha_s" !kind; nl ();
  begin match (dependencies ["aS"] params,
               dependencies_arrays ["aS"] params) with
  | [], [] →
     printf "␣␣␣␣!␣'aS'␣not␣among␣the␣input␣parameters"; nl ();
  | deps, deps_arrays →
     printf "␣␣␣␣aS␣=␣alpha_s"; nl ();
     List.iter eval_parameter deps;
     List.iter eval_parameter_pair deps_arrays
  end;
  printf "␣␣end␣subroutine␣model_update_alpha_s"; nl ();
  if !no_write then begin
    printf "!␣No␣print_parameters"; nl ();
  end else begin
    printf "␣␣subroutine␣print_parameters␣()"; nl ();
    printf "␣␣␣␣@[<2>character(len=*),␣parameter␣::";
    printf "@␣fmt_real␣=␣\"(A12,4X,'␣=␣',E25.18)\",";
    printf "@␣fmt_complex␣=␣\"(A12,4X,'␣=␣',E25.18,'␣+␣i*',E25.18)\",";
    printf "@␣fmt_real_array␣=␣\"(A12,'(',I2.2,')','␣=␣',E25.18)\",";
    printf "@␣fmt_complex_array␣=␣";
    printf "\"(A12,'(',I2.2,')','␣=␣',E25.18,'␣+␣i*',E25.18)\""; nl ();
    printf "␣␣␣␣@[<2>write␣(unit␣=␣*,␣fmt␣=␣\"(A)\")␣@,";
    printf "\"default␣values␣for␣the␣input␣parameters:\""; nl ();
    List.iter (fun (p, _) → print_echo "real" p) params.input;
    printf "␣␣␣␣@[<2>write␣(unit␣=␣*,␣fmt␣=␣\"(A)\")␣@,";
    printf "\"derived␣parameters:\""; nl ();
    List.iter (print_echo "real") declarations.real_singles;
```

```
      List.iter (print_echo "complex") declarations.complex_singles;
      List.iter (print_echo_array "real") declarations.real_arrays;
      List.iter (print_echo_array "complex") declarations.complex_arrays;
      printf "␣␣end␣subroutine␣print_parameters"; nl ();
    end;
    printf "end␣module␣%s" !parameter_module; nl ()
```

Run-Time Diagnostics

```
type diagnostic = All | Arguments | Momenta | Gauge

type diagnostic_mode = Off | Warn | Panic

let warn mode =
  match !mode with
  | Off → false
  | Warn → true
  | Panic → true

let panic mode =
  match !mode with
  | Off → false
  | Warn → false
  | Panic → true

let suffix mode =
  if panic mode then
    "panic"
  else
    "warn"

let diagnose_arguments = ref Off
let diagnose_momenta = ref Off
let diagnose_gauge = ref Off

let rec parse_diagnostic = function
  | All, panic →
      parse_diagnostic (Arguments, panic);
      parse_diagnostic (Momenta, panic);
      parse_diagnostic (Gauge, panic)
  | Arguments, panic →
      diagnose_arguments := if panic then Panic else Warn
  | Momenta, panic →
      diagnose_momenta := if panic then Panic else Warn
  | Gauge, panic →
      diagnose_gauge := if panic then Panic else Warn
```

If diagnostics are required, we have to switch off Fortran95 features like pure functions.

```
let parse_diagnostics = function
  | [] → ()
  | diagnostics →
      fortran95 := false;
      List.iter parse_diagnostic diagnostics
```

Amplitude

```
let declare_momenta_chunk = function
  | [] → ()
  | momenta →
      printf "␣␣␣␣@[<2>type(momentum)␣::␣";
      print_list (List.map format_momentum momenta); nl ()

let declare_momenta = function
```

```
    | [] → ()
    | momenta →
        List.iter
          declare_momenta_chunk
          (ThoList.chopn declaration_chunk_size momenta)

let declare_wavefunctions multiplicity wfs =
  let wfs′ = classify_wfs wfs in
  declare_list multiplicity ("complex(kind=" ^ !kind ^ ")")
    (wfs′.scalars @ wfs′.brs_scalars);
  declare_list multiplicity ("type(" ^ Names.psi_type ^ ")")
    (wfs′.spinors @ wfs′.brs_spinors);
  declare_list multiplicity ("type(" ^ Names.psibar_type ^ ")")
    (wfs′.conjspinors @ wfs′.brs_conjspinors);
  declare_list multiplicity ("type(" ^ Names.chi_type ^ ")")
    (wfs′.realspinors @ wfs′.brs_realspinors @ wfs′.ghostspinors);
  declare_list multiplicity ("type(" ^ Names.grav_type ^ ")") wfs′.vectorspinors;
  declare_list multiplicity "type(vector)" (wfs′.vectors @ wfs′.massive_vectors @
      wfs′.brs_vectors @ wfs′.brs_massive_vectors @ wfs′.ward_vectors);
  declare_list multiplicity "type(tensor2odd)" wfs′.tensors_1;
  declare_list multiplicity "type(tensor)" wfs′.tensors_2

let flavors a = F.incoming a @ F.outgoing a

let declare_brakets_chunk = function
  | [] → ()
  | amplitudes →
      printf "␣␣␣␣@[<2>complex(kind=%s)␣::␣" !kind;
      print_list (List.map (fun a → flavors_symbol ~decl :true (flavors a)) amplitudes); nl ()

let declare_brakets = function
  | [] → ()
  | amplitudes →
      List.iter
        declare_brakets_chunk
        (ThoList.chopn declaration_chunk_size amplitudes)

let print_variable_declarations amplitudes =
  let multiplicity = CF.multiplicity amplitudes
  and processes = CF.processes amplitudes in
  if ¬ !amp_triv then begin
    declare_momenta
      (PSet.elements
         (List.fold_left
            (fun set a →
              PSet.union set (List.fold_right
                                   (fun wf → PSet.add (F.momentum_list wf))
                                   (F.externals a) PSet.empty))
            PSet.empty processes));
    declare_momenta
      (PSet.elements
         (List.fold_left
            (fun set a →
              PSet.union set (List.fold_right
                                   (fun wf → PSet.add (F.momentum_list wf))
                                   (F.variables a) PSet.empty))
            PSet.empty processes));
    if !openmp then begin
      printf "␣␣type␣%s@[<2>" openmp_tld_type;
      nl ();
    end ;
    declare_wavefunctions multiplicity
      (WFSet.elements
         (List.fold_left
            (fun set a →
               WFSet.union set (List.fold_right WFSet.add (F.externals a) WFSet.empty))
            WFSet.empty processes));
    declare_wavefunctions multiplicity
      (WFSet.elements
         (List.fold_left
            (fun set a →
               WFSet.union set (List.fold_right WFSet.add (F.variables a) WFSet.empty))
            WFSet.empty processes));
    declare_brakets processes;
    if !openmp then begin
      printf "@]␣␣end␣type␣%s\n" openmp_tld_type;
      printf "␣␣type(%s)␣::␣%s" openmp_tld_type openmp_tld;
      nl ();
    end;
  end
```

print_current is the most important function that has to match the functions in `omega95` (see appendix AB). It offers plentiful opportunities for making mistakes, in particular those related to signs. We start with a few auxiliary functions:

```
  let children2 rhs =
    match F.children rhs with
    | [wf1; wf2] → (wf1, wf2)
    | _ → failwith "Targets.children2:␣can't␣happen"

  let children3 rhs =
    match F.children rhs with
    | [wf1; wf2; wf3] → (wf1, wf2, wf3)
    | _ → invalid_arg "Targets.children3:␣can't␣happen"

  let print_current amplitude dictionary rhs =
    let module Vintage = Targets_vintage.Make_Fortran(Names)(Vintage_Fermions)(Fusion_Maker)(P)(M) in
    match F.coupling rhs with
    | V3 (vertex, fusion, constant) →
        Vintage.print_current_V3 multiple_variable momentum amplitude dictionary rhs vertex fusion constant
    | V4 (vertex, fusion, constant) →
        Vintage.print_current_V4 multiple_variable momentum amplitude dictionary rhs vertex fusion constant
```

This reproduces the hack on page 508 and gives the correct results up to quartic vertices. Make sure that it is also correct in light of (20.1), i. e.

$$\mathrm{i}T = \mathrm{i}^{\#\text{vertices}}\mathrm{i}^{\#\text{propagators}}\cdots = \mathrm{i}^{n-2}\mathrm{i}^{n-3}\cdots = -\mathrm{i}(-1)^n\cdots$$

```
    | Vn (UFO (c, v, s, fl, color), fusion, constant) →
       if Birdtracks.is_unit color then
         let g = CM.constant_symbol constant
         and chn = F.children rhs in
         let wfs = List.map (multiple_variable amplitude dictionary) chn
         and ps = List.map momentum chn in
         let n = List.length fusion in
         let eps = if n mod 2 = 0 then -1 else 1 in
         printf "@,␣%s␣" (if (eps × F.sign rhs) < 0 then "-" else "+");
         UFO.Targets.Fortran.fuse c v s fl g wfs ps fusion
       else
         failwith "print_current:␣nontrivial␣color␣structure"

  let print_propagator f p m gamma =
    let minus_third = "(-1.0_" ^ !kind ^ "/3.0_" ^ !kind ^ ")" in
    let w =
      begin match SCM.width f with
        | Vanishing | Fudged → "0.0_" ^ !kind
        | Constant | Complex_Mass → gamma
```

```
          | Timelike → "wd_tl(" ^ p ^ "," ^ gamma ^ ")"
          | Running → "wd_run(" ^ p ^ "," ^ m ^ "," ^ gamma ^ ")"
          | Custom f → f ^ "(" ^ p ^ "," ^ gamma ^ ")"
        end in
      let cms =
        begin match SCM.width f with
          | Complex_Mass → ".true."
          | _ → ".false."
        end in
      match SCM.propagator f with
        | Prop_Scalar →
          printf "pr_phi(%s,%s,%s," p m w
        | Prop_Col_Scalar →
          printf "%s␣*␣pr_phi(%s,%s,%s," minus_third p m w
        | Prop_Ghost → printf "(0,1)␣*␣pr_phi(%s,␣%s,␣%s," p m w
        | Prop_Spinor →
          printf "%s(%s,%s,%s,%s," Names.psi_propagator p m w cms
        | Prop_ConjSpinor →
          printf "%s(%s,%s,%s,%s," Names.psibar_propagator p m w cms
        | Prop_Majorana →
          printf "%s(%s,%s,%s,%s," Names.chi_propagator p m w cms
        | Prop_Col_Majorana →
          printf "%s␣*␣%s(%s,%s,%s,%s," minus_third Names.chi_propagator p m w cms
        | Prop_Unitarity →
          printf "pr_unitarity(%s,%s,%s,%s," p m w cms
        | Prop_Col_Unitarity →
          printf "%s␣*␣pr_unitarity(%s,%s,%s,%s," minus_third p m w cms
        | Prop_Feynman →
          printf "pr_feynman(%s," p
        | Prop_Col_Feynman →
          printf "%s␣*␣pr_feynman(%s," minus_third p
        | Prop_Gauge xi →
          printf "pr_gauge(%s,%s," p (SCM.gauge_symbol xi)
        | Prop_Rxi xi →
          printf "pr_rxi(%s,%s,%s,%s," p m w (SCM.gauge_symbol xi)
        | Prop_Tensor_2 →
          printf "pr_tensor(%s,%s,%s," p m w
        | Prop_Tensor_pure →
          printf "pr_tensor_pure(%s,%s,%s," p m w
        | Prop_Vector_pure →
          printf "pr_vector_pure(%s,%s,%s," p m w
        | Prop_Vectorspinor →
          printf "pr_grav(%s,%s,%s," p m w
        | Aux_Scalar | Aux_Spinor | Aux_ConjSpinor | Aux_Majorana
        | Aux_Vector | Aux_Tensor_1 → printf "("
        | Aux_Col_Scalar | Aux_Col_Vector | Aux_Col_Tensor_1 → printf "%s␣*␣(" minus_third
        | Only_Insertion → printf "("
        | Prop_UFO name →
          printf "pr_U_%s(%s,%s,%s," name p m w

    let print_projector f p m gamma =
      let minus_third = "(-1.0_" ^ !kind ^ "/3.0_" ^ !kind ^ ")" in
      match SCM.propagator f with
      | Prop_Scalar →
          printf "pj_phi(%s,%s," m gamma
      | Prop_Col_Scalar →
          printf "%s␣*␣pj_phi(%s,%s," minus_third m gamma
      | Prop_Ghost →
          printf "(0,1)␣*␣pj_phi(%s,%s," m gamma
      | Prop_Spinor →
          printf "%s(%s,%s,%s," Names.psi_projector p m gamma
      | Prop_ConjSpinor →
          printf "%s(%s,%s,%s," Names.psibar_projector p m gamma
      | Prop_Majorana →
          printf "%s(%s,%s,%s," Names.chi_projector p m gamma
      | Prop_Col_Majorana →
          printf "%s␣*␣%s(%s,%s,%s," minus_third Names.chi_projector p m gamma
      | Prop_Unitarity →
          printf "pj_unitarity(%s,%s,%s," p m gamma
      | Prop_Col_Unitarity →
          printf "%s␣*␣pj_unitarity(%s,%s,%s," minus_third p m gamma
      | Prop_Feynman | Prop_Col_Feynman →
          invalid_arg "no␣on-shell␣Feynman␣propagator!"
      | Prop_Gauge _ →
          invalid_arg "no␣on-shell␣massless␣gauge␣propagator!"
      | Prop_Rxi _ →
          invalid_arg "no␣on-shell␣Rxi␣propagator!"
      | Prop_Vectorspinor →
          printf "pj_grav(%s,%s,%s," p m gamma
      | Prop_Tensor_2 →
          printf "pj_tensor(%s,%s,%s," p m gamma
      | Prop_Tensor_pure →
          invalid_arg "no␣on-shell␣pure␣Tensor␣propagator!"
      | Prop_Vector_pure →
          invalid_arg "no␣on-shell␣pure␣Vector␣propagator!"
      | Aux_Scalar | Aux_Spinor | Aux_ConjSpinor | Aux_Majorana
      | Aux_Vector | Aux_Tensor_1 → printf "("
      | Aux_Col_Scalar | Aux_Col_Vector | Aux_Col_Tensor_1 → printf "%s␣*␣(" minus_third
      | Only_Insertion → printf "("
      | Prop_UFO name →
         invalid_arg "no␣on␣shell␣UFO␣propagator"

    let print_gauss f p m gamma =
      let minus_third = "(-1.0_" ^ !kind ^ "/3.0_" ^ !kind ^ ")" in
      match SCM.propagator f with
      | Prop_Scalar →
          printf "pg_phi(%s,%s,%s," p m gamma
      | Prop_Ghost →
          printf "(0,1)␣*␣pg_phi(%s,%s,%s," p m gamma
      | Prop_Spinor →
          printf "%s(%s,%s,%s," Names.psi_projector p m gamma
      | Prop_ConjSpinor →
          printf "%s(%s,%s,%s," Names.psibar_projector p m gamma
      | Prop_Majorana →
          printf "%s(%s,%s,%s," Names.chi_projector p m gamma
      | Prop_Col_Majorana →
          printf "%s␣*␣%s(%s,%s,%s," minus_third Names.chi_projector p m gamma
      | Prop_Unitarity →
          printf "pg_unitarity(%s,%s,%s," p m gamma
      | Prop_Feynman | Prop_Col_Feynman →
          invalid_arg "no␣on-shell␣Feynman␣propagator!"
      | Prop_Gauge _ →
          invalid_arg "no␣on-shell␣massless␣gauge␣propagator!"
      | Prop_Rxi _ →
          invalid_arg "no␣on-shell␣Rxi␣propagator!"
      | Prop_Tensor_2 →
          printf "pg_tensor(%s,%s,%s," p m gamma
      | Prop_Tensor_pure →
          invalid_arg "no␣pure␣tensor␣propagator!"
      | Prop_Vector_pure →
          invalid_arg "no␣pure␣vector␣propagator!"
      | Aux_Scalar | Aux_Spinor | Aux_ConjSpinor | Aux_Majorana
```

```
    | Aux_Vector | Aux_Tensor_1 → printf "("
    | Only_Insertion → printf "("
    | Prop_UFO name →
        invalid_arg "no␣UFO␣gauss␣insertion"
    | _ → invalid_arg "targets:print_gauss:␣not␣available"

let print_fusion_diagnostics amplitude dictionary fusion =
  if warn diagnose_gauge then begin
    let lhs = F.lhs fusion in
    let f = F.flavor lhs
    and v = variable lhs
    and p = momentum lhs in
    let mass = SCM.mass_symbol f in
    match SCM.propagator f with
    | Prop_Gauge _ | Prop_Feynman
    | Prop_Rxi _ | Prop_Unitarity →
        printf "␣␣␣␣␣␣@[<2>%s␣=" v;
        List.iter (print_current amplitude dictionary) (F.rhs fusion); nl ();
        begin match SCM.goldstone f with
        | None →
            printf "␣␣␣␣␣␣call␣omega_ward_%s(\"%s\",%s,%s,%s)"
              (suffix diagnose_gauge) v mass p v; nl ()
        | Some (g, phase) →
            let gv = SCM.flavor_symbol g ^ "_" ^ format_p lhs in
            printf "␣␣␣␣␣␣call␣omega_slavnov_%s"
              (suffix diagnose_gauge);
            printf "(@[\"%s\",%s,%s,%s,@,%s*%s)"
              v mass p v (format_constant phase) gv; nl ()
        end
    | _ → ()
  end

let print_fusion amplitude dictionary fusion =
  let lhs = F.lhs fusion in
  let f = F.flavor lhs in
  printf "␣␣␣␣␣␣@[<2>%s␣=@,␣" (multiple_variable amplitude dictionary lhs);
  if F.on_shell amplitude lhs then
    print_projector f (momentum lhs)
      (SCM.mass_symbol f) (SCM.width_symbol f)
  else
    if F.is_gauss amplitude lhs then
      print_gauss f (momentum lhs)
        (SCM.mass_symbol f) (SCM.width_symbol f)
    else
      print_propagator f (momentum lhs)
        (SCM.mass_symbol f) (SCM.width_symbol f);
  List.iter (print_current amplitude dictionary) (F.rhs fusion);
  printf ")"; nl ()

let print_momenta seen_momenta amplitude =
  List.fold_left (fun seen f →
    let wf = F.lhs f in
    let p = F.momentum_list wf in
    if ¬ (PSet.mem p seen) then begin
      let rhs1 = List.hd (F.rhs f) in
      printf "␣␣␣␣%s␣=␣%s" (momentum wf)
        (String.concat "␣+␣"
            (List.map momentum (F.children rhs1))); nl ()
    end;
    PSet.add p seen)
    seen_momenta (F.fusions amplitude)

let print_fusions dictionary fusions =
  List.iter
    (fun (f, amplitude) →
      print_fusion_diagnostics amplitude dictionary f;
      print_fusion amplitude dictionary f)
    fusions
```

The following will need a bit more work, because the decision when to *reverse_braket* for UFO models with Majorana fermions needs collaboration from *UFO.Targets.Fortran.fuse* which is called by *print_current*. See the function *UFO_targets.Fortran.jrr_print_majorana_current_transposing* for illustration (the function is never used and only for documentation).

```
let spins_of_rhs rhs =
  List.map (fun wf → SCM.lorentz (F.flavor wf)) (F.children rhs)

let spins_of_ket ket =
  match ThoList.uniq (List.map spins_of_rhs ket) with
  | [spins] → spins
  | [] → failwith "Targets.Fortran.spins_of_ket:␣empty"
  | _ → [] (* HACK! *)

let print_braket amplitude dictionary name braket =
  let bra = F.bra braket
  and ket = F.ket braket in
  let spin_bra = SCM.lorentz (F.flavor bra)
  and spins_ket = spins_of_ket ket in
  let vintage = true (* F.vintage *) in
  printf "␣␣␣␣␣␣@[<2>%s␣=@␣%s@,␣+␣" name name;
  if Fermions.reverse_braket vintage spin_bra spins_ket then
    begin
      printf "@,(";
      List.iter (print_current amplitude dictionary) ket;
      printf ")*%s" (multiple_variable amplitude dictionary bra)
    end
  else
    begin
      printf "%s*@,(" (multiple_variable amplitude dictionary bra);
      List.iter (print_current amplitude dictionary) ket;
      printf ")"
    end;
  nl ()
```

$$\mathrm{i}T = \mathrm{i}^{\#\text{vertices}}\mathrm{i}^{\#\text{propagators}}\cdots = \mathrm{i}^{n-2}\mathrm{i}^{n-3}\cdots = -\mathrm{i}(-1)^n\cdots \tag{20.1}$$

tho : we write some brakets twice using different names. Is it useful to cache them?

```
let print_braket_slice ?orders dictionary amplitude brakets =
  let name = flavors_symbol ?orders (flavors amplitude) in
  printf "␣␣␣␣␣␣%s␣=␣0" name; nl ();
  List.iter (print_braket amplitude dictionary name) brakets;
  let n = List.length (F.externals amplitude) in
  if n mod 2 = 0 then begin
    printf "␣␣␣␣␣␣@[<2>%s␣=@,␣-␣%s␣!␣%d␣vertices,␣%d␣propagators"
      name name (n − 2) (n − 3); nl ()
  end else begin
    printf "␣␣␣␣␣␣!␣%s␣=␣%s␣!␣%d␣vertices,␣%d␣propagators"
      name name (n − 2) (n − 3); nl ()
  end;
  let s = F.symmetry amplitude in
  if s > 1 then
    printf "␣␣␣␣␣␣@[<2>%s␣=@,␣%s@,␣/␣sqrt(%d.0_%s)␣!␣symmetry␣factor" name name s !kind
  else
    printf "␣␣␣␣␣␣!␣unit␣symmetry␣factor";
  nl ()
let print_brakets dictionary amplitude =
  match F.brakets amplitude with
  | [([], brakets)] → print_braket_slice dictionary amplitude brakets
  | [(orders, brakets)] →
    Printf.eprintf "omega:␣implementation␣of␣coupling␣order␣slices␣not␣complete␣yet!\n";
    print_braket_slice ~orders dictionary amplitude brakets
  | slices →
     Printf.eprintf "omega:␣implementation␣of␣coupling␣order␣slices␣not␣complete␣yet!\n";
     List.iter
       (fun (orders, brakets) → print_braket_slice ~orders dictionary amplitude brakets)
       slices
let print_incoming wf =
  let p = momentum wf
  and s = spin wf
  and f = F.flavor wf in
  let m = SCM.mass_symbol f in
  match SCM.lorentz f with
  | Scalar → printf "1"
  | BRS Scalar → printf "(0,-1)␣*␣(%s␣*␣%s␣-␣%s**2)" p p m
  | Spinor →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.psi_incoming m p s
  | BRS Spinor →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.brs_psi_incoming m p s
  | ConjSpinor →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.psibar_incoming m p s
  | BRS ConjSpinor →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.brs_psibar_incoming m p s
  | Majorana →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.chi_incoming m p s
  | Maj_Ghost → printf "ghost␣(%s,␣-␣%s,␣%s)" m p s
  | BRS Majorana →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.brs_chi_incoming m p s
  | Vector | Massive_Vector →
      printf "eps␣(%s,␣-␣%s,␣%s)" m p s
  | BRS Vector | BRS Massive_Vector → printf
        "(0,1)␣*␣(%s␣*␣%s␣-␣%s**2)␣*␣eps␣(%s,␣-%s,␣%s)" p p m m p s
  | Vectorspinor | BRS Vectorspinor →
      printf "%s␣(%s,␣-␣%s,␣%s)" Names.grav_incoming m p s
  | Tensor_1 → invalid_arg "Tensor_1␣only␣internal"
  | Tensor_2 → printf "eps2␣(%s,␣-␣%s,␣%s)" m p s
  | _ → invalid_arg "no␣such␣BRST␣transformations"
let print_outgoing wf =
  let p = momentum wf
  and s = spin wf
  and f = F.flavor wf in
  let m = SCM.mass_symbol f in
  match SCM.lorentz f with
  | Scalar → printf "1"
  | BRS Scalar → printf "(0,-1)␣*␣(%s␣*␣%s␣-␣%s**2)" p p m
  | Spinor →
      printf "%s␣(%s,␣%s,␣%s)" Names.psi_outgoing m p s
  | BRS Spinor →
      printf "%s␣(%s,␣%s,␣%s)" Names.brs_psi_outgoing m p s
  | ConjSpinor →
      printf "%s␣(%s,␣%s,␣%s)" Names.psibar_outgoing m p s
  | BRS ConjSpinor →
      printf "%s␣(%s,␣%s,␣%s)" Names.brs_psibar_outgoing m p s
  | Majorana →
      printf "%s␣(%s,␣%s,␣%s)" Names.chi_outgoing m p s
  | BRS Majorana →
      printf "%s␣(%s,␣%s,␣%s)" Names.brs_chi_outgoing m p s
  | Maj_Ghost → printf "ghost␣(%s,␣%s,␣%s)" m p s
  | Vector | Massive_Vector →
      printf "conjg␣(eps␣(%s,␣%s,␣%s))" m p s
  | BRS Vector | BRS Massive_Vector → printf
        "(0,1)␣*␣(%s*%s-%s**2)␣*␣(conjg␣(eps␣(%s,␣%s,␣%s)))" p p m m p s
  | Vectorspinor | BRS Vectorspinor →
      printf "%s␣(%s,␣%s,␣%s)" Names.grav_incoming m p s
  | Tensor_1 → invalid_arg "Tensor_1␣only␣internal"
  | Tensor_2 → printf "conjg␣(eps2␣(%s,␣%s,␣%s))" m p s
  | BRS _ → invalid_arg "no␣such␣BRST␣transformations"

let print_external_momenta amplitude =
  let externals =
    List.combine
      (F.externals amplitude)
      (List.map (fun _ → true) (F.incoming amplitude) @
       List.map (fun _ → false) (F.outgoing amplitude)) in
  List.iter (fun (wf, incoming) →
    if incoming then
      printf "␣␣␣␣%s␣=␣-␣k(:,%d)␣!␣incoming"
        (momentum wf) (ext_momentum wf)
    else
      printf "␣␣␣␣%s␣=␣␣␣k(:,%d)␣!␣outgoing"
        (momentum wf) (ext_momentum wf); nl ()) externals

let print_externals seen_wfs amplitude =
  let externals =
    List.combine
      (F.externals amplitude)
      (List.map (fun _ → true) (F.incoming amplitude) @
       List.map (fun _ → false) (F.outgoing amplitude)) in
  List.fold_left (fun seen (wf, incoming) →
    if ¬ (WFSet.mem wf seen) then begin
      printf "␣␣␣␣␣␣@[<2>%s␣=@,␣" (variable wf);
      (if incoming then print_incoming else print_outgoing) wf; nl ()
    end;
    WFSet.add wf seen) seen_wfs externals

let flavors_to_string flavors =
  String.concat "␣" (List.map (fun f → CM.flavor_to_string (SCM.flavor_all_orders f)) flavors)

let process_to_string amplitude =
  flavors_to_string (F.incoming amplitude) ^ "␣->␣" ^
  flavors_to_string (F.outgoing amplitude)

let flavors_sans_color_to_string flavors =
  String.concat "␣" (List.map M.flavor_to_string flavors)

let process_sans_color_to_string (fin, fout) =
  flavors_sans_color_to_string fin ^ "␣->␣" ^
  flavors_sans_color_to_string fout

let print_fudge_factor amplitude =
  let name = flavors_symbol (flavors amplitude) in
  List.iter (fun wf →
    let p = momentum wf
    and f = F.flavor wf in
    match SCM.width f with
    | Fudged →
        let m = SCM.mass_symbol f
        and w = SCM.width_symbol f in
        printf "␣␣␣␣␣␣if␣(%s␣>␣0.0_%s)␣then" w !kind; nl ();
        printf "␣␣␣␣␣␣␣␣@[<2>%s␣=␣%s@␣*␣(%s*%s␣-␣%s**2)"
```

```
          name name p p m;
        printf "@␣/␣cmplx␣(%s*%s␣-␣%s**2,␣%s*%s,␣kind=%s)"
          p p m m w !kind; nl ();
        printf "␣␣␣␣␣␣end␣if"; nl ()
  | _ → ()) (F.s_channel amplitude)

let num_helicities amplitudes =
  List.length (CF.helicities amplitudes)

let num_coupling_orders amplitudes =
  match CF.coupling_orders amplitudes with
  | None → 0
  | Some (co_list, _) → List.length co_list

let num_coupling_order_powers amplitudes =
  match CF.coupling_orders amplitudes with
  | None → 0
  | Some (_, powers) → List.length powers
```

Spin, Flavor & Color Tables

The following abomination is required to keep the number of continuation lines as low as possible. FORTRAN77-style `DATA` statements are actually a bit nicer here, but they are not available for *constant* arrays.

We used to have a more elegant design with a sentinel 0 added to each initializer, but some revisions of the Compaq/Digital Compiler have a bug that causes them to reject this variant.

The actual table writing code using `reshape` should be factored, since it's the same algorithm every time.

```
let print_integer_parameter name value =
  printf "␣␣@[<2>integer,␣parameter␣::␣%s␣=␣%d" name value; nl ()

let print_real_parameter name value =
  printf "␣␣@[<2>real(kind=%s),␣parameter␣::␣%s␣=␣%d"
    !kind name value; nl ()

let print_logical_parameter name value =
  printf "␣␣@[<2>logical,␣parameter␣::␣%s␣=␣.%s."
    name (if value then "true" else "false"); nl ()

let num_particles_in amplitudes =
  match CF.flavors amplitudes with
  | [] → 0
  | (fin, _) :: _ → List.length fin

let num_particles_out amplitudes =
  match CF.flavors amplitudes with
  | [] → 0
  | (_, fout) :: _ → List.length fout

let num_particles amplitudes =
  match CF.flavors amplitudes with
  | [] → 0
  | (fin, fout) :: _ → List.length fin + List.length fout

module CFlow = Color.Flow

let num_color_flows amplitudes =
  if !amp_triv then
    1
  else
    List.length (CF.color_flows amplitudes)

let num_color_indices_default = 2 (* Standard model *)

let num_color_indices amplitudes =
  try CFlow.rank (List.hd (CF.color_flows amplitudes)) with _ → num_color_indices_default
```

```
let color_to_string c =
  "(" ^ (String.concat "," (List.map (Printf.sprintf "%3d") c)) ^ ")"

let cflow_to_string cflow =
  String.concat "␣" (List.map color_to_string (CFlow.in_to_lists cflow)) ^ "␣->␣" ^
  String.concat "␣" (List.map color_to_string (CFlow.out_to_lists cflow))

let protected = ",␣protected" (* Fortran 2003! *)

let print_coupling_orders_table amplitudes =
  printf "␣␣@[<2>integer,␣dimension(n_co,n_cop),␣save%s␣::␣table_coupling_orders" protected; nl ();
  begin match CF.coupling_orders amplitudes with
  | None | Some (_, []) → ()
  | Some (_, powers) →
     List.iteri
       (fun i powers →
         printf "␣␣@[<2>data␣table_coupling_orders(:,%4d)␣/␣%s␣/" (succ i)
           (String.concat ",␣" (List.map (Printf.sprintf "%2d") powers));
         nl ())
       powers
  end;
  nl ()

let print_spin_table name tuples =
  printf "␣␣@[<2>integer,␣dimension(n_prt,n_hel),␣save%s␣::␣table_spin_%s"
    protected name; nl ();
  match tuples with
  | [] → ()
  | _ →
     List.iteri
       (fun i (tuple1, tuple2) →
         printf "␣␣@[<2>data␣table_spin_%s(:,%4d)␣/␣%s␣/" name (succ i)
           (String.concat ",␣" (List.map (Printf.sprintf "%2d") (tuple1 @ tuple2)));
         nl ())
       tuples

let print_spin_tables amplitudes =
  print_spin_table "states" (CF.helicities amplitudes);
  nl ()

let print_flavor_table name tuples =
  printf "␣␣@[<2>integer,␣dimension(n_prt,n_flv),␣save%s␣::␣table_flavor_%s"
    protected name; nl ();
  match tuples with
  | [] → ()
  | _ →
     List.iteri
       (fun i tuple →
         printf "␣␣@[<2>data␣table_flavor_%s(:,%4d)␣/␣%s␣/␣!␣%s" name (succ i)
           (String.concat ",␣"
              (List.map (fun f → Printf.sprintf "%3d" (M.pdg f)) tuple))
           (String.concat "␣" (List.map M.flavor_to_string tuple));
         nl ())
       tuples

let print_flavor_tables amplitudes =
  print_flavor_table "states"
    (List.map (fun (fin, fout) → fin @ fout) (CF.flavors amplitudes));
  nl ()

let num_flavors amplitudes =
  List.length (CF.flavors amplitudes)

let print_color_flows_table tuples =
  if !amp_triv then begin
    printf
```

```
        "␣␣@[<2>integer,␣dimension(n_cindex,n_prt,n_cflow),␣save%s␣::␣table_color_flows␣=␣0"
        protected; nl ();
      end
    else begin
      printf
        "␣␣@[<2>integer,␣dimension(n_cindex,n_prt,n_cflow),␣save%s␣::␣table_color_flows"
        protected; nl ();
    end;
    if ¬ !amp_triv then begin
      match tuples with
      | [] → ()
      | _ :: _ as tuples →
          List.iteri
            (fun i tuple →
              begin match CFlow.to_lists tuple with
              | [] → ()
              | cf1 :: cfn →
                  printf "␣␣@[<2>data␣table_color_flows(:,:,%4d)␣/" (succ i);
                  printf "@␣%s" (String.concat "," (List.map string_of_int cf1));
                  List.iter (fun cf → printf ",@␣␣%s" (String.concat "," (List.map string_of_int cf))) cfn;
                  printf "@␣/"; nl ()
              end)
            tuples
    end

  let print_ghost_flags_table tuples =
    if !amp_triv then begin
      printf
        "␣␣@[<2>logical,␣dimension(n_prt,n_cflow),␣save%s␣::␣table_ghost_flags␣=␣F"
        protected; nl ();
      end
    else begin
      printf
        "␣␣@[<2>logical,␣dimension(n_prt,n_cflow),␣save%s␣::␣table_ghost_flags"
        protected; nl ();
      match tuples with
      | [] → ()
      | _ →
          List.iteri
            (fun i tuple →
              begin match CFlow.ghost_flags tuple with
              | [] → ()
              | gf1 :: gfn →
                  printf "␣␣@[<2>data␣table_ghost_flags(:,%4d)␣/" (succ i);
                  printf "@␣%s" (if gf1 then "T" else "F");
                  List.iter (fun gf → printf ",@␣␣%s" (if gf then "T" else "F")) gfn;
                  printf "␣/";
                  nl ()
              end)
            tuples
    end

  let format_power_of x
      { Color.Flow.num = num; Color.Flow.den = den; Color.Flow.power = pwr } =
    match num, den, pwr with
    | _, 0, _ → invalid_arg "format_power_of:␣zero␣denominator"
    | 0, _, _ → "+zero"
    | 1, 1, 0 | −1, −1, 0 → "+one"
    | −1, 1, 0 | 1, −1, 0 → "-one"
    | 1, 1, 1 | −1, −1, 1 → "+" ^ x
    | −1, 1, 1 | 1, −1, 1 → "-" ^ x
    | 1, 1, −1 | −1, −1, −1 → "+1/" ^ x
    | −1, 1, −1 | 1, −1, −1 → "-1/" ^ x
    | 1, 1, p | −1, −1, p →
        "+" ^ (if p > 0 then "" else "1/") ^ x ^ "**" ^ string_of_int (abs p)
    | −1, 1, p | 1, −1, p →
        "-" ^ (if p > 0 then "" else "1/") ^ x ^ "**" ^ string_of_int (abs p)
    | n, 1, 0 →
        (if n < 0 then "-" else "+") ^ string_of_int (abs n) ^ ".0_" ^ !kind
    | n, d, 0 →
        (if n × d < 0 then "-" else "+") ^
        string_of_int (abs n) ^ ".0_" ^ !kind ^ "/" ^
        string_of_int (abs d)
    | n, 1, 1 →
        (if n < 0 then "-" else "+") ^ string_of_int (abs n) ^ "*" ^ x
    | n, 1, −1 →
        (if n < 0 then "-" else "+") ^ string_of_int (abs n) ^ "/" ^ x
    | n, d, 1 →
        (if n × d < 0 then "-" else "+") ^
        string_of_int (abs n) ^ ".0_" ^ !kind ^ "/" ^
        string_of_int (abs d) ^ "*" ^ x
    | n, d, −1 →
        (if n × d < 0 then "-" else "+") ^
        string_of_int (abs n) ^ ".0_" ^ !kind ^ "/" ^
        string_of_int (abs d) ^ "/" ^ x
    | n, 1, p →
        (if n < 0 then "-" else "+") ^ string_of_int (abs n) ^
        (if p > 0 then "*" else "/") ^ x ^ "**" ^ string_of_int (abs p)
    | n, d, p →
        (if n × d < 0 then "-" else "+") ^
        string_of_int (abs n) ^ ".0_" ^ !kind ^ "/" ^
        string_of_int (abs d) ^
        (if p > 0 then "*" else "/") ^ x ^ "**" ^ string_of_int (abs p)

  let format_powers_of x = function
    | [] → "zero"
    | powers → String.concat "" (List.map (format_power_of x) powers)
```

We can optimize the following slightly by reusing common color factor *parameter*s.

```
  let print_color_factor_table table =
    let n_cflow = Array.length table in
    let n_cfactors = ref 0 in
    for c1 = 0 to pred n_cflow do
      for c2 = 0 to pred n_cflow do
        match table.(c1).(c2) with
        | [] → ()
        | _ → incr n_cfactors
      done
    done;
    print_integer_parameter "n_cfactors" !n_cfactors;
    printf "␣␣@[<2>type(%s),␣dimension(n_cfactors),␣save%s␣::"
      omega_color_factor_abbrev protected;
    printf "@␣table_color_factors"; nl ();
    if ¬ !amp_triv then begin
      let i = ref 1 in
      if n_cflow > 0 then begin
        for c1 = 0 to pred n_cflow do
          for c2 = 0 to pred n_cflow do
            match table.(c1).(c2) with
            | [] → ()
            | cf →
                printf "␣␣@[<2>real(kind=%s),␣parameter,␣private␣::␣color_factor_%06d␣=␣%s"
```

```
                  !kind !i (format_powers_of nc_parameter cf);
                nl ();
                printf "␣␣@[<2>data␣table_color_factors(%6d)␣/␣%s(%d,%d,color_factor_%06d)␣/"
                  !i omega_color_factor_abbrev (succ c1) (succ c2) !i;
                incr i;
                nl ();
            done
          done
        end;
      end

    let print_color_tables amplitudes =
      let cflows = CF.color_flows amplitudes
      and cfactors = CF.color_factors amplitudes in
      (* print_color_flows_table_old "c" cflows; nl (); *)
      print_color_flows_table cflows; nl ();
      (* print_ghost_flags_table_old "g" cflows; nl (); *)
      print_ghost_flags_table cflows; nl ();
      (* print_color_factor_table_old cfactors; nl (); *)
      print_color_factor_table cfactors; nl ()

    let option_to_logical = function
      | Some _ → "T"
      | None → "F"

    let print_flavor_color_table n_flv n_cflow table =
      if !amp_triv then begin
        printf
          "␣␣@[<2>logical,␣dimension(n_flv,␣n_cflow),␣save%s␣::␣@␣flv_col_is_allowed␣=␣T"
        protected; nl ();
        end
      else begin
        printf
          "␣␣@[<2>logical,␣dimension(n_flv,␣n_cflow),␣save%s␣::␣@␣flv_col_is_allowed"
        protected; nl ();
        if n_flv > 0 then begin
          for c = 0 to pred n_cflow do
            printf
              "␣␣@[<2>data␣flv_col_is_allowed(:,%4d)␣/" (succ c);
            printf "@␣%s" (option_to_logical table.(0).(c));
            for f = 1 to pred n_flv do
              printf ",@␣%s" (option_to_logical table.(f).(c))
            done;
            printf "@␣/"; nl ()
          done;
        end;
      end

    let print_amplitude_table a =
      (* print_flavor_color_table_old "a" (num_flavors a) (List.length (CF.color_flows a)) (CF.process_table a); nl ();
*)
      print_flavor_color_table
        (num_flavors a) (List.length (CF.color_flows a)) (CF.process_table a);
      nl ();
      printf
        "␣␣@[<2>complex(kind=%s),␣dimension(n_flv,␣n_cflow,␣n_hel),␣save␣::␣amp" !kind;
      nl ();
      nl ()

    let print_helicity_selection_table () =
      printf "␣␣@[<2>logical,␣dimension(n_hel),␣save␣::␣";
      printf "hel_is_allowed␣=␣T"; nl ();
      printf "␣␣@[<2>real(kind=%s),␣dimension(n_hel),␣save␣::␣" !kind;
      printf "hel_max_abs␣=␣0"; nl ();
      printf "␣␣@[<2>real(kind=%s),␣save␣::␣" !kind;
      printf "hel_sum_abs␣=␣0,␣";
      printf "hel_threshold␣=␣1E10_%s" !kind; nl ();
      printf "␣␣@[<2>integer,␣save␣::␣";
      printf "hel_count␣=␣0,␣";
      printf "hel_cutoff␣=␣100"; nl ();
      printf "␣␣@[<2>integer␣::␣";
      printf "i"; nl ();
      printf "␣␣@[<2>integer,␣save,␣dimension(n_hel)␣::␣";
      printf "hel_map␣=␣(/(i,␣i␣=␣1,␣n_hel)/)"; nl ();
      printf "␣␣@[<2>integer,␣save␣::␣hel_finite␣=␣n_hel"; nl ();
      nl ()
```

Optional MD5 sum function

```
let print_md5sum_functions = function
  | Some s →
      printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
      printf "function␣md5sum␣()"; nl ();
      printf "␣␣␣␣character(len=32)␣::␣md5sum"; nl ();
      printf "␣␣␣␣!␣DON'T␣EVEN␣THINK␣of␣modifying␣the␣following␣line!"; nl ();
      printf "␣␣␣␣md5sum␣=␣\"%s\"" s; nl ();
      printf "␣␣end␣function␣md5sum"; nl ();
      nl ()
  | None → ()
```

Maintenance & Inquiry Functions

```
let print_maintenance_functions () =
  if !whizard then begin
    printf "␣␣subroutine␣init␣(par,␣scheme)"; nl ();
    printf "␣␣␣␣real(kind=%s),␣dimension(*),␣intent(in)␣::␣par" !kind; nl ();
    printf "␣␣␣␣integer,␣intent(in)␣::␣scheme"; nl ();
    printf "␣␣␣␣call␣import_from_whizard␣(par,␣scheme)"; nl ();
    printf "␣␣end␣subroutine␣init"; nl ();
    nl ();
    printf "␣␣subroutine␣final␣()"; nl ();
    printf "␣␣end␣subroutine␣final"; nl ();
    nl ();
    printf "␣␣subroutine␣update_alpha_s␣(alpha_s)"; nl ();
    printf "␣␣␣␣real(kind=%s),␣intent(in)␣::␣alpha_s" !kind; nl ();
    printf "␣␣␣␣call␣model_update_alpha_s␣(alpha_s)"; nl ();
    printf "␣␣end␣subroutine␣update_alpha_s"; nl ();
    nl ()
  end

let print_inquiry_function_openmp () = begin
  printf "␣␣pure␣function␣openmp_supported␣()␣result␣(status)"; nl ();
  printf "␣␣␣␣logical␣::␣status"; nl ();
  printf "␣␣␣␣status␣=␣%s" (if !openmp then ".true." else ".false."); nl ();
  printf "␣␣end␣function␣openmp_supported"; nl ();
  nl ()
end

let print_external_mass_case flv (fin, fout) =
  printf "␣␣␣␣case␣(%3d)" (succ flv); nl ();
  List.iteri
    (fun i f →
      printf "␣␣␣␣␣␣m(%2d)␣=␣%s" (succ i) (M.mass_symbol f); nl ())
    (fin @ fout)
```

```
let print_external_masses amplitudes =
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "subroutine␣external_masses␣(m,␣flv)"; nl ();
  printf "␣␣␣␣real(kind=%s),␣dimension(:),␣intent(out)␣::␣m" !kind; nl ();
  printf "␣␣␣␣integer,␣intent(in)␣::␣flv"; nl ();
  printf "␣␣␣␣select␣case␣(flv)"; nl ();
  List.iteri print_external_mass_case (CF.flavors amplitudes);
  printf "␣␣␣␣end␣select"; nl ();
  printf "␣␣end␣subroutine␣external_masses"; nl ();
  nl ()
let print_numeric_inquiry_functions (f, v) =
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "function␣%s␣()␣result␣(n)" f; nl ();
  printf "␣␣␣␣integer␣::␣n"; nl ();
  printf "␣␣␣␣n␣=␣%s" v; nl ();
  printf "␣␣end␣function␣%s" f; nl ();
  nl ()
let print_inquiry_functions name =
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "function␣number_%s␣()␣result␣(n)" name; nl ();
  printf "␣␣␣␣integer␣::␣n"; nl ();
  printf "␣␣␣␣n␣=␣size␣(table_%s,␣dim=2)" name; nl ();
  printf "␣␣end␣function␣number_%s" name; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "subroutine␣%s␣(a)" name; nl ();
  printf "␣␣␣␣integer,␣dimension(:,:),␣intent(out)␣::␣a"; nl ();
  printf "␣␣␣␣a␣=␣table_%s" name; nl ();
  printf "␣␣end␣subroutine␣%s" name; nl ();
  nl ()
let print_color_flows () =
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "function␣number_color_indices␣()␣result␣(n)"; nl ();
  printf "␣␣␣␣integer␣::␣n"; nl ();
  if !amp_triv then begin
    printf "␣␣␣␣n␣=␣n_cindex"; nl ();
    end
  else begin
    printf "␣␣␣␣n␣=␣size␣(table_color_flows,␣dim=1)"; nl ();
  end;
  printf "␣␣end␣function␣number_color_indices"; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "function␣number_color_flows␣()␣result␣(n)"; nl ();
  printf "␣␣␣␣integer␣::␣n"; nl ();
  if !amp_triv then begin
    printf "␣␣␣␣n␣=␣n_cflow"; nl ();
    end
  else begin
    printf "␣␣␣␣n␣=␣size␣(table_color_flows,␣dim=3)"; nl ();
  end;
  printf "␣␣end␣function␣number_color_flows"; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "subroutine␣color_flows␣(a,␣g)"; nl ();
  printf "␣␣␣␣integer,␣dimension(:,:,:),␣intent(out)␣::␣a"; nl ();
  printf "␣␣␣␣logical,␣dimension(:,:),␣intent(out)␣::␣g"; nl ();
  printf "␣␣␣␣a␣=␣table_color_flows"; nl ();
  printf "␣␣␣␣g␣=␣table_ghost_flags"; nl ();
  printf "␣␣end␣subroutine␣color_flows"; nl ();
  nl ()

let print_color_factors () =
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "function␣number_color_factors␣()␣result␣(n)"; nl ();
  printf "␣␣␣␣integer␣::␣n"; nl ();
  printf "␣␣␣␣n␣=␣size␣(table_color_factors)"; nl ();
  printf "␣␣end␣function␣number_color_factors"; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
  printf "subroutine␣color_factors␣(cf)"; nl ();
  printf "␣␣␣␣type(%s),␣dimension(:),␣intent(out)␣::␣cf"
    omega_color_factor_abbrev; nl ();
  printf "␣␣␣␣cf␣=␣table_color_factors"; nl ();
  printf "␣␣end␣subroutine␣color_factors"; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 ∧ pure_unless_openmp then printf "pure␣";
  printf "function␣color_sum␣(flv,␣hel)␣result␣(amp2)"; nl ();
  printf "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel"; nl ();
  printf "␣␣␣␣real(kind=%s)␣::␣amp2" !kind; nl ();
  printf "␣␣␣␣amp2␣=␣real␣(omega_color_sum␣(flv,␣hel,␣amp,␣table_color_factors))"; nl ();
  printf "␣␣end␣function␣color_sum"; nl ();
  nl ()

let print_dispatch_functions () =
  printf "␣␣@[<5>";
  printf "subroutine␣new_event␣(p)"; nl ();
  printf "␣␣␣␣real(kind=%s),␣dimension(0:3,*),␣intent(in)␣::␣p" !kind; nl ();
  printf "␣␣␣␣logical␣::␣mask_dirty"; nl ();
  printf "␣␣␣␣integer␣::␣hel"; nl ();
  printf "␣␣␣␣call␣calculate_amplitudes␣(amp,␣p,␣hel_is_allowed)"; nl ();
  printf "␣␣␣␣if␣((hel_threshold␣.gt.␣0)␣.and.␣(hel_count␣.le.␣hel_cutoff))␣then"; nl ();
  printf "␣␣␣␣␣␣call␣@[<3>omega_update_helicity_selection@␣(hel_count,@␣amp,@␣";
  printf "hel_max_abs,@␣hel_sum_abs,@␣hel_is_allowed,@␣hel_threshold,@␣hel_cutoff,@␣mask_dirty)"; nl ()
  printf "␣␣␣␣␣␣if␣(mask_dirty)␣then"; nl ();
  printf "␣␣␣␣␣␣␣␣hel_finite␣=␣0"; nl ();
  printf "␣␣␣␣␣␣␣␣do␣hel␣=␣1,␣n_hel"; nl ();
  printf "␣␣␣␣␣␣␣␣␣␣if␣(hel_is_allowed(hel))␣then"; nl ();
  printf "␣␣␣␣␣␣␣␣␣␣␣␣hel_finite␣=␣hel_finite␣+␣1"; nl ();
  printf "␣␣␣␣␣␣␣␣␣␣␣␣hel_map(hel_finite)␣=␣hel"; nl ();
  printf "␣␣␣␣␣␣␣␣␣␣end␣if"; nl ();
  printf "␣␣␣␣␣␣␣␣end␣do"; nl ();
  printf "␣␣␣␣␣␣end␣if"; nl ();
  printf "␣␣␣␣end␣if"; nl ();
  printf "␣␣end␣subroutine␣new_event"; nl ();
  nl ();
  printf "␣␣@[<5>";
  printf "subroutine␣reset_helicity_selection␣(threshold,␣cutoff)"; nl ();
  printf "␣␣␣␣real(kind=%s),␣intent(in)␣::␣threshold" !kind; nl ();
  printf "␣␣␣␣integer,␣intent(in)␣::␣cutoff"; nl ();
  printf "␣␣␣␣integer␣::␣i"; nl ();
  printf "␣␣␣␣hel_is_allowed␣=␣T"; nl ();
  printf "␣␣␣␣hel_max_abs␣=␣0"; nl ();
  printf "␣␣␣␣hel_sum_abs␣=␣0"; nl ();
  printf "␣␣␣␣hel_count␣=␣0"; nl ();
  printf "␣␣␣␣hel_threshold␣=␣threshold"; nl ();
  printf "␣␣␣␣hel_cutoff␣=␣cutoff"; nl ();
  printf "␣␣␣␣hel_map␣=␣(/(i,␣i␣=␣1,␣n_hel)/)"; nl ();
  printf "␣␣␣␣hel_finite␣=␣n_hel"; nl ();
  printf "␣␣end␣subroutine␣reset_helicity_selection"; nl ();
  nl ();
  printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
printf "function␣is_allowed␣(flv,␣hel,␣col)␣result␣(yorn)"; nl ();
printf "␣␣␣␣logical␣::␣yorn"; nl ();
printf "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel,␣col"; nl ();
if !amp_triv then begin
    printf "␣␣␣␣!␣print␣*,␣'inside␣is_allowed'"; nl ();
end;
if ¬ !amp_triv then begin
    printf "␣␣␣␣yorn␣=␣hel_is_allowed(hel)␣.and.␣";
    printf "flv_col_is_allowed(flv,col)"; nl ();
    end
else begin
    printf "␣␣␣␣yorn␣=␣.false."; nl ();
end;
printf "␣␣end␣function␣is_allowed"; nl ();
nl ();
printf "␣␣@[<5>"; if !fortran95 then printf "pure␣";
printf "function␣get_amplitude␣(flv,␣hel,␣col)␣result␣(amp_result)"; nl ();
printf "␣␣␣␣complex(kind=%s)␣::␣amp_result" !kind; nl ();
printf "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel,␣col"; nl ();
printf "␣␣␣␣amp_result␣=␣amp(flv,␣col,␣hel)"; nl ();
printf "␣␣end␣function␣get_amplitude"; nl ();
nl ()
```

Main Function

```
let format_power_of_nc
    { Color.Flow.num = num; Color.Flow.den = den; Color.Flow.power = pwr } =
  match num, den, pwr with
  | _, 0, _ → invalid_arg "format_power_of_nc:␣zero␣denominator"
  | 0, _, _ → ""
  | 1, 1, 0 | −1, −1, 0 → "+␣1"
  | −1, 1, 0 | 1, −1, 0 → "-␣1"
  | 1, 1, 1 | −1, −1, 1 → "+␣N"
  | −1, 1, 1 | 1, −1, 1 → "-␣N"
  | 1, 1, −1 | −1, −1, −1 → "+␣1/N"
  | −1, 1, −1 | 1, −1, −1 → "-␣1/N"
  | 1, 1, p | −1, −1, p →
      "+␣" ^ (if p > 0 then "" else "1/") ^ "N^" ^ string_of_int (abs p)
  | −1, 1, p | 1, −1, p →
      "-␣" ^ (if p > 0 then "" else "1/") ^ "N^" ^ string_of_int (abs p)
  | n, 1, 0 →
      (if n < 0 then "-␣" else "+␣") ^ string_of_int (abs n)
  | n, d, 0 →
      (if n × d < 0 then "-␣" else "+␣") ^
      string_of_int (abs n) ^ "/" ^ string_of_int (abs d)
  | n, 1, 1 →
      (if n < 0 then "-␣" else "+␣") ^ string_of_int (abs n) ^ "N"
  | n, 1, −1 →
      (if n < 0 then "-␣" else "+␣") ^ string_of_int (abs n) ^ "/N"
  | n, d, 1 →
      (if n × d < 0 then "-␣" else "+␣") ^
      string_of_int (abs n) ^ "/" ^ string_of_int (abs d) ^ "N"
  | n, d, −1 →
      (if n × d < 0 then "-␣" else "+␣") ^
      string_of_int (abs n) ^ "/" ^ string_of_int (abs d) ^ "/N"
  | n, 1, p →
      (if n < 0 then "-␣" else "+␣") ^ string_of_int (abs n) ^
      (if p > 0 then "*" else "/") ^ "N^" ^ string_of_int (abs p)
  | n, d, p →
      (if n × d < 0 then "-␣" else "+␣") ^ string_of_int (abs n) ^ "/" ^
      string_of_int (abs d) ^ (if p > 0 then "*" else "/") ^ "N^" ^ string_of_int (abs p)
let format_powers_of_nc = function
  | [] → "0"
  | powers → String.concat "␣" (List.map format_power_of_nc powers)
let dump_amplitude_slices amplitudes =
  match CF.coupling_orders amplitudes with
  | None → ()
  | Some (co_list, cop_list) →
     printf "!␣␣␣coupling␣orders:"; nl ();
     printf "!"; nl ();
     printf "!␣␣␣␣␣␣%s" (String.concat ",␣" (List.map CM.coupling_order_to_string co_list)); nl ();
     List.iter
       (fun cop_list →
         printf "!␣␣␣␣␣␣%s" (String.concat ",␣" (List.map string_of_int cop_list)); nl ())
       cop_list;
     printf "!"; nl ();
     List.iter
       (fun amplitude →
         printf "!␣␣␣␣␣%s" (process_to_string amplitude); nl ();
         match F.brakets amplitude with
         | [] → ()
         | lines →
            let order_to_string (order, n) =
              Printf.sprintf "%s␣=␣%d" (CM.coupling_order_to_string order) n in
            let orders_to_string orders =
              String.concat ",␣" (List.map order_to_string orders) in
            List.iter (fun (orders, _) → printf "!␣␣␣␣␣%s" (orders_to_string orders); nl ()) lines;
            printf "!"; nl ())
       (CF.processes amplitudes);
     printf "!"; nl ()
let print_description cmdline amplitudes () =
  printf
    "!␣File␣generated␣automatically␣by␣O'Mega␣%s␣%s␣%s"
    Config.version Config.status Config.date; nl ();
  List.iter (fun s → printf "!␣%s" s; nl ()) (M.caveats ());
  printf "!"; nl ();
  printf "!␣␣␣%s" cmdline; nl ();
  printf "!"; nl ();
  printf "!␣with␣all␣scattering␣amplitudes␣for␣the␣process(es)"; nl ();
  printf "!"; nl ();
  printf "!␣␣␣flavor␣combinations:"; nl ();
  printf "!"; nl ();
  ThoList.iteri
    (fun i process →
      printf "!␣␣␣␣␣%3d:␣%s" i (process_sans_color_to_string process); nl ())
    1 (CF.flavors amplitudes);
  printf "!"; nl ();
  printf "!␣␣␣color␣flows:"; nl ();
  if ¬ !amp_triv then begin
    printf "!"; nl ();
    ThoList.iteri
      (fun i cflow →
        printf "!␣␣␣␣␣%3d:␣%s" i (cflow_to_string cflow); nl ())
      1 (CF.color_flows amplitudes);
    printf "!"; nl ();
    printf "!␣␣␣␣␣NB:␣i.g.␣not␣all␣color␣flows␣contribute␣to␣all␣flavor"; nl ();
    printf "!␣␣␣␣␣combinations.␣␣Consult␣the␣array␣FLV_COL_IS_ALLOWED"; nl ();
    printf "!␣␣␣␣␣below␣for␣the␣allowed␣combinations."; nl ();
  end;
  printf "!"; nl ();
  printf "!␣␣␣Color␣Factors:"; nl ();
  printf "!"; nl ();
  if ¬ !amp_triv then begin
    let cfactors = CF.color_factors amplitudes in
    for c1 = 0 to pred (Array.length cfactors) do
      for c2 = 0 to c1 do
        match cfactors.(c1).(c2) with
        | [] → ()
        | cfactor →
            printf "!␣␣␣␣␣(%3d,%3d):␣%s"
              (succ c1) (succ c2) (format_powers_of_nc cfactor); nl ()
      done
    done;
  end;
  if ¬ !amp_triv then begin
      printf "!"; nl ();
      printf "!␣␣␣vanishing␣or␣redundant␣flavor␣combinations:"; nl ();
      printf "!"; nl ();
      List.iter (fun process →
        printf "!␣␣␣␣␣␣␣␣␣␣%s" (process_sans_color_to_string process); nl ())
        (CF.vanishing_flavors amplitudes);
      printf "!"; nl ();
  end;
  begin
    match CF.constraints amplitudes with
    | None → ()
    | Some s →
        printf
          "!␣␣␣diagram␣selection␣(MIGHT␣BREAK␣GAUGE␣INVARIANCE!!!):"; nl ();
        printf "!"; nl ();
        printf "!␣␣␣␣␣%s" s; nl ();
        printf "!"; nl ()
  end;
  begin
    match CF.slicings amplitudes with
    | [] → ()
    | lines →
        printf
          "!␣␣␣coupling␣constant␣selections␣('slicings'):"; nl ();
        printf "!"; nl ();
        List.iter (fun s → printf "!␣␣␣␣␣%s" s; nl ()) lines;
        printf "!"; nl ()
  end;
  dump_amplitude_slices amplitudes;
  printf "!"; nl ()
```

Printing Modules

```
type accessibility =
  | Public
  | Private
  | Protected (* Fortran 2003 *)

let accessibility_to_string = function
  | Public → "public"
  | Private → "private"
  | Protected → "protected"

type used_symbol =
  | As_Is of string
  | Aliased of string × string
```

```
let print_used_symbol = function
  | As_Is name → printf "%s" name
  | Aliased (orig, alias) → printf "%s␣=>␣%s" alias orig

type used_module =
  | Full of string
  | Full_Aliased of string × (string × string) list
  | Subset of string × used_symbol list

let print_used_module = function
  | Full name
  | Full_Aliased (name, [])
  | Subset (name, []) →
      printf "␣␣use␣%s" name;
      nl ()
  | Full_Aliased (name, aliases) →
      printf "␣␣@[<5>use␣%s" name;
      List.iter
        (fun (orig, alias) → printf ",␣%s␣=>␣%s" alias orig)
        aliases;
      nl ()
  | Subset (name, used_symbol :: used_symbols) →
      printf "␣␣@[<5>use␣%s,␣only:␣" name;
      print_used_symbol used_symbol;
      List.iter (fun s → printf ",␣"; print_used_symbol s) used_symbols;
      nl ()

type fortran_module =
    { module_name : string;
      default_accessibility : accessibility;
      used_modules : used_module list;
      public_symbols : string list;
      print_declarations : (unit → unit) list;
      print_implementations : (unit → unit) list }

let print_public = function
  | name1 :: names →
      printf "␣␣@[<2>public␣::␣%s" name1;
      List.iter (fun n → printf ",@␣%s" n) names; nl ()
  | [] → ()

let print_module m =
  printf "module␣%s" m.module_name; nl ();
  List.iter print_used_module m.used_modules;
  printf "␣␣implicit␣none"; nl ();
  printf "␣␣%s" (accessibility_to_string m.default_accessibility); nl ();
  print_public m.public_symbols; nl ();
  begin match m.print_declarations with
  | [] → ()
  | print_declarations →
      List.iter (fun f → f ()) print_declarations; nl ()
  end;
  begin match m.print_implementations with
  | [] → ()
  | print_implementations →
      printf "contains"; nl (); nl ();
      List.iter (fun f → f ()) print_implementations; nl ();
  end;
  printf "end␣module␣%s" m.module_name; nl ()

let print_modules modules =
  List.iter print_module modules;
  print_flush ()

let module_to_file line_length oc prelude m =
    output_string oc (m.module_name ^ "\n");
    let filename = m.module_name ^ ".f90" in
    let channel = open_out filename in
    Format_Fortran.set_formatter_out_channel ~width : line_length channel;
    prelude ();
    print_modules [m];
    close_out channel

let modules_to_file line_length oc prelude = function
  | [] → ()
  | m :: mlist →
      module_to_file line_length oc prelude m;
      List.iter (module_to_file line_length oc (fun () → ())) mlist
```

Chopping Up Amplitudes

```
let all_brakets process =
  ThoList.flatmap snd (F.brakets process)

let num_fusions_brakets size amplitudes =
  let num_fusions =
    max 1 size in
  let count_brakets =
    List.fold_left
      (fun sum process → sum + List.length (all_brakets process))
      0 (CF.processes amplitudes)
  and count_processes =
    List.length (CF.processes amplitudes) in
  if count_brakets > 0 then
    let num_brakets =
      max 1 ((num_fusions × count_processes) / count_brakets) in
    (num_fusions, num_brakets)
  else
    (num_fusions, 1)

let chop_amplitudes size amplitudes =
  let num_fusions, num_brakets = num_fusions_brakets size amplitudes in
  (ThoList.enumerate 1 (ThoList.chopn num_fusions (CF.fusions amplitudes)),
   ThoList.enumerate 1 (ThoList.chopn num_brakets (CF.processes amplitudes)))

let print_compute_fusions1 dictionary (n, fusions) =
  if ¬ !amp_triv then begin
    if !openmp then begin
      printf "␣␣subroutine␣compute_fusions_%04d␣(%s)" n openmp_tld; nl ();
      printf "␣␣@[<5>type(%s),␣intent(inout)␣::␣%s" openmp_tld_type openmp_tld; nl ();
    end else begin
      printf "␣␣@[<5>subroutine␣compute_fusions_%04d␣()" n; nl ();
    end;
    print_fusions dictionary fusions;
    printf "␣␣end␣subroutine␣compute_fusions_%04d" n; nl ();
  end

and print_compute_brakets1 dictionary (n, processes) =
  if ¬ !amp_triv then begin
    if !openmp then begin
      printf "␣␣subroutine␣compute_brakets_%04d␣(%s)" n openmp_tld; nl ();
      printf "␣␣@[<5>type(%s),␣intent(inout)␣::␣%s" openmp_tld_type openmp_tld; nl ();
    end else begin
      printf "␣␣@[<5>subroutine␣compute_brakets_%04d␣()" n; nl ();
    end;
    List.iter (print_brakets dictionary) processes;
    printf "␣␣end␣subroutine␣compute_brakets_%04d" n; nl ();
  end
```

Common Stuff

```
let omega_public_symbols =
  ["number_particles_in"; "number_particles_out";
   "number_color_indices";
   "reset_helicity_selection"; "new_event";
   "is_allowed"; "get_amplitude"; "color_sum";
   "external_masses"; "openmp_supported"] @
  ThoList.flatmap
    (fun n → ["number_" ^ n; n])
    ["spin_states"; "flavor_states"; "color_flows"; "color_factors"]

let whizard_public_symbols md5sum =
  ["init"; "final"; "update_alpha_s"] @
  (match md5sum with Some _ → ["md5sum"] | None → [])

let used_modules () =
  [Full "kinds";
   Full Names.use_module;
   Full_Aliased ("omega_color", ["omega_color_factor", omega_color_factor_abbrev])] @
  List.map
    (fun m → Full m)
    (match !parameter_module with
     | "" → !use_modules
     | pm → pm :: !use_modules)

let public_symbols () =
  if !whizard then
    omega_public_symbols @ (whizard_public_symbols !md5sum)
  else
    omega_public_symbols

let print_constants amplitudes =

  printf "␣␣!␣DON'T␣EVEN␣THINK␣of␣removing␣the␣following!"; nl ();
  printf "␣␣!␣If␣the␣compiler␣complains␣about␣undeclared"; nl ();
  printf "␣␣!␣or␣undefined␣variables,␣you␣are␣compiling"; nl ();
  printf "␣␣!␣against␣an␣incompatible␣omega95␣module!"; nl ();
  printf "␣␣@[<2>integer,␣dimension(%d),␣parameter,␣private␣::␣"
    (List.length require_library);
  printf "require␣=@␣(/␣@[";
  print_list require_library;
  printf "␣/)"; nl (); nl ();
```

Using these parameters makes sense for documentation, but in practice, there is no need to ever change them.

```
  List.iter
    (function name, value → print_integer_parameter name (value amplitudes))
    [ ("n_prt", num_particles);
      ("n_in", num_particles_in);
      ("n_out", num_particles_out);
      ("n_cflow", num_color_flows); (* Number of different color amplitudes. *)
      ("n_cindex", num_color_indices); (* Maximum rank of color tensors. *)
      ("n_flv", num_flavors); (* Number of different flavor amplitudes. *)
      ("n_hel", num_helicities); (* Number of different helicity amplitudes. *)
      ("n_co", num_coupling_orders); (* Number of different coupling orders. *)
      ("n_cop", num_coupling_order_powers) (* Number of different powers of coupling orders. *) ];
  nl ();
```

Abbreviations.

```
  printf "␣␣!␣NB:␣you␣MUST␣NOT␣change␣the␣value␣of␣%s␣here!!!" nc_parameter;
  nl ();
  printf "␣␣!␣␣␣␣␣It␣is␣defined␣here␣for␣convenience␣only␣and␣must␣be"; nl ();
  printf "␣␣!␣␣␣␣␣compatible␣with␣hardcoded␣values␣in␣the␣amplitude!"; nl ();
  print_real_parameter nc_parameter (SCM.nc ()); (* N_C *)
```

```
    List.iter
      (function name, value → print_logical_parameter name value)
      [ ("F", false); ("T", true) ]; nl ();
    print_coupling_orders_table amplitudes;
    print_spin_tables amplitudes;
    print_flavor_tables amplitudes;
    print_color_tables amplitudes;
    print_amplitude_table amplitudes;
    print_helicity_selection_table ()

let print_interface amplitudes =
    print_md5sum_functions !md5sum;
    print_maintenance_functions ();
    List.iter print_numeric_inquiry_functions
      [("number_particles_in", "n_in");
       ("number_particles_out", "n_out")];
    List.iter print_inquiry_functions
      ["spin_states"; "flavor_states"];
    print_external_masses amplitudes;
    print_inquiry_function_openmp ();
    print_color_flows ();
    print_color_factors ();
    print_dispatch_functions ();
    nl ();
    (* Is this really necessary? *)
    Format_Fortran.switch_line_continuation false;
    if !km_write ∨ !km_pure then (Targets_Kmatrix.Fortran.print !km_pure);
    if !km_2_write ∨ !km_2_pure then (Targets_Kmatrix_2.Fortran.print !km_2_pure);
    Format_Fortran.switch_line_continuation true;
    nl ()

let print_calculate_amplitudes declarations computations amplitudes =
    printf "␣␣@[<5>subroutine␣calculate_amplitudes␣(amp,␣k,␣mask)"; nl ();
    printf "␣␣␣␣complex(kind=%s),␣dimension(:,:,:),␣intent(out)␣::␣amp" !kind; nl ();
    printf "␣␣␣␣real(kind=%s),␣dimension(0:3,*),␣intent(in)␣::␣k" !kind; nl ();
    printf "␣␣␣␣logical,␣dimension(:),␣intent(in)␣::␣mask"; nl ();
    printf "␣␣␣␣integer,␣dimension(n_prt)␣::␣s"; nl ();
    printf "␣␣␣␣integer␣::␣h,␣hi"; nl ();
    declarations ();
    if ¬ !amp_triv then begin
      begin match CF.processes amplitudes with
      | p :: _ → print_external_momenta p
      | _ → ()
      end;
      ignore (List.fold_left print_momenta PSet.empty (CF.processes amplitudes));
    end;
    printf "␣␣␣␣amp␣=␣0"; nl ();
    if ¬ !amp_triv then begin
      if num_helicities amplitudes > 0 then begin
        printf "␣␣␣␣if␣(hel_finite␣==␣0)␣return"; nl ();
        if !openmp then begin
          printf "!$OMP␣PARALLEL␣DO␣DEFAULT(SHARED)␣PRIVATE(s,␣h,␣%s)␣SCHEDULE(STATIC)" openmp_tld; nl ();
        end;
        printf "␣␣␣␣do␣hi␣=␣1,␣hel_finite"; nl ();
        printf "␣␣␣␣␣␣h␣=␣hel_map(hi)"; nl ();
        printf "␣␣␣␣␣␣s␣=␣table_spin_states(:,h)"; nl ();
        ignore (List.fold_left print_externals WFSet.empty (CF.processes amplitudes));
        computations ();
        List.iter print_fudge_factor (CF.processes amplitudes);
      (* This sorting should slightly improve cache locality. *)
        let triple_snd = fun (_, x, _) → x
        in let triple_fst = fun (x, _, _) → x
```

```
            in let rec builder1 flvi flowi flows = match flows with
            | (Some a) :: tl → (flvi, flowi, flavors_symbol (flavors a)) :: (builder1 flvi (flowi + 1) tl)
            | None :: tl → builder1 flvi (flowi + 1) tl
            | [] → []
               in let rec builder2 flvi flvs = match flvs with
               | flv :: tl → (builder1 flvi 1 flv) @ (builder2 (flvi + 1) tl)
               | [] → []
                  in let unsorted = builder2 1 (List.map Array.to_list (Array.to_list (CF.process_table amplitudes)))
                     in let sorted = List.sort (fun a b →
                        if (triple_snd a ≢ triple_snd b) then triple_snd a − triple_snd b else (triple_fst a − triple_fst b))
                           unsorted
                        in List.iter (fun (flvi, flowi, flv) →
                           (printf "␣␣␣␣␣␣amp(%d,%d,h)␣=␣%s" flvi flowi flv; nl ();)) sorted;

                        printf "␣␣␣␣end␣do"; nl ();
                        if !openmp then begin
                           printf "!$OMP␣END␣PARALLEL␣DO"; nl ();
                        end;
    end;
  end;
  printf "␣␣end␣subroutine␣calculate_amplitudes"; nl ()

let print_compute_chops chopped_fusions chopped_brakets () =
  List.iter
    (fun (i, _) → printf "␣␣␣␣␣␣call␣compute_fusions_%04d␣(%s)" i
       (if !openmp then openmp_tld else ""); nl ())
    chopped_fusions;
  List.iter
    (fun (i, _) → printf "␣␣␣␣␣␣call␣compute_brakets_%04d␣(%s)" i
       (if !openmp then openmp_tld else ""); nl ())
    chopped_brakets
```

UFO Fusions

```
module VSet =
  Set.Make (struct type t = F.constant Coupling.t let compare = compare end)

let ufo_fusions_used amplitudes =
  let couplings =
    List.fold_left
      (fun acc p →
        let fusions = ThoList.flatmap F.rhs (F.fusions p)
        and brakets = ThoList.flatmap F.ket (all_brakets p) in
        let couplings =
          VSet.of_list (List.map F.coupling (fusions @ brakets)) in
        VSet.union acc couplings)
      VSet.empty (CF.processes amplitudes) in
  VSet.fold
    (fun v acc →
      match v with
      | Coupling.Vn (Coupling.UFO (_, v, _, _, _), _, _) →
         Sets.String.add v acc
      | _ → acc)
    couplings Sets.String.empty
```

Single Function

```
let amplitudes_to_channel_single_function cmdline oc amplitudes =

  let print_declarations () =
    print_constants amplitudes
  and print_implementations () =
    print_interface amplitudes;
    print_calculate_amplitudes
      (fun () → print_variable_declarations amplitudes)
      (fun () →
        print_fusions (CF.dictionary amplitudes) (CF.fusions amplitudes);
        List.iter
          (print_brakets (CF.dictionary amplitudes))
          (CF.processes amplitudes))
      amplitudes in

  let fortran_module =
    { module_name = !module_name;
      used_modules = used_modules ();
      default_accessibility = Private;
      public_symbols = public_symbols ();
      print_declarations = [print_declarations];
      print_implementations = [print_implementations] } in

  Format_Fortran.set_formatter_out_channel ~width :!line_length oc;
  print_description cmdline amplitudes ();
  print_modules [fortran_module]
```

Single Module

```
let amplitudes_to_channel_single_module cmdline oc size amplitudes =

  let print_declarations () =
    print_constants amplitudes;
    print_variable_declarations amplitudes

  and print_implementations () =
    print_interface amplitudes in

  let chopped_fusions, chopped_brakets =
    chop_amplitudes size amplitudes in

  let dictionary = CF.dictionary amplitudes in

  let print_compute_amplitudes () =
    print_calculate_amplitudes
      (fun () → ())
      (print_compute_chops chopped_fusions chopped_brakets)
      amplitudes

  and print_compute_fusions () =
    List.iter (print_compute_fusions1 dictionary) chopped_fusions

  and print_compute_brakets () =
    List.iter (print_compute_brakets1 dictionary) chopped_brakets in

  let fortran_module =
    { module_name = !module_name;
      used_modules = used_modules ();
      default_accessibility = Private;
      public_symbols = public_symbols ();
      print_declarations = [print_declarations];
      print_implementations = [print_implementations;
                               print_compute_amplitudes;
                               print_compute_fusions;
                               print_compute_brakets] } in

  Format_Fortran.set_formatter_out_channel ~width :!line_length oc;
  print_description cmdline amplitudes ();
  print_modules [fortran_module]
```

Multiple Modules

```
let modules_of_amplitudes _ _ size amplitudes =
  let name = !module_name in
  let print_declarations () =
    print_constants amplitudes
  and print_variables () =
    print_variable_declarations amplitudes in
  let constants_module =
    { module_name = name ^ "_constants";
      used_modules = used_modules ();
      default_accessibility = Public;
      public_symbols = [];
      print_declarations = [print_declarations];
      print_implementations = [] } in
  let variables_module =
    { module_name = name ^ "_variables";
      used_modules = used_modules ();
      default_accessibility = Public;
      public_symbols = [];
      print_declarations = [print_variables];
      print_implementations = [] } in
  let dictionary = CF.dictionary amplitudes in
  let print_compute_fusions (n, fusions) () =
    if ¬ !amp_triv then begin
      if !openmp then begin
        printf "␣␣subroutine␣compute_fusions_%04d␣(%s)" n openmp_tld; nl ();
        printf "␣␣@[<5>type(%s),␣intent(inout)␣::␣%s" openmp_tld_type openmp_tld; nl ();
      end else begin
        printf "␣␣@[<5>subroutine␣compute_fusions_%04d␣()" n; nl ();
      end;
      print_fusions dictionary fusions;
      printf "␣␣end␣subroutine␣compute_fusions_%04d" n; nl ();
    end in
  let print_compute_brakets (n, processes) () =
    if ¬ !amp_triv then begin
      if !openmp then begin
        printf "␣␣subroutine␣compute_brakets_%04d␣(%s)" n openmp_tld; nl ();
        printf "␣␣@[<5>type(%s),␣intent(inout)␣::␣%s" openmp_tld_type openmp_tld; nl ();
      end else begin
        printf "␣␣@[<5>subroutine␣compute_brakets_%04d␣()" n; nl ();
      end;
      List.iter (print_brakets dictionary) processes;
      printf "␣␣end␣subroutine␣compute_brakets_%04d" n; nl ();
    end in
  let fusions_module (n, _ as fusions) =
    let tag = Printf.sprintf "_fusions_%04d" n in
    { module_name = name ^ tag;
      used_modules = (used_modules () @
                      [Full constants_module.module_name;
                       Full variables_module.module_name]);
      default_accessibility = Private;
      public_symbols = ["compute" ^ tag];
      print_declarations = [];
      print_implementations = [print_compute_fusions fusions] } in
  let brakets_module (n, _ as processes) =
    let tag = Printf.sprintf "_brakets_%04d" n in
    { module_name = name ^ tag;
      used_modules = (used_modules () @
                      [Full constants_module.module_name;
                       Full variables_module.module_name]);
      default_accessibility = Private;
      public_symbols = ["compute" ^ tag];
      print_declarations = [];
      print_implementations = [print_compute_brakets processes] } in

  let chopped_fusions, chopped_brakets =
    chop_amplitudes size amplitudes in

  let fusions_modules =
    List.map fusions_module chopped_fusions in

  let brakets_modules =
    List.map brakets_module chopped_brakets in

  let print_implementations () =
    print_interface amplitudes;
    print_calculate_amplitudes
      (fun () → ())
      (print_compute_chops chopped_fusions chopped_brakets)
      amplitudes in

  let public_module =
    { module_name = name;
       used_modules = (used_modules () @
                       [Full constants_module.module_name;
                        Full variables_module.module_name ] @
                       List.map
                         (fun m → Full m.module_name)
                         (fusions_modules @ brakets_modules));
      default_accessibility = Private;
      public_symbols = public_symbols ();
      print_declarations = [];
      print_implementations = [print_implementations] }
  and private_modules =
    [constants_module; variables_module] @
      fusions_modules @ brakets_modules in
  (public_module, private_modules)

let amplitudes_to_channel_single_file cmdline oc size amplitudes =
  let public_module, private_modules =
    modules_of_amplitudes cmdline oc size amplitudes in
  Format_Fortran.set_formatter_out_channel ~width :!line_length oc;
  print_description cmdline amplitudes ();
  print_modules (private_modules @ [public_module])

let amplitudes_to_channel_multi_file cmdline oc size amplitudes =
  let public_module, private_modules =
    modules_of_amplitudes cmdline oc size amplitudes in
  modules_to_file !line_length oc
    (print_description cmdline amplitudes)
    (public_module :: private_modules)
```

Dispatch

```
let amplitudes_to_channel cmdline oc diagnostics amplitudes =
  parse_diagnostics diagnostics;
  let ufo_fusions =
    let ufo_fusions_set = ufo_fusions_used amplitudes in
    if Sets.String.is_empty ufo_fusions_set then
      None
```

```
          else
            Some ufo_fusions_set in
        begin match ufo_fusions with
        | Some only →
            let name = !module_name ^ "_ufo"
            and fortran_module = Names.use_module in
            use_modules := name :: !use_modules;
            UFO.Targets.Fortran.lorentz_module
              ~only ~name ~fortran_module ~parameter_module :!parameter_module
              (Format_Fortran.formatter_of_out_channel oc) ()
        | None → ()
        end;
        match !output_mode with
        | Single_Function →
            amplitudes_to_channel_single_function cmdline oc amplitudes
        | Single_Module size →
            amplitudes_to_channel_single_module cmdline oc size amplitudes
        | Single_File size →
            amplitudes_to_channel_single_file cmdline oc size amplitudes
        | Multi_File size →
            amplitudes_to_channel_multi_file cmdline oc size amplitudes

    let parameters_to_channel oc =
      parameters_to_fortran oc (CM.parameters ())

  end

module Make =
  Make_Fortran(Target_Fortran_Names.Dirac)(Targets_vintage.Fortran_Fermions)
module Make_Majorana =
  Make_Fortran(Target_Fortran_Names.Majorana)(Targets_vintage.Fortran_Majorana_Fermions)
```

20.7 Interface of Targets_vintage

This is the original implementation of *Target_Fortran().print_current* for hard coded models with *Coupling.V3* and *Coupling.V4* vertices only. It was adequate for the Standard Model and simple extensions upto the MSSM. The extension to higher dimensional operators became more and more baroque — to the extent to be almost unmaintainable. In order to make *Target_Fortran* maintainable, this code has been factored out.
Output routines for fermion couplings.

```
module type Fermions =
  sig
    open Coupling
    val print_current : int × fermionbar × boson × fermion →
      string → string → string → fuse2 → unit
    val print_current_mom : int × fermionbar × boson × fermion →
      string → string → string → string → string → string → fuse2 → unit
    val print_current_p : int × fermion × boson × fermion →
      string → string → string → fuse2 → unit
    val print_current_b : int × fermionbar × boson × fermionbar →
      string → string → string → fuse2 → unit
    val print_current_g : int × fermionbar × boson × fermion →
      string → string → string → string → string → string → fuse2 → unit
    val print_current_g4 : int × fermionbar × boson2 × fermion →
      string → string → string → string → fuse3 → unit
    val reverse_braket : bool → lorentz → lorentz list → bool
  end
```

We need to use the names of Fortran types, wave function variables and propagator functions consistently with `omegalib` and *Target_Fortran*.

```
module type Fermion_Maker = functor (N : Target_Fortran_Names.T) → Fermions

module Fortran_Fermions : Fermion_Maker
module Fortran_Majorana_Fermions : Fermion_Maker
```

Output routines triple and quartic vertices.

```
module type T =
  sig

    type amplitude
    type constant
    type wf
    type rhs
```

print_current_V3 format_wf format_p amplitude dictionary amplitude dictionary rhs vertex fusion constant writes code combining the children *rhs* into a current, using the vertex factor *vertex*, coupling *constant* and the permutation *fusion* of its legs. *amplitude* is used with *dictionary* to disambiguate wavefunctions with the same flavor and momentum. The formatting functions *format_wf* and *format_p* must be compatible with the remaining implementation of *Target*.

The type is probably unnecessarily higher order. It was natural in the monolithic implementation and has been kept in the first refactoring step.

```
    val print_current_V3 :
      (amplitude → (amplitude → wf → int) → wf → string) → (wf → string) →
      amplitude → (amplitude → wf → int) → rhs →
      constant Coupling.vertex3 → Coupling.fuse2 → constant → unit

    val print_current_V4 :
      (amplitude → (amplitude → wf → int) → wf → string) → (wf → string) →
      amplitude → (amplitude → wf → int) → rhs →
      constant Coupling.vertex4 → Coupling.fuse3 → constant → unit

  end

module type Maker =
  functor (N : Target_Fortran_Names.T) → functor (F : Fermion_Maker) →
  functor (FM : Fusion.Maker) → functor (P : Momentum.T) → functor (M : Model.T) → T
  with type amplitude = Fusion.Multi(FM)(P)(M).amplitude
   and type constant = Orders.Slice(Colorize.It(M)).constant
   and type wf = FM(P)(M).wf
   and type rhs = FM(P)(M).rhs

 module Make_Fortran : Maker
```

20.8 Implementation of *Targets_vintage*

20.8.1 `Fortran` 90/95

Dirac Fermions

We factor out the code for fermions so that we can use the simpler implementation for Dirac fermions if the model contains no Majorana fermions.

```
module type Fermions =
  sig
    open Coupling
    val print_current : int × fermionbar × boson × fermion →
      string → string → string → fuse2 → unit
    val print_current_mom : int × fermionbar × boson × fermion →
      string → string → string → string → string → string
      → fuse2 → unit
    val print_current_p : int × fermion × boson × fermion →
      string → string → string → fuse2 → unit
    val print_current_b : int × fermionbar × boson × fermionbar →
      string → string → string → fuse2 → unit
    val print_current_g : int × fermionbar × boson × fermion →
```

```
      string → string → string → string → string → string
      → fuse2 → unit
    val print_current_g4 : int × fermionbar × boson2 × fermion →
      string → string → string → string → fuse3 → unit
    val reverse_braket : bool → lorentz → lorentz list → bool
  end

module type Fermion_Maker = functor (N : Target_Fortran_Names.T) → Fermions

module Fortran_Fermions (Names : Target_Fortran_Names.T) : Fermions =
  struct

    open Coupling
    open Format

    let format_coupling coeff c =
      match coeff with
      | 1 → c
      | −1 → "(-" ^ c ^")"
      | coeff → string_of_int coeff ^ "*" ^ c

    let format_coupling_2 coeff c =
      match coeff with
      | 1 → c
      | −1 → "-" ^ c
      | coeff → string_of_int coeff ^ "*" ^ c
```

JR's coupling constant HACK, necessitated by tho's bad design descition.

```
    let fastener s i ?p ?q () =
      try
        let offset = (String.index s '(') in
        if ((String.get s (String.length s − 1)) ≢ ')') then
          failwith "fastener:␣wrong␣usage␣of␣parentheses"
        else
          let func_name = (String.sub s 0 offset) and
              tail =
            (String.sub s (succ offset) (String.length s − offset − 2)) in
          if (String.contains func_name ')') ∨
             (String.contains tail '(') ∨
             (String.contains tail ')') then
            failwith "fastener:␣wrong␣usage␣of␣parentheses"
          else
            func_name ^ "(" ^ string_of_int i ^ "," ^ tail ^ ")"
      with
      | Not_found →
          if (String.contains s ')') then
            failwith "fastener:␣wrong␣usage␣of␣parentheses"
          else
            match p with
            | None → s ^ "(" ^ string_of_int i ^ ")"
            | Some p →
              match q with
              | None → s ^ "(" ^ p ^ "*" ^ p ^ "," ^ string_of_int i ^ ")"
              | Some q → s ^ "(" ^ p ^ "," ^ q ^ "," ^ string_of_int i ^ ")"

    let print_fermion_current coeff f c wf1 wf2 fusion =
      let c = format_coupling coeff c in
      match fusion with
      | F13 → printf "%s_ff(%s,%s,%s)" f c wf1 wf2
      | F31 → printf "%s_ff(%s,%s,%s)" f c wf2 wf1
      | F23 → printf "f_%sf(%s,%s,%s)" f c wf1 wf2
      | F32 → printf "f_%sf(%s,%s,%s)" f c wf2 wf1
      | F12 → printf "f_f%s(%s,%s,%s)" f c wf1 wf2
    | F21 → printf "f_f%s(%s,%s,%s)" f c wf2 wf1
```

Using a two element array for the combined vector-axial and scalar-pseudo couplings helps to support HELAS as well. Since we will probably never support general boson couplings with HELAS, it might be retired in favor of two separate variables. For this *Model.constant_symbol* has to be generalized.

NB: passing the array instead of two separate constants would be a *bad* idea, because the support for Majorana spinors below will have to flip signs!

```
  let print_fermion_current2 coeff f c wf1 wf2 fusion =
    let c = format_coupling_2 coeff c in
    let c1 = fastener c 1 ()
    and c2 = fastener c 2 () in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F31 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf2 wf1
    | F23 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1
    | F12 → printf "f_f%s(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F21 → printf "f_f%s(%s,%s,%s,%s)" f c1 c2 wf2 wf1

  let print_fermion_current_mom_v1 coeff f c wf1 wf2 p1 p2 p12 fusion =
    let c = format_coupling coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s,%s)" f (c1 ~p : p12 ()) (c2 ~p : p12 ()) wf1 wf2
    | F31 → printf "%s_ff(%s,%s,%s,%s)" f (c1 ~p : p12 ()) (c2 ~p : p12 ()) wf2 wf1
    | F23 → printf "f_%sf(%s,%s,%s,%s)" f (c1 ~p : p1 ()) (c2 ~p : p1 ()) wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s,%s)" f (c1 ~p : p2 ()) (c2 ~p : p2 ()) wf2 wf1
    | F12 → printf "f_f%s(%s,%s,%s,%s)" f (c1 ~p : p2 ()) (c2 ~p : p2 ()) wf1 wf2
    | F21 → printf "f_f%s(%s,%s,%s,%s)" f (c1 ~p : p1 ()) (c2 ~p : p1 ()) wf2 wf1

  let print_fermion_current_mom_v2 coeff f c wf1 wf2 p1 p2 p12 fusion =
    let c = format_coupling coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,@,%s,%s,%s)" f (c1 ~p : p12 ()) (c2 ~p : p12 ()) wf1 wf2 p12
    | F31 → printf "%s_ff(%s,%s,@,%s,%s,%s)" f (c1 ~p : p12 ()) (c2 ~p : p12 ()) wf2 wf1 p12
    | F23 → printf "f_%sf(%s,%s,@,%s,%s,%s)" f (c1 ~p : p1 ()) (c2 ~p : p1 ()) wf1 wf2 p1
    | F32 → printf "f_%sf(%s,%s,@,%s,%s,%s)" f (c1 ~p : p2 ()) (c2 ~p : p2 ()) wf2 wf1 p2
    | F12 → printf "f_f%s(%s,%s,@,%s,%s,%s)" f (c1 ~p : p2 ()) (c2 ~p : p2 ()) wf1 wf2 p2
    | F21 → printf "f_f%s(%s,%s,@,%s,%s,%s)" f (c1 ~p : p1 ()) (c2 ~p : p1 ()) wf2 wf1 p1

  let print_fermion_current_mom_ff coeff f c wf1 wf2 p1 p2 p12 fusion =
    let c = format_coupling coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s,%s)" f (c1 ~p : p1 ~q : p2 ()) (c2 ~p : p1 ~q : p2 ()) wf1 wf2
    | F31 → printf "%s_ff(%s,%s,%s,%s)" f (c1 ~p : p1 ~q : p2 ()) (c2 ~p : p1 ~q : p2 ()) wf2 wf1
    | F23 → printf "f_%sf(%s,%s,%s,%s)" f (c1 ~p : p12 ~q : p2 ()) (c2 ~p : p12 ~q : p2 ()) wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s,%s)" f (c1 ~p : p12 ~q : p1 ()) (c2 ~p : p12 ~q : p1 ()) wf2 wf1
    | F12 → printf "f_f%s(%s,%s,%s,%s)" f (c1 ~p : p12 ~q : p1 ()) (c2 ~p : p12 ~q : p1 ()) wf1 wf2
    | F21 → printf "f_f%s(%s,%s,%s,%s)" f (c1 ~p : p12 ~q : p2 ()) (c2 ~p : p12 ~q : p2 ()) wf2 wf1

  let print_current = function
    | coeff, Psibar, VA, Psi → print_fermion_current2 coeff "va"
    | coeff, Psibar, VA2, Psi → print_fermion_current coeff "va2"
    | coeff, Psibar, VA3, Psi → print_fermion_current coeff "va3"
    | coeff, Psibar, V, Psi → print_fermion_current coeff "v"
    | coeff, Psibar, A, Psi → print_fermion_current coeff "a"
    | coeff, Psibar, VL, Psi → print_fermion_current coeff "vl"
    | coeff, Psibar, VR, Psi → print_fermion_current coeff "vr"
    | coeff, Psibar, VLR, Psi → print_fermion_current2 coeff "vlr"
    | coeff, Psibar, SP, Psi → print_fermion_current2 coeff "sp"
    | coeff, Psibar, S, Psi → print_fermion_current coeff "s"
    | coeff, Psibar, P, Psi → print_fermion_current coeff "p"
    | coeff, Psibar, SL, Psi → print_fermion_current coeff "sl"
    | coeff, Psibar, SR, Psi → print_fermion_current coeff "sr"
    | coeff, Psibar, SLR, Psi → print_fermion_current2 coeff "slr"
    | _, Psibar, _, Psi → invalid_arg
          "Targets.Fortran_Fermions:␣no␣superpotential␣here"
    | _, Chibar, _, _ | _, _, _, Chi → invalid_arg
          "Targets.Fortran_Fermions:␣Majorana␣spinors␣not␣handled"
    | _, Gravbar, _, _ | _, _, _, Grav → invalid_arg
          "Targets.Fortran_Fermions:␣Gravitinos␣not␣handled"

  let print_current_mom = function
    | coeff, Psibar, VLRM, Psi → print_fermion_current_mom_v1 coeff "vlr"
    | coeff, Psibar, VAM, Psi → print_fermion_current_mom_ff coeff "va"
    | coeff, Psibar, VA3M, Psi → print_fermion_current_mom_ff coeff "va3"
    | coeff, Psibar, SPM, Psi → print_fermion_current_mom_v1 coeff "sp"
    | coeff, Psibar, TVA, Psi → print_fermion_current_mom_v1 coeff "tva"
    | coeff, Psibar, TVAM, Psi → print_fermion_current_mom_v2 coeff "tvam"
    | coeff, Psibar, TLR, Psi → print_fermion_current_mom_v1 coeff "tlr"
    | coeff, Psibar, TLRM, Psi → print_fermion_current_mom_v2 coeff "tlrm"
    | coeff, Psibar, TRL, Psi → print_fermion_current_mom_v1 coeff "trl"
    | coeff, Psibar, TRLM, Psi → print_fermion_current_mom_v2 coeff "trlm"
    | _, Psibar, _, Psi → invalid_arg
          "Targets.Fortran_Fermions:␣only␣sigma␣tensor␣coupling␣here"
    | _, Chibar, _, _ | _, _, _, Chi → invalid_arg
          "Targets.Fortran_Fermions:␣Majorana␣spinors␣not␣handled"
    | _, Gravbar, _, _ | _, _, _, Grav → invalid_arg
          "Targets.Fortran_Fermions:␣Gravitinos␣not␣handled"

  let print_current_p = function
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Fermions:␣No␣clashing␣arrows␣here"

  let print_current_b = function
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Fermions:␣No␣clashing␣arrows␣here"

  let print_current_g = function
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Fermions:␣No␣gravitinos␣here"

  let print_current_g4 = function
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Fermions:␣No␣gravitinos␣here"

  let reverse_braket vintage bra ket =
    match bra with
    | Spinor → true
    | _ → false

end
```

Majorana Fermions

JR sez' (regarding the Majorana Feynman rules): For this function we need a different approach due to our aim of implementing the fermion vertices with the right line as ingoing (in a calculational sense) and the left line in a fusion as outgoing. In defining all external lines and the fermionic wavefunctions built out of them as ingoing we have to invert the left lines to make them outgoing. This happens by multiplying them with the inverse charge conjugation matrix in an appropriate representation and then transposing it. We must distinguish whether the direction of calculation and the physical direction of the fermion number flow are parallel or antiparallel. In the first case we can use the "normal" Feynman rules for Dirac particles, while in the second, according to the paper of Denner et al., we have to reverse the sign of the vector and antisymmetric bilinears of the Dirac spinors, cf. the *Coupling* module.

Note the subtlety for the left- and righthanded couplings: Only the vector part of these couplings changes in the appropriate cases its sign, changing the chirality to the negative of the opposite. *(JR's probably right, but I need to check myself . . .)*

```
module Fortran_Majorana_Fermions (Names : Target_Fortran_Names.T) : Fermions =
 struct

   open Coupling
   open Format

   let format_coupling coeff c =
     match coeff with
     | 1 → c
     | −1 → "(-" ^ c ^")"
     | coeff → string_of_int coeff ^ "*" ^ c

   let format_coupling_2 coeff c =
     match coeff with
     | 1 → c
     | −1 → "-" ^ c
     | coeff → string_of_int coeff ^ "*" ^ c
```

JR's coupling constant HACK, necessitated by tho's bad design descition.

```
   let fastener s i =
     try
       let offset = (String.index s '(') in
       if ((String.get s (String.length s − 1)) ≢ ')') then
         failwith "fastener:␣wrong␣usage␣of␣parentheses"
       else
         let func_name = (String.sub s 0 offset) and
             tail =
               (String.sub s (succ offset) (String.length s − offset − 2)) in
         if (String.contains func_name ')') ∨
            (String.contains tail '(') ∨
            (String.contains tail ')') then
           failwith "fastener:␣wrong␣usage␣of␣parentheses"
         else
           func_name ^ "(" ^ string_of_int i ^ "," ^ tail ^ ")"
     with
     | Not_found →
         if (String.contains s ')') then
           failwith "fastener:␣wrong␣usage␣of␣parentheses"
         else
           s ^ "(" ^ string_of_int i ^ ")"

   let print_fermion_current coeff f c wf1 wf2 fusion =
     let c = format_coupling coeff c in
     match fusion with
     | F13 | F31 → printf "%s_ff(%s,%s,%s)" f c wf1 wf2
     | F23 | F21 → printf "f_%sf(%s,%s,%s)" f c wf1 wf2
     | F32 | F12 → printf "f_%sf(%s,%s,%s)" f c wf2 wf1

   let print_fermion_current2 coeff f c wf1 wf2 fusion =
     let c = format_coupling_2 coeff c in
     let c1 = fastener c 1 and
         c2 = fastener c 2 in
     match fusion with
     | F13 | F31 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
```

```
  | F23 | F21 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
  | F32 | F12 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1

let print_fermion_current_mom_v1 coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
  | F31 → printf "%s_ff(-(%s),%s,%s,%s)" f c1 c2 wf1 wf2
  | F23 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
  | F32 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1
  | F12 → printf "f_f%s(-(%s),%s,%s,%s)" f c1 c2 wf2 wf1
  | F21 → printf "f_f%s(-(%s),%s,%s,%s)" f c1 c2 wf1 wf2

let print_fermion_current_mom_v1_chiral coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
  | F31 → printf "%s_ff(-(%s),-(%s),%s,%s)" f c2 c1 wf1 wf2
  | F23 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
  | F32 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1
  | F12 → printf "f_f%s(-(%s),-(%s),%s,%s)" f c2 c1 wf2 wf1
  | F21 → printf "f_f%s(-(%s),-(%s),%s,%s)" f c2 c1 wf2 wf1

let print_fermion_current_mom_v2 coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F31 → printf "%s_ff(-(%s),%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F12 → printf "f_f%s(-(%s),%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F21 → printf "f_f%s(-(%s),%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1

let print_fermion_current_mom_v2_chiral coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F31 → printf "%s_ff(-(%s),-(%s),%s,%s,%s)" f c2 c1 wf2 wf1 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F12 → printf "f_f%s(-(%s),-(%s),%s,%s,%s)" f c2 c1 wf1 wf2 p2
  | F21 → printf "f_f%s(-(%s),-(%s),%s,%s,%s)" f c2 c1 wf2 wf1 p1

let print_fermion_current_vector coeff f c wf1 wf2 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s)" f c wf1 wf2
  | F31 → printf "%s_ff(-%s,%s,%s)" f c wf1 wf2
  | F23 → printf "f_%sf(%s,%s,%s)" f c wf1 wf2
  | F32 → printf "f_%sf(%s,%s,%s)" f c wf2 wf1
  | F12 → printf "f_%sf(-%s,%s,%s)" f c wf2 wf1
  | F21 → printf "f_%sf(-%s,%s,%s)" f c wf1 wf2

let print_fermion_current2_vector coeff f c wf1 wf2 fusion =
  let c = format_coupling_2 coeff c in
  let c1 = fastener c 1 and
```

```
        c2 = fastener c 2 in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F31 → printf "%s_ff(-(%s),%s,%s,%s)" f c1 c2 wf1 wf2
    | F23 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1
    | F12 → printf "f_%sf(-(%s),%s,%s,%s)" f c1 c2 wf2 wf1
    | F21 → printf "f_%sf(-(%s),%s,%s,%s)" f c1 c2 wf1 wf2

  let print_fermion_current_chiral coeff f1 f2 c wf1 wf2 fusion =
    let c = format_coupling coeff c in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s)" f1 c wf1 wf2
    | F31 → printf "%s_ff(-%s,%s,%s)" f2 c wf1 wf2
    | F23 → printf "f_%sf(%s,%s,%s)" f1 c wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s)" f1 c wf2 wf1
    | F12 → printf "f_%sf(-%s,%s,%s)" f2 c wf2 wf1
    | F21 → printf "f_%sf(-%s,%s,%s)" f2 c wf1 wf2

  let print_fermion_current2_chiral coeff f c wf1 wf2 fusion =
    let c = format_coupling_2 coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    match fusion with
    | F13 → printf "%s_ff(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F31 → printf "%s_ff(-(%s),-(%s),%s,%s)" f c2 c1 wf1 wf2
    | F23 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf1 wf2
    | F32 → printf "f_%sf(%s,%s,%s,%s)" f c1 c2 wf2 wf1
    | F12 → printf "f_%sf(-(%s),-(%s),%s,%s)" f c2 c1 wf2 wf1
    | F21 → printf "f_%sf(-(%s),-(%s),%s,%s)" f c2 c1 wf1 wf2

  let print_current = function
    | coeff, _, VA, _ → print_fermion_current2_vector coeff "va"
    | coeff, _, V, _ → print_fermion_current_vector coeff "v"
    | coeff, _, A, _ → print_fermion_current coeff "a"
    | coeff, _, VL, _ → print_fermion_current_chiral coeff "vl" "vr"
    | coeff, _, VR, _ → print_fermion_current_chiral coeff "vr" "vl"
    | coeff, _, VLR, _ → print_fermion_current2_chiral coeff "vlr"
    | coeff, _, SP, _ → print_fermion_current2 coeff "sp"
    | coeff, _, S, _ → print_fermion_current coeff "s"
    | coeff, _, P, _ → print_fermion_current coeff "p"
    | coeff, _, SL, _ → print_fermion_current coeff "sl"
    | coeff, _, SR, _ → print_fermion_current coeff "sr"
    | coeff, _, SLR, _ → print_fermion_current2 coeff "slr"
    | coeff, _, POT, _ → print_fermion_current_vector coeff "pot"
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Majorana_Fermions:␣Not␣needed␣in␣the␣models"

  let print_current_p = function
    | coeff, Psi, SL, Psi → print_fermion_current coeff "sl"
    | coeff, Psi, SR, Psi → print_fermion_current coeff "sr"
    | coeff, Psi, SLR, Psi → print_fermion_current2 coeff "slr"
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Majorana_Fermions:␣Not␣needed␣in␣the␣used␣models"

  let print_current_b = function
    | coeff, Psibar, SL, Psibar → print_fermion_current coeff "sl"
    | coeff, Psibar, SR, Psibar → print_fermion_current coeff "sr"
    | coeff, Psibar, SLR, Psibar → print_fermion_current2 coeff "slr"
    | _, _, _, _ → invalid_arg
          "Targets.Fortran_Majorana_Fermions:␣Not␣needed␣in␣the␣used␣models"
```

This function is for the vertices with three particles including two fermions but also a momentum, therefore with a dimensionful coupling constant, e.g. the gravitino vertices. One has to dinstinguish between the two

kinds of canonical orders in the string of gamma matrices. Of course, the direction of the string of gamma matrices is reversed if one goes from the *Gravbar*, _, *Psi* to the *Psibar*, _, *Grav* vertices, and the same is true for the couplings of the gravitino to the Majorana fermions. For more details see the tables in the *coupling* implementation.
We now have to fix the directions of the momenta. For making the compiler happy and because we don't want to make constructions of infinite complexity we list the momentum including vertices without gravitinos here; the pattern matching says that's better. Perhaps we have to find a better name now.

For the cases of MOM, $MOM5$, $MOML$ and $MOMR$ which arise only in BRST transformations we take the mass as a coupling constant. For $VMOM$ we don't need a mass either. These vertices are like kinetic terms and so need not have a coupling constant. By this we avoid a strange and awful construction with a new variable. But be careful with a generalization if you want to use these vertices for other purposes.

```
let format_coupling_mom coeff c =
  match coeff with
  | 1 → c
  | −1 → "(-" ^ c ^")"
  | coeff → string_of_int coeff ^ "*" ^ c

let commute_proj f =
  match f with
  | "moml" → "lmom"
  | "momr" → "rmom"
  | "lmom" → "moml"
  | "rmom" → "momr"
  | "svl" → "svr"
  | "svr" → "svl"
  | "sl" → "sr"
  | "sr" → "sl"
  | "s" → "s"
  | "p" → "p"
  | _ → invalid_arg "Targets:Fortran_Majorana_Fermions:␣wrong␣case"

let print_fermion_current_mom coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling_mom coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F31 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F12 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F21 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1

let print_fermion_current_mom_sign coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling_mom coeff c in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F31 → printf "%s_ff(%s,%s,%s,%s,-(%s))" f c1 c2 wf1 wf2 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F12 → printf "f_%sf(%s,%s,%s,%s,-(%s))" f c1 c2 wf2 wf1 p2
  | F21 → printf "f_%sf(%s,%s,%s,%s,-(%s))" f c1 c2 wf1 wf2 p1

let print_fermion_current_mom_sign_1 coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s)" f c wf1 wf2 p12
  | F31 → printf "%s_ff(%s,%s,%s,-(%s))" f c wf1 wf2 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s)" f c wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s)" f c wf2 wf1 p2
  | F12 → printf "f_%sf(%s,%s,%s,-(%s))" f c wf2 wf1 p2
```

```
    | F21 → printf "f_%sf(%s,%s,%s,-(%s))" f c wf1 wf2 p1

let print_fermion_current_mom_chiral coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling_mom coeff c and
      cf = commute_proj f in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | F13 → printf "%s_ff(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p12
  | F31 → printf "%s_ff(%s,%s,%s,␣%s,-(%s))" cf c1 c2 wf1 wf2 p12
  | F23 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf1 wf2 p1
  | F32 → printf "f_%sf(%s,%s,%s,%s,%s)" f c1 c2 wf2 wf1 p2
  | F12 → printf "f_%sf(%s,%s,%s,%s,-(%s))" cf c1 c2 wf2 wf1 p2
  | F21 → printf "f_%sf(%s,%s,%s,%s,-(%s))" cf c1 c2 wf1 wf2 p1

let print_fermion_g_current coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_grf(%s,%s,%s,%s)" f c wf1 wf2 p12
  | F31 → printf "%s_fgr(%s,%s,%s,%s)" f c wf1 wf2 p12
  | F23 → printf "gr_%sf(%s,%s,%s,%s)" f c wf1 wf2 p1
  | F32 → printf "gr_%sf(%s,%s,%s,%s)" f c wf2 wf1 p2
  | F12 → printf "f_%sgr(%s,%s,%s,%s)" f c wf2 wf1 p2
  | F21 → printf "f_%sgr(%s,%s,%s,%s)" f c wf1 wf2 p1

let print_fermion_g_2_current coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_grf(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p12
  | F31 → printf "%s_fgr(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p12
  | F23 → printf "gr_%sf(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p1
  | F32 → printf "gr_%sf(%s(1),%s(2),%s,%s,%s)" f c c wf2 wf1 p2
  | F12 → printf "f_%sgr(%s(1),%s(2),%s,%s,%s)" f c c wf2 wf1 p2
  | F21 → printf "f_%sgr(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p1

let print_fermion_g_current_rev coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_fgr(%s,%s,%s,%s)" f c wf1 wf2 p12
  | F31 → printf "%s_grf(%s,%s,%s,%s)" f c wf1 wf2 p12
  | F23 → printf "f_%sgr(%s,%s,%s,%s)" f c wf1 wf2 p1
  | F32 → printf "f_%sgr(%s,%s,%s,%s)" f c wf2 wf1 p2
  | F12 → printf "gr_%sf(%s,%s,%s,%s)" f c wf2 wf1 p2
  | F21 → printf "gr_%sf(%s,%s,%s,%s)" f c wf1 wf2 p1

let print_fermion_g_2_current_rev coeff f c wf1 wf2 p1 p2 p12 fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_fgr(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p12
  | F31 → printf "%s_grf(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p12
  | F23 → printf "f_%sgr(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p1
  | F32 → printf "f_%sgr(%s(1),%s(2),%s,%s,%s)" f c c wf2 wf1 p2
  | F12 → printf "gr_%sf(%s(1),%s(2),%s,%s,%s)" f c c wf2 wf1 p2
  | F21 → printf "gr_%sf(%s(1),%s(2),%s,%s,%s)" f c c wf1 wf2 p1

let print_fermion_g_current_vector coeff f c wf1 wf2 _ _ _ fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_grf(%s,%s,%s)" f c wf1 wf2
  | F31 → printf "%s_fgr(-%s,%s,%s)" f c wf1 wf2
  | F23 → printf "gr_%sf(%s,%s,%s)" f c wf1 wf2
  | F32 → printf "gr_%sf(%s,%s,%s)" f c wf2 wf1
  | F12 → printf "f_%sgr(-%s,%s,%s)" f c wf2 wf1
  | F21 → printf "f_%sgr(-%s,%s,%s)" f c wf1 wf2
```

```
let print_fermion_g_current_vector_rev coeff f c wf1 wf2 _ _ _ fusion =
  let c = format_coupling coeff c in
  match fusion with
  | F13 → printf "%s_fgr(%s,%s,%s)" f c wf1 wf2
  | F31 → printf "%s_grf(-%s,%s,%s)" f c wf1 wf2
  | F23 → printf "f_%sgr(%s,%s,%s)" f c wf1 wf2
  | F32 → printf "f_%sgr(%s,%s,%s)" f c wf2 wf1
  | F12 → printf "gr_%sf(-%s,%s,%s)" f c wf2 wf1
  | F21 → printf "gr_%sf(-%s,%s,%s)" f c wf1 wf2

let print_current_g = function
  | coeff, _, MOM, _ → print_fermion_current_mom_sign coeff "mom"
  | coeff, _, MOM5, _ → print_fermion_current_mom coeff "mom5"
  | coeff, _, MOML, _ → print_fermion_current_mom_chiral coeff "moml"
  | coeff, _, MOMR, _ → print_fermion_current_mom_chiral coeff "momr"
  | coeff, _, LMOM, _ → print_fermion_current_mom_chiral coeff "lmom"
  | coeff, _, RMOM, _ → print_fermion_current_mom_chiral coeff "rmom"
  | coeff, _, VMOM, _ → print_fermion_current_mom_sign_1 coeff "vmom"
  | coeff, Gravbar, S, _ → print_fermion_g_current coeff "s"
  | coeff, Gravbar, SL, _ → print_fermion_g_current coeff "sl"
  | coeff, Gravbar, SR, _ → print_fermion_g_current coeff "sr"
  | coeff, Gravbar, SLR, _ → print_fermion_g_2_current coeff "slr"
  | coeff, Gravbar, P, _ → print_fermion_g_current coeff "p"
  | coeff, Gravbar, V, _ → print_fermion_g_current coeff "v"
  | coeff, Gravbar, VLR, _ → print_fermion_g_2_current coeff "vlr"
  | coeff, Gravbar, POT, _ → print_fermion_g_current_vector coeff "pot"
  | coeff, _, S, Grav → print_fermion_g_current_rev coeff "s"
  | coeff, _, SL, Grav → print_fermion_g_current_rev coeff "sl"
  | coeff, _, SR, Grav → print_fermion_g_current_rev coeff "sr"
  | coeff, _, SLR, Grav → print_fermion_g_2_current_rev coeff "slr"
  | coeff, _, P, Grav → print_fermion_g_current_rev (−coeff) "p"
  | coeff, _, V, Grav → print_fermion_g_current_rev coeff "v"
  | coeff, _, VLR, Grav → print_fermion_g_2_current_rev coeff "vlr"
  | coeff, _, POT, Grav → print_fermion_g_current_vector_rev coeff "pot"
  | _, _, _, _ → invalid_arg
      "Targets.Fortran_Majorana_Fermions:␣not␣used␣in␣the␣models"

let print_current_mom = function
  | coeff, _, TVA, _ → print_fermion_current_mom_v1 coeff "tva"
  | coeff, _, TVAM, _ → print_fermion_current_mom_v2 coeff "tvam"
  | coeff, _, TLR, _ → print_fermion_current_mom_v1_chiral coeff "tlr"
  | coeff, _, TLRM, _ → print_fermion_current_mom_v2_chiral coeff "tlrm"
  | _, _, _, _ → invalid_arg
        "Targets.Fortran_Majorana_Fermions:␣Not␣needed␣in␣the␣models"
```

We need support for dimension-5 vertices with two fermions and two bosons, appearing in theories of supergravity and also together with in insertions of the supersymmetric current. There is a canonical order *fermionbar*, *boson_1*, *boson_2*, *fermion*, so what one has to do is a mapping from the fusions *F123* etc. to the order of the three wave functions *wf1*, *wf2* and *wf3*.
The function *d_p* (for distinct the particle) distinguishes which particle (scalar or vector) must be fused to in the special functions.

```
let d_p = function
  | 1, ("sv"|"pv"|"svl"|"svr"|"slrv") → "1"
  | 1, _ → ""
  | 2, ("sv"|"pv"|"svl"|"svr"|"slrv") → "2"
  | 2, _ → ""
  | _, _ → invalid_arg "Targets.Fortran_Majorana_Fermions:␣not␣used"

let wf_of_f wf1 wf2 wf3 f =
  match f with
  | (F123 | F423) → [wf2; wf3; wf1]
  | (F213 | F243 | F143 | F142 | F413 | F412) → [wf1; wf3; wf2]
  | (F132 | F432) → [wf3; wf2; wf1]
```

```
    | (F231 | F234 | F134 | F124 | F431 | F421) → [wf1; wf2; wf3]
    | (F312 | F342) → [wf3; wf1; wf2]
    | (F321 | F324 | F314 | F214 | F341 | F241) → [wf2; wf1; wf3]

let print_fermion_g4_brs_vector_current coeff f c wf1 wf2 wf3 fusion =
  let cf = commute_proj f and
      cp = format_coupling coeff c and
      cm = if f = "pv" then
        format_coupling coeff c
      else
        format_coupling (−coeff) c
  and
      d1 = d_p (1,f) and
      d2 = d_p (2,f) and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
      printf "f_%sf(%s,%s,%s,%s)" cf cm f1 f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
      printf "f_%sf(%s,%s,%s,%s)" f cp f1 f2 f3
  | (F134 | F143 | F314) → printf "%s%s_ff(%s,%s,%s,%s)" f d1 cp f1 f2 f3
  | (F124 | F142 | F214) → printf "%s%s_ff(%s,%s,%s,%s)" f d2 cp f1 f2 f3
  | (F413 | F431 | F341) → printf "%s%s_ff(%s,%s,%s,%s)" cf d1 cm f1 f2 f3
  | (F241 | F412 | F421) → printf "%s%s_ff(%s,%s,%s,%s)" cf d2 cm f1 f2 f3

let print_fermion_g4_svlr_current coeff _ c wf1 wf2 wf3 fusion =
  let c = format_coupling_2 coeff c and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
      printf "f_svlrf(-(%s),-(%s),%s,%s,%s)" c2 c1 f1 f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
      printf "f_svlrf(%s,%s,%s,%s,%s)" c1 c2 f1 f2 f3
  | (F134 | F143 | F314) →
      printf "svlr2_ff(%s,%s,%s,%s,%s)" c1 c2 f1 f2 f3
  | (F124 | F142 | F214) →
      printf "svlr1_ff(%s,%s,%s,%s,%s)" c1 c2 f1 f2 f3
  | (F413 | F431 | F341) →
      printf "svlr2_ff(-(%s),-(%s),%s,%s,%s)" c2 c1 f1 f2 f3
  | (F241 | F412 | F421) →
      printf "svlr1_ff(-(%s),-(%s),%s,%s,%s)" c2 c1 f1 f2 f3

let print_fermion_s2_current coeff f c wf1 wf2 wf3 fusion =
  let cp = format_coupling coeff c and
      cm = if f = "p" then
        format_coupling (−coeff) c
      else
        format_coupling coeff c
  and
      cf = commute_proj f and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
      printf "%s␣*␣f_%sf(%s,%s,%s)" f1 cf cm f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
```

```
        printf "%s␣*␣f_%sf(%s,%s,%s)" f1 f cp f2 f3
  | (F134 | F143 | F314) →
        printf "%s␣*␣%s_ff(%s,%s,%s)" f2 f cp f1 f3
  | (F124 | F142 | F214) →
        printf "%s␣*␣%s_ff(%s,%s,%s)" f2 f cp f1 f3
  | (F413 | F431 | F341) →
        printf "%s␣*␣%s_ff(%s,%s,%s)" f2 cf cm f1 f3
  | (F241 | F412 | F421) →
        printf "%s␣*␣%s_ff(%s,%s,%s)" f2 cf cm f1 f3

let print_fermion_s2p_current coeff f c wf1 wf2 wf3 fusion =
  let c = format_coupling_2 coeff c and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "%s␣*␣f_%sf(%s,-(%s),%s,%s)" f1 f c1 c2 f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "%s␣*␣f_%sf(%s,%s,%s,%s)" f1 f c1 c2 f2 f3
  | (F134 | F143 | F314) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c1 c2 f1 f3
  | (F124 | F142 | F214) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c1 c2 f1 f3
  | (F413 | F431 | F341) →
        printf "%s␣*␣%s_ff(%s,-(%s),%s,%s)" f2 f c1 c2 f1 f3
  | (F241 | F412 | F421) →
        printf "%s␣*␣%s_ff(%s,-(%s),%s,%s)" f2 f c1 c2 f1 f3

let print_fermion_s2lr_current coeff f c wf1 wf2 wf3 fusion =
  let c = format_coupling_2 coeff c and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  let c1 = fastener c 1 and
      c2 = fastener c 2 in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "%s␣*␣f_%sf(%s,%s,%s,%s)" f1 f c2 c1 f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "%s␣*␣f_%sf(%s,%s,%s,%s)" f1 f c1 c2 f2 f3
  | (F134 | F143 | F314) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c1 c2 f1 f3
  | (F124 | F142 | F214) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c1 c2 f1 f3
  | (F413 | F431 | F341) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c2 c1 f1 f3
  | (F241 | F412 | F421) →
        printf "%s␣*␣%s_ff(%s,%s,%s,%s)" f2 f c2 c1 f1 f3

let print_fermion_g4_current coeff f c wf1 wf2 wf3 fusion =
  let c = format_coupling coeff c and
      f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
      f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
      f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
  match fusion with
  | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "f_%sgr(-%s,%s,%s,%s)" f c f1 f2 f3
  | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "gr_%sf(%s,%s,%s,%s)" f c f1 f2 f3
  | (F134 | F143 | F314 | F124 | F142 | F214) →
```

```
          printf "%s_grf(%s,%s,%s,%s)" f c f1 f2 f3
    | (F413 | F431 | F341 | F241 | F412 | F421) →
          printf "%s_fgr(-%s,%s,%s,%s)" f c f1 f2 f3

  let print_fermion_2_g4_current coeff f c wf1 wf2 wf3 fusion =
    let f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    let c = format_coupling_2 coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
          printf "f_%sgr(-(%s),-(%s),%s,%s,%s)" f c2 c1 f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
          printf "gr_%sf(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F134 | F143 | F314 | F124 | F142 | F214) →
          printf "%s_grf(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F413 | F431 | F341 | F241 | F412 | F421) →
          printf "%s_fgr(-(%s),-(%s),%s,%s,%s)" f c2 c1 f1 f2 f3

  let print_fermion_g4_current_rev coeff f c wf1 wf2 wf3 fusion =
    let c = format_coupling coeff c and
        f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
          printf "f_%sgr(%s,%s,%s,%s)" f c f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
          printf "gr_%sf(-%s,%s,%s,%s)" f c f1 f2 f3
    | (F134 | F143 | F314 | F124 | F142 | F214) →
          printf "%s_grf(-%s,%s,%s,%s)" f c f1 f2 f3
    | (F413 | F431 | F341 | F241 | F412 | F421) →
          printf "%s_fgr(%s,%s,%s,%s)" f c f1 f2 f3
```

Here we have to distinguish which of the two bosons is produced in the fusion of three particles which include both fermions.

```
  let print_fermion_g4_vector_current coeff f c wf1 wf2 wf3 fusion =
    let c = format_coupling coeff c and
        d1 = d_p (1,f) and
        d2 = d_p (2,f) and
        f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
          printf "f_%sgr(%s,%s,%s,%s)" f c f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
          printf "gr_%sf(%s,%s,%s,%s)" f c f1 f2 f3
    | (F134 | F143 | F314) → printf "%s%s_grf(%s,%s,%s,%s)" f d1 c f1 f2 f3
    | (F124 | F142 | F214) → printf "%s%s_grf(%s,%s,%s,%s)" f d2 c f1 f2 f3
    | (F413 | F431 | F341) → printf "%s%s_fgr(%s,%s,%s,%s)" f d1 c f1 f2 f3
    | (F241 | F412 | F421) → printf "%s%s_fgr(%s,%s,%s,%s)" f d2 c f1 f2 f3

  let print_fermion_2_g4_vector_current coeff f c wf1 wf2 wf3 fusion =
    let d1 = d_p (1,f) and
        d2 = d_p (2,f) and
        f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    let c = format_coupling_2 coeff c in
    let c1 = fastener c 1 and
```

```
        c2 = fastener c 2 in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "f_%sgr(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "gr_%sf(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F134 | F143 | F314) → printf "%s%s_grf(%s,%s,%s,%s,%s)" f d1 c1 c2 f1 f2 f3
    | (F124 | F142 | F214) → printf "%s%s_grf(%s,%s,%s,%s,%s)" f d2 c1 c2 f1 f2 f3
    | (F413 | F431 | F341) → printf "%s%s_fgr(%s,%s,%s,%s,%s)" f d1 c1 c2 f1 f2 f3
    | (F241 | F412 | F421) → printf "%s%s_fgr(%s,%s,%s,%s,%s)" f d2 c1 c2 f1 f2 f3

let print_fermion_g4_vector_current_rev coeff f c wf1 wf2 wf3 fusion =
    let c = format_coupling coeff c and
        d1 = d_p (1,f) and
        d2 = d_p (2,f) and
        f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "gr_%sf(%s,%s,%s,%s)" f c f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "f_%sgr(%s,%s,%s,%s)" f c f1 f2 f3
    | (F134 | F143 | F314) → printf "%s%s_fgr(%s,%s,%s,%s)" f d1 c f1 f2 f3
    | (F124 | F142 | F214) → printf "%s%s_fgr(%s,%s,%s,%s)" f d2 c f1 f2 f3
    | (F413 | F431 | F341) → printf "%s%s_grf(%s,%s,%s,%s)" f d1 c f1 f2 f3
    | (F241 | F412 | F421) → printf "%s%s_grf(%s,%s,%s,%s)" f d2 c f1 f2 f3

let print_fermion_2_g4_current_rev coeff f c wf1 wf2 wf3 fusion =
    let c = format_coupling_2 coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 and
        d1 = d_p (1,f) and
        d2 = d_p (2,f) in
    let f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
        printf "gr_%sf(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F423 | F243 | F432 | F234 | F342 | F324) →
        printf "f_%sgr(-(%s),-(%s),%s,%s,%s)" f c1 c2 f1 f2 f3
    | (F134 | F143 | F314) →
        printf "%s%s_fgr(-(%s),-(%s),%s,%s,%s)" f d1 c1 c2 f1 f2 f3
    | (F124 | F142 | F214) →
        printf "%s%s_fgr(-(%s),-(%s),%s,%s,%s)" f d2 c1 c2 f1 f2 f3
    | (F413 | F431 | F341) →
        printf "%s%s_grf(%s,%s,%s,%s,%s)" f d1 c1 c2 f1 f2 f3
    | (F241 | F412 | F421) →
        printf "%s%s_grf(%s,%s,%s,%s,%s)" f d2 c1 c2 f1 f2 f3

let print_fermion_2_g4_vector_current_rev coeff f c wf1 wf2 wf3 fusion =
    (* Here we put in the extra minus sign from the coeff. *)
    let c = format_coupling coeff c in
    let c1 = fastener c 1 and
        c2 = fastener c 2 in
    let d1 = d_p (1,f) and
        d2 = d_p (2,f) and
        f1 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 0) and
        f2 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 1) and
        f3 = (List.nth (wf_of_f wf1 wf2 wf3 fusion) 2) in
    match fusion with
    | (F123 | F213 | F132 | F231 | F312 | F321) →
```

```
          printf "gr_%sf(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
      | (F423 | F243 | F432 | F234 | F342 | F324) →
          printf "f_%sgr(%s,%s,%s,%s,%s)" f c1 c2 f1 f2 f3
      | (F134 | F143 | F314) → printf "%s%s_fgr(%s,%s,%s,%s,%s)" f d1 c1 c2 f1 f2 f3
      | (F124 | F142 | F214) → printf "%s%s_fgr(%s,%s,%s,%s,%s)" f d2 c1 c2 f1 f2 f3
      | (F413 | F431 | F341) → printf "%s%s_grf(%s,%s,%s,%s,%s)" f d1 c1 c2 f1 f2 f3
      | (F241 | F412 | F421) → printf "%s%s_grf(%s,%s,%s,%s,%s)" f d2 c1 c2 f1 f2 f3

    let print_current_g4 = function
      | coeff, Gravbar, S2, _ → print_fermion_g4_current coeff "s2"
      | coeff, Gravbar, SV, _ → print_fermion_g4_vector_current coeff "sv"
      | coeff, Gravbar, SLV, _ → print_fermion_g4_vector_current coeff "slv"
      | coeff, Gravbar, SRV, _ → print_fermion_g4_vector_current coeff "srv"
      | coeff, Gravbar, SLRV, _ → print_fermion_2_g4_vector_current coeff "slrv"
      | coeff, Gravbar, PV, _ → print_fermion_g4_vector_current coeff "pv"
      | coeff, Gravbar, V2, _ → print_fermion_g4_current coeff "v2"
      | coeff, Gravbar, V2LR, _ → print_fermion_2_g4_current coeff "v2lr"
      | _, Gravbar, _, _ → invalid_arg "print_current_g4:␣not␣implemented"
      | coeff, _, S2, Grav → print_fermion_g4_current_rev coeff "s2"
      | coeff, _, SV, Grav → print_fermion_g4_vector_current_rev (−coeff) "sv"
      | coeff, _, SLV, Grav → print_fermion_g4_vector_current_rev (−coeff) "slv"
      | coeff, _, SRV, Grav → print_fermion_g4_vector_current_rev (−coeff) "srv"
      | coeff, _, SLRV, Grav → print_fermion_2_g4_vector_current_rev coeff "slrv"
      | coeff, _, PV, Grav → print_fermion_g4_vector_current_rev coeff "pv"
      | coeff, _, V2, Grav → print_fermion_g4_vector_current_rev coeff "v2"
      | coeff, _, V2LR, Grav → print_fermion_2_g4_current_rev coeff "v2lr"
      | _, _, _, Grav → invalid_arg "print_current_g4:␣not␣implemented"
      | coeff, _, S2, _ → print_fermion_s2_current coeff "s"
      | coeff, _, P2, _ → print_fermion_s2_current coeff "p"
      | coeff, _, S2P, _ → print_fermion_s2p_current coeff "sp"
      | coeff, _, S2L, _ → print_fermion_s2_current coeff "sl"
      | coeff, _, S2R, _ → print_fermion_s2_current coeff "sr"
      | coeff, _, S2LR, _ → print_fermion_s2lr_current coeff "slr"
      | coeff, _, V2, _ → print_fermion_g4_brs_vector_current coeff "v2"
      | coeff, _, SV, _ → print_fermion_g4_brs_vector_current coeff "sv"
      | coeff, _, PV, _ → print_fermion_g4_brs_vector_current coeff "pv"
      | coeff, _, SLV, _ → print_fermion_g4_brs_vector_current coeff "svl"
      | coeff, _, SRV, _ → print_fermion_g4_brs_vector_current coeff "svr"
      | coeff, _, SLRV, _ → print_fermion_g4_svlr_current coeff "svlr"
      | _, _, V2LR, _ → invalid_arg "Targets.print_current:␣not␣available"

    let reverse_braket vintage bra ket =
      if vintage then
        false
      else
        match bra, ket with
        | Majorana, Majorana :: _ → true
        | _, _ → false

  end
```

Currents for Coupling.V3 and Coupling.V4

```
module type T =
  sig
    type amplitude
    type constant
    type wf
    type rhs
    val print_current_V3 :
      (amplitude → (amplitude → wf → int) → wf → string) → (wf → string) →
      amplitude → (amplitude → wf → int) → rhs →
      constant Coupling.vertex3 → Coupling.fuse2 → constant → unit
    val print_current_V4 :
      (amplitude → (amplitude → wf → int) → wf → string) → (wf → string) →
      amplitude → (amplitude → wf → int) → rhs →
      constant Coupling.vertex4 → Coupling.fuse3 → constant → unit
  end

module type Maker =
  functor (N : Target_Fortran_Names.T) → functor (F : Fermion_Maker) →
  functor (FM : Fusion.Maker) → functor (P : Momentum.T) → functor (M : Model.T) → T
  with type amplitude = Fusion.Multi(FM)(P)(M).amplitude
   and type constant = Orders.Slice(Colorize.It(M)).constant
   and type wf = FM(P)(M).wf
   and type rhs = FM(P)(M).rhs

module Make_Fortran (Names : Target_Fortran_Names.T) (Fermion_Maker : Fermion_Maker)
          (FM : Fusion.Maker) (P : Momentum.T) (M : Model.T) =
  struct

    open Coupling
    open Format

    module Fermions = Fermion_Maker(Names)

    module CM = Colorize.It(M)
    module SCM = Orders.Slice(Colorize.It(M))
    module F = FM(P)(M)
    module CF = Fusion.Multi(FM)(P)(M)

    type amplitude = CF.amplitude
    type constant = Orders.Slice(Colorize.It(M)).constant
    type wf = F.wf
    type rhs = F.rhs

    let children2 rhs =
      match F.children rhs with
      | [wf1; wf2] → (wf1, wf2)
      | _ → failwith "Targets.children2:␣can't␣happen"

    let children3 rhs =
      match F.children rhs with
      | [wf1; wf2; wf3] → (wf1, wf2, wf3)
      | _ → invalid_arg "Targets.children3:␣can't␣happen"
```

Note that it is (marginally) faster to multiply the two scalar products with the coupling constant than the four vector components.

This could be part of `omegalib` as well …

```
    let format_coeff = function
      | 1 → ""
      | −1 → "-"
      | coeff → "(" ^ string_of_int coeff ^ ")*"

    let format_coupling coeff c =
      match coeff with
      | 1 → c
      | −1 → "(-" ^ c ^")"
      | coeff → string_of_int coeff ^ "*" ^ c
```

The following is error prone and should be generated automagically.

```
    let print_vector4 c wf1 wf2 wf3 fusion (coeff, contraction) =
      match contraction, fusion with
      | C_12_34, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
      | C_13_42, (F241 | F421 | F243 | F423 | F132 | F312 | F134 | F314)
```

```
  | C_14_23, (F231 | F321 | F234 | F324 | F142 | F412 | F143 | F413) →
      printf "((%s%s)*(%s*%s))*%s" (format_coeff coeff) c wf1 wf2 wf3
  | C_12_34, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
  | C_13_42, (F124 | F142 | F324 | F342 | F213 | F231 | F413 | F431)
  | C_14_23, (F123 | F132 | F423 | F432 | F214 | F241 | F314 | F341) →
      printf "((%s%s)*(%s*%s))*%s" (format_coeff coeff) c wf2 wf3 wf1
  | C_12_34, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241)
  | C_13_42, (F214 | F412 | F234 | F432 | F123 | F321 | F143 | F341)
  | C_14_23, (F213 | F312 | F243 | F342 | F124 | F421 | F134 | F431) →
      printf "((%s%s)*(%s*%s))*%s" (format_coeff coeff) c wf1 wf3 wf2

let print_vector4_t_0 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
  | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
  | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
  | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
      printf "g_dim8g3_t_0(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
  | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
  | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
  | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
      printf "g_dim8g3_t_0(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
  | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
  | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
  | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
      printf "g_dim8g3_t_0(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

let print_vector4_t_1 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
  | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
  | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
  | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
      printf "g_dim8g3_t_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
  | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
  | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
  | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
      printf "g_dim8g3_t_1(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
  | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
  | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
  | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
      printf "g_dim8g3_t_1(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

let print_vector4_t_2 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
  | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
  | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
  | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
      printf "g_dim8g3_t_2(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
  | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
  | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
  | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
      printf "g_dim8g3_t_2(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
  | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
  | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
  | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
      printf "g_dim8g3_t_2(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

let print_vector4_m_0 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
  | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
  | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
  | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
      printf "g_dim8g3_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
  | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
```

```
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "g_dim8g3_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "g_dim8g3_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

  let print_vector4_m_1 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "g_dim8g3_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "g_dim8g3_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "g_dim8g3_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

   let print_vector4_m_7 c wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "g_dim8g3_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "g_dim8g3_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "g_dim8g3_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2

  let print_add_vector4 c wf1 wf2 wf3 fusion (coeff, contraction) =
    printf "@␣+␣";
    print_vector4 c wf1 wf2 wf3 fusion (coeff, contraction)

  let print_vector4_km c pa pb wf1 wf2 wf3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
    | C_13_42, (F241 | F421 | F243 | F423 | F132 | F312 | F134 | F314)
    | C_14_23, (F231 | F321 | F234 | F324 | F142 | F412 | F143 | F413) →
        printf "((%s%s%s+%s))*(%s*%s))*%s"
          (format_coeff coeff) c pa pb wf1 wf2 wf3
    | C_12_34, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
    | C_13_42, (F124 | F142 | F324 | F342 | F213 | F231 | F413 | F431)
    | C_14_23, (F123 | F132 | F423 | F432 | F214 | F241 | F314 | F341) →
        printf "((%s%s%s+%s))*(%s*%s))*%s"
          (format_coeff coeff) c pa pb wf2 wf3 wf1
    | C_12_34, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241)
    | C_13_42, (F214 | F412 | F234 | F432 | F123 | F321 | F143 | F341)
    | C_14_23, (F213 | F312 | F243 | F342 | F124 | F421 | F134 | F431) →
        printf "((%s%s%s+%s))*(%s*%s))*%s"
          (format_coeff coeff) c pa pb wf1 wf3 wf2

  let print_vector4_km_t_0 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

  let print_vector4_km_t_1 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

  let print_vector4_km_t_2 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_2(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_2(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_2(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

  let print_vector4_km_t_rsi c pa pb pc wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))*((%s+%s)*(%s+%s)
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3 pa pb pa pb pb pc pb pc
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
```

```
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        printf "@[(%s%s%s+%s)*g_dim8g3_t_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))*((%s+%s)*(%s+%s)
          (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2 pa pb pa pb pa pc pa pc

let print_vector4_km_m_0 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_0(cmplx(1,kind=default),cmplx(1,kind=default),@␣%s,%s,%s,%s
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
        else
          printf "@[((%s%s%s+%s))*g_dim8g3_m_0(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=de
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_0(cmplx(1,kind=default),cmplx(1,kind=default),@␣%s,%s,%s,%s
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
        else
          printf "@[(%s%s%s+%s)*g_dim8g3_m_0(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=defa
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_0(cmplx(1,kind=default),cmplx(1,kind=default),@␣%s,%s,%s,%s
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2
        else
          printf "@[(%s%s%s+%s)*g_dim8g3_m_0(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=defa
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

let print_vector4_km_m_1 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
  match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(1,kind=default),cmplx(1,kind=default),@␣%s,%s,%s,%s
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
        else
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=defa
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(1,kind=default),cmplx(1,kind=default),@␣␣%s,%s,%s,%
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
        else
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=defa
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(1,kind=default),cmplx(1,kind=default),@␣%s,%s,%s,%s
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2
        else
          printf "@[(%s%s%s+%s)*g_dim8g3_m_1(cmplx(costhw**(-2),kind=default),cmplx(costhw**2,kind=def
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

let print_vector4_km_m_7 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F234 | F243 | F134 | F143 | F421 | F321 | F412 | F312)
    | C_13_42, (F324 | F342 | F124 | F142 | F431 | F231 | F413 | F213)
    | C_14_23, (F423 | F432 | F123 | F132 | F341 | F241 | F314 | F214) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(1,kind=default),cmplx(1,kind=default),cmplx(1,kin
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
        else
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(costhw**(-2),kind=default),cmplx(1,kind=default),
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F324 | F314 | F423 | F413 | F142 | F132 | F241 | F231)
    | C_13_42, (F234 | F214 | F432 | F412 | F143 | F123 | F341 | F321)
    | C_14_23, (F243 | F213 | F342 | F312 | F134 | F124 | F431 | F421) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(1,kind=default),cmplx(1,kind=default),cmplx(1,kin
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
        else
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(costhw**(-2),kind=default),cmplx(1,kind=default),
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F342 | F341 | F432 | F431 | F124 | F123 | F214 | F213)
    | C_13_42, (F243 | F241 | F423 | F421 | F134 | F132 | F314 | F312)
    | C_14_23, (F234 | F231 | F324 | F321 | F143 | F142 | F413 | F412) →
        if (String.contains c 'w' ∨ String.contains c '4') then
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(1,kind=default),cmplx(1,kind=default),cmplx(1,kin
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2
        else
          printf "@[(%s%s%s+%s)*@␣g_dim8g3_m_7(cmplx(costhw**(-2),kind=default),cmplx(1,kind=default),
            (format_coeff coeff) c pa pb wf3 p3 wf1 p1 wf2 p2

let print_add_vector4_km c pa pb wf1 wf2 wf3 fusion (coeff, contraction) =
    printf "@␣+␣";
    print_vector4_km c pa pb wf1 wf2 wf3 fusion (coeff, contraction)

let print_dscalar4 c wf1 wf2 wf3 p1 p2 p3 p123
    fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
    | C_13_42, (F241 | F421 | F243 | F423 | F132 | F312 | F134 | F314)
    | C_14_23, (F231 | F321 | F234 | F324 | F142 | F412 | F143 | F413) →
        printf "((%s%s)*(%s*%s)*(%s*%s)*%s*%s*%s)"
          (format_coeff coeff) c p1 p2 p3 p123 wf1 wf2 wf3
    | C_12_34, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
    | C_13_42, (F124 | F142 | F324 | F342 | F213 | F231 | F413 | F431)
    | C_14_23, (F123 | F132 | F423 | F432 | F214 | F241 | F314 | F341) →
        printf "((%s%s)*(%s*%s)*(%s*%s)*%s*%s*%s)"
          (format_coeff coeff) c p2 p3 p1 p123 wf1 wf2 wf3
    | C_12_34, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241)
    | C_13_42, (F214 | F412 | F234 | F432 | F123 | F321 | F143 | F341)
    | C_14_23, (F213 | F312 | F243 | F342 | F124 | F421 | F134 | F431) →
        printf "((%s%s)*(%s*%s)*(%s*%s)*%s*%s*%s)"
          (format_coeff coeff) c p1 p3 p2 p123 wf1 wf2 wf3

let print_add_dscalar4 c wf1 wf2 wf3 p1 p2 p3 p123
    fusion (coeff, contraction) =
    printf "@␣+␣";
    print_dscalar4 c wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction)

let print_dscalar2_vector2 c wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F123 | F213 | F124 | F214) →
```

```
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p1 p2 wf1 wf2 wf3
  | C_12_34, (F134 | F143 | F234 | F243) →
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p1 p123 wf2 wf3 wf1
  | C_12_34, (F132 | F231 | F142 | F241) →
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p1 p3 wf1 wf3 wf2
  | C_12_34, (F312 | F321 | F412 | F421) →
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p2 p3 wf2 wf3 wf1
  | C_12_34, (F314 | F413 | F324 | F423) →
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p2 p123 wf1 wf3 wf2
  | C_12_34, (F341 | F431 | F342 | F432) →
      printf "(%s%s)*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c p3 p123 wf1 wf2 wf3
  | C_13_42, (F123 | F214)
  | C_14_23, (F124 | F213) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf1 p1 wf3 wf2 p2
  | C_13_42, (F124 | F213)
  | C_14_23, (F123 | F214) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf2 p2 wf3 wf1 p1
  | C_13_42, (F132 | F241)
  | C_14_23, (F142 | F231) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf1 p1 wf2 wf3 p3
  | C_13_42, (F142 | F231)
  | C_14_23, (F132 | F241) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf3 p3 wf2 wf1 p1
  | C_13_42, (F312 | F421)
  | C_14_23, (F412 | F321) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf2 p2 wf1 wf3 p3
  | C_13_42, (F321 | F412)
  | C_14_23, (F421 | F312) →
      printf "((%s%s)*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c wf3 p3 wf1 wf2 p2
  | C_13_42, (F134 | F243)
  | C_14_23, (F143 | F234) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf3 p123 wf1 p1 wf2
  | C_13_42, (F143 | F234)
  | C_14_23, (F134 | F243) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf2 p123 wf1 p1 wf3
  | C_13_42, (F314 | F423)
  | C_14_23, (F413 | F324) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf3 p123 wf2 p2 wf1
  | C_13_42, (F324 | F413)
  | C_14_23, (F423 | F314) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf1 p123 wf2 p2 wf3
  | C_13_42, (F341 | F432)
  | C_14_23, (F431 | F342) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf2 p123 wf3 p3 wf1
  | C_13_42, (F342 | F431)
  | C_14_23, (F432 | F341) →
      printf "((%s%s)*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c wf1 p123 wf3 p3 wf2

let print_add_dscalar2_vector2 c wf1 wf2 wf3 p1 p2 p3 p123
    fusion (coeff, contraction) =
  printf "@␣+␣";
  print_dscalar2_vector2 c wf1 wf2 wf3 p1 p2 p3 p123
    fusion (coeff, contraction)

let print_dscalar2_vector2_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction) =
  match contraction, fusion with
  | C_12_34, (F123 | F213 | F124 | F214) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p1 p2 wf1 wf2 wf3
  | C_12_34, (F134 | F143 | F234 | F243) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p1 p123 wf2 wf3 wf1
  | C_12_34, (F132 | F231 | F142 | F241) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p1 p3 wf1 wf3 wf2
  | C_12_34, (F312 | F321 | F412 | F421) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p2 p3 wf2 wf3 wf1
  | C_12_34, (F314 | F413 | F324 | F423) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p2 p123 wf1 wf3 wf2
  | C_12_34, (F341 | F431 | F342 | F432) →
      printf "(%s%s%s+%s))*(%s*%s)*(%s*%s)*%s"
        (format_coeff coeff) c pa pb p3 p123 wf1 wf2 wf3
  | C_13_42, (F123 | F214)
  | C_14_23, (F124 | F213) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf1 p1 wf3 wf2 p2
  | C_13_42, (F124 | F213)
  | C_14_23, (F123 | F214) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf2 p2 wf3 wf1 p1
  | C_13_42, (F132 | F241)
  | C_14_23, (F142 | F231) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf1 p1 wf2 wf3 p3
  | C_13_42, (F142 | F231)
  | C_14_23, (F132 | F241) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf3 p3 wf2 wf1 p1
  | C_13_42, (F312 | F421)
  | C_14_23, (F412 | F321) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf2 p2 wf1 wf3 p3
  | C_13_42, (F321 | F412)
  | C_14_23, (F421 | F312) →
      printf "((%s%s%s+%s))*(%s*%s*%s)*%s*%s)"
        (format_coeff coeff) c pa pb wf3 p3 wf1 wf2 p2
  | C_13_42, (F134 | F243)
  | C_14_23, (F143 | F234) →
      printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
        (format_coeff coeff) c pa pb wf3 p123 wf1 p1 wf2
  | C_13_42, (F143 | F234)
  | C_14_23, (F134 | F243) →
      printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
```

```
            (format_coeff coeff) c pa pb wf2 p123 wf1 p1 wf3
    | C_13_42, (F314 | F423)
    | C_14_23, (F413 | F324) →
        printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
            (format_coeff coeff) c pa pb wf3 p123 wf2 p2 wf1
    | C_13_42, (F324 | F413)
    | C_14_23, (F423 | F314) →
        printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
            (format_coeff coeff) c pa pb wf1 p123 wf2 p2 wf3
    | C_13_42, (F341 | F432)
    | C_14_23, (F431 | F342) →
        printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
            (format_coeff coeff) c pa pb wf2 p123 wf3 p3 wf1
    | C_13_42, (F342 | F431)
    | C_14_23, (F432 | F341) →
        printf "((%s%s%s+%s))*(%s*%s)*(%s*%s*%s))"
            (format_coeff coeff) c pa pb wf1 p123 wf3 p3 wf2

let print_add_dscalar2_vector2_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction) =
    printf "@␣+␣";
    print_dscalar2_vector2_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction)

let print_dscalar2_vector2_m_0_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F123 | F213 | F124 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F134 | F143 | F234 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F132 | F231 | F142 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf1 p1 wf3 p3 wf2 p2
    | C_12_34, (F312 | F321 | F412 | F421) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_12_34, (F314 | F413 | F324 | F423) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F341 | F431 | F342 | F432) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_13_42, (F123 | F214)
    | C_14_23, (F124 | F213) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf1 p1 wf2 p3 wf3 p2
    | C_13_42, (F124 | F213)
    | C_14_23, (F123 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf2 p2 wf1 p3 wf3 p1
    | C_13_42, (F132 | F241)
    | C_14_23, (F142 | F231) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf1 p1 wf3 p2 wf2 p3
    | C_13_42, (F142 | F231)
    | C_14_23, (F132 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf3 p3 wf1 p2 wf2 p1
    | C_13_42, (F312 | F421)
    | C_14_23, (F412 | F321) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
            (format_coeff coeff) c pa pb wf2 p2 wf3 p1 wf1 p3
    | C_13_42, (F321 | F412)
    | C_14_23, (F421 | F312) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p1 wf1 p2
    | C_13_42, (F134 | F243)
    | C_14_23, (F143 | F234) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p3 wf3 p1 wf2 p2
    | C_13_42, (F143 | F234)
    | C_14_23, (F134 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p2 wf2 p1 wf3 p3
    | C_13_42, (F314 | F423)
    | C_14_23, (F413 | F324) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p3 wf3 p2 wf1 p1
    | C_13_42, (F324 | F413)
    | C_14_23, (F423 | F314) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p1 wf1 p2 wf3 p3
    | C_13_42, (F341 | F432)
    | C_14_23, (F431 | F342) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p2 wf2 p3 wf1 p1
    | C_13_42, (F342 | F431)
    | C_14_23, (F432 | F341) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_0(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p1 wf1 p3 wf2 p2

  let print_add_dscalar2_vector2_m_0_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
    printf "@␣+␣";
    print_dscalar2_vector2_m_0_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction)

let print_dscalar2_vector2_m_1_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F123 | F213 | F124 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F134 | F143 | F234 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F132 | F231 | F142 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf3 p3 wf2 p2
    | C_12_34, (F312 | F321 | F412 | F421) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_12_34, (F314 | F413 | F324 | F423) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F341 | F431 | F342 | F432) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_13_42, (F123 | F214)
    | C_14_23, (F124 | F213) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p3 wf3 p2
    | C_13_42, (F124 | F213)
    | C_14_23, (F123 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p3 wf3 p1
    | C_13_42, (F132 | F241)
    | C_14_23, (F142 | F231) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf3 p2 wf2 p3
    | C_13_42, (F142 | F231)
    | C_14_23, (F132 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf1 p2 wf2 p1
    | C_13_42, (F312 | F421)
    | C_14_23, (F412 | F321) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf3 p1 wf1 p3
    | C_13_42, (F321 | F412)
    | C_14_23, (F421 | F312) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p1 wf1 p2
    | C_13_42, (F134 | F243)
    | C_14_23, (F143 | F234) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p3 wf3 p1 wf2 p2
    | C_13_42, (F143 | F234)
    | C_14_23, (F134 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p2 wf2 p1 wf3 p3
    | C_13_42, (F314 | F423)
    | C_14_23, (F413 | F324) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p3 wf3 p2 wf1 p1
    | C_13_42, (F324 | F413)
    | C_14_23, (F423 | F314) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p1 wf1 p2 wf3 p3
    | C_13_42, (F341 | F432)
    | C_14_23, (F431 | F342) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p2 wf2 p3 wf1 p1
    | C_13_42, (F342 | F431)
    | C_14_23, (F432 | F341) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_1(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p1 wf1 p3 wf2 p2

  let print_add_dscalar2_vector2_m_1_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
    printf "@␣+␣";
    print_dscalar2_vector2_m_1_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction)

let print_dscalar2_vector2_m_7_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
    match contraction, fusion with
    | C_12_34, (F123 | F213 | F124 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F134 | F143 | F234 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p2 wf3 p3
    | C_12_34, (F132 | F231 | F142 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf3 p3 wf2 p2
    | C_12_34, (F312 | F321 | F412 | F421) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_12_34, (F314 | F413 | F324 | F423) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p1 wf3 p3
    | C_12_34, (F341 | F431 | F342 | F432) →
```

```
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p2 wf1 p1
    | C_13_42, (F123 | F214)
    | C_14_23, (F124 | F213) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf2 p3 wf3 p2
    | C_13_42, (F124 | F213)
    | C_14_23, (F123 | F214) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf1 p3 wf3 p1
    | C_13_42, (F132 | F241)
    | C_14_23, (F142 | F231) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p1 wf3 p2 wf2 p3
    | C_13_42, (F142 | F231)
    | C_14_23, (F132 | F241) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf1 p2 wf2 p1
    | C_13_42, (F312 | F421)
    | C_14_23, (F412 | F321) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p2 wf3 p1 wf1 p3
    | C_13_42, (F321 | F412)
    | C_14_23, (F421 | F312) →
        printf "@[((%s%s%s+%s))*v_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p3 wf2 p1 wf1 p2
    | C_13_42, (F134 | F243)
    | C_14_23, (F143 | F234) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p3 wf3 p1 wf2 p2
    | C_13_42, (F143 | F234)
    | C_14_23, (F134 | F243) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf1 p2 wf2 p1 wf3 p3
    | C_13_42, (F314 | F423)
    | C_14_23, (F413 | F324) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p3 wf3 p2 wf1 p1
    | C_13_42, (F324 | F413)
    | C_14_23, (F423 | F314) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf2 p1 wf1 p2 wf3 p3
    | C_13_42, (F341 | F432)
    | C_14_23, (F431 | F342) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p2 wf2 p3 wf1 p1
    | C_13_42, (F342 | F431)
    | C_14_23, (F432 | F341) →
        printf "@[((%s%s%s+%s))*phi_phi2v_m_7(cmplx(1,kind=default),@␣%s,%s,%s,%s,%s,%s))@]"
          (format_coeff coeff) c pa pb wf3 p1 wf1 p3 wf2 p2

let print_add_dscalar2_vector2_m_7_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction) =
  printf "@␣+␣";
  print_dscalar2_vector2_m_7_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion (coeff, contraction)

let print_dscalar4_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction) =
  match contraction, fusion with
    | C_12_34, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
    | C_13_42, (F241 | F421 | F243 | F423 | F132 | F312 | F134 | F314)
    | C_14_23, (F231 | F321 | F234 | F324 | F142 | F412 | F143 | F413) →
        printf "((%s%s%s+%s))*(%s*%s)*(%s*%s)*%s*%s*%s)"
          (format_coeff coeff) c pa pb p1 p2 p3 p123 wf1 wf2 wf3
```

```
      | C_12_34, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
      | C_13_42, (F124 | F142 | F324 | F342 | F213 | F231 | F413 | F431)
      | C_14_23, (F123 | F132 | F423 | F432 | F214 | F241 | F314 | F341) →
          printf "((%s%s%s+%s))*(%s*%s)*(%s*%s)*%s*%s*%s)"
            (format_coeff coeff) c pa pb p2 p3 p1 p123 wf1 wf2 wf3
      | C_12_34, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241)
      | C_13_42, (F214 | F412 | F234 | F432 | F123 | F321 | F143 | F341)
      | C_14_23, (F213 | F312 | F243 | F342 | F124 | F421 | F134 | F431) →
          printf "((%s%s%s+%s))*(%s*%s)*(%s*%s)*%s*%s*%s)"
            (format_coeff coeff) c pa pb p1 p3 p2 p123 wf1 wf2 wf3

    let print_add_dscalar4_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction) =
      printf "@␣+␣";
      print_dscalar4_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion (coeff, contraction)

    let print_current_V3 format_wf format_p amplitude dictionary rhs vertex fusion constant =
      let ch1, ch2 = children2 rhs in
      let wf1 = format_wf amplitude dictionary ch1
      and wf2 = format_wf amplitude dictionary ch2
      and p1 = format_p ch1
      and p2 = format_p ch2
      and m1 = SCM.mass_symbol (F.flavor ch1)
      and m2 = SCM.mass_symbol (F.flavor ch2) in
      let c = SCM.constant_symbol constant in
      printf "@,␣%s␣" (if (F.sign rhs) < 0 then "-" else "+");
      begin match vertex with
```

Fermionic currents $\bar\psi \not{A}\psi$ and $\bar\psi\phi\psi$ are handled by the *Fermions* module, since they depend on the choice of Feynman rules: Dirac or Majorana.

```
      | FBF (coeff, fb, b, f) →
         begin match coeff, fb, b, f with
         | _, _, (VLRM | SPM | VAM | VA3M | TVA | TVAM | TLR | TLRM | TRL | TRLM), _ →
            let p12 = Printf.sprintf "(-%s-%s)" p1 p2 in
            Fermions.print_current_mom (coeff, fb, b, f) c wf1 wf2 p1 p2
              p12 fusion
         | _, _, _, _ →
            Fermions.print_current (coeff, fb, b, f) c wf1 wf2 fusion
         end
      | PBP (coeff, f1, b, f2) →
         Fermions.print_current_p (coeff, f1, b, f2) c wf1 wf2 fusion
      | BBB (coeff, fb1, b, fb2) →
         Fermions.print_current_b (coeff, fb1, b, fb2) c wf1 wf2 fusion
      | GBG (coeff, fb, b, f) →
         let p12 = Printf.sprintf "(-%s-%s)" p1 p2 in
         Fermions.print_current_g (coeff, fb, b, f) c wf1 wf2 p1 p2 p12 fusion
```

Table 16.13 is a bit misleading, since if includes totally antisymmetric structure constants. The space-time part alone is also totally antisymmetric:

```
      | Gauge_Gauge_Gauge coeff →
         let c = format_coupling coeff c in
         begin match fusion with
         | (F23 | F31 | F12) → printf "g_gg(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
         | (F32 | F13 | F21) → printf "g_gg(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
         end

      | I_Gauge_Gauge_Gauge coeff →
         let c = format_coupling coeff c in
         begin match fusion with
         | (F23 | F31 | F12) → printf "g_gg((0,1)*(%s),%s,%s,%s,%s)" c wf1 p1 wf2 p2
         | (F32 | F13 | F21) → printf "g_gg((0,1)*(%s),%s,%s,%s,%s)" c wf2 p2 wf1 p1
         end
```

In *Aux_Gauge_Gauge*, we can not rely on antisymmetry alone, because of the different Lorentz representations of the auxiliary and the gauge field. Instead we have to provide the sign in

$$(V_2 \wedge V_3) \cdot T_1 = \begin{cases} V_2 \cdot (T_1 \cdot V_3) = -V_2 \cdot (V_3 \cdot T_1) \\ V_3 \cdot (V_2 \cdot T_1) = -V_3 \cdot (T_1 \cdot V_2) \end{cases} \tag{20.2}$$

ourselves. Alternatively, one could provide `g_xg` mirroring `g_gx`.

```
| Aux_Gauge_Gauge coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "x_gg(%s,%s,%s)" c wf1 wf2
    | F32 → printf "x_gg(%s,%s,%s)" c wf2 wf1
    | F12 → printf "g_gx(%s,%s,%s)" c wf2 wf1
    | F21 → printf "g_gx(%s,%s,%s)" c wf1 wf2
    | F13 → printf "(-1)*g_gx(%s,%s,%s)" c wf2 wf1
    | F31 → printf "(-1)*g_gx(%s,%s,%s)" c wf1 wf2
    end
```

These cases are symmetric and we just have to juxtapose the correct fields and provide parentheses to minimize the number of multiplications.

```
| Scalar_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "%s*(%s*%s)" c wf1 wf2
    | (F12 | F13) → printf "(%s*%s)*%s" c wf1 wf2
    | (F21 | F31) → printf "(%s*%s)*%s" c wf2 wf1
    end

| Aux_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "%s*(%s*%s)" c wf1 wf2
    | (F12 | F13) → printf "(%s*%s)*%s" c wf1 wf2
    | (F21 | F31) → printf "(%s*%s)*%s" c wf2 wf1
    end
```

Even simpler:

```
| Scalar_Scalar_Scalar coeff →
    printf "(%s*%s*%s)" (format_coupling coeff c) wf1 wf2

| Aux_Scalar_Scalar coeff →
    printf "(%s*%s*%s)" (format_coupling coeff c) wf1 wf2

| Aux_Scalar_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F13 | F31) → printf "%s*(%s*%s)" c wf1 wf2
    | (F23 | F21) → printf "(%s*%s)*%s" c wf1 wf2
    | (F32 | F12) → printf "(%s*%s)*%s" c wf2 wf1
    end

| Vector_Scalar_Scalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "v_ss(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "v_ss(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "s_vs(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "s_vs(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)*s_vs(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)*s_vs(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Graviton_Scalar_Scalar coeff →
```

```
let c = format_coupling coeff c in
begin match fusion with
| F12 → printf "s_gravs(%s,%s,-(%s+%s),%s,%s,%s)" c m2 p1 p2 p2 wf1 wf2
| F21 → printf "s_gravs(%s,%s,-(%s+%s),%s,%s,%s)" c m1 p1 p2 p1 wf2 wf1
| F13 → printf "s_gravs(%s,%s,%s,-(%s+%s),%s,%s)" c m2 p2 p1 p2 wf1 wf2
| F31 → printf "s_gravs(%s,%s,%s,-(%s+%s),%s,%s)" c m1 p1 p1 p2 wf2 wf1
| F23 → printf "grav_ss(%s,%s,%s,%s,%s,%s)" c m1 p1 p2 wf1 wf2
| F32 → printf "grav_ss(%s,%s,%s,%s,%s,%s)" c m1 p2 p1 wf2 wf1
end
```

In producing a vector in the fusion we always contract the rightmost index with the vector wavefunction from *rhs*. So the first momentum is always the one of the vector boson produced in the fusion, while the second one is that from the *rhs*. This makes the cases *F12* and *F13* as well as *F21* and *F31* equal. In principle, we could have already done this for the *Graviton_Scalar_Scalar* case.

```
| Graviton_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F12 | F13) → printf "v_gravv(%s,%s,-(%s+%s),%s,%s,%s)" c m2 p1 p2 p2 wf1 wf2
    | (F21 | F31) → printf "v_gravv(%s,%s,-(%s+%s),%s,%s,%s)" c m1 p1 p2 p1 wf2 wf1
    | F23 → printf "grav_vv(%s,%s,%s,%s,%s,%s)" c m1 p1 p2 wf1 wf2
    | F32 → printf "grav_vv(%s,%s,%s,%s,%s,%s)" c m1 p2 p1 wf2 wf1
    end

| Graviton_Spinor_Spinor coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "f_gravf(%s,%s,-(%s+%s),(-%s),%s,%s)" c m2 p1 p2 p2 wf1 wf2
    | F32 → printf "f_gravf(%s,%s,-(%s+%s),(-%s),%s,%s)" c m1 p1 p2 p1 wf2 wf1
    | F12 → printf "f_fgrav(%s,%s,%s,%s+%s,%s,%s)" c m1 p1 p1 p2 wf1 wf2
    | F21 → printf "f_fgrav(%s,%s,%s,%s+%s,%s,%s)" c m2 p2 p1 p2 wf2 wf1
    | F13 → printf "grav_ff(%s,%s,%s,(-%s),%s,%s)" c m1 p1 p2 wf1 wf2
    | F31 → printf "grav_ff(%s,%s,%s,(-%s),%s,%s)" c m1 p2 p1 wf2 wf1
    end

| Dim4_Vector_Vector_Vector_T coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "tkv_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "tkv_vv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "tv_kvv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "tv_kvv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)*tv_kvv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)*tv_kvv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Dim4_Vector_Vector_Vector_L coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "lkv_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "lkv_vv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 | F13 → printf "lv_kvv(%s,%s,%s,%s)" c wf1 p1 wf2
    | F21 | F31 → printf "lv_kvv(%s,%s,%s,%s)" c wf2 p2 wf1
    end

| Dim6_Gauge_Gauge_Gauge coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 | F31 | F12 → printf "kg_kgkg(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 | F13 | F21 → printf "kg_kgkg(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Dim4_Vector_Vector_Vector_T5 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "t5kv_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "t5kv_vv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 | F13 → printf "t5v_kvv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 | F31 → printf "t5v_kvv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Dim4_Vector_Vector_Vector_L5 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "l5kv_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "l5kv_vv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "l5v_kvv(%s,%s,%s,%s)" c wf1 p1 wf2
    | F21 → printf "l5v_kvv(%s,%s,%s,%s)" c wf2 p2 wf1
    | F13 → printf "(-1)*l5v_kvv(%s,%s,%s,%s)" c wf1 p1 wf2
    | F31 → printf "(-1)*l5v_kvv(%s,%s,%s,%s)" c wf2 p2 wf1
    end

  | Dim6_Gauge_Gauge_Gauge_5 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "kg5_kgkg(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "kg5_kgkg(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "kg_kg5kg(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "kg_kg5kg(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)*kg_kg5kg(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)*kg_kg5kg(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Aux_DScalar_DScalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "%s*(%s*%s)*(%s*%s)" c p1 p2 wf1 wf2
    | (F12 | F13) → printf "%s*(-((%s+%s)*%s))*(%s*%s)" c p1 p2 p2 wf1 wf2
    | (F21 | F31) → printf "%s*(-((%s+%s)*%s))*(%s*%s)" c p1 p2 p1 wf1 wf2
    end

  | Aux_Vector_DScalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "%s*(%s*%s)*%s" c wf1 p2 wf2
    | F32 → printf "%s*(%s*%s)*%s" c wf2 p1 wf1
    | F12 → printf "%s*(-((%s+%s)*%s))*%s" c p1 p2 wf2 wf1
    | F21 → printf "%s*(-((%s+%s)*%s))*%s" c p1 p2 wf1 wf2
    | (F13 | F31) → printf "(-(%s+%s))*(%s*%s*%s)" p1 p2 c wf1 wf2
    end

  | Dim5_Scalar_Gauge2 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) →
        printf "(%s)*((%s*%s)*(%s*%s)␣-␣(%s*%s)*(%s*%s))" c p1 wf2 p2 wf1 p1 p2 wf2 wf1
    | (F12 | F13) →
        printf "(%s)*%s*((-((%s+%s)*%s))*%s␣-␣((-(%s+%s)*%s))*%s)" c wf1 p1 p2 wf2 p2 p1 p2 p2 wf2
    | (F21 | F31) →
        printf "(%s)*%s*((-((%s+%s)*%s))*%s␣-␣((-(%s+%s)*%s))*%s)" c wf2 p2 p1 wf1 p1 p1 p2 p1 wf1
    end

  | Dim5_Scalar_Gauge2_Skew coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "(-␣phi_vv␣(%s,␣%s,␣%s,␣%s,␣%s))" c p1 p2 wf1 wf2
    | (F12 | F13) → printf "(-␣v_phiv␣(%s,␣%s,␣%s,␣%s,␣%s))" c wf1 p1 p2 wf2
    | (F21 | F31) → printf "v_phiv␣(%s,␣%s,␣%s,␣%s,␣%s)" c wf2 p1 p2 wf1
    end
```

```
| Dim5_Scalar_Vector_Vector_T coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "(%s)*(%s*%s)*(%s*%s)" c p1 wf2 p2 wf1
    | (F12 | F13) → printf "(%s)*%s*(-((%s+%s)*%s))*%s" c wf1 p1 p2 wf2 p2
    | (F21 | F31) → printf "(%s)*%s*(-((%s+%s)*%s))*%s" c wf2 p2 p1 wf1 p1
    end
| Dim5_Scalar_Vector_Vector_U coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "phi_u_vv␣(%s,␣%s,␣%s,␣%s,␣%s)" c p1 p2 wf1 wf2
    | (F12 | F13) → printf "v_u_phiv␣(%s,␣%s,␣%s,␣%s,␣%s)" c wf1 p1 p2 wf2
    | (F21 | F31) → printf "v_u_phiv␣(%s,␣%s,␣%s,␣%s,␣%s)" c wf2 p2 p1 wf1
    end
| Dim5_Scalar_Vector_Vector_TU coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "(%s)*((%s*%s)*(-(%s+%s)*%s)␣-␣(-(%s+%s)*%s)*(%s*%s))"
                c p1 wf2 p1 p2 wf1 p1 p2 p1 wf1 wf2
    | F32 → printf "(%s)*((%s*%s)*(-(%s+%s)*%s)␣-␣(-(%s+%s)*%s)*(%s*%s))"
                c p2 wf1 p1 p2 wf2 p1 p2 p2 wf1 wf2
    | F12 → printf "(%s)*%s*((%s*%s)*%s␣-␣(%s*%s)*%s)"
                c wf1 p1 wf2 p2 p1 p2 wf2
    | F21 → printf "(%s)*%s*((%s*%s)*%s␣-␣(%s*%s)*%s)"
                c wf2 p2 wf1 p1 p1 p2 wf1
    | F13 → printf "(%s)*%s*((-(%s+%s)*%s)*%s␣-␣(-(%s+%s)*%s)*%s)"
                c wf1 p1 p2 wf2 p1 p1 p2 p1 wf2
    | F31 → printf "(%s)*%s*((-(%s+%s)*%s)*%s␣-␣(-(%s+%s)*%s)*%s)"
                c wf2 p1 p2 wf1 p2 p1 p2 p2 wf1
    end
| Dim5_Scalar_Scalar2 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) →
        printf "phi_dim5s2(%s,␣%s␣,%s,␣%s,␣%s)" c wf1 p1 wf2 p2
    | (F12 | F13) →
        let p12 = Printf.sprintf "(-%s-%s)" p1 p2 in
        printf "phi_dim5s2(%s,%s,%s,%s,%s)" c wf1 p12 wf2 p2
    | (F21 | F31) →
        let p12 = Printf.sprintf "(-%s-%s)" p1 p2 in
        printf "phi_dim5s2(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p12
    end
| Scalar_Vector_Vector_t coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "s_vv_t(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_sv_t(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_sv_t(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_Vector_Vector_Vector_T coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "(%s)*(%s*%s)*(%s*%s)*(%s-%s)" c p2 wf1 p1 wf2 p1 p2
    | F32 → printf "(%s)*(%s*%s)*(%s*%s)*(%s-%s)" c p1 wf2 p2 wf1 p2 p1
    | (F12 | F13) →
        printf "(%s)*((%s+2*%s)*%s)*(-((%s+%s)*%s))*%s" c p1 p2 wf1 p1 p2 wf2 p2
    | (F21 | F31) →
        printf "(%s)*((-((%s+%s)*%s))*(%s+2*%s)*%s)*%s" c p2 p1 wf1 p2 p1 wf2 p1
    end
```

```
| Tensor_2_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_vv(%s,%s,%s)" c wf1 wf2
    | (F12 | F13) → printf "v_t2v(%s,%s,%s)" c wf1 wf2
    | (F21 | F31) → printf "v_t2v(%s,%s,%s)" c wf2 wf1
    end

| Tensor_2_Scalar_Scalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_phi2(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "phi_t2phi(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "phi_t2phi(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Tensor_2_Vector_Vector_1 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_vv_1(%s,%s,%s)" c wf1 wf2
    | (F12 | F13) → printf "v_t2v_1(%s,%s,%s)" c wf1 wf2
    | (F21 | F31) → printf "v_t2v_1(%s,%s,%s)" c wf2 wf1
    end

| Tensor_2_Vector_Vector_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_vv_cf(%s,%s,%s)" c wf1 wf2
    | (F12 | F13) → printf "v_t2v_cf(%s,%s,%s)" c wf1 wf2
    | (F21 | F31) → printf "v_t2v_cf(%s,%s,%s)" c wf2 wf1
    end

| Tensor_2_Scalar_Scalar_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_phi2_cf(%s,%s,%s,%s,␣%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "phi_t2phi_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "phi_t2phi_cf(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Dim5_Tensor_2_Vector_Vector_1 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_vv_d5_1(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_t2v_d5_1(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_t2v_d5_1(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Tensor_2_Vector_Vector_t coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "t2_vv_t(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_t2v_t(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_t2v_t(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

| Dim5_Tensor_2_Vector_Vector_2 coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "t2_vv_d5_2(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "t2_vv_d5_2(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | (F12 | F13) → printf "v_t2v_d5_2(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_t2v_d5_2(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
```

```
| TensorVector_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "dv_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_dvv(%s,%s,%s,%s)" c wf1 p1 wf2
    | (F21 | F31) → printf "v_dvv(%s,%s,%s,%s)" c wf2 p2 wf1
    end
| TensorVector_Vector_Vector_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "dv_vv_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_dvv_cf(%s,%s,%s,%s)" c wf1 p1 wf2
    | (F21 | F31) → printf "v_dvv_cf(%s,%s,%s,%s)" c wf2 p2 wf1
    end
| TensorVector_Scalar_Scalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "dv_phi2(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "phi_dvphi(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "phi_dvphi(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| TensorVector_Scalar_Scalar_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "dv_phi2_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "phi_dvphi_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "phi_dvphi_cf(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| TensorScalar_Vector_Vector coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "tphi_vv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_tphiv(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_tphiv(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| TensorScalar_Vector_Vector_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "tphi_vv_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_tphiv_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_tphiv_cf(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| TensorScalar_Scalar_Scalar coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "tphi_ss(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "s_tphis(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "s_tphis(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| TensorScalar_Scalar_Scalar_cf coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "tphi_ss_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "s_tphis_cf(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "s_tphis_cf(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim7_Tensor_2_Vector_Vector_T coeff →
```

```
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "t2_vv_d7(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "t2_vv_d7(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | (F12 | F13) → printf "v_t2v_d7(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_t2v_d7(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Dim6_Scalar_Vector_Vector_D coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "s_vv_6D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_sv_6D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_sv_6D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Dim6_Scalar_Vector_Vector_DP coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32) → printf "s_vv_6DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F12 | F13) → printf "v_sv_6DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | (F21 | F31) → printf "v_sv_6DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Dim6_HAZ_D coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "h_az_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "h_az_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "a_hz_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "a_hz_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "z_ah_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F21 → printf "z_ah_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    end
  | Dim6_HAZ_DP coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "h_az_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "h_az_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "a_hz_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "a_hz_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "z_ah_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F21 → printf "z_ah_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    end
  | Gauge_Gauge_Gauge_i coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "g_gg_23(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "g_gg_23(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "g_gg_13(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "g_gg_13(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "(-1)␣*␣g_gg_13(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "(-1)␣*␣g_gg_13(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end

  | Dim6_GGG coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "g_gg_6(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "g_gg_6(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "g_gg_6(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "g_gg_6(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)␣*␣g_gg_6(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)␣*␣g_gg_6(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_AWW_DP coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "a_ww_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "a_ww_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "w_aw_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "w_aw_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "(-1)␣*␣w_aw_DP(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "(-1)␣*␣w_aw_DP(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_AWW_DW coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "a_ww_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "a_ww_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)␣*␣a_ww_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)␣*␣a_ww_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "a_ww_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "a_ww_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_Gauge_Gauge_Gauge_i coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 | F31 | F12 →
        printf "kg_kgkg_i(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 | F13 | F21 →
        printf "kg_kgkg_i(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_HHH coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | (F23 | F32 | F12 | F21 | F13 | F31) →
        printf "h_hh_6(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    end
| Dim6_WWZ_DPWDW coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "w_wz_DPW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "w_wz_DPW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)␣*␣w_wz_DPW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)␣*␣w_wz_DPW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "z_ww_DPW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "z_ww_DPW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_WWZ_DW coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F23 → printf "w_wz_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F32 → printf "w_wz_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F13 → printf "(-1)␣*␣w_wz_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F31 → printf "(-1)␣*␣w_wz_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    | F12 → printf "z_ww_DW(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
    | F21 → printf "z_ww_DW(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
    end
| Dim6_WWZ_D coeff →
    let c = format_coupling coeff c in
    begin match fusion with
```

```
        | F23 → printf "w_wz_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
        | F32 → printf "w_wz_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
        | F13 → printf "(-1)␣*␣w_wz_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
        | F31 → printf "(-1)␣*␣w_wz_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
        | F12 → printf "z_ww_D(%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2
        | F21 → printf "z_ww_D(%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1
        end
    end
```

Flip the sign to account for the i^2 relative to diagrams with only cubic couplings.

That's an *slightly dangerous* hack!!! How do we accnount for such signs when treating n-ary vertices uniformly?

```
  let print_current_V4 format_wf format_p amplitude dictionary rhs vertex fusion constant =
    let c = CM.constant_symbol constant
    and ch1, ch2, ch3 = children3 rhs in
    let wf1 = format_wf amplitude dictionary ch1
    and wf2 = format_wf amplitude dictionary ch2
    and wf3 = format_wf amplitude dictionary ch3
    and p1 = format_p ch1
    and p2 = format_p ch2
    and p3 = format_p ch3 in
    printf "@,␣%s␣" (if (F.sign rhs) < 0 then "+" else "-");
    begin match vertex with
    | Scalar4 coeff → printf "(%s*%s*%s*%s)" (format_coupling coeff c) wf1 wf2 wf3
    | Scalar2_Vector2 coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F134 | F143 | F234 | F243 → printf "%s*%s*(%s*%s)" c wf1 wf2 wf3
        | F314 | F413 | F324 | F423 → printf "%s*%s*(%s*%s)" c wf2 wf1 wf3
        | F341 | F431 | F342 | F432 → printf "%s*%s*(%s*%s)" c wf3 wf1 wf2
        | F312 | F321 | F412 | F421 → printf "(%s*%s*%s)*%s" c wf2 wf3 wf1
        | F231 | F132 | F241 | F142 → printf "(%s*%s*%s)*%s" c wf1 wf3 wf2
        | F123 | F213 | F124 | F214 → printf "(%s*%s*%s)*%s" c wf1 wf2 wf3
        end
    | Vector4 contractions →
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
        | head :: tail →
            printf "(";
            print_vector4 c wf1 wf2 wf3 fusion head;
            List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
            printf ")"
        end
    | Dim8_Vector4_t_0 contractions →
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
        | head :: tail →
            print_vector4_t_0 c wf1 p1 wf2 p2 wf3 p3 fusion head;
            List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
        end
    | Dim8_Vector4_t_1 contractions →
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
        | head :: tail →
            print_vector4_t_1 c wf1 p1 wf2 p2 wf3 p3 fusion head;
            List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
        end
    | Dim8_Vector4_t_2 contractions →
        begin match contractions with
```

```
      | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
      | head :: tail →
          print_vector4_t_2 c wf1 p1 wf2 p2 wf3 p3 fusion head;
          List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
      end
  | Dim8_Vector4_m_0 contractions →
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
      | head :: tail →
          print_vector4_m_0 c wf1 p1 wf2 p2 wf3 p3 fusion head;
          List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
      end
  | Dim8_Vector4_m_1 contractions →
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
      | head :: tail →
          print_vector4_m_1 c wf1 p1 wf2 p2 wf3 p3 fusion head;
          List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
      end
  | Dim8_Vector4_m_7 contractions →
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣Vector4␣[]"
      | head :: tail →
          print_vector4_m_7 c wf1 p1 wf2 p2 wf3 p3 fusion head;
          List.iter (print_add_vector4 c wf1 wf2 wf3 fusion) tail;
      end
  | Vector4_K_Matrix_tho (_, poles) →
      let pa, pb =
        begin match fusion with
        | (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
        | (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
        | (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
        end in
      printf "(%s*(%s*%s)*(%s*%s)*(%s*%s)@,*("
        c p1 wf1 p2 wf2 p3 wf3;
      List.iter (fun (coeff, pole) →
            printf "+%s/((%s+%s)*(%s+%s)-%s)"
              (SCM.constant_symbol coeff) pa pb pa pb
              (SCM.constant_symbol pole))
        poles;
      printf ")*(-%s-%s-%s))" p1 p2 p3
  | Vector4_K_Matrix_jr (disc, contractions) →
      let pa, pb =
        begin match disc, fusion with
        | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
        | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
        | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
        end in
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_jr␣[]"
      | head :: tail →
          printf "(";
          print_vector4_km c pa pb wf1 wf2 wf3 fusion head;
          List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion)
            tail;
          printf ")"
      end
  | Vector4_K_Matrix_cf_t0 (disc, contractions) →
```

```
    let pa, pb, pc =
      begin match disc, fusion with
      | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2, p3)
      | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3, p1)
      | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3, p2)
      | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2, p3)
      | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3, p1)
      | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3, p2)
      end in
    begin match contractions with
    | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_t0␣[]"
    | head :: tail →
        printf "(";
        print_vector4_km_t_0 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
        List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion) tail;
        printf ")"
    end
| Vector4_K_Matrix_cf_t1 (disc, contractions) →
    let pa, pb =
      begin match disc, fusion with
      | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
      | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
      | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
      | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
      | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
      | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
      end in
    begin match contractions with
    | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_t1␣[]"
    | head :: tail →
        printf "(";
        print_vector4_km_t_1 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
        List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion) tail;
        printf ")"
    end
| Vector4_K_Matrix_cf_t2 (disc, contractions) →
    let pa, pb =
      begin match disc, fusion with
      | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
      | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
      | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
      | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
      | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
      | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
      end in
    begin match contractions with
    | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_t2␣[]"
    | head :: tail →
        printf "(";
        print_vector4_km_t_2 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
        List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion)
          tail;
        printf ")"
    end
| Vector4_K_Matrix_cf_t_rsi (disc, contractions) →
    let pa, pb, pc =
      begin match disc, fusion with
      | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2, p3)
      | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3, p1)
      | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3, p2)
      | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2, p3)
```

```
          | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3, p1)
          | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3, p2)
          end in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_t_rsi␣[]"
        | head :: tail →
            printf "(";
            print_vector4_km_t_rsi c pa pb pc wf1 p1 wf2 p2 wf3 p3 fusion head;
            List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion)
              tail;
            printf ")"
        end
    | Vector4_K_Matrix_cf_m0 (disc, contractions) →
        let pa, pb =
          begin match disc, fusion with
          | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
          | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
          | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
          | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
          | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
          | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
          end in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_m0␣[]"
        | head :: tail →
            printf "(";
            print_vector4_km_m_0 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
            List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion) tail;
            printf ")"
        end
    | Vector4_K_Matrix_cf_m1 (disc, contractions) →
        let pa, pb =
          begin match disc, fusion with
          | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
          | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
          | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
          | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
          | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
          | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
          end in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_m1␣[]"
        | head :: tail →
            printf "(";
            print_vector4_km_m_1 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
            List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion)
              tail;
            printf ")"
        end
    | Vector4_K_Matrix_cf_m7 (disc, contractions) →
        let pa, pb =
          begin match disc, fusion with
          | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
          | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
          | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
          | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
          | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
          | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
          end in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣Vector4_K_Matrix_cf_m7␣[]"
```

```
      | head :: tail →
          printf "(";
          print_vector4_km_m_7 c pa pb wf1 p1 wf2 p2 wf3 p3 fusion head;
          List.iter (print_add_vector4_km c pa pb wf1 wf2 wf3 fusion) tail;
          printf ")"
      end
  | DScalar2_Vector2_K_Matrix_ms (disc, contractions) →
      let p123 = Printf.sprintf "(-%s-%s-%s)" p1 p2 p3 in
      let pa, pb =
        begin match disc, fusion with
        | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 4, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 4, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 4, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 5, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 5, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 5, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 6, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 6, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 6, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 7, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 7, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 7, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 8, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 8, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 8, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
        | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
        | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
        end in
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣DScalar2_Vector4_K_Matrix_ms␣[]"
      | head :: tail →
          printf "(";
          print_dscalar2_vector2_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion head;
          List.iter (print_add_dscalar2_vector2_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion) tail;
          printf ")"
      end
  | DScalar2_Vector2_m_0_K_Matrix_cf (disc, contractions) →
      let pa, pb =
        begin match disc, fusion with
        | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 4, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 4, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 4, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 5, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 5, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 5, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 6, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 6, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 6, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 7, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 7, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 7, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 8, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 8, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 8, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
        | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
        | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
        end in
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣DScalar2_Vector4_K_Matrix_cf_m0␣[]"
      | head :: tail →
          printf "(";
          print_dscalar2_vector2_m_0_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion head;
          List.iter (print_add_dscalar2_vector2_m_0_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion) tail;
          printf ")"
      end
  | DScalar2_Vector2_m_1_K_Matrix_cf (disc, contractions) →
      let pa, pb =
        begin match disc, fusion with
        | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 4, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 4, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 4, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 5, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 5, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 5, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 6, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 6, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 6, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 7, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 7, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 7, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 8, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 8, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 8, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
        | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
        | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
        end in
      begin match contractions with
      | [] → invalid_arg "Targets.print_current:␣DScalar2_Vector4_K_Matrix_cf_m1␣[]"
      | head :: tail →
          printf "(";
          print_dscalar2_vector2_m_1_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion head;
          List.iter (print_add_dscalar2_vector2_m_1_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion) tail;
          printf ")"
      end
  | DScalar2_Vector2_m_7_K_Matrix_cf (disc, contractions) →
      let pa, pb =
        begin match disc, fusion with
        | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 4, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 4, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 4, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 5, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
        | 5, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
        | 5, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
        | 6, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
        | 6, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
        | 6, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
        | 7, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
```

```
            | 7, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
            | 7, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
            | 8, (F134 | F132 | F314 | F312 | F241 | F243 | F421 | F423) → (p1, p2)
            | 8, (F213 | F413 | F231 | F431 | F124 | F324 | F142 | F342) → (p2, p3)
            | 8, (F143 | F123 | F341 | F321 | F412 | F214 | F432 | F234) → (p1, p3)
            | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
            | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
            | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
            end in
          begin match contractions with
          | [] → invalid_arg "Targets.print_current:␣DScalar2_Vector4_K_Matrix_cf_m7␣[]"
          | head :: tail →
              printf "(";
              print_dscalar2_vector2_m_7_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion head;
              List.iter (print_add_dscalar2_vector2_m_7_km c pa pb wf1 wf2 wf3 p1 p2 p3 fusion) tail;
              printf ")"
          end
      | DScalar4_K_Matrix_ms (disc, contractions) →
          let p123 = Printf.sprintf "(-%s-%s-%s)" p1 p2 p3 in
          let pa, pb =
            begin match disc, fusion with
            | 3, (F143 | F413 | F142 | F412 | F321 | F231 | F324 | F234) → (p1, p2)
            | 3, (F314 | F341 | F214 | F241 | F132 | F123 | F432 | F423) → (p2, p3)
            | 3, (F134 | F431 | F124 | F421 | F312 | F213 | F342 | F243) → (p1, p3)
            | _, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214) → (p1, p2)
            | _, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421) → (p2, p3)
            | _, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241) → (p1, p3)
            end in
          begin match contractions with
          | [] → invalid_arg "Targets.print_current:␣DScalar4_K_Matrix_ms␣[]"
          | head :: tail →
              printf "(";
              print_dscalar4_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion head;
              List.iter (print_add_dscalar4_km c pa pb wf1 wf2 wf3 p1 p2 p3 p123 fusion) tail;
              printf ")"
          end
      | Dim8_Scalar2_Vector2_1 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 →
              printf "phi_phi2v_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          | F314 | F413 | F324 | F423 →
              printf "phi_phi2v_1(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
          | F341 | F431 | F342 | F432 →
              printf "phi_phi2v_1(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F312 | F321 | F412 | F421 →
              printf "v_phi2v_1(%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1
          | F231 | F132 | F241 | F142 →
              printf "v_phi2v_1(%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2
          | F123 | F213 | F124 | F214 →
              printf "v_phi2v_1(%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3
          end
      | Dim8_Scalar2_Vector2_2 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 →
              printf "phi_phi2v_2(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          | F314 | F413 | F324 | F423 →
              printf "phi_phi2v_2(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
          | F341 | F431 | F342 | F432 →
              printf "phi_phi2v_2(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F312 | F321 | F412 | F421 →
              printf "v_phi2v_2(%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1
          | F231 | F132 | F241 | F142 →
              printf "v_phi2v_2(%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2
          | F123 | F213 | F124 | F214 →
              printf "v_phi2v_2(%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3
          end
      | Dim8_Scalar2_Vector2_m_0 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 →
              printf "phi_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          | F314 | F413 | F324 | F423 →
              printf "phi_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
          | F341 | F431 | F342 | F432 →
              printf "phi_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F312 | F321 | F412 | F421 →
              printf "v_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F231 | F132 | F241 | F142 →
              printf "v_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
          | F123 | F213 | F124 | F214 →
              printf "v_phi2v_m_0(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          end
      | Dim8_Scalar2_Vector2_m_1 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 →
              printf "phi_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          | F314 | F413 | F324 | F423 →
              printf "phi_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
          | F341 | F431 | F342 | F432 →
              printf "phi_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F312 | F321 | F412 | F421 →
              printf "v_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F231 | F132 | F241 | F142 →
              printf "v_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
          | F123 | F213 | F124 | F214 →
              printf "v_phi2v_m_1(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          end
      | Dim8_Scalar2_Vector2_m_7 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 →
              printf "phi_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          | F314 | F413 | F324 | F423 →
              printf "phi_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
          | F341 | F431 | F342 | F432 →
              printf "phi_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F312 | F321 | F412 | F421 →
              printf "v_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
          | F231 | F132 | F241 | F142 →
              printf "v_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
          | F123 | F213 | F124 | F214 →
              printf "v_phi2v_m_7(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
          end
      | Dim8_Scalar4 coeff →
          let c = format_coupling coeff c in
          begin match fusion with
          | F134 | F143 | F234 | F243 | F314 | F413 | F324 | F423
          | F341 | F431 | F342 | F432 | F312 | F321 | F412 | F421
          | F231 | F132 | F241 | F142 | F123 | F213 | F124 | F214 →
```

```
        printf "s_dim8s3␣(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      end
  | GBBG (coeff, fb, b, f) →
      Fermions.print_current_g4 (coeff, fb, b, f) c wf1 wf2 wf3 fusion

  | Dim6_H4_P2 coeff →
      let c = format_coupling coeff c in
      begin match fusion with
      | F134 | F143 | F234 | F243 | F314 | F413 | F324 | F423
      | F341 | F431 | F342 | F432 | F312 | F321 | F412 | F421
      | F231 | F132 | F241 | F142 | F123 | F213 | F124 | F214 →
          printf "hhhh_p2␣(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      end

  | Dim6_AHWW_DPB coeff →
      let c = format_coupling coeff c in
      begin match fusion with
      | F234 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F243 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F342 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F324 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F423 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F432 → printf "a_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      | F134 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F143 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F341 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F314 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F413 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F431 → printf "h_aww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      | F124 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F142 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F241 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F214 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F412 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F421 → printf "w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      | F123 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F132 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F231 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F213 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F312 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F321 → printf "(-1)*w_ahw_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      end
  | Dim6_AHWW_DPW coeff →
      let c = format_coupling coeff c in
      begin match fusion with
      | F234 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F243 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F342 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F324 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F423 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F432 → printf "a_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      | F134 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F143 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F341 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F314 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F413 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
      | F431 → printf "h_aww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
      | F124 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
      | F142 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
      | F241 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
      | F214 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
      | F412 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
```

```
        | F421 → printf "w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "(-1)*w_ahw_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
    | Dim6_AHWW_DW coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "a_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "h_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "w3_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "w4_ahw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
                end
    | Dim6_Scalar2_Vector2_D coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 | F134 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 | F143 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 | F341 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 | F314 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 | F413 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 | F431 → printf "h_hww_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 | F123 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 | F132 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 | F231 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 | F213 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 | F312 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 | F321 → printf "w_hhw_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
    | Dim6_Scalar2_Vector2_DP coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 | F134 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F342 | F341 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F423 | F413 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F243 | F143 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F324 | F314 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F432 | F431 → printf "h_hww_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 | F124 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F231 | F241 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F312 | F412 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F132 | F142 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F213 | F214 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F321 | F421 → printf "w_hhw_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
              end
| Dim6_Scalar2_Vector2_PB coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F234 | F134 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | F342 | F341 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
    | F423 | F413 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
    | F243 | F143 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
    | F324 | F314 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | F432 | F431 → printf "h_hvv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
    | F123 | F124 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | F231 | F241 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
    | F312 | F412 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
    | F132 | F142 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
    | F213 | F214 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | F321 | F421 → printf "v_hhv_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
    end
| Dim6_HHZZ_T coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F234 | F134 → printf "(%s)*(%s)*(%s)*(%s)" c wf1 wf2 wf3
    | F342 | F341 → printf "(%s)*(%s)*(%s)*(%s)" c wf3 wf1 wf2
    | F423 | F413 → printf "(%s)*(%s)*(%s)*(%s)" c wf2 wf3 wf1
    | F243 | F143 → printf "(%s)*(%s)*(%s)*(%s)" c wf1 wf3 wf2
    | F324 | F314 → printf "(%s)*(%s)*(%s)*(%s)" c wf2 wf1 wf3
    | F432 | F431 → printf "(%s)*(%s)*(%s)*(%s)" c wf3 wf2 wf1
    | F123 | F124 | F231 | F241 | F312 | F412 → printf "(%s)*(%s)*(%s)*(%s)" c wf1 wf2 wf3
    | F132 | F142 | F213 | F214 | F321 | F421 → printf "(%s)*(%s)*(%s)*(%s)" c wf1 wf2 wf3
    end
| Dim6_Vector4_DW coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F234 | F134 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | F342 | F341 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
    | F423 | F413 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
    | F243 | F143 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
    | F324 | F314 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | F432 | F431 → printf "a_aww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
    | F124 | F123 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | F241 | F231 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
    | F412 | F312 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
    | F142 | F132 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
    | F214 | F213 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
    | F421 | F321 → printf "w_aaw_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
    end
| Dim6_Vector4_W coeff →
    let c = format_coupling coeff c in
    begin match fusion with
    | F234 | F134 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
    | F342 | F341 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
    | F423 | F413 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
    | F243 | F143 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
```

```
        | F324 | F314 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F432 | F431 → printf "a_aww_W(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 | F124 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F231 | F241 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F312 | F412 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F132 | F142 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F213 | F214 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F321 | F421 → printf "w_aaw_W(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end

    | Dim6_HWWZ_DW coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "h_wwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "(-1)*w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "w_hwz_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_hww_DW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end

    | Dim6_HWWZ_DPB coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "h_wwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "(-1)*w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "w_hwz_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_hww_DPB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
     | Dim6_HWWZ_DDPW coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "h_wwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "(-1)*w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "w_hwz_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_hww_DDPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
     | Dim6_HWWZ_DPW coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "h_wwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "(-1)*w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "w_hwz_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_hww_DPW(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
     | Dim6_AHHZ_D coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "a_hhz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "h_ahz_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_ahh_D(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
     | Dim6_AHHZ_DP coeff →
        let c = format_coupling coeff c in
        begin match fusion with
        | F234 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "a_hhz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "h_ahz_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_ahh_DP(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
     | Dim6_AHHZ_PB coeff →
        let c = format_coupling coeff c in
        begin match fusion with
```

```
        | F234 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F243 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F342 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F324 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F423 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F432 → printf "a_hhz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F124 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F142 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F241 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F214 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F412 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F421 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F134 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F143 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F341 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F314 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F413 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F431 → printf "h_ahz_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        | F123 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf2 p2 wf3 p3
        | F132 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf1 p1 wf3 p3 wf2 p2
        | F231 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf1 p1 wf2 p2
        | F213 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf1 p1 wf3 p3
        | F312 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf2 p2 wf3 p3 wf1 p1
        | F321 → printf "z_ahh_PB(%s,%s,%s,%s,%s,%s,%s)" c wf3 p3 wf2 p2 wf1 p1
        end
```

In principle, *p4* could be obtained from the left hand side …

```
      | DScalar4 contractions →
        let p123 = Printf.sprintf "(-%s-%s-%s)" p1 p2 p3 in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣DScalar4␣[]"
        | head :: tail →
           printf "(";
           print_dscalar4 c wf1 wf2 wf3 p1 p2 p3 p123 fusion head;
           List.iter (print_add_dscalar4 c wf1 wf2 wf3 p1 p2 p3 p123 fusion) tail;
           printf ")"
        end
      | DScalar2_Vector2 contractions →
        let p123 = Printf.sprintf "(-%s-%s-%s)" p1 p2 p3 in
        begin match contractions with
        | [] → invalid_arg "Targets.print_current:␣DScalar4␣[]"
        | head :: tail →
           printf "(";
           print_dscalar2_vector2 c wf1 wf2 wf3 p1 p2 p3 p123 fusion head;
           List.iter (print_add_dscalar2_vector2 c wf1 wf2 wf3 p1 p2 p3 p123 fusion) tail;
           printf ")"
        end

      end
  end
```

20.9 Interface of Targets_Kmatrix

```
module Fortran : sig val print : bool → unit end
```

20.10 Implementation of Targets_Kmatrix

```
module Fortran =
struct
  open Format
  let nl = print_newline
```

Special functions for the K matrix approach. This might be generalized to other functions that have to have access to the parameters and coupling constants. At the moment, this is hardcoded.

```
  let print pure_functions =
    let pure =
      if pure_functions then
        "pure␣"
      else
        "" in
    printf "␣␣!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"; nl ();
    printf "␣␣!!!␣Special␣K␣matrix␣functions"; nl ();
    printf "␣␣!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"; nl ();
    nl();
    printf "␣␣%sfunction␣width_res␣(z,res,w_wkm,m,g)␣result␣(w)" pure; nl ();
    printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣z,␣w_wkm,␣m,␣g"; nl ();
    printf "␣␣␣␣␣␣integer,␣intent(in)␣::␣res"; nl ();
    printf "␣␣␣␣␣␣real(kind=default)␣::␣w"; nl ();
    printf "␣␣␣␣␣␣if␣(z.eq.0␣.AND.␣w_wkm.eq.0␣)␣then"; nl ();
    printf "␣␣␣␣␣␣␣␣w␣=␣0"; nl ();
    printf "␣␣␣␣␣␣else"; nl ();
    printf "␣␣␣␣␣␣␣␣if␣(w_wkm.eq.0)␣then"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣select␣case␣(res)"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(1)␣!!!␣Scalar␣isosinglet"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣3.*g**2/32./Pi␣*␣m**3/vev**2"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(2)␣!!!␣Scalar␣isoquintet"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/64./Pi␣*␣m**3/vev**2"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(3)␣!!!␣Vector␣isotriplet"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/48./Pi␣*␣m"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(4)␣!!!␣Tensor␣isosinglet"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/320./Pi␣*␣m**3/vev**2"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(5)␣!!!␣Tensor␣isoquintet"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/1920./Pi␣*␣m**3/vev**2"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣default"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣0"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣end␣select"; nl ();
    printf "␣␣␣␣␣␣␣␣else"; nl ();
    printf "␣␣␣␣␣␣␣␣␣␣w␣=␣w_wkm"; nl ();
    printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
    printf "␣␣␣␣␣␣end␣if"; nl ();
    printf "␣␣end␣function␣width_res"; nl ();
    nl ();
    printf "␣␣%sfunction␣s0stu␣(s,␣m)␣result␣(s0)" pure; nl ();
    printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
    printf "␣␣␣␣␣␣real(kind=default)␣::␣s0"; nl ();
    printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
    printf "␣␣␣␣␣␣␣␣s0␣=␣0"; nl ();
    printf "␣␣␣␣␣␣else"; nl ();
    printf "␣␣␣␣␣␣␣␣s0␣=␣m**2␣-␣s/2␣+␣m**4/s␣*␣log(m**2/(s+m**2))"; nl ();
    printf "␣␣␣␣␣␣end␣if"; nl ();
    printf "␣␣end␣function␣s0stu"; nl();
    nl ();
    printf "␣␣%sfunction␣s1stu␣(s,␣m)␣result␣(s1)" pure; nl ();
    printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
    printf "␣␣␣␣␣␣real(kind=default)␣::␣s1"; nl ();
    printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
    printf "␣␣␣␣␣␣␣␣s1␣=␣0"; nl ();
    printf "␣␣␣␣␣␣else"; nl ();
```

```
printf "␣␣␣␣␣␣␣␣s1␣=␣2*m**4/s␣+␣s/6␣+␣m**4/s**2*(2*m**2+s)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s1stu"; nl();
nl ();
printf "␣␣%sfunction␣s2stu␣(s,␣m)␣result␣(s2)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s2"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣s2␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣s2␣=␣m**4/s**2␣*␣(6*m**2␣+␣3*s)␣+␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣m**4/s**3␣*␣(6*m**4␣+␣6*m**2*s␣+␣s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s2stu"; nl();
nl ();
printf "␣!!␣%sfunction␣s3stu␣(s,␣m)␣result␣(s3)" pure; nl ();
printf "␣!!␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣!!␣␣␣␣␣real(kind=default)␣::␣s3"; nl ();
printf "␣!!␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣!!␣␣␣␣␣␣␣s3␣=␣0"; nl ();
printf "␣!!␣␣␣␣␣else"; nl ();
printf "␣!!␣␣␣␣␣␣␣s3␣=␣m**4/s**3␣*␣(60*m**4␣+␣60*m**2*s+11*s**2)␣+␣&"; nl();
printf "␣!!␣␣␣␣␣␣␣␣␣␣␣␣m**4/s**4␣*(2*m**2+s)␣(10*m**4␣+␣10*m**2*s␣+␣s**2)␣&"; nl();
printf "␣!!␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣!!␣␣␣␣␣end␣if"; nl ();
printf "␣!!␣␣end␣function␣s3stu"; nl();
nl ();
printf "␣␣%sfunction␣p0stu␣(s,␣m)␣result␣(p0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣p0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣p0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣p0␣=␣1␣+␣(2*s+m**2)*log(m**2/(s+m**2))/s"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣p0stu"; nl();
nl ();
printf "␣␣%sfunction␣p1stu␣(s,␣m)␣result␣(p1)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣p1"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣p1␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣p1␣=␣(m**2␣+␣2*s)/s**2␣*␣(2*s+(2*m**2+s)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2)))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣p1stu"; nl();
nl ();
printf "␣␣%sfunction␣d0stu␣(s,␣m)␣result␣(d0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣d0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣d0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣d0␣=␣(2*m**2+11*s)/2␣+␣(m**4+6*m**2*s+6*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣/s␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣d0stu"; nl();
nl ();
```

```
printf "␣␣%sfunction␣d1stu␣(s,␣m)␣result␣(d1)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣d1"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣d1␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣d1␣=␣(s*(12*m**4␣+␣72*m**2*s␣+␣73*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣+␣6*(2*m**2␣+␣s)*(m**4␣+␣6*m**2*s␣+␣6*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2)))/6/s**2"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣d1stu"; nl();
nl ();
printf "␣␣%sfunction␣da00␣(cc,␣s,␣m)␣result␣(amp_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a00_0,␣a00_1,␣a00_a,␣a00_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a00"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_00"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1.0/32.0/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2/vev**4*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(1)␣=␣-2.0␣*␣cc(1)**2/vev**2␣*␣s0stu(s,m(1))␣"; nl ();
printf "␣␣␣␣␣␣if␣(cc(1)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(1)␣=␣a00(1)␣-␣3.0*cc(1)**2/vev**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(1)**2,m(1)*wkm(1),default)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a00(2)␣=␣-5.0*cc(2)**2/vev**2␣*␣s0stu(s,m(2))␣/␣3.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a00(3)␣=␣-cc(3)**2*(4.0*p0stu(s,m(3))␣+␣6.0*s/m(3)**2)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(4)␣=␣-cc(4)**2/vev**2/3␣*␣(d0stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*kappal*s0stu(s,m(4)))"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(kappal␣/=␣0))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(4)␣=␣a00(4)␣-␣cc(4)**2/vev**2*kappal␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(4)**2,m(4)␣*␣wkm(4),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a00(5)␣=␣-5.0*cc(5)**2/vev**2*(d0stu(s,m(5))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣/3.0)/6.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(6)␣=␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/4*␣s**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣((2-2*s/m(4)**2+s**2/m(4)**4)+kappat/2␣)"; nl ();
printf "␣␣␣␣␣␣if␣(a00(6)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(6)␣=␣a00(6)/cmplx(s-m(4)**2,␣-␣w_res/32/Pi␣*␣real(a00(6),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a00(6)␣=␣a00(6)␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12␣*␣(s0stu(s,m(4))␣&"; n
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(7)␣=␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/4␣*␣s**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣␣((1-4*s/m(4)**2+2*s**2/m(4)**4)+kappam␣)"; nl ();
printf "␣␣␣␣␣␣if␣(a00(7)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(7)␣=␣a00(7)/cmplx(s-m(4)**2,␣-␣w_res/32/Pi␣*␣real(a00(7),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a00(7)␣=␣a00(7)␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12␣*␣(s0stu(
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a00_f␣=␣2.*fudge_higgs*s/vev**2"; nl ();
printf "␣␣␣␣␣␣a00_f␣=␣a00_f␣!!!␣-␣0*5.*(1-ghvva)**2/vev**2*mass(25)**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a00_0␣=␣8.*(7.*a4␣+␣11.*a5)/3.*s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a00_1␣=␣(25.*log(lam_reg**2/s)/9␣+␣11./54.0_default)*s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a00_a␣=␣␣a00_0␣!!!␣+␣a00_1/16./Pi**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_00␣=␣sum(a00)+a00_f+a00_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_00␣=␣-␣a00_a␣-␣a00_f␣-␣part_r␣*␣(sum(a00)␣-␣a00(3))␣+␣1/(real(1/amp_00,defaul
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_00␣=␣(1-part_r)␣*␣sum(a00)␣+␣part_r␣*␣a00(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣amp_00␣=␣vev**4/s**2␣*␣amp_00"; nl ();
printf "␣␣end␣function␣da00"; nl();
nl ();
printf "␣␣%sfunction␣da02␣(cc,␣s,␣m)␣result␣(amp_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a02_0,␣a02_1,␣a02_a"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a02"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_02"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1.0/32.0/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2/vev**4*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(1)␣=␣-2.0*cc(1)**2/vev**2␣*␣s2stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a02(2)␣=␣-5.0*cc(2)**2/vev**2␣*␣s2stu(s,m(2))␣/␣3.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a02(3)␣=␣-4.0*cc(3)**2*(2*s+m(3)**2)*s2stu(s,m(3))/m(3)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(4)␣=␣-␣cc(4)**2/vev**2/3␣*␣␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣((1.+6.*s/m(4)**2+6.*s**2/m(4)**4)-2*kappal)␣*␣s2stu(s,m(4))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(4)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(4)␣=␣a02(4)␣-␣cc(4)**2/vev**2/10.␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s**2/cmplx(s-m(4)**2,m(4)*wkm(4),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a02(5)␣=␣-cc(5)**2/vev**2*(5.0*(1.0+6.0*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s/m(5)**2+6.0*s**2/m(5)**4)*s2stu(s,m(5))/3.0␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣)/6.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(6)␣=␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/40*␣s**2"; nl ();
printf "␣␣␣␣␣␣if␣(a02(6)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(6)␣=␣a02(6)/cmplx(s-m(4)**2,␣-␣w_res/32/Pi␣*␣real(a02(6),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a02(6)␣=␣a02(6)␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12␣*␣(s2stu(s,m(4))␣&"; n
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(7)␣=␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/20␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s**2"; nl ();
printf "␣␣␣␣␣␣if␣(a02(7)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(7)␣=␣a02(7)/cmplx(s-m(4)**2,␣-␣w_res/32/Pi␣*␣real(a02(7),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a02(7)␣=␣a02(7)␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12␣*␣(s2stu(
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a02_0␣=␣(8.*(2.*a4␣+␣a5)/15.)␣*␣␣s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a02_1␣=␣(log(lam_reg**2/s)/9.␣-␣7./135.0_default)␣*␣␣s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a02_a␣=␣a02_0␣!!!␣+␣a02_1/16/Pi**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_02␣=␣sum(a02)+a02_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_02␣=␣-␣a02_a␣-␣part_r␣*␣(sum(a02)␣-␣a02(3))␣+␣1/(real(1/amp_02,default)-ii)";
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_02␣=␣(1-part_r)␣*␣sum(a02)␣+␣part_r␣*␣a02(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_02␣=␣vev**4/s**2␣*␣amp_02"; nl ();
printf "␣␣end␣function␣da02"; nl();
nl ();
printf "␣␣%sfunction␣da11␣(cc,␣s,␣m)␣result␣(amp_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a11_0,␣a11_1,␣a11_a,␣a11_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a11"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_11"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1.0/32.0/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2/vev**4*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(1)␣=␣-␣2.0*cc(1)**2/vev**2␣*␣s1stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a11(2)␣=␣5.0*cc(2)**2/vev**2␣*␣s1stu(s,m(2))␣/␣6.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a11(3)␣=␣-␣cc(3)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(s/m(3)**2␣+␣2.␣*␣p1stu(s,m(3)))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(3)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a11(3)␣=␣a11(3)␣-2./3.␣*␣cc(3)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s/cmplx(s-m(3)**2,m(3)*wkm(3),default)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(4)␣=␣-␣cc(4)**2/vev**2*(d1stu(s,m(4)-2*kappal*s1stu(s,m(4)))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/3.0)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a11(5)␣=␣␣5.0*cc(5)**2/vev**2*(d1stu(s,m(5))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣)/36.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12␣*␣(s1stu(s,m(4))␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣(3*(1+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣)␣-␣(s/m(4)**2+s**2/m(4)**4)*s)"; nl ()
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12␣*␣(s1stu(s,m(4))␣
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣)␣-␣2*(s/m(4)**2+s**2/m(4)**4)*s)";
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a11_f␣=␣fudge_higgs*s/3./vev**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a11_0␣=␣4.*(a4␣-␣2*a5)/3.␣*␣s**2/vev**4␣"; nl ();
printf "␣␣␣␣␣␣a11_1␣=␣-␣1.0/54.0_default␣*␣s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a11_a␣=␣a11_0␣!!!␣+␣a11_1/16/Pi**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_11␣=␣sum(a11)+a11_f+a11_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_11␣=␣-␣a11_a␣-␣part_r␣*␣(sum(a11)␣-␣a11(3))␣+␣1/(real(1/amp_11,default)-ii)";
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_11␣=␣(1-part_r)␣*␣sum(a11)␣+␣part_r␣*␣a11(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_11␣=␣vev**4/s**2␣*␣amp_11"; nl ();
printf "␣␣end␣function␣da11"; nl();
nl ();
printf "␣␣%sfunction␣da20␣(cc,␣s,␣m)␣result␣(amp_20)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a20_0,␣a20_1,␣a20_a,␣a20_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a20"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_20"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1.0/32.0/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2/vev**4*ii"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣a20(1)␣=␣-2.0*cc(1)**2/vev**2␣*␣s0stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a20(2)␣=␣-␣cc(2)**2/vev**2/6.␣*␣s0stu(s,m(2))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(2)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a20(2)␣=␣a20(2)␣-␣cc(2)**2/vev**2/2.␣*&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(2)**2,m(2)*wkm(2),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a20(3)␣=␣cc(3)**2*(2.0*p0stu(s,m(3))␣+␣3.0*s/m(3)**2)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(4)␣=␣-␣cc(4)**2/vev**2*(d0stu(s,m(4)-2*kappal*s0stu(s,m(4)))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/3.0)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a20(5)␣=␣-␣cc(5)**2/vev**2*(d0stu(s,m(5))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣)/36.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12␣*␣(s0stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣)␣-␣3*(s/m(4)**2-s**2/m(4)**4)*s)";
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12␣*␣(s0stu(s,m(4))␣
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣)␣-␣6*(s/m(4)**2-s**2/m(4)**4)*s)";
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a20_f␣=␣-␣fudge_higgs*s/vev**2"; nl ();
printf "␣␣␣␣␣␣a20_f␣=␣a20_f␣-␣0*2*(1-ghvva)**2/vev**2*mass(25)**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a20_0␣=␣␣16*(2*a4␣+␣a5)/3*s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a20_1␣=␣(10*log(lam_reg**2/s)/9␣+␣25/108.0_default)␣*␣s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a20_a␣=␣a20_0␣!!!␣+␣a20_1/16/Pi**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_20␣=␣sum(a20)+a20_f+a20_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_20␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_20␣=␣-␣a20_a␣-␣a20_f␣-␣part_r␣*␣(sum(a20)␣-␣a20(3))␣+␣1/(real(1/amp_20,defaul
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_20␣=␣(1-part_r)␣*␣sum(a20)␣+␣part_r␣*␣a20(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_20␣=␣vev**4/s**2␣*␣amp_20"; nl ();
printf "␣␣end␣function␣da20"; nl();
nl ();
printf "␣␣%sfunction␣da22␣(cc,␣s,␣m)␣result␣(amp_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a22_0,␣a22_1,␣a22_a,␣a22_r"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a22"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_22"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1.0/32.0/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2/vev**4*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(1)␣=␣-␣2.0*cc(1)**2/vev**2␣*␣s2stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a22(2)␣=␣-␣cc(2)**2/vev**2␣*␣s2stu(s,m(2))␣/␣6.0"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣triplet"; nl ();
printf "␣␣␣␣␣␣a22(3)␣=␣2.0*cc(3)**2*(2*s+m(3)**2)*s2stu(s,m(3))/m(3)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(4)␣=␣-␣cc(4)**2/vev**2*((1.0␣+␣6.0*s/m(4)**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣+6.0*s**2/m(4)**4-2*kappal)*s2stu(s,m(4))/3.0)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a22(5)␣=␣-␣cc(5)**2/vev**2/36.␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣((1.+6.*s/m(5)**2+6.*s**2/m(5)**4␣)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s2stu(s,m(5)))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(5)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a22(5)␣=␣a22(5)␣-␣cc(5)**2/vev**2/60␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(5)**2,m(5)*wkm(5),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12␣*␣(s2stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12␣*␣(s2stu(s,m(4))␣
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a22_0␣=␣4*(a4␣+␣2*a5)/15*s**2/vev**4␣"; nl ();
printf "␣␣␣␣␣␣a22_1␣=␣(2*log(lam_reg**2/s)/45␣-␣247/5400.0_default)*s**2/vev**4"; nl ();
printf "␣␣␣␣␣␣a22_a␣=␣a22_0␣!!!␣+␣a22_1/16/Pi**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_22␣=␣sum(a22)+a22_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_22␣=␣-␣a22_a␣-␣part_r␣*␣(sum(a22)␣-␣a22(3))␣+␣1/(real(1/amp_22,default)-ii)";
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_22␣=␣(1-part_r)␣*␣sum(a22)␣+␣part_r␣*␣a22(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_22␣=␣vev**4/s**2␣*␣amp_22"; nl ();
printf "␣␣end␣function␣da22"; nl();
nl ();
printf "␣␣%sfunction␣dalzz0_s␣(cc,m,k)␣result␣(alzz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz0_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz0_s␣=␣2*g**4/costhw**2*((da00(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣da20(cc,s,m))/24␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣(5.)*(da02(cc,s,m)␣-␣da22(cc,s,m))/12)"; nl ();
printf "␣␣end␣function␣dalzz0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz0_t␣(cc,m,k)␣result␣(alzz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz0_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz0_t␣=␣(5.)*g**4/costhw**2*(da02(cc,s,m)␣-␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣da22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣dalzz0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz1_s␣(cc,m,k)␣result␣(alzz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_s␣=␣g**4/costhw**2*(da20(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣(5.)*da22(cc,s,m)/4)"; nl ();
printf "␣␣end␣function␣dalzz1_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz1_t␣(cc,m,k)␣result␣(alzz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_t␣=␣g**4/costhw**2*(-␣(3.)*da11(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*(5.)*da22(cc,s,m)/8)"; nl ();
printf "␣␣end␣function␣dalzz1_t"; nl ();
nl ();
```

```
printf "␣␣%sfunction␣dalzz1_u␣(cc,m,k)␣result␣(alzz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_u"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_u␣=␣g**4/costhw**2*((3.)*da11(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*(5.)*da22(cc,s,m)/8)"; nl ();
printf "␣␣end␣function␣dalzz1_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_s␣(cc,m,k)␣result␣(alww0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_s␣=␣g**4*((2*da00(cc,s,m)␣+␣da20(cc,s,m))/24␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣(5.)*(2*da02(cc,s,m)␣+␣da22(cc,s,m))/12)"; nl ();
printf "␣␣end␣function␣dalww0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_t␣(cc,m,k)␣result␣(alww0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_t␣=␣g**4*(2*(5.)*da02(cc,s,m)␣-␣(3.)*da11(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣(5.)*da22(cc,s,m))/8"; nl ();
printf "␣␣end␣function␣dalww0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_u␣(cc,m,k)␣result␣(alww0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_u"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_u␣=␣g**4*(2*(5.)*da02(cc,s,m)␣+␣(3.)*da11(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣(5.)*da22(cc,s,m))/8"; nl ();
printf "␣␣end␣function␣dalww0_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalww2_s␣(cc,m,k)␣result␣(alww2_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww2_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww2_s␣=␣g**4*(da20(cc,s,m)␣-␣2*(5.)*da22(cc,s,m))/4␣"; nl ();
printf "␣␣end␣function␣dalww2_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalww2_t␣(cc,m,k)␣result␣(alww2_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww2_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
```

```
      printf "␣␣␣␣␣␣alww2_t␣=␣3*(5.)*g**4*da22(cc,s,m)/4"; nl ();
      printf "␣␣end␣function␣dalww2_t"; nl ();
      nl ();
      printf "␣␣%sfunction␣dalz4_s␣(cc,m,k)␣result␣(alz4_s)" pure; nl ();
      printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
      printf "␣␣␣␣␣␣complex(kind=default)␣::␣alz4_s"; nl ();
      printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
      printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
      printf "␣␣␣␣␣␣alz4_s␣=␣g**4/costhw**4*((da00(cc,s,m)␣&"; nl ();
      printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da20(cc,s,m))/12␣&"; nl ();
      printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣(5.)*(da02(cc,s,m)+2*da22(cc,s,m))/6)"; nl ();
      printf "␣␣end␣function␣dalz4_s"; nl ();
      nl ();
      printf "␣␣@[<5>";
      printf "␣␣%sfunction␣dalz4_t␣(cc,m,k)␣result␣(alz4_t)" pure; nl ();
      printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:12),␣intent(in)␣::␣cc"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
      printf "␣␣␣␣␣␣complex(kind=default)␣::␣alz4_t"; nl ();
      printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
      printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
      printf "␣␣␣␣␣␣alz4_t␣=␣g**4/costhw**4*(5.)*(da02(cc,s,m)␣&"; nl ();
      printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da22(cc,s,m))/4"; nl ();
      printf "␣␣end␣function␣dalz4_t"; nl ();
      nl ();
  end
```

20.11 Interface of Targets_Kmatrix_2

```
module Fortran : sig val print : bool → unit end
```

20.12 Implementation of Targets_Kmatrix_2

```
module Fortran =
  struct

    open Format

    let nl = print_newline
```

Special functions for the K matrix approach. This might be generalized to other functions that have to have access to the parameters and coupling constants. At the moment, this is hardcoded.

```
    let print pure_functions =
      let pure =
        if pure_functions then
          "pure␣"
        else
          "" in
      printf "␣␣!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"; nl ();
      printf "␣␣!!!␣Special␣K␣matrix␣functions"; nl ();
      printf "␣␣!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"; nl ();
      nl();
      printf "␣␣%sfunction␣width_res␣(z,res,w_wkm,m,g)␣result␣(w)" pure; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣z,␣w_wkm,␣m,␣g"; nl ();
      printf "␣␣␣␣␣␣integer,␣intent(in)␣::␣res"; nl ();
      printf "␣␣␣␣␣␣real(kind=default)␣::␣w"; nl ();
      printf "␣␣␣␣␣␣if␣(z.eq.0␣.AND.␣w_wkm.eq.0␣)␣then"; nl ();
      printf "␣␣␣␣␣␣␣␣w␣=␣0"; nl ();
```

```
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣if␣(w_wkm.eq.0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣select␣case␣(res)"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(1)␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2␣/32._default/Pi␣*␣m**3"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(2)␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/128._default/Pi␣*␣m**3"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(3)␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/48._default/Pi␣*␣m"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(4)␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/960._default/Pi␣*␣m**3"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣(5)␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣g**2/3840._default/Pi␣*␣m**3"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣case␣default"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣0"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣end␣select"; nl ();
printf "␣␣␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣w␣=␣w_wkm"; nl ();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣width_res"; nl ();
nl ();
printf "␣␣%sfunction␣s0stu␣(s,␣m)␣result␣(s0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣s0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣s0␣=␣m**2␣-␣s/2␣+␣m**4/s␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s0stu"; nl();
nl ();
printf "␣␣%sfunction␣s1stu␣(s,␣m)␣result␣(s1)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s1"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣s1␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣s1␣=␣2*m**4/s␣+␣s/6␣+␣m**4/s**2*(2*m**2+s)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s1stu"; nl();
nl ();
printf "␣␣%sfunction␣s2stu␣(s,␣m)␣result␣(s2)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s2"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣s2␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣s2␣=␣m**4/s**2␣*␣(6*m**2␣+␣3*s)␣+␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣m**4/s**3␣*␣(6*m**4␣+␣6*m**2*s␣+␣s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s2stu"; nl();
nl ();
printf "␣!!␣%sfunction␣s3stu␣(s,␣m)␣result␣(s3)" pure; nl ();
printf "␣!!␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣!!␣␣␣␣␣real(kind=default)␣::␣s3"; nl ();
printf "␣!!␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣!!␣␣␣␣␣␣␣s3␣=␣0"; nl ();
printf "␣!!␣␣␣␣␣else"; nl ();
printf "␣!!␣␣␣␣␣␣␣s3␣=␣m**4/s**3␣*␣(60*m**4␣+␣60*m**2*s+11*s**2)␣+␣&"; nl();
printf "␣!!␣␣␣␣␣␣␣␣␣␣␣␣m**4/s**4␣*(2*m**2+s)␣(10*m**4␣+␣10*m**2*s␣+␣s**2)␣&"; nl();
printf "␣!!␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣!!␣␣␣␣␣end␣if"; nl ();
printf "␣!!␣␣end␣function␣s3stu"; nl();
nl ();
printf "␣␣%sfunction␣p0stu␣(s,␣m)␣result␣(p0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣p0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣p0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣p0␣=␣1._default␣+␣(2*s+m**2)*log(m**2/(s+m**2))/s"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣p0stu"; nl();
nl ();
printf "␣␣%sfunction␣p1stu␣(s,␣m)␣result␣(p1)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣p1"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣p1␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣p1␣=␣(m**2␣+␣2*s)/s**2␣*␣(2*s+(2*m**2+s)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2)))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣p1stu"; nl();
nl ();
printf "␣␣%sfunction␣d0stu␣(s,␣m)␣result␣(d0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣d0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣d0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣d0␣=␣(2*m**2+11*s)/2␣+␣(m**4+6*m**2*s+6*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣/s␣*␣log(m**2/(s+m**2))"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣d0stu"; nl();
nl ();
printf "␣␣%sfunction␣d1stu␣(s,␣m)␣result␣(d1)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣d1"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣d1␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣d1␣=␣(s*(12*m**4␣+␣72*m**2*s␣+␣73*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣+␣6*(2*m**2␣+␣s)*(m**4␣+␣6*m**2*s␣+␣6*s**2)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣*␣log(m**2/(s+m**2)))/6/s**2"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣d1stu"; nl();
nl ();
printf "␣␣%sfunction␣da00␣(cc,␣s,␣m)␣result␣(amp_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a00_0,␣a00_a,␣a00_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a00"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_00"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2␣*␣ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣0*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)"; nl (
printf "␣␣␣␣␣␣kappam␣=␣0*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣0*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(1)␣=␣-␣cc(1)**2␣*␣s0stu(s,m(1))␣/␣2␣"; nl ();
printf "␣␣␣␣␣␣if␣(cc(1)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(1)␣=␣a00(1)␣-␣3/4.0_default␣*cc(1)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(1)**2,m(1)*wkm(1),default)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a00(2)␣=␣-7*cc(2)**2␣*␣s0stu(s,m(2))/8.0_default"; nl ();
printf "␣␣␣␣␣␣if␣(cc(2)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(2)␣=␣a00(2)␣-␣1/16.0_default␣*cc(2)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(2)**2,m(2)*wkm(2),default)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a00(3)␣=␣-cc(3)**2*(4*p0stu(s,m(3))␣+␣6*s/m(3)**2)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(4)␣=␣-cc(4)**2␣*␣(d0stu(s,m(4))/12.0_default)"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(cc(13)␣/=␣2))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(4)␣=␣a00(4)␣+␣cc(4)**2␣*␣(cc(13)␣-␣2.0_default)␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cc(13)*␣(␣1.0_default␣/␣16.0_default␣*␣␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(4)**2,m(4)*wkm(4),default)&␣"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣s0stu(s,m(4))/24.0_default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a00(5)␣=␣-7*cc(5)**2␣*␣d0stu(s,m(5))/␣48.0_default␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(6)␣=␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2*␣s**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣((2-2*s/m(4)**2+s**2/m(4)**4)+kappat/2._default␣)/4"; nl ();
printf "␣␣␣␣␣␣if␣(a00(6)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(6)␣=␣a00(6)/cmplx(s-m(4)**2,␣-␣w_res/Pi␣*␣real(a00(6),default)/32,default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a00(6)␣=␣a00(6)␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2␣*␣(s0stu(s,m(4))␣&"; nl ()
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1._default+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a00(7)␣=␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)␣*␣s**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣␣((1._default-4*s/m(4)**2+2*s**2/m(4)**4)+kappam␣)/4"; nl ();
printf "␣␣␣␣␣␣if␣(a00(7)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a00(7)␣=␣a00(7)/cmplx(s-m(4)**2,␣-␣w_res/Pi/32␣*␣real(a00(7),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a00(7)␣=␣a00(7)␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)␣*␣(s0stu(s,m
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a00_f␣=␣2.*fudge_higgs*s/vev**2"; nl ();
printf "␣␣␣␣␣␣a00_f␣=␣a00_f␣!!!␣-␣0*5*(1._default-ghvva)**2/vev**2*mass(25)**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a00_0␣=␣(7*fs0␣+␣11*fs1)/␣6.0_default␣*s**2"; nl ();
printf "␣␣␣␣␣␣a00_a␣=␣␣a00_0␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_00␣=␣sum(a00)+a00_f+a00_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_00␣=␣-␣a00_a␣-␣a00_f␣-␣part_r␣*␣(sum(a00)␣-␣a00(3))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣+␣1/(real(1/amp_00,default)-ii)"; nl();
printf "␣␣␣␣␣␣!!!␣Validation␣!!!"; nl ();
printf "␣␣␣␣␣␣!!!␣amp_00␣=␣-␣a00_a␣-␣a00_f␣-␣part_r␣*␣(sum(a00)␣-␣a00(3))"; nl();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_00␣=␣(1-part_r)␣*␣sum(a00)␣+␣part_r␣*␣a00(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(unit_limit)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_00␣=␣32␣*␣Pi␣*␣(␣(0._default,0.5_default)␣+␣0.5_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣exp(cmplx(0._default,Pi␣*␣(amp00␣-␣0.5_default),default)␣)␣)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣amp_00␣=␣vev**4/s**2␣*␣amp_00"; nl ();
printf "␣␣end␣function␣da00"; nl();
nl ();
printf "␣␣%sfunction␣da02␣(cc,␣s,␣m)␣result␣(amp_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a02_0,␣a02_a"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a02"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_02"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(1)␣=␣-cc(1)**2/2.0_default␣*␣s2stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a02(2)␣=␣-7*cc(2)**2␣*␣s2stu(s,m(2))␣/␣8.0_default"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a02(3)␣=␣-4*cc(3)**2*(2*s+m(3)**2)*s2stu(s,m(3))/m(3)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(4)␣=␣-␣cc(4)**2␣/␣12.0_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣((1._default+6*s/m(4)**2+6*s**2/m(4)**4))␣*␣s2stu(s,m(4))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(4)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(4)␣=␣a02(4)␣-␣cc(4)**2/40.0_default␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s**2/cmplx(s-m(4)**2,m(4)*wkm(4),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(cc(13)␣/=␣2))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(4)␣=␣a02(4)␣+␣cc(4)**2␣*␣(cc(13)␣-␣2.0_default)␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cc(13)␣*␣s2stu(s,m(4))/24.0_default"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a02(5)␣=␣-cc(5)**2␣*␣7.0_default␣/␣48.0_default␣*(1._default+6*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s/m(5)**2+6*s**2/m(5)**4)*s2stu(s,m(5))␣"; nl ();
printf "␣␣␣␣␣␣if␣(cc(5)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(5)␣=␣a02(5)␣-␣cc(5)**2/480.0_default␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s**2/cmplx(s-m(5)**2,m(5)*wkm(5),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(6)␣=␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2␣*␣s**2/40"; nl ();
printf "␣␣␣␣␣␣if␣(a02(6)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(6)␣=␣a02(6)/cmplx(s-m(4)**2,␣-␣w_res/32/Pi␣*␣real(a02(6),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a02(6)␣=␣a02(6)␣-␣cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2␣*␣(s2stu(s,m(4))␣&"; nl ()
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1._default+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a02(7)␣=␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s**2/20"; nl ();
printf "␣␣␣␣␣␣if␣(a02(7)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a02(7)␣=␣a02(7)/cmplx(s-m(4)**2,␣-␣w_res/Pi/32␣*␣real(a02(7),default),default)␣"; nl
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣a02(7)␣=␣a02(7)␣-␣cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)␣*␣(s2stu(s,m
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a02_0␣=␣(2*fs0␣+␣fs1)␣/␣30._default␣*␣␣s**2"; nl ();
printf "␣␣␣␣␣␣a02_a␣=␣a02_0␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_02␣=␣sum(a02)+a02_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_02␣=␣-␣a02_a␣-␣part_r␣*␣(sum(a02)␣-␣a02(3))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣1/(real(1/amp_02,default)-ii)"; nl();
printf "␣␣␣␣␣␣!!!␣Validation␣!!!"; nl ();
printf "␣␣␣␣␣␣!!!␣amp_02␣=␣-␣a02_a␣-␣part_r␣*␣(sum(a02)␣-␣a02(3))"; nl();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_02␣=␣(1._default-part_r)␣*␣sum(a02)␣+␣part_r␣*␣a02(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(unit_limit)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_02␣=␣32␣*␣Pi␣*␣(␣(0.0_default,0.5_default)␣+␣0.5_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣exp(cmplx(0._default,Pi␣*␣(amp02␣-␣0.5_default),default)␣)␣)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_02␣=␣vev**4/s**2␣*␣amp_02"; nl ();
printf "␣␣end␣function␣da02"; nl();
nl ();
printf "␣␣%sfunction␣da11␣(cc,␣s,␣m)␣result␣(amp_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a11_0,␣a11_a,␣a11_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a11"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_11"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(1)␣=␣-␣cc(1)**2/2.0_default␣*␣s1stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a11(2)␣=␣3*cc(2)**2␣*␣s1stu(s,m(2))␣/␣8.0_default"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a11(3)␣=␣-␣cc(3)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(s/m(3)**2␣+␣2␣*␣p1stu(s,m(3)))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(3)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a11(3)␣=␣a11(3)␣-2␣*␣cc(3)**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s/cmplx(s-m(3)**2,m(3)*wkm(3),default)/3␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(4)␣=␣-␣cc(4)**2*␣d1stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/12.0_default␣"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(cc(13)␣/=␣2))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a11(4)␣=␣a11(4)␣+␣cc(4)**2␣*␣(cc(13)␣-␣2.0_default)␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cc(13)␣*␣s1stu(s,m(4))/24.0_default␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a11(5)␣=␣␣cc(5)**2␣/␣16.0_default␣*␣d1stu(s,m(5))␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12._default␣*␣(s1stu(s,m(4))␣*␣&
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣(3*(1._default+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣)␣-␣(s/m(4)**2+s**2/m(4)**4)
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a11(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12._default␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(s1stu(s,m(4))␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*(s/m(4)**2+s**2/m(4)**4)*s)"; nl ();
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a11_f␣=␣fudge_higgs*s/vev**2/3"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a11_0␣=␣(fs0␣-␣2*fs1)␣/␣12.0_default␣*␣s**2␣␣"; nl ();
printf "␣␣␣␣␣␣a11_a␣=␣a11_0"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_11␣=␣sum(a11)+a11_f+a11_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_11␣=␣-a11_a␣-␣part_r␣*␣(sum(a11)␣-␣a11(3))&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣1/(real(1/amp_11,default)-ii)"; nl();
printf "␣␣␣␣␣␣!!!␣Validation␣!!!"; nl ();
printf "␣␣␣␣␣␣!!!␣amp_11␣=␣-␣a11_a␣-␣part_r␣*␣(sum(a11)␣-␣a11(3))"; nl();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_11␣=␣(1._default-part_r)␣*␣sum(a11)␣+␣part_r␣*␣a11(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(unit_limit)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_11␣=␣32␣*␣Pi␣*␣(␣(0.0_default,0.5_default)␣+␣0.5_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣exp(cmplx(0._default,Pi␣*␣(amp11␣-␣0.5_default),default)␣)␣)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_11␣=␣vev**4/s**2␣*␣amp_11"; nl ();
printf "␣␣end␣function␣da11"; nl();
nl ();
printf "␣␣%sfunction␣da20␣(cc,␣s,␣m)␣result␣(amp_20)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a20_0,␣a20_a,␣a20_f"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a20"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_20"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2*ii"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣a20(1)␣=␣-cc(1)**2/2.0_default␣*␣s0stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a20(2)␣=␣-␣cc(2)**2␣*␣s0stu(s,m(2))␣/8.0_default"; nl ();
printf "␣␣␣␣␣␣if␣(cc(2)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a20(2)␣=␣a20(2)␣-␣cc(2)**2␣/4.0_default␣*&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(2)**2,m(2)*wkm(2),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣isotriplet"; nl ();
printf "␣␣␣␣␣␣a20(3)␣=␣cc(3)**2*(2*p0stu(s,m(3))␣+␣3*s/m(3)**2)"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(4)␣=␣-␣cc(4)**2*␣d0stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/12.0_default␣"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(cc(13)␣/=␣2))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a20(4)␣=␣a20(4)␣+␣cc(4)**2␣*␣(cc(13)␣-␣2.0_default)␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cc(13)␣*␣s0stu(s,m(4))/24.0_default␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a20(5)␣=␣-␣cc(5)**2/␣48.0_default␣*␣d0stu(s,m(5))␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2/12._default␣*␣(s0stu(s,m(4))␣&";
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1._default+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣-␣3*(s/m(4)**2-s**2/m(4)**4)*s)"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a20(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)/12._default␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(s0stu(s,m(4))␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣6*(s/m(4)**2-s**2/m(4)**4)*s)"; nl ();
printf "␣␣␣␣␣␣!!!␣Fudge-Higgs"; nl ();
printf "␣␣␣␣␣␣a20_f␣=␣-␣fudge_higgs*s/vev**2"; nl ();
printf "␣␣␣␣␣␣a20_f␣=␣a20_f␣!!!␣-␣0*2*(1._default-ghvva)**2/vev**2*mass(25)**2"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a20_0␣=␣␣(2*fs0␣+␣fs1)␣/␣3.0_default␣*␣s**2"; nl ();
printf "␣␣␣␣␣␣a20_a␣=␣a20_0␣"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_20␣=␣sum(a20)+a20_f+a20_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_20␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_20␣=␣-␣a20_a␣-␣a20_f␣-␣part_r␣*␣(sum(a20)␣-␣a20(3))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣1/(real(1/amp_20,default)-ii)"; nl();
printf "␣␣␣␣␣␣!!!␣Validation␣!!!"; nl ();
printf "␣␣␣␣␣␣!!!␣amp_20␣=␣-␣a20_a␣-␣a20_f␣-␣part_r␣*␣(sum(a20)␣-␣a20(3))␣"; nl();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_20␣=␣(1-part_r)␣*␣sum(a20)␣+␣part_r␣*␣a20(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(unit_limit)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_20␣=␣32␣*␣Pi␣*␣(␣(0.0_default,0.5_default)␣+␣0.5_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣exp(cmplx(0._default,Pi␣*␣(amp20␣-␣0.5_default),default)␣)␣)␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_20␣=␣vev**4/s**2␣*␣amp_20"; nl ();
printf "␣␣end␣function␣da20"; nl();
nl ();
printf "␣␣%sfunction␣da22␣(cc,␣s,␣m)␣result␣(amp_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣a22_0,␣a22_1,␣a22_a,␣a22_r"; nl ();
printf "␣␣␣␣␣␣complex(kind=default),␣dimension(1:7)␣::␣a22"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣jj,␣amp_22"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣kappal,␣kappam,␣kappat"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣jj␣=␣s**2*ii"; nl ();
printf "␣␣␣␣␣␣kappal␣=␣cc(12)*((mass(23)**2+mass(24)**2)/m(4)**2-2*mass(23)**2*mass(24)**2/m(4)**4)
printf "␣␣␣␣␣␣kappam␣=␣cc(12)*((mass(23)**4+mass(24)**4)/m(4)**2/(mass(23)**2+mass(24)**2)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣2*mass(23)**2*mass(24)**2/m(4)**4)"; nl ();
printf "␣␣␣␣␣␣kappat␣=␣cc(12)*mass(23)**2*mass(24)**2/m(4)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Longitudinal"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(1)␣=␣-␣cc(1)**2/2.0_default␣*␣s2stu(s,m(1))"; nl ();
printf "␣␣␣␣␣␣!!!␣Scalar␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a22(2)␣=␣-␣cc(2)**2␣*␣s2stu(s,m(2))/␣8.0_default"; nl ();
printf "␣␣␣␣␣␣!!!␣Vector␣triplet"; nl ();
printf "␣␣␣␣␣␣a22(3)␣=␣2*cc(3)**2*(2*s+m(3)**2)*s2stu(s,m(3))/m(3)**4"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(4)␣=␣-␣cc(4)**2*((1._default␣+␣6*s/m(4)**2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣+6*s**2/m(4)**4)*s2stu(s,m(4))/12.0_default)"; nl ();
printf "␣␣␣␣␣␣if␣(␣(cc(4)␣/=␣0).and.(cc(13)␣/=␣2))␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a22(4)␣=␣a22(4)␣+␣cc(4)**2␣*␣(cc(13)␣-␣2.0_default)␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cc(13)␣*␣s2stu(s,m(4))␣/␣24.0_default␣"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isoquintet"; nl ();
printf "␣␣␣␣␣␣a22(5)␣=␣-␣cc(5)**2/48._default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣((1._default+6*s/m(5)**2+6*s**2/m(5)**4␣)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣s2stu(s,m(5)))"; nl ();
printf "␣␣␣␣␣␣if␣(cc(5)␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣a22(5)␣=␣a22(5)␣-␣cc(5)**2/120._default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s**2/cmplx(s-m(5)**2,m(5)*wkm(5),default)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣!!!␣Transversal"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(6)␣=␣-cc(9)**2/Pi/vev**6*mass(23)**2*mass(24)**2␣*␣(s2stu(s,m(4))␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(3*(1._default+2*s/m(4)**2+2*s**2/m(4)**4)+kappat␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Mixed"; nl ();
printf "␣␣␣␣␣␣!!!␣Tensor␣isosinglet"; nl ();
printf "␣␣␣␣␣␣a22(7)␣=␣-cc(11)*cc(9)*cc(4)/Pi/vev**4*(mass(23)**2+mass(24)**2)␣*␣(s2stu(s,m(4))␣&";
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣(12*s/m(4)**2+12*s**2/m(4)**4+2*kappam␣))/12"; nl ();
printf "␣␣␣␣␣␣!!!␣Low␣energy␣theory␣alphas"; nl ();
printf "␣␣␣␣␣␣a22_0␣=␣(fs0␣+␣2*fs1)/60._default*s**2␣"; nl ();
printf "␣␣␣␣␣␣a22_a␣=␣a22_0"; nl ();
printf "␣␣␣␣␣␣!!!␣Unitarize"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_22␣=␣sum(a22)+a22_a"; nl();
printf "␣␣␣␣␣␣␣␣if␣(amp_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣amp_22␣=␣-␣a22_a␣-␣part_r␣*␣(sum(a22)␣-␣a22(3))␣&␣"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣1/(real(1/amp_22,default)-ii)"; nl();
printf "␣␣␣␣␣␣!!!␣Validation␣!!!"; nl ();
printf "␣␣␣␣␣␣!!!␣amp_22␣=␣-␣a22_a␣-␣part_r␣*␣(sum(a22)␣-␣a22(3))"; nl();
printf "␣␣␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣amp_22␣=␣(1-part_r)␣*␣sum(a22)␣+␣part_r␣*␣a22(3)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣if␣(unit_limit)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣amp_22␣=␣32␣*␣Pi␣*␣(␣(0.0_default,0.5_default)␣+␣0.5_default␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣exp(cmplx(0._default,Pi␣*␣(amp22␣-␣0.5_default),default)␣)␣)"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣amp_22␣=␣vev**4/s**2␣*␣amp_22"; nl ();
printf "␣␣end␣function␣da22"; nl();
nl ();
printf "␣␣%sfunction␣s0stu_t␣(s,␣m)␣result␣(s0)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s,␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s0"; nl ();
printf "␣␣␣␣␣␣if␣(m.ge.1.0e08)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣s0␣=␣0"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣s0␣=␣(m**2/3␣-␣m**4/(2*s)␣+␣m**6/s**2␣-␣s/4␣+␣m**8/s**3␣*␣log(m**2/(m**2+s)))"; nl ()
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣end␣function␣s0stu_t"; nl();
nl ();
printf "␣␣%sfunction␣dat00_0␣(cc,␣s,␣m)␣result␣(ampt_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣-g**4*6*ft0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣1/(1/ampt_00␣-␣ii)␣-␣ampt_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣ampt_00␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat00_0"; nl ();
nl ();
printf "␣␣%sfunction␣dat02_0␣(cc,␣s,␣m)␣result␣(ampt_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣-g**4*2/5*ft0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣1/(1/ampt_02␣-␣ii)␣-␣ampt_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣ampt_02␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat02_0"; nl ();
nl ();
printf "␣␣%sfunction␣dat12_0␣(cc,␣s,␣m)␣result␣(ampt_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣-g**4*2/5*ft0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣1/(1/ampt_12␣-␣ii)␣-␣ampt_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣ampt_12␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat12_0"; nl ();
nl ();
printf "␣␣%sfunction␣dat22_0␣(cc,␣s,␣m)␣result␣(ampt_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣-g**4*2/5*ft0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣1/(1/ampt_22␣-␣ii)␣-␣ampt_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣ampt_22␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat22_0"; nl ();
nl ();
printf "␣␣%sfunction␣dat00_1␣(cc,␣s,␣m)␣result␣(ampt_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣-␣g**4*3*ft1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣1/(1/ampt_00␣-␣ii)␣-␣ampt_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣ampt_00␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat00_1"; nl ();
nl ();
printf "␣␣%sfunction␣dat02_1␣(cc,␣s,␣m)␣result␣(ampt_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣-␣g**4*1/5*ft1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣1/(1/ampt_02␣-␣ii)␣-␣ampt_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣ampt_02␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat02_1"; nl ();
nl ();
printf "␣␣%sfunction␣dat12_1␣(cc,␣s,␣m)␣result␣(ampt_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣-␣g**4*1/5*ft1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣1/(1/ampt_12␣-␣ii)␣-␣ampt_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣ampt_12␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat12_1"; nl ();
nl ();
```

```
printf "␣␣%sfunction␣dat22_1␣(cc,␣s,␣m)␣result␣(ampt_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣-␣g**4*1/5*ft1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣1/(1/ampt_22␣-␣ii)␣-␣ampt_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣ampt_22␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat22_1"; nl ();
nl ();
printf "␣␣%sfunction␣dat00_2␣(cc,␣s,␣m)␣result␣(ampt_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣-g**4*3/2*ft2*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣1/(1/ampt_00␣-␣ii)␣-␣ampt_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣ampt_00␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat00_2"; nl ();
nl ();
printf "␣␣%sfunction␣dat02_2␣(cc,␣s,␣m)␣result␣(ampt_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣-g**4*1/10*ft2*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣1/(1/ampt_02␣-␣ii)␣-␣ampt_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣ampt_02␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat02_2"; nl ();
nl ();
printf "␣␣%sfunction␣dat11_2␣(cc,␣s,␣m)␣result␣(ampt_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_11"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_11␣=␣-g**4*1/6*ft2*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_11␣=␣1/(1/ampt_11␣-␣ii)␣-␣ampt_11"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_11␣=␣ampt_11␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat11_2"; nl ();
nl ();
printf "␣␣%sfunction␣dat22_2␣(cc,␣s,␣m)␣result␣(ampt_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣-g**4*1/10*ft2*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣1/(1/ampt_22␣-␣ii)␣-␣ampt_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣ampt_22␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat22_2"; nl ();
nl ();
printf "␣␣%sfunction␣dat00_rsi␣(cc,␣s,␣m)␣result␣(ampt_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣3*g**4*gkm(6)**2*s**2/cmplx(s-m(1)**2,m(1)*wkm(1),default)"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣1/(1/ampt_00␣-␣ii)␣-␣ampt_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_00␣=␣ampt_00␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat00_rsi"; nl ();
nl ();
printf "␣␣%sfunction␣dat02_rsi␣(cc,␣s,␣m)␣result␣(ampt_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣-␣g**4*4/5*cc(6)**2*s0stu_t(s,m(1))"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣1/(1/ampt_02␣-␣ii)␣-␣ampt_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_02␣=␣ampt_02␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat02_rsi"; nl ();
nl ();
printf "␣␣%sfunction␣dat12_rsi␣(cc,␣s,␣m)␣result␣(ampt_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣-␣g**4*4/5*cc(6)**2*s0stu_t(s,m(1))"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣1/(1/ampt_12␣-␣ii)␣-␣ampt_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_12␣=␣ampt_12␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat12_rsi"; nl ();
nl ();
printf "␣␣%sfunction␣dat22_rsi␣(cc,␣s,␣m)␣result␣(ampt_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampt_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣-␣g**4*4/5*cc(6)**2*s0stu_t(s,m(1))"; nl ();
printf "␣␣␣␣␣␣if␣(ampt_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣1/(1/ampt_22␣-␣ii)␣-␣ampt_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampt_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampt_22␣=␣ampt_22␣/␣(s**2␣*␣g**4)"; nl ();
printf "␣␣end␣function␣dat22_rsi"; nl ();
nl ();
printf "␣␣%sfunction␣dam00␣(cc,␣s,␣m)␣result␣(ampm_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣g**2*3/2*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣1/(1/ampm_00␣-␣ii)␣-␣ampm_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣ampm_00␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam00"; nl ();
nl ();
printf "␣␣%sfunction␣dam01␣(cc,␣s,␣m)␣result␣(ampm_01)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_01"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣-g**2*1/8*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_01␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣1/(1/ampm_01␣-␣ii)␣-␣ampm_01"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣ampm_01␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam01"; nl ();
nl ();
printf "␣␣%sfunction␣dam02␣(cc,␣s,␣m)␣result␣(ampm_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣g**2*1/40*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣1/(1/ampm_02␣-␣ii)␣-␣ampm_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣ampm_02␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam02"; nl ();
nl ();
printf "␣␣%sfunction␣dam11␣(cc,␣s,␣m)␣result␣(ampm_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_11"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣-g**2*1/8*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣1/(1/ampm_11␣-␣ii)␣-␣ampm_11"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣ampm_11␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam11"; nl ();
nl ();
printf "␣␣%sfunction␣dam12␣(cc,␣s,␣m)␣result␣(ampm_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣g**2*1/40*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣1/(1/ampm_12␣-␣ii)␣-␣ampm_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣ampm_12␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam12"; nl ();
nl ();
printf "␣␣%sfunction␣dam21␣(cc,␣s,␣m)␣result␣(ampm_21)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_21"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣-g**2*1/8*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_21␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣1/(1/ampm_21␣-␣ii)␣-␣ampm_21"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣ampm_21␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam21"; nl ();
nl ();
printf "␣␣%sfunction␣dam22␣(cc,␣s,␣m)␣result␣(ampm_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣g**2*1/40*fm0*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣1/(1/ampm_22␣-␣ii)␣-␣ampm_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣ampm_22␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam22"; nl ();
nl ();
printf "␣␣%sfunction␣dam00_1␣(cc,␣s,␣m)␣result␣(ampm_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣g**2*3/8*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣1/(1/ampm_00␣-␣ii)␣-␣ampm_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣ampm_00␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam00_1"; nl ();
```

```
nl ();
printf "␣␣%sfunction␣dam01_1␣(cc,␣s,␣m)␣result␣(ampm_01)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_01"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣-g**2*1/32*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_01␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣1/(1/ampm_01␣-␣ii)␣-␣ampm_01"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣ampm_01␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam01_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam02_1␣(cc,␣s,␣m)␣result␣(ampm_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣g**2*1/160*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣1/(1/ampm_02␣-␣ii)␣-␣ampm_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣ampm_02␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam02_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam11_1␣(cc,␣s,␣m)␣result␣(ampm_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_11"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣-g**2*1/32*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣1/(1/ampm_11␣-␣ii)␣-␣ampm_11"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣ampm_11␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam11_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam12_1␣(cc,␣s,␣m)␣result␣(ampm_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣g**2*1/160*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣1/(1/ampm_12␣-␣ii)␣-␣ampm_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣ampm_12␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam12_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam21_1␣(cc,␣s,␣m)␣result␣(ampm_21)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_21"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣-g**2*1/32*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_21␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣1/(1/ampm_21␣-␣ii)␣-␣ampm_21"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣ampm_21␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam21_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam22_1␣(cc,␣s,␣m)␣result␣(ampm_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣g**2*1/160*fm1*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣1/(1/ampm_22␣-␣ii)␣-␣ampm_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣ampm_22␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam22_1"; nl ();
nl ();
printf "␣␣%sfunction␣dam00_7␣(cc,␣s,␣m)␣result␣(ampm_00)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_00"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣g**2*3/16*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_00␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣1/(1/ampm_00␣-␣ii)␣-␣ampm_00"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_00␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_00␣=␣ampm_00␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam00_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam01_7␣(cc,␣s,␣m)␣result␣(ampm_01)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_01"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣g**2*3/32*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_01␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣1/(1/ampm_01␣-␣ii)␣-␣ampm_01"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_01␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_01␣=␣ampm_01␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam01_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam02_7␣(cc,␣s,␣m)␣result␣(ampm_02)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_02"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣g**2*1/160*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_02␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣1/(1/ampm_02␣-␣ii)␣-␣ampm_02"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_02␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_02␣=␣ampm_02␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam02_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam11_7␣(cc,␣s,␣m)␣result␣(ampm_11)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_11"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣g**2*3/32*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_11␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣1/(1/ampm_11␣-␣ii)␣-␣ampm_11"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_11␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_11␣=␣ampm_11␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam11_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam12_7␣(cc,␣s,␣m)␣result␣(ampm_12)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_12"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣g**2*1/160*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_12␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣1/(1/ampm_12␣-␣ii)␣-␣ampm_12"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_12␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_12␣=␣ampm_12␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam12_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam21_7␣(cc,␣s,␣m)␣result␣(ampm_21)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_21"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣g**2*3/32*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_21␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣1/(1/ampm_21␣-␣ii)␣-␣ampm_21"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_21␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_21␣=␣ampm_21␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam21_7"; nl ();
nl ();
printf "␣␣%sfunction␣dam22_7␣(cc,␣s,␣m)␣result␣(ampm_22)" pure; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣intent(in)␣::␣s"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ii,␣ampm_22"; nl ();
printf "␣␣␣␣␣␣ii␣=␣cmplx(0.0,1/32._default/Pi,default)"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣g**2*1/160*fm7*s**2"; nl ();
printf "␣␣␣␣␣␣if␣(ampm_22␣/=␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣1/(1/ampm_22␣-␣ii)␣-␣ampm_22"; nl ();
printf "␣␣␣␣␣␣else"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if␣"; nl ();
printf "␣␣␣␣␣␣if␣(fudge_km␣==␣0)␣then"; nl ();
printf "␣␣␣␣␣␣␣␣ampm_22␣=␣0"; nl ();
printf "␣␣␣␣␣␣end␣if"; nl ();
printf "␣␣␣␣␣␣ampm_22␣=␣ampm_22␣/␣(s**2)"; nl ();
printf "␣␣end␣function␣dam22_7"; nl ();
nl ();
printf "␣␣%sfunction␣dalh4_s␣(cc,m,k)␣result␣(alh4_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alh4_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alh4_s␣=␣16.0_default␣*␣cc(14)␣/␣vev**4␣*((da00(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da20(cc,s,m))/12␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*(da02(cc,s,m)+2*da22(cc,s,m))/6)"; nl ();
printf "␣␣end␣function␣dalh4_s"; nl ();
nl ();
printf "␣␣@[<5>";
printf "␣␣%sfunction␣dalh4_t␣(cc,m,k)␣result␣(alh4_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alh4_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alh4_t␣=␣80.0_default␣*␣cc(14)␣/␣vev**4␣*(da02(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣dalh4_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalhw0_s␣(cc,m,k)␣result␣(alhw0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhw0_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alhw0_s␣=␣-␣8.0_default␣*␣cc(14)␣*␣g**2/vev**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣((da00(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣da20(cc,s,m))/24␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*(da02(cc,s,m)␣-␣da22(cc,s,m))/12)"; nl ();
printf "␣␣end␣function␣dalhw0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalhw0_t␣(cc,m,k)␣result␣(alhw0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhw0_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alhw0_t␣=␣-␣5.0_default␣*␣cc(14)␣*␣g**2/␣vev**2␣*␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(da02(cc,s,m)␣-␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣da22(cc,s,m))"; nl ();
printf "␣␣end␣function␣dalhw0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalhz0_s␣(cc,m,k)␣result␣(alhz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhz0_s"; nl ();
printf "␣␣␣␣␣␣alhz0_s␣=␣dalhw0_s(cc,m,k)␣/␣costhw**2"; nl();
printf "␣␣end␣function␣dalhz0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalhz0_t␣(cc,m,k)␣result␣(alhz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhz0_t"; nl ();
printf "␣␣␣␣␣␣alhz0_t␣=␣dalhw0_t(cc,m,k)␣/␣costhw**2"; nl();
printf "␣␣end␣function␣dalhz0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalhw1_s␣(cc,m,k)␣result␣(alhw1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhw1_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alhw1_s␣=␣-␣cc(14)*g**2/vev**2*(da20(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*da22(cc,s,m))"; nl ();
printf "␣␣end␣function␣dalhw1_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalhw1_t␣(cc,m,k)␣result␣(alhw1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhw1_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alhw1_t␣=␣-␣cc(14)*g**2/vev**2*(-␣3*da11(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*5*da22(cc,s,m)/2)"; nl ();
printf "␣␣end␣function␣dalhw1_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalhw1_u␣(cc,m,k)␣result␣(alhw1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhw1_u"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alhw1_u␣=␣-␣cc(14)*g**2/vev**2*(3*da11(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*5*da22(cc,s,m)/2)"; nl ();
printf "␣␣end␣function␣dalhw1_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalhz1_s␣(cc,m,k)␣result␣(alhz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhz1_s"; nl ();
printf "␣␣␣␣␣␣alhz1_s␣=␣dalhw1_s(cc,m,k)␣/␣costhw**2"; nl ();
printf "␣␣end␣function␣dalhz1_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalhz1_t␣(cc,m,k)␣result␣(alhz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhz1_t"; nl ();
printf "␣␣␣␣␣␣alhz1_t␣=␣dalhw1_t(cc,m,k)␣/␣costhw**2"; nl ();
printf "␣␣end␣function␣dalhz1_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalhz1_u␣(cc,m,k)␣result␣(alhz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alhz1_u"; nl ();
printf "␣␣␣␣␣␣alhz1_u␣=␣dalhw1_u(cc,m,k)␣/␣costhw**2"; nl ();
printf "␣␣end␣function␣dalhz1_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz0_s␣(cc,m,k)␣result␣(alzz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz0_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz0_s␣=␣2*g**4/costhw**2*((da00(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣da20(cc,s,m))/24␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*(da02(cc,s,m)␣-␣da22(cc,s,m))/12)"; nl ();
printf "␣␣end␣function␣dalzz0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz0_t␣(cc,m,k)␣result␣(alzz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz0_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz0_t␣=␣5*g**4/costhw**2*(da02(cc,s,m)␣-␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣da22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣dalzz0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz1_s␣(cc,m,k)␣result␣(alzz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_s␣=␣g**4/costhw**2*(da20(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*da22(cc,s,m)/4)"; nl ();
printf "␣␣end␣function␣dalzz1_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz1_t␣(cc,m,k)␣result␣(alzz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_t␣=␣g**4/costhw**2*(-␣3*da11(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*5*da22(cc,s,m)/8)"; nl ();
printf "␣␣end␣function␣dalzz1_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalzz1_u␣(cc,m,k)␣result␣(alzz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alzz1_u"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alzz1_u␣=␣g**4/costhw**2*(3*da11(cc,s,m)/8␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣3*5*da22(cc,s,m)/8)"; nl ();
printf "␣␣end␣function␣dalzz1_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_s␣(cc,m,k)␣result␣(alww0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_s␣=␣g**4*((2*da00(cc,s,m)␣+␣da20(cc,s,m))/24␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*(2*da02(cc,s,m)␣+␣da22(cc,s,m))/12)"; nl ();
printf "␣␣end␣function␣dalww0_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_t␣(cc,m,k)␣result␣(alww0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_t␣=␣g**4*(2*(5.)*da02(cc,s,m)␣-␣3*da11(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣5*da22(cc,s,m))/8"; nl ();
printf "␣␣end␣function␣dalww0_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalww0_u␣(cc,m,k)␣result␣(alww0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww0_u"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww0_u␣=␣g**4*(2*(5.)*da02(cc,s,m)␣+␣3*da11(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣5*da22(cc,s,m))/8"; nl ();
printf "␣␣end␣function␣dalww0_u"; nl ();
nl ();
printf "␣␣%sfunction␣dalww2_s␣(cc,m,k)␣result␣(alww2_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww2_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww2_s␣=␣g**4*(da20(cc,s,m)␣-␣2*5*da22(cc,s,m))/4␣"; nl ();
printf "␣␣end␣function␣dalww2_s"; nl ();
nl ();
printf "␣␣%sfunction␣dalww2_t␣(cc,m,k)␣result␣(alww2_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alww2_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alww2_t␣=␣3*5*g**4*da22(cc,s,m)/4"; nl ();
printf "␣␣end␣function␣dalww2_t"; nl ();
nl ();
printf "␣␣%sfunction␣dalz4_s␣(cc,m,k)␣result␣(alz4_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alz4_s"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alz4_s␣=␣g**4/costhw**4*((da00(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da20(cc,s,m))/12␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣5*(da02(cc,s,m)+2*da22(cc,s,m))/6)"; nl ();
printf "␣␣end␣function␣dalz4_s"; nl ();
nl ();
printf "␣␣@[<5>";
printf "␣␣%sfunction␣dalz4_t␣(cc,m,k)␣result␣(alz4_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣alz4_t"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣alz4_t␣=␣g**4/costhw**4*5*(da02(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣2*da22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣dalz4_t"; nl ();
nl ();
printf "␣%sfunction␣datzz0_s_0␣(cc,m,k)␣result␣(atzz0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_s␣=␣-4*g**4*costhw**2*dat00_0(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datzz0_s_0"; nl ();
nl ();
printf "␣%sfunction␣datzz0_t_0␣(cc,m,k)␣result␣(atzz0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_t␣=␣-4*g**4*costhw**2*5*(dat02_0(cc,s,m)␣-␣dat22_0(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_t_0"; nl ();
nl ();
printf "␣%sfunction␣datzz0_u_0␣(cc,m,k)␣result␣(atzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_u␣=␣-4*g**4*costhw**2*5*(dat02_0(cc,s,m)␣-␣dat22_0(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_u_0"; nl ();
nl ();
printf "␣%sfunction␣datzz1_s_0␣(cc,m,k)␣result␣(atzz1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_s␣=␣0"; nl();
printf "␣␣end␣function␣datzz1_s_0"; nl ();
nl ();
printf "␣%sfunction␣datzz1_t_0␣(cc,m,k)␣result␣(atzz1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_t␣=␣-4*g**4*costhw**2*5*(dat12_0(cc,s,m)␣+␣dat22_0(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_t_0"; nl ();
nl ();
printf "␣%sfunction␣datzz1_u_0␣(cc,m,k)␣result␣(atzz1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_u␣=␣-4*g**4*costhw**2*5*(-dat12_0(cc,s,m)␣+␣dat22_0(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_u_0"; nl ();
```

```
nl ();
printf "␣%sfunction␣datww0_s_0␣(cc,m,k)␣result␣(atww0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_s␣=␣-4*g**4*2*dat00_0(cc,s,m)/6"; nl();
printf "␣␣end␣function␣datww0_s_0"; nl ();
nl ();
printf "␣%sfunction␣datww0_t_0␣(cc,m,k)␣result␣(atww0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_t␣=␣-4*g**4*5*(2*dat02_0(cc,s,m)␣+␣3*dat12_0(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_0(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_t_0"; nl ();
nl ();
printf "␣%sfunction␣datww0_u_0␣(cc,m,k)␣result␣(atww0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_u␣=␣-4*g**4*5*(2*dat02_0(cc,s,m)␣-␣3*dat12_0(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_0(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_u_0"; nl ();
nl ();
printf "␣%sfunction␣datww2_s_0␣(cc,m,k)␣result␣(atww2_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_s␣=␣0"; nl();
printf "␣␣end␣function␣datww2_s_0"; nl ();
nl ();
printf "␣%sfunction␣datww2_t_0␣(cc,m,k)␣result␣(atww2_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_t␣=␣-4*g**4*5*dat22_0(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_t_0"; nl ();
nl ();
printf "␣%sfunction␣datww2_u_0␣(cc,m,k)␣result␣(atww2_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_u␣=␣-4*g**4*5*dat22_0(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_u_0"; nl ();
nl ();
printf "␣%sfunction␣datz4_s_0␣(cc,m,k)␣result␣(atz4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_s␣=␣-4*g**4*costhw**4*dat00_0(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datz4_s_0"; nl ();
nl ();
printf "␣%sfunction␣datz4_t_0␣(cc,m,k)␣result␣(atz4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_t␣=␣-4*g**4*costhw**4*5*(dat02_0(cc,s,m)␣+␣2*dat22_0(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_t_0"; nl ();
nl ();
printf "␣%sfunction␣datz4_u_0␣(cc,m,k)␣result␣(atz4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_u␣=␣-4*g**4*costhw**4*5*(dat02_0(cc,s,m)␣+␣2*dat22_0(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_u_0"; nl ();
nl ();
printf "␣%sfunction␣data4_s_0␣(cc,m,k)␣result␣(ata4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_s␣=␣datz4_s_0␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_s_0"; nl ();
nl ();
printf "␣%sfunction␣data4_t_0␣(cc,m,k)␣result␣(ata4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_t␣=␣datz4_t_0␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_t_0"; nl ();
nl ();
printf "␣%sfunction␣data4_u_0␣(cc,m,k)␣result␣(ata4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_u␣=␣datz4_u_0␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_u_0"; nl ();
```

```
nl ();
printf "␣%sfunction␣dataw0_s_0␣(cc,m,k)␣result␣(ataw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_s␣=␣datzz0_s_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_s_0"; nl ();
nl ();
printf "␣%sfunction␣dataw0_t_0␣(cc,m,k)␣result␣(ataw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_t␣=␣datzz0_t_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_t_0"; nl ();
nl ();
printf "␣%sfunction␣dataw0_u_0␣(cc,m,k)␣result␣(ataw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_u␣=␣datzz0_u_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_u_0"; nl ();
nl ();
printf "␣%sfunction␣dataw1_s_0␣(cc,m,k)␣result␣(ataw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_s␣=␣datzz1_s_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_s_0"; nl ();
nl ();
printf "␣%sfunction␣dataw1_t_0␣(cc,m,k)␣result␣(ataw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_t␣=␣datzz1_t_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_t_0"; nl ();
nl ();
printf "␣%sfunction␣dataw1_u_0␣(cc,m,k)␣result␣(ataw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_u␣=␣datzz1_u_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_u_0"; nl ();
nl ();
```

```
printf "␣%sfunction␣dataz_s_0␣(cc,m,k)␣result␣(ataz_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_s␣=␣datz4_s_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_s_0"; nl ();
nl ();
printf "␣%sfunction␣dataz_t_0␣(cc,m,k)␣result␣(ataz_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_t␣=␣datz4_t_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_t_0"; nl ();
nl ();
printf "␣%sfunction␣dataz_u_0␣(cc,m,k)␣result␣(ataz_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_u␣=␣datz4_u_0(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_u_0"; nl ();
nl ();
printf "␣%sfunction␣datazw0_s_0␣(cc,m,k)␣result␣(atazw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_s␣=␣datzz0_s_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_s_0"; nl ();
nl ();
printf "␣%sfunction␣datazw0_t_0␣(cc,m,k)␣result␣(atazw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_t␣=␣datzz0_t_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_t_0"; nl ();
nl ();
printf "␣%sfunction␣datazw0_u_0␣(cc,m,k)␣result␣(atazw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_u␣=␣datzz0_u_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_u_0"; nl ();
nl ();
printf "␣%sfunction␣datazw1_s_0␣(cc,m,k)␣result␣(atazw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_s␣=␣datzz1_s_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_s_0"; nl ();
nl ();
printf "␣%sfunction␣datazw1_t_0␣(cc,m,k)␣result␣(atazw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_t␣=␣datzz1_t_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_t_0"; nl ();
nl ();
printf "␣%sfunction␣datazw1_u_0␣(cc,m,k)␣result␣(atazw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_u␣=␣datzz1_u_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_u_0"; nl ();
nl ();
printf "␣%sfunction␣dat3az_s_0␣(cc,m,k)␣result␣(at3az_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_s␣=␣datz4_s_0(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_s_0"; nl ();
nl ();
printf "␣%sfunction␣dat3az_t_0␣(cc,m,k)␣result␣(at3az_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_t␣=␣datz4_t_0(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_t_0"; nl ();
nl ();
printf "␣%sfunction␣dat3az_u_0␣(cc,m,k)␣result␣(at3az_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_u␣=␣datz4_u_0(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_u_0"; nl ();
nl ();
printf "␣%sfunction␣data3z_s_0␣(cc,m,k)␣result␣(ata3z_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_s␣=␣datz4_s_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_s_0"; nl ();
nl ();
printf "␣%sfunction␣data3z_t_0␣(cc,m,k)␣result␣(ata3z_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_t␣=␣datz4_t_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_t_0"; nl ();
nl ();
printf "␣%sfunction␣data3z_u_0␣(cc,m,k)␣result␣(ata3z_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_u␣=␣datz4_u_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_u_0"; nl ();
nl ();
printf "␣%sfunction␣datzz0_s_1␣(cc,m,k)␣result␣(atzz0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_s␣=␣-4*g**4*costhw**2*dat00_1(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datzz0_s_1"; nl ();
nl ();
printf "␣%sfunction␣datzz0_t_1␣(cc,m,k)␣result␣(atzz0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_t␣=␣-4*g**4*costhw**2*5*(dat02_1(cc,s,m)␣-␣dat22_1(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_t_1"; nl ();
nl ();
printf "␣%sfunction␣datzz0_u_1␣(cc,m,k)␣result␣(atzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_u␣=␣-4*g**4*costhw**2*5*(dat02_1(cc,s,m)␣-␣dat22_1(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_u_1"; nl ();
nl ();
printf "␣%sfunction␣datzz1_s_1␣(cc,m,k)␣result␣(atzz1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_s␣=␣0"; nl();
printf "␣␣end␣function␣datzz1_s_1"; nl ();
nl ();
printf "␣%sfunction␣datzz1_t_1␣(cc,m,k)␣result␣(atzz1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_t␣=␣-4*g**4*costhw**2*5*(dat12_1(cc,s,m)␣+␣dat22_1(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_t_1"; nl ();
nl ();
printf "␣%sfunction␣datzz1_u_1␣(cc,m,k)␣result␣(atzz1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_u␣=␣-4*g**4*costhw**2*5*(-dat12_1(cc,s,m)␣+␣dat22_1(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_u_1"; nl ();
nl ();
printf "␣%sfunction␣datww0_s_1␣(cc,m,k)␣result␣(atww0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_s␣=␣-4*g**4*2*dat00_1(cc,s,m)/6"; nl();
printf "␣␣end␣function␣datww0_s_1"; nl ();
nl ();
printf "␣%sfunction␣datww0_t_1␣(cc,m,k)␣result␣(atww0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_t␣=␣-4*g**4*5*(2*dat02_1(cc,s,m)␣+␣3*dat12_1(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_1(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_t_1"; nl ();
nl ();
printf "␣%sfunction␣datww0_u_1␣(cc,m,k)␣result␣(atww0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_u␣=␣-4*g**4*5*(2*dat02_1(cc,s,m)␣-␣3*dat12_1(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_1(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_u_1"; nl ();
nl ();
printf "␣%sfunction␣datww2_s_1␣(cc,m,k)␣result␣(atww2_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_s␣=␣0"; nl();
printf "␣␣end␣function␣datww2_s_1"; nl ();
nl ();
printf "␣%sfunction␣datww2_t_1␣(cc,m,k)␣result␣(atww2_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_t␣=␣-4*g**4*5*dat22_1(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_t_1"; nl ();
nl ();
printf "␣%sfunction␣datww2_u_1␣(cc,m,k)␣result␣(atww2_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_u␣=␣-4*g**4*5*dat22_1(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_u_1"; nl ();
nl ();
printf "␣%sfunction␣datz4_s_1␣(cc,m,k)␣result␣(atz4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_s␣=␣-4*g**4*costhw**4*(dat00_1(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_s_1"; nl ();
nl ();
printf "␣%sfunction␣datz4_t_1␣(cc,m,k)␣result␣(atz4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_t␣=␣-4*g**4*costhw**4*5*(dat02_1(cc,s,m)␣+␣2*dat22_1(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_t_1"; nl ();
nl ();
printf "␣%sfunction␣datz4_u_1␣(cc,m,k)␣result␣(atz4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_u␣=␣-4*g**4*costhw**4*5*(dat02_1(cc,s,m)␣+␣2*dat22_1(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_u_1"; nl ();
nl ();
printf "␣%sfunction␣data4_s_1␣(cc,m,k)␣result␣(ata4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_s␣=␣datz4_s_1␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_s_1"; nl ();
nl ();
printf "␣%sfunction␣data4_t_1␣(cc,m,k)␣result␣(ata4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_t␣=␣datz4_t_1␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_t_1"; nl ();
nl ();
printf "␣%sfunction␣data4_u_1␣(cc,m,k)␣result␣(ata4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_u␣=␣datz4_u_1␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_u_1"; nl ();
nl ();
printf "␣%sfunction␣dataw0_s_1␣(cc,m,k)␣result␣(ataw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_s␣=␣datzz0_s_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_s_1"; nl ();
nl ();
printf "␣%sfunction␣dataw0_t_1␣(cc,m,k)␣result␣(ataw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_t␣=␣datzz0_t_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_t_1"; nl ();
nl ();
printf "␣%sfunction␣dataw0_u_1␣(cc,m,k)␣result␣(ataw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_u␣=␣datzz0_u_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_u_1"; nl ();
nl ();
printf "␣%sfunction␣dataw1_s_1␣(cc,m,k)␣result␣(ataw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_s␣=␣datzz1_s_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_s_1"; nl ();
nl ();
printf "␣%sfunction␣dataw1_t_1␣(cc,m,k)␣result␣(ataw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_t␣=␣datzz1_t_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_t_1"; nl ();
nl ();
printf "␣%sfunction␣dataw1_u_1␣(cc,m,k)␣result␣(ataw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_u␣=␣datzz1_u_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_u_1"; nl ();
nl ();
printf "␣%sfunction␣dataz_s_1␣(cc,m,k)␣result␣(ataz_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_s␣=␣datz4_s_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_s_1"; nl ();
nl ();
printf "␣%sfunction␣dataz_t_1␣(cc,m,k)␣result␣(ataz_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_t␣=␣datz4_t_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_t_1"; nl ();
nl ();
printf "␣%sfunction␣dataz_u_1␣(cc,m,k)␣result␣(ataz_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_u␣=␣datz4_u_1(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_u_1"; nl ();
nl ();
printf "␣%sfunction␣datazw0_s_1␣(cc,m,k)␣result␣(atazw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_s␣=␣datzz0_s_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_s_1"; nl ();
nl ();
printf "␣%sfunction␣datazw0_t_1␣(cc,m,k)␣result␣(atazw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_t␣=␣datzz0_t_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_t_1"; nl ();
nl ();
printf "␣%sfunction␣datazw0_u_1␣(cc,m,k)␣result␣(atazw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_u␣=␣datzz1_u_0(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_u_1"; nl ();
nl ();
printf "␣%sfunction␣datazw1_s_1␣(cc,m,k)␣result␣(atazw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_s␣=␣datzz1_s_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_s_1"; nl ();
nl ();
printf "␣%sfunction␣datazw1_t_1␣(cc,m,k)␣result␣(atazw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_t␣=␣datzz1_t_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_t_1"; nl ();
nl ();
printf "␣%sfunction␣datazw1_u_1␣(cc,m,k)␣result␣(atazw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_u␣=␣datzz1_u_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_u_1"; nl ();
nl ();
printf "␣%sfunction␣dat3az_s_1␣(cc,m,k)␣result␣(at3az_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_s␣=␣datz4_s_1(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_s_1"; nl ();
nl ();
printf "␣%sfunction␣dat3az_t_1␣(cc,m,k)␣result␣(at3az_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_t␣=␣datz4_t_1(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_t_1"; nl ();
nl ();
printf "␣%sfunction␣dat3az_u_1␣(cc,m,k)␣result␣(at3az_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_u␣=␣datz4_u_1(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_u_1"; nl ();
nl ();
printf "␣%sfunction␣data3z_s_1␣(cc,m,k)␣result␣(ata3z_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_s␣=␣datz4_s_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_s_1"; nl ();
nl ();
printf "␣%sfunction␣data3z_t_1␣(cc,m,k)␣result␣(ata3z_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_t␣=␣datz4_t_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_t_1"; nl ();
nl ();
printf "␣%sfunction␣data3z_u_1␣(cc,m,k)␣result␣(ata3z_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_u␣=␣datz4_u_1(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_u_1"; nl ();
nl ();
printf "␣%sfunction␣datzz0_s_2␣(cc,m,k)␣result␣(atzz0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_s␣=␣-2*g**4*costhw**2*dat00_2(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datzz0_s_2"; nl ();
nl ();
printf "␣%sfunction␣datzz0_t_2␣(cc,m,k)␣result␣(atzz0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_t␣=␣-2*g**4*costhw**2*5*(dat02_2(cc,s,m)␣-␣dat22_2(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_t_2"; nl ();
nl ();
printf "␣%sfunction␣datzz0_u_2␣(cc,m,k)␣result␣(atzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_u␣=␣-2*g**4*costhw**2*5*(dat02_2(cc,s,m)␣-␣dat22_2(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_u_2"; nl ();
nl ();
printf "␣%sfunction␣datzz1_s_2␣(cc,m,k)␣result␣(atzz1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_s␣=␣0"; nl();
printf "␣␣end␣function␣datzz1_s_2"; nl ();
nl ();
printf "␣%sfunction␣datzz1_t_2␣(cc,m,k)␣result␣(atzz1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_t␣=␣-2*g**4*costhw**2*(3*dat11_2(cc,s,m)␣+␣5*dat22_2(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_t_2"; nl ();
nl ();
printf "␣%sfunction␣datzz1_u_2␣(cc,m,k)␣result␣(atzz1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_u␣=␣-2*g**4*costhw**2*(-3*dat11_2(cc,s,m)␣+␣5*dat22_2(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_u_2"; nl ();
nl ();
printf "␣%sfunction␣datww0_s_2␣(cc,m,k)␣result␣(atww0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_s␣=␣-2*g**4*dat00_2(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datww0_s_2"; nl ();
nl ();
printf "␣%sfunction␣datww0_t_2␣(cc,m,k)␣result␣(atww0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_t␣=␣-2*g**4*(9*dat11_2(cc,s,m)␣+␣10*dat02_2(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣5*dat22_2(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_t_2"; nl ();
nl ();
printf "␣%sfunction␣datww0_u_2␣(cc,m,k)␣result␣(atww0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_u␣=␣-2*g**4*(-9*dat11_2(cc,s,m)␣+␣10*dat02_2(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣5*dat22_2(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_u_2"; nl ();
nl ();
printf "␣%sfunction␣datww2_s_2␣(cc,m,k)␣result␣(atww2_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_s␣=␣0"; nl();
printf "␣␣end␣function␣datww2_s_2"; nl ();
nl ();
printf "␣%sfunction␣datww2_t_2␣(cc,m,k)␣result␣(atww2_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_t␣=␣-2*g**4*5*dat22_2(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_t_2"; nl ();
nl ();
printf "␣%sfunction␣datww2_u_2␣(cc,m,k)␣result␣(atww2_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_u␣=␣-2*g**4*5*dat22_2(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_u_2"; nl ();
nl ();
printf "␣%sfunction␣datz4_s_2␣(cc,m,k)␣result␣(atz4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_s␣=␣-2*g**4*costhw**4*dat00_2(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datz4_s_2"; nl ();
nl ();
printf "␣%sfunction␣datz4_t_2␣(cc,m,k)␣result␣(atz4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_t␣=␣-2*g**4*costhw**4*5*(dat02_2(cc,s,m)␣+␣2*dat22_2(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_t_2"; nl ();
nl ();
printf "␣%sfunction␣datz4_u_2␣(cc,m,k)␣result␣(atz4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_u␣=␣-2*g**4*costhw**4*5*(dat02_2(cc,s,m)␣+␣2*dat22_2(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_u_2"; nl ();
nl ();
printf "␣%sfunction␣data4_s_2␣(cc,m,k)␣result␣(ata4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_s␣=␣datz4_s_2␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_s_2"; nl ();
nl ();
printf "␣%sfunction␣data4_t_2␣(cc,m,k)␣result␣(ata4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_t␣=␣datz4_t_2␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_t_2"; nl ();
nl ();
printf "␣%sfunction␣data4_u_2␣(cc,m,k)␣result␣(ata4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_u␣=␣datz4_u_2␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_u_2"; nl ();
nl ();
printf "␣%sfunction␣dataw0_s_2␣(cc,m,k)␣result␣(ataw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_s␣=␣datzz0_s_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_s_2"; nl ();
nl ();
printf "␣%sfunction␣dataw0_t_2␣(cc,m,k)␣result␣(ataw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_t␣=␣datzz0_t_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_t_2"; nl ();
nl ();
printf "␣%sfunction␣dataw0_u_2␣(cc,m,k)␣result␣(ataw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_u␣=␣datzz0_u_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_u_2"; nl ();
nl ();
printf "␣%sfunction␣dataw1_s_2␣(cc,m,k)␣result␣(ataw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_s␣=␣datzz1_s_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_s_2"; nl ();
nl ();
printf "␣%sfunction␣dataw1_t_2␣(cc,m,k)␣result␣(ataw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_t␣=␣datzz1_t_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_t_2"; nl ();
nl ();
printf "␣%sfunction␣dataw1_u_2␣(cc,m,k)␣result␣(ataw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_u␣=␣datzz1_u_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_u_2"; nl ();
nl ();
printf "␣%sfunction␣dataz_s_2␣(cc,m,k)␣result␣(ataz_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_s␣=␣datz4_s_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_s_2"; nl ();
```

```
nl ();
printf "␣%sfunction␣dataz_t_2␣(cc,m,k)␣result␣(ataz_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_t␣=␣datz4_t_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_t_2"; nl ();
nl ();
printf "␣%sfunction␣dataz_u_2␣(cc,m,k)␣result␣(ataz_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_u␣=␣datz4_u_2(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_u_2"; nl ();
nl ();
printf "␣%sfunction␣datazw0_s_2␣(cc,m,k)␣result␣(atazw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_s␣=␣datzz0_s_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_s_2"; nl ();
nl ();
printf "␣%sfunction␣datazw0_t_2␣(cc,m,k)␣result␣(atazw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_t␣=␣datzz0_t_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_t_2"; nl ();
nl ();
printf "␣%sfunction␣datazw0_u_2␣(cc,m,k)␣result␣(atazw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_u␣=␣datzz0_u_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_u_2"; nl ();
nl ();
printf "␣%sfunction␣datazw1_s_2␣(cc,m,k)␣result␣(atazw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_s␣=␣datzz1_s_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_s_2"; nl ();
nl ();
```

```
printf "␣%sfunction␣datazw1_t_2␣(cc,m,k)␣result␣(atazw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_t␣=␣datzz1_t_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_t_2"; nl ();
nl ();
printf "␣%sfunction␣datazw1_u_2␣(cc,m,k)␣result␣(atazw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_u␣=␣datzz1_u_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_u_2"; nl ();
nl ();
printf "␣%sfunction␣dat3az_s_2␣(cc,m,k)␣result␣(at3az_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_s␣=␣datz4_s_2(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_s_2"; nl ();
nl ();
printf "␣%sfunction␣dat3az_t_2␣(cc,m,k)␣result␣(at3az_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_t␣=␣datz4_t_2(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_t_2"; nl ();
nl ();
printf "␣%sfunction␣dat3az_u_2␣(cc,m,k)␣result␣(at3az_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_u␣=␣datz4_u_2(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_u_2"; nl ();
nl ();
printf "␣%sfunction␣data3z_s_2␣(cc,m,k)␣result␣(ata3z_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_s␣=␣datz4_s_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_s_2"; nl ();
nl ();
printf "␣%sfunction␣data3z_t_2␣(cc,m,k)␣result␣(ata3z_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_t␣=␣datz4_t_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_t_2"; nl ();
nl ();
printf "␣%sfunction␣data3z_u_2␣(cc,m,k)␣result␣(ata3z_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_u␣=␣datz4_u_2(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_u_2"; nl ();
nl ();
printf "␣%sfunction␣datzz0_s_rsi␣(cc,m,k)␣result␣(atzz0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_s␣=␣-4*g**4*costhw**2*dat00_rsi(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datzz0_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datzz0_t_rsi␣(cc,m,k)␣result␣(atzz0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_t␣=␣-4*g**4*costhw**2*5*(dat02_rsi(cc,s,m)␣-␣dat22_rsi(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datzz0_u_rsi␣(cc,m,k)␣result␣(atzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz0_u␣=␣-4*g**4*costhw**2*5*(dat02_rsi(cc,s,m)␣-␣dat22_rsi(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datzz0_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datzz1_s_rsi␣(cc,m,k)␣result␣(atzz1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_s␣=␣0"; nl();
printf "␣␣end␣function␣datzz1_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datzz1_t_rsi␣(cc,m,k)␣result␣(atzz1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_t␣=␣-4*g**4*costhw**2*5*(dat12_rsi(cc,s,m)␣+␣dat22_rsi(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datzz1_u_rsi␣(cc,m,k)␣result␣(atzz1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atzz1_u␣=␣-4*g**4*costhw**2*5*(-dat12_rsi(cc,s,m)␣+␣dat22_rsi(cc,s,m))/2"; nl();
printf "␣␣end␣function␣datzz1_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww0_s_rsi␣(cc,m,k)␣result␣(atww0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_s␣=␣-4*g**4*2*dat00_rsi(cc,s,m)/6"; nl();
printf "␣␣end␣function␣datww0_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww0_t_rsi␣(cc,m,k)␣result␣(atww0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_t␣=␣-4*g**4*5*(2*dat02_rsi(cc,s,m)␣+␣3*dat12_rsi(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_rsi(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww0_u_rsi␣(cc,m,k)␣result␣(atww0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww0_u␣=␣-4*g**4*5*(2*dat02_rsi(cc,s,m)␣-␣3*dat12_rsi(cc,s,m)␣&"; nl();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dat22_rsi(cc,s,m))/6"; nl ();
printf "␣␣end␣function␣datww0_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww2_s_rsi␣(cc,m,k)␣result␣(atww2_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_s␣=␣0"; nl();
printf "␣␣end␣function␣datww2_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww2_t_rsi␣(cc,m,k)␣result␣(atww2_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_t␣=␣-4*g**4*5*dat22_rsi(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datww2_u_rsi␣(cc,m,k)␣result␣(atww2_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atww2_u␣=␣-4*g**4*5*dat22_rsi(cc,s,m)"; nl();
printf "␣␣end␣function␣datww2_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datz4_s_rsi␣(cc,m,k)␣result␣(atz4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_s␣=␣-4*g**4*costhw**4*dat00_rsi(cc,s,m)/3"; nl();
printf "␣␣end␣function␣datz4_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datz4_t_rsi␣(cc,m,k)␣result␣(atz4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_t␣=␣-4*g**4*costhw**4*5*(dat02_rsi(cc,s,m)␣+␣2*dat22_rsi(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datz4_u_rsi␣(cc,m,k)␣result␣(atz4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atz4_u␣=␣-4*g**4*costhw**4*5*(dat02_rsi(cc,s,m)␣+␣2*dat22_rsi(cc,s,m))/3"; nl();
printf "␣␣end␣function␣datz4_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣data4_s_rsi␣(cc,m,k)␣result␣(ata4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_s␣=␣datz4_s_rsi␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣data4_t_rsi␣(cc,m,k)␣result␣(ata4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_t␣=␣datz4_t_rsi␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣data4_u_rsi␣(cc,m,k)␣result␣(ata4_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata4_u␣=␣datz4_u_rsi␣(cc,m,k)␣/␣costhw**4␣*␣sinthw**4"; nl();
printf "␣␣end␣function␣data4_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw0_s_rsi␣(cc,m,k)␣result␣(ataw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_s␣=␣datzz0_s_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw0_t_rsi␣(cc,m,k)␣result␣(ataw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_t␣=␣datzz0_t_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw0_u_rsi␣(cc,m,k)␣result␣(ataw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw0_u␣=␣datzz0_u_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw0_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw1_s_rsi␣(cc,m,k)␣result␣(ataw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_s␣=␣datzz1_s_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw1_t_rsi␣(cc,m,k)␣result␣(ataw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_t␣=␣datzz1_t_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataw1_u_rsi␣(cc,m,k)␣result␣(ataw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataw1_u␣=␣datzz1_u_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataw1_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataz_s_rsi␣(cc,m,k)␣result␣(ataz_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_s␣=␣datz4_s_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataz_t_rsi␣(cc,m,k)␣result␣(ataz_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_t␣=␣datz4_t_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣dataz_u_rsi␣(cc,m,k)␣result␣(ataz_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ataz_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ataz_u␣=␣datz4_u_rsi(cc,m,k)␣/␣costhw**2␣*␣sinthw**2"; nl();
printf "␣␣end␣function␣dataz_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw0_s_rsi␣(cc,m,k)␣result␣(atazw0_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_s␣=␣datzz0_s_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw0_t_rsi␣(cc,m,k)␣result␣(atazw0_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_t␣=␣datzz0_t_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw0_u_rsi␣(cc,m,k)␣result␣(atazw0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw0_u␣=␣datzz0_u_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw0_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw1_s_rsi␣(cc,m,k)␣result␣(atazw1_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_s␣=␣datzz1_s_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw1_t_rsi␣(cc,m,k)␣result␣(atazw1_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_t␣=␣datzz1_t_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣datazw1_u_rsi␣(cc,m,k)␣result␣(atazw1_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣atazw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣atazw1_u␣=␣datzz1_u_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣datazw1_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣dat3az_s_rsi␣(cc,m,k)␣result␣(at3az_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_s␣=␣datz4_s_rsi(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣dat3az_t_rsi␣(cc,m,k)␣result␣(at3az_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_t␣=␣datz4_t_rsi(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣dat3az_u_rsi␣(cc,m,k)␣result␣(at3az_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣at3az_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣at3az_u␣=␣datz4_u_rsi(cc,m,k)␣/␣costhw**3␣*␣sinthw**3"; nl();
printf "␣␣end␣function␣dat3az_u_rsi"; nl ();
nl ();
printf "␣%sfunction␣data3z_s_rsi␣(cc,m,k)␣result␣(ata3z_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_s␣=␣datz4_s_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_s_rsi"; nl ();
nl ();
printf "␣%sfunction␣data3z_t_rsi␣(cc,m,k)␣result␣(ata3z_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_t␣=␣datz4_t_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_t_rsi"; nl ();
nl ();
printf "␣%sfunction␣data3z_u_rsi␣(cc,m,k)␣result␣(ata3z_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣ata3z_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣ata3z_u␣=␣datz4_u_rsi(cc,m,k)␣/␣costhw␣*␣sinthw"; nl();
printf "␣␣end␣function␣data3z_u_rsi"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_s_0␣(cc,m,k)␣result␣(amhw0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_s␣=␣4*cc(14)*dam00(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damhw0_s_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_t_0␣(cc,m,k)␣result␣(amhw0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_t␣=␣␣-␣4*cc(14)*(dam01(cc,s,m)␣-␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02(cc,s,m)/3␣+␣25*dam22(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_t_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_u_0␣(cc,m,k)␣result␣(amhw0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_u␣=␣-␣cc(14)*4*(dam01(cc,s,m)␣-␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02(cc,s,m)/3␣-␣25*dam22(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_u_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_s_0␣(cc,m,k)␣result␣(amhz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_s"; nl ();
printf "␣␣␣␣␣␣amhz0_s␣=␣damhw0_s_0(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_s_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_t_0␣(cc,m,k)␣result␣(amhz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_t"; nl ();
printf "␣␣␣␣␣␣amhz0_t␣=␣damhw0_t_0(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_t_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_u_0␣(cc,m,k)␣result␣(amhz0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_u"; nl ();
printf "␣␣␣␣␣␣amhz0_u␣=␣damhw0_u_0(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_u_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_s_0␣(cc,m,k)␣result␣(amhw1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damhw1_s_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_t_0␣(cc,m,k)␣result␣(amhw1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_t␣=␣-␣4*cc(14)*(3*dam11(cc,s,m)␣+␣3*dam21(cc,s,m)␣&"; nl ();
printf "                          -␣25*dam12(cc,s,m)␣-␣25*dam22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_t_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_u_0␣(cc,m,k)␣result␣(amhw1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_u␣=␣-␣4*cc(14)*(-3*dam11(cc,s,m)␣+␣3*dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12(cc,s,m)␣+␣25*dam22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_u_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_s_0␣(cc,m,k)␣result␣(amhz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_s"; nl ();
printf "␣␣␣␣␣␣amhz1_s␣=␣damhw1_s_0(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_s_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_t_0␣(cc,m,k)␣result␣(amhz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_t"; nl ();
printf "␣␣␣␣␣␣amhz1_t␣=␣damhw1_t_0(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_t_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_u_0␣(cc,m,k)␣result␣(amhz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_u"; nl ();
printf "␣␣␣␣␣␣amhz1_u␣=␣damhw1_u_0(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_u_0"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_s_1␣(cc,m,k)␣result␣(amhw0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_s␣=␣4*cc(14)*dam00_1(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damhw0_s_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_t_1␣(cc,m,k)␣result␣(amhw0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_t␣=␣␣-␣4*cc(14)*(dam01_1(cc,s,m)␣-␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_1(cc,s,m)/3␣+␣25*dam22_1(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_t_1"; nl ();
```

```
nl ();
printf "␣␣%sfunction␣damhw0_u_1␣(cc,m,k)␣result␣(amhw0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_u␣=␣-␣4*cc(14)*(dam01_1(cc,s,m)␣-␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02_1(cc,s,m)/3␣-␣25*dam22_1(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_u_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_s_1␣(cc,m,k)␣result␣(amhz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_s"; nl ();
printf "␣␣␣␣␣␣amhz0_s␣=␣damhw0_s_1(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_s_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_t_1␣(cc,m,k)␣result␣(amhz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_t"; nl ();
printf "␣␣␣␣␣␣amhz0_t␣=␣damhw0_t_1(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_t_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_u_1␣(cc,m,k)␣result␣(amhz0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_u"; nl ();
printf "␣␣␣␣␣␣amhz0_u␣=␣damhw0_u_1(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_u_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_s_1␣(cc,m,k)␣result␣(amhw1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damhw1_s_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_t_1␣(cc,m,k)␣result␣(amhw1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_t␣=␣-␣4*cc(14)*(3*dam11_1(cc,s,m)␣+␣3*dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)␣-␣25*dam22_1(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_t_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_u_1␣(cc,m,k)␣result␣(amhw1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_u␣=␣-␣4*cc(14)*(-3*dam11_1(cc,s,m)␣+␣3*dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)␣+␣25*dam22_1(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_u_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_s_1␣(cc,m,k)␣result␣(amhz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_s"; nl ();
printf "␣␣␣␣␣␣amhz1_s␣=␣damhw1_s_1(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_s_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_t_1␣(cc,m,k)␣result␣(amhz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_t"; nl ();
printf "␣␣␣␣␣␣amhz1_t␣=␣damhw1_t_1(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_t_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_u_1␣(cc,m,k)␣result␣(amhz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_u"; nl ();
printf "␣␣␣␣␣␣amhz1_u␣=␣damhw1_u_1(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_u_1"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_s_7␣(cc,m,k)␣result␣(amhw0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_s␣=␣4*cc(14)*dam00_7(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damhw0_s_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_t_7␣(cc,m,k)␣result␣(amhw0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_t␣=␣␣4*cc(14)*(dam01_7(cc,s,m)␣-␣dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_7(cc,s,m)/3␣+␣25*dam22_7(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_t_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhw0_u_7␣(cc,m,k)␣result␣(amhw0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw0_u␣=␣4*cc(14)*(dam01_7(cc,s,m)␣-␣dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02_7(cc,s,m)/3␣-␣25*dam22_7(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damhw0_u_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_s_7␣(cc,m,k)␣result␣(amhz0_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_s"; nl ();
printf "␣␣␣␣␣␣amhz0_s␣=␣damhw0_s_7(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_s_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_t_7␣(cc,m,k)␣result␣(amhz0_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_t"; nl ();
printf "␣␣␣␣␣␣amhz0_t␣=␣damhw0_t_7(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_t_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz0_u_7␣(cc,m,k)␣result␣(amhz0_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz0_u"; nl ();
printf "␣␣␣␣␣␣amhz0_u␣=␣damhw0_u_7(cc,m,k)␣*␣costhw**2"; nl();
printf "␣␣end␣function␣damhz0_u_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_s_7␣(cc,m,k)␣result␣(amhw1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damhw1_s_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_t_7␣(cc,m,k)␣result␣(amhw1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_t␣=␣4*cc(14)*(3*dam11_7(cc,s,m)␣+␣3*dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)␣-␣25*dam22_7(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_t_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhw1_u_7␣(cc,m,k)␣result␣(amhw1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhw1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amhw1_u␣=␣4*cc(14)*(-3*dam11_7(cc,s,m)␣+␣3*dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)␣+␣25*dam22_7(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damhw1_u_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_s_7␣(cc,m,k)␣result␣(amhz1_s)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_s"; nl ();
printf "␣␣␣␣␣␣amhz1_s␣=␣damhw1_s_7(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_s_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_t_7␣(cc,m,k)␣result␣(amhz1_t)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_t"; nl ();
printf "␣␣␣␣␣␣amhz1_t␣=␣damhw1_t_7(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_t_7"; nl ();
nl ();
printf "␣␣%sfunction␣damhz1_u_7␣(cc,m,k)␣result␣(amhz1_u)" pure; nl ();
printf "␣␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amhz1_u"; nl ();
printf "␣␣␣␣␣␣amhz1_u␣=␣damhw1_u_7(cc,m,k)␣*␣costhw**2"; nl ();
printf "␣␣end␣function␣damhz1_u_7"; nl ();
nl ();
printf "␣%sfunction␣damzz0_s_0␣(cc,m,k)␣result␣(amzz0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_s␣=␣4*mass(24)**2*dam00(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damzz0_s_0"; nl ();
nl ();
printf "␣%sfunction␣damzz0_t_0␣(cc,m,k)␣result␣(amzz0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_t␣=␣-4*mass(24)**2*(dam01(cc,s,m)␣-␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02(cc,s,m)/3␣+␣25*dam22(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_t_0"; nl ();
nl ();
printf "␣%sfunction␣damzz0_u_0␣(cc,m,k)␣result␣(amzz0_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_u␣=␣-4*mass(24)**2*(dam01(cc,s,m)␣-␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02(cc,s,m)/3␣-␣25*dam22(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_u_0"; nl ();
nl ();
printf "␣%sfunction␣damzz1_s_0␣(cc,m,k)␣result␣(amzz1_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damzz1_s_0"; nl ();
nl ();
printf "␣%sfunction␣damzz1_t_0␣(cc,m,k)␣result␣(amzz1_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_t␣=␣-4*mass(24)**2*(3*dam11(cc,s,m)␣+␣3*dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12(cc,s,m)␣-␣25*dam22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_t_0"; nl ();
nl ();
printf "␣%sfunction␣damzz1_u_0␣(cc,m,k)␣result␣(amzz1_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_u␣=␣-4*mass(24)**2*(-3*dam11(cc,s,m)␣+␣3*dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12(cc,s,m)␣+␣25*dam22(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_u_0"; nl ();
nl ();
printf "␣%sfunction␣damww0_s_0␣(cc,m,k)␣result␣(amww0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_s␣=␣4*mass(24)**2*dam00(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damww0_s_0"; nl ();
nl ();
printf "␣%sfunction␣damww0_t_0␣(cc,m,k)␣result␣(amww0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_t␣=␣-4*mass(24)**2*(dam01(cc,s,m)␣+␣3*dam11(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21(cc,s,m)/2␣-␣25*dam02(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12(cc,s,m)/2␣-␣25*dam22(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_t_0"; nl ();
nl ();
printf "␣%sfunction␣damww0_u_0␣(cc,m,k)␣result␣(amww0_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_u␣=␣-4*mass(24)**2*(dam01(cc,s,m)␣-␣3*dam11(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21(cc,s,m)/2␣+␣25*dam02(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12(cc,s,m)/2␣+␣25*dam22(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_u_0"; nl ();
nl ();
printf "␣%sfunction␣damww2_s_0␣(cc,m,k)␣result␣(amww2_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_s␣=␣0"; nl ();
printf "␣␣end␣function␣damww2_s_0"; nl ();
nl ();
printf "␣%sfunction␣damww2_t_0␣(cc,m,k)␣result␣(amww2_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_t␣=␣-4*mass(24)**2*(3*dam21(cc,s,m)␣-␣25*dam22(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_t_0"; nl ();
nl ();
printf "␣%sfunction␣damww2_u_0␣(cc,m,k)␣result␣(amww2_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_u␣=␣-4*mass(24)**2*(3*dam21(cc,s,m)␣-␣25*dam22(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_u_0"; nl ();
nl ();
printf "␣%sfunction␣damz4_s_0␣(cc,m,k)␣result␣(amz4_s)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_s␣=␣4*mass(24)**2*dam00(cc,s,m)/3"; nl();
printf "␣␣end␣function␣damz4_s_0"; nl ();
nl ();
printf "␣%sfunction␣damz4_t_0␣(cc,m,k)␣result␣(amz4_t)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_t␣=␣-4*mass(24)**2*(dam01(cc,s,m)/2␣+␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-25*dam02(cc,s,m)/6␣-␣25*dam22(cc,s,m)/3)"; nl ();
printf "␣␣end␣function␣damz4_t_0"; nl ();
nl ();
printf "␣%sfunction␣damz4_u_0␣(cc,m,k)␣result␣(amz4_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_u␣=␣-4*mass(24)**2*(dam01(cc,s,m)/2␣+␣dam21(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-25*dam02(cc,s,m)/6␣-␣25*dam22(cc,s,m)/3)"; nl ();
printf "␣␣end␣function␣damz4_u_0"; nl ();
nl ();
printf "␣%sfunction␣damzz0_s_1␣(cc,m,k)␣result␣(amzz0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_s␣=␣4*mass(24)**2*dam00_1(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damzz0_s_1"; nl ();
nl ();
printf "␣%sfunction␣damzz0_t_1␣(cc,m,k)␣result␣(amzz0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_t␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)␣-␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_1(cc,s,m)/3␣+␣25*dam22_1(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_t_1"; nl ();
nl ();
printf "␣%sfunction␣damzz0_u_1␣(cc,m,k)␣result␣(amzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_u␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)␣-␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02_1(cc,s,m)/3␣-␣25*dam22_1(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_u_1"; nl ();
nl ();
printf "␣%sfunction␣damzz1_s_1␣(cc,m,k)␣result␣(amzz1_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damzz1_s_1"; nl ();
nl ();
printf "␣%sfunction␣damzz1_t_1␣(cc,m,k)␣result␣(amzz1_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_t␣=␣-4*mass(24)**2*(3*dam11_1(cc,s,m)␣+␣3*dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)␣-␣25*dam22_1(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_t_1"; nl ();
nl ();
printf "␣%sfunction␣damzz1_u_1␣(cc,m,k)␣result␣(amzz1_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_u␣=␣-4*mass(24)**2*(-3*dam11_1(cc,s,m)␣+␣3*dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)␣+␣25*dam22_1(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_u_1"; nl ();
nl ();
printf "␣%sfunction␣damww0_s_1␣(cc,m,k)␣result␣(amww0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_s␣=␣4*mass(24)**2*dam00_1(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damww0_s_1"; nl ();
nl ();
printf "␣%sfunction␣damww0_t_1␣(cc,m,k)␣result␣(amww0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_t␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)␣+␣3*dam11_1(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21_1(cc,s,m)/2␣-␣25*dam02_1(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)/2␣-␣25*dam22_1(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_t_1"; nl ();
nl ();
printf "␣%sfunction␣damww0_u_1␣(cc,m,k)␣result␣(amww0_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_u␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)␣-␣3*dam11_1(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21_1(cc,s,m)/2␣+␣25*dam02_1(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_1(cc,s,m)/2␣+␣25*dam22_1(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_u_1"; nl ();
nl ();
printf "␣%sfunction␣damww2_s_1␣(cc,m,k)␣result␣(amww2_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_s␣=␣0"; nl ();
printf "␣␣end␣function␣damww2_s_1"; nl ();
nl ();
printf "␣%sfunction␣damww2_t_1␣(cc,m,k)␣result␣(amww2_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_t␣=␣-4*mass(24)**2*(3*dam21_1(cc,s,m)␣-␣25*dam22_1(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_t_1"; nl ();
nl ();
printf "␣%sfunction␣damww2_u_1␣(cc,m,k)␣result␣(amww2_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_u␣=␣-4*mass(24)**2*(3*dam21_1(cc,s,m)␣-␣25*dam22_1(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_u_1"; nl ();
nl ();
printf "␣%sfunction␣damz4_s_1␣(cc,m,k)␣result␣(amz4_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_s␣=␣4*mass(24)**2*dam00_1(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damz4_s_1"; nl ();
nl ();
printf "␣%sfunction␣damz4_t_1␣(cc,m,k)␣result␣(amz4_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_t␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)/2␣+␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_1(cc,s,m)/6␣-␣25*dam22_1(cc,s,m)/3)"; nl ();
printf "␣␣end␣function␣damz4_t_1"; nl ();
nl ();
printf "␣%sfunction␣damz4_u_1␣(cc,m,k)␣result␣(amz4_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amz4_u␣=␣-4*mass(24)**2*(dam01_1(cc,s,m)/2␣+␣dam21_1(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_1(cc,s,m)/6␣-␣25*dam22_1(cc,s,m)/3)"; nl ();
printf "␣␣end␣function␣damz4_u_1"; nl ();
nl ();
printf "␣%sfunction␣damzz0_s_7␣(cc,m,k)␣result␣(amzz0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_s␣=␣4*mass(24)**2*dam00_7(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damzz0_s_7"; nl ();
nl ();
printf "␣%sfunction␣damzz0_t_7␣(cc,m,k)␣result␣(amzz0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_t␣=␣4*mass(24)**2*(dam01_7(cc,s,m)␣-␣dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_7(cc,s,m)/3␣+␣25*dam22_7(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_t_7"; nl ();
nl ();
printf "␣%sfunction␣damzz0_u_7␣(cc,m,k)␣result␣(amzz0_u)" pure; nl();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz0_u␣=␣4*mass(24)**2*(dam01_7(cc,s,m)␣-␣dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣25*dam02_7(cc,s,m)/3␣-␣25*dam22_7(cc,s,m)/3)/6"; nl ();
printf "␣␣end␣function␣damzz0_u_7"; nl ();
nl ();
printf "␣%sfunction␣damzz1_s_7␣(cc,m,k)␣result␣(amzz1_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_s␣=␣0"; nl ();
printf "␣␣end␣function␣damzz1_s_7"; nl ();
nl ();
printf "␣%sfunction␣damzz1_t_7␣(cc,m,k)␣result␣(amzz1_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_t␣=␣4*mass(24)**2*(3*dam11_7(cc,s,m)␣+␣3*dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)␣-␣25*dam22_7(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_t_7"; nl ();
nl ();
printf "␣%sfunction␣damzz1_u_7␣(cc,m,k)␣result␣(amzz1_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amzz1_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amzz1_u␣=␣4*mass(24)**2*(-3*dam11_7(cc,s,m)␣+␣3*dam21_7(cc,s,m)␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)␣+␣25*dam22_7(cc,s,m))/4"; nl ();
printf "␣␣end␣function␣damzz1_u_7"; nl ();
nl ();
printf "␣%sfunction␣damww0_s_7␣(cc,m,k)␣result␣(amww0_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_s␣=␣4*mass(24)**2*dam00_7(cc,s,m)/3"; nl ();
printf "␣␣end␣function␣damww0_s_7"; nl ();
nl ();
```

```
printf "␣%sfunction␣damww0_t_7␣(cc,m,k)␣result␣(amww0_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_t␣=␣4*mass(24)**2*(dam01_7(cc,s,m)␣+␣3*dam11_7(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21_7(cc,s,m)/2␣-␣25*dam02_7(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)/2␣-␣25*dam22_7(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_t_7"; nl ();
nl ();
printf "␣%sfunction␣damww0_u_7␣(cc,m,k)␣result␣(amww0_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww0_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww0_u␣=␣4*mass(24)**2*(dam01_7(cc,s,m)␣-␣3*dam11_7(cc,s,m)/2␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+␣dam21_7(cc,s,m)/2␣+␣25*dam02_7(cc,s,m)/3␣&"; nl ();
printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam12_7(cc,s,m)/2␣+␣25*dam22_7(cc,s,m)/6)/2"; nl ();
printf "␣␣end␣function␣damww0_u_7"; nl ();
nl ();
printf "␣%sfunction␣damww2_s_7␣(cc,m,k)␣result␣(amww2_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_s␣=␣0"; nl ();
printf "␣␣end␣function␣damww2_s_7"; nl ();
nl ();
printf "␣%sfunction␣damww2_t_7␣(cc,m,k)␣result␣(amww2_t)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_t"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_t␣=␣4*mass(24)**2*(3*dam21_7(cc,s,m)␣-␣25*dam22_7(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_t_7"; nl ();
nl ();
printf "␣%sfunction␣damww2_u_7␣(cc,m,k)␣result␣(amww2_u)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amww2_u"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
printf "␣␣␣␣␣␣amww2_u␣=␣4*mass(24)**2*(3*dam21_7(cc,s,m)␣-␣25*dam22_7(cc,s,m))/2"; nl ();
printf "␣␣end␣function␣damww2_u_7"; nl ();
nl ();
printf "␣%sfunction␣damz4_s_7␣(cc,m,k)␣result␣(amz4_s)" pure; nl ();
printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_s"; nl ();
printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
```

```
      printf "␣␣␣␣␣␣amz4_s␣=␣4*mass(24)**2*dam00_7(cc,s,m)/3"; nl ();
      printf "␣␣end␣function␣damz4_s_7"; nl ();
      nl ();
      printf "␣%sfunction␣damz4_t_7␣(cc,m,k)␣result␣(amz4_t)" pure; nl ();
      printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
      printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
      printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_t"; nl ();
      printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
      printf "␣␣␣␣␣␣amz4_t␣=␣4*mass(24)**2*(dam01_7(cc,s,m)/2␣+␣dam21_7(cc,s,m)␣&"; nl ();
      printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_7(cc,s,m)/6␣-␣25*dam22_7(cc,s,m)/3)"; nl ();
      printf "␣␣end␣function␣damz4_t_7"; nl ();
      nl ();
      printf "␣%sfunction␣damz4_u_7␣(cc,m,k)␣result␣(amz4_u)" pure; nl ();
      printf "␣␣␣␣␣type(momentum),␣intent(in)␣::␣k"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:14),␣intent(in)␣::␣cc"; nl ();
      printf "␣␣␣␣␣␣real(kind=default),␣dimension(1:5),␣intent(in)␣::␣m"; nl ();
      printf "␣␣␣␣␣␣real(kind=default)␣::␣s"; nl ();
      printf "␣␣␣␣␣␣complex(kind=default)␣::␣amz4_u"; nl ();
      printf "␣␣␣␣␣␣s␣=␣k*k"; nl ();
      printf "␣␣␣␣␣␣amz4_u␣=␣4*mass(24)**2*(dam01_7(cc,s,m)/2␣+␣dam21_7(cc,s,m)␣&"; nl ();
      printf "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣25*dam02_7(cc,s,m)/6␣-␣25*dam22_7(cc,s,m)/3)"; nl ();
      printf "␣␣end␣function␣damz4_u_7"; nl ();
      nl ();
  end
```

20.13 Interface of *Target_VM*

```
module Make : Target.Maker
```

20.14 Implementation of *Target_VM*

20.14.1 O'Mega Virtual Machine with `Fortran 90/95`

```
module Make (Fusion_Maker : Fusion.Maker) (P : Momentum.T) (M : Model.T) =
  struct

    open Coupling
    open Format

    module CM = Colorize.It(M)
    module SCM = Orders.Slice(Colorize.It(M))
    module F = Fusion_Maker(P)(M)
    module CF = Fusion.Multi(Fusion_Maker)(P)(M)
    module CFlow = Color.Flow
    type amplitudes = CF.amplitudes
```

Options.

```
    type diagnostic = All | Arguments | Momenta | Gauge

    let wrapper_module = ref "ovm_wrapper"
    let parameter_module_external = ref "some_external_module_with_model_info"
    let bytecode_file = ref "bytecode.hbc"
    let md5sum = ref None
    let openmp = ref false
    let kind = ref "default"
    let whizard = ref false

    let options = Options.create
      [ "wrapper_module", Arg.String (fun s → wrapper_module := s),
        "name␣name␣of␣wrapper␣module";
        "bytecode_file", Arg.String (fun s → bytecode_file := s),
        "name␣bytecode␣file␣to␣be␣used␣in␣wrapper";
        "parameter_module_external", Arg.String (fun s →
                                        parameter_module_external := s),
        "name␣external␣parameter␣module␣to␣be␣used␣in␣wrapper";
        "md5sum", Arg.String (fun s → md5sum := Some s),
        "checksum␣transfer␣MD5␣checksum␣in␣wrapper";
        "whizard", Arg.Set whizard, "␣include␣WHIZARD␣interface␣in␣wrapper";
        "openmp", Arg.Set openmp,
        "activate␣parallel␣computation␣of␣amplitude␣with␣OpenMP"]
```

Integers encode the opcodes (operation codes).

```
    let ovm_ADD_MOMENTA = 1
    let ovm_CALC_BRAKET = 2

    let ovm_LOAD_SCALAR = 10
    let ovm_LOAD_SPINOR_INC = 11
    let ovm_LOAD_SPINOR_OUT = 12
    let ovm_LOAD_CONJSPINOR_INC = 13
    let ovm_LOAD_CONJSPINOR_OUT = 14
    let ovm_LOAD_MAJORANA_INC = 15
    let ovm_LOAD_MAJORANA_OUT = 16
    let ovm_LOAD_VECTOR_INC = 17
    let ovm_LOAD_VECTOR_OUT = 18
    let ovm_LOAD_VECTORSPINOR_INC = 19
    let ovm_LOAD_VECTORSPINOR_OUT = 20
    let ovm_LOAD_TENSOR2_INC = 21
    let ovm_LOAD_TENSOR2_OUT = 22
    let ovm_LOAD_BRS_SCALAR = 30
    let ovm_LOAD_BRS_SPINOR_INC = 31
    let ovm_LOAD_BRS_SPINOR_OUT = 32
    let ovm_LOAD_BRS_CONJSPINOR_INC = 33
    let ovm_LOAD_BRS_CONJSPINOR_OUT = 34
    let ovm_LOAD_BRS_VECTOR_INC = 37
    let ovm_LOAD_BRS_VECTOR_OUT = 38
    let ovm_LOAD_MAJORANA_GHOST_INC = 23
    let ovm_LOAD_MAJORANA_GHOST_OUT = 24
    let ovm_LOAD_BRS_MAJORANA_INC = 35
    let ovm_LOAD_BRS_MAJORANA_OUT = 36

    let ovm_PROPAGATE_SCALAR = 51
    let ovm_PROPAGATE_COL_SCALAR = 52
    let ovm_PROPAGATE_GHOST = 53
    let ovm_PROPAGATE_SPINOR = 54
    let ovm_PROPAGATE_CONJSPINOR = 55
    let ovm_PROPAGATE_MAJORANA = 56
    let ovm_PROPAGATE_COL_MAJORANA = 57
    let ovm_PROPAGATE_UNITARITY = 58
    let ovm_PROPAGATE_COL_UNITARITY = 59
    let ovm_PROPAGATE_FEYNMAN = 60
    let ovm_PROPAGATE_COL_FEYNMAN = 61
    let ovm_PROPAGATE_VECTORSPINOR = 62
    let ovm_PROPAGATE_TENSOR2 = 63
```

ovm_PROPAGATE_NONE has to be split up to different types to work in conjunction with color MC ...

```
    let ovm_PROPAGATE_NONE = 64

    let ovm_FUSE_V_FF = − 1
    let ovm_FUSE_F_VF = − 2
    let ovm_FUSE_F_FV = − 3
```

```
let ovm_FUSE_VA_FF = -4
let ovm_FUSE_F_VAF = -5
let ovm_FUSE_F_FVA = -6
let ovm_FUSE_VA2_FF = -7
let ovm_FUSE_F_VA2F = -8
let ovm_FUSE_F_FVA2 = -9
let ovm_FUSE_A_FF = -10
let ovm_FUSE_F_AF = -11
let ovm_FUSE_F_FA = -12
let ovm_FUSE_VL_FF = -13
let ovm_FUSE_F_VLF = -14
let ovm_FUSE_F_FVL = -15
let ovm_FUSE_VR_FF = -16
let ovm_FUSE_F_VRF = -17
let ovm_FUSE_F_FVR = -18
let ovm_FUSE_VLR_FF = -19
let ovm_FUSE_F_VLRF = -20
let ovm_FUSE_F_FVLR = -21
let ovm_FUSE_SP_FF = -22
let ovm_FUSE_F_SPF = -23
let ovm_FUSE_F_FSP = -24
let ovm_FUSE_S_FF = -25
let ovm_FUSE_F_SF = -26
let ovm_FUSE_F_FS = -27
let ovm_FUSE_P_FF = -28
let ovm_FUSE_F_PF = -29
let ovm_FUSE_F_FP = -30
let ovm_FUSE_SL_FF = -31
let ovm_FUSE_F_SLF = -32
let ovm_FUSE_F_FSL = -33
let ovm_FUSE_SR_FF = -34
let ovm_FUSE_F_SRF = -35
let ovm_FUSE_F_FSR = -36
let ovm_FUSE_SLR_FF = -37
let ovm_FUSE_F_SLRF = -38
let ovm_FUSE_F_FSLR = -39

let ovm_FUSE_G_GG = -40
let ovm_FUSE_V_SS = -41
let ovm_FUSE_S_VV = -42
let ovm_FUSE_S_VS = -43
let ovm_FUSE_V_SV = -44
let ovm_FUSE_S_SS = -45
let ovm_FUSE_S_SVV = -46
let ovm_FUSE_V_SSV = -47
let ovm_FUSE_S_SSS = -48
let ovm_FUSE_V_VVV = -49

let ovm_FUSE_S_G2 = -50
let ovm_FUSE_G_SG = -51
let ovm_FUSE_G_GS = -52
let ovm_FUSE_S_G2_SKEW = -53
let ovm_FUSE_G_SG_SKEW = -54
let ovm_FUSE_G_GS_SKEW = -55

let inst_length = 8
```

Some helper functions.

```
let printi ~lhs : l ~rhs1 : r1 ?coupl : (cp = 0) ?coeff : (co = 0)
            ?rhs2 : (r2 = 0) ?rhs3 : (r3 = 0) ?rhs4 : (r4 = 0) code =
  printf "@\n%d␣%d␣%d␣%d␣%d␣%d␣%d␣%d" code cp co l r1 r2 r3 r4

let nl () = printf "@\n"
let print_int_lst lst = nl (); lst |> List.iter (printf "%d   ")

let print_str_lst lst = nl (); lst |> List.iter (printf "%s ")

let break () = printi ~lhs:0 ~rhs1:0 0
```

Copied from below. Needed for header.

Could be fused with *lorentz_ordering*.

```
type declarations =
  { scalars : F.wf list;
    spinors : F.wf list;
    conjspinors : F.wf list;
    realspinors : F.wf list;
    ghostspinors : F.wf list;
    vectorspinors : F.wf list;
    vectors : F.wf list;
    ward_vectors : F.wf list;
    massive_vectors : F.wf list;
    tensors_1 : F.wf list;
    tensors_2 : F.wf list;
    brs_scalars : F.wf list;
    brs_spinors : F.wf list;
    brs_conjspinors : F.wf list;
    brs_realspinors : F.wf list;
    brs_vectorspinors : F.wf list;
    brs_vectors : F.wf list;
    brs_massive_vectors : F.wf list }

let rec classify_wfs' acc = function
  | [] → acc
  | wf :: rest →
      classify_wfs'
        (match SCM.lorentz (F.flavor wf) with
        | Scalar → {acc with scalars = wf :: acc.scalars}
        | Spinor → {acc with spinors = wf :: acc.spinors}
        | ConjSpinor → {acc with conjspinors = wf :: acc.conjspinors}
        | Majorana → {acc with realspinors = wf :: acc.realspinors}
        | Maj_Ghost → {acc with ghostspinors = wf :: acc.ghostspinors}
        | Vectorspinor →
            {acc with vectorspinors = wf :: acc.vectorspinors}
        | Vector → {acc with vectors = wf :: acc.vectors}
        | Massive_Vector →
            {acc with massive_vectors = wf :: acc.massive_vectors}
        | Tensor_1 → {acc with tensors_1 = wf :: acc.tensors_1}
        | Tensor_2 → {acc with tensors_2 = wf :: acc.tensors_2}
        | BRS Scalar → {acc with brs_scalars = wf :: acc.brs_scalars}
        | BRS Spinor → {acc with brs_spinors = wf :: acc.brs_spinors}
        | BRS ConjSpinor → {acc with brs_conjspinors =
                                  wf :: acc.brs_conjspinors}
        | BRS Majorana → {acc with brs_realspinors =
                                wf :: acc.brs_realspinors}
        | BRS Vectorspinor → {acc with brs_vectorspinors =
                                    wf :: acc.brs_vectorspinors}
        | BRS Vector → {acc with brs_vectors = wf :: acc.brs_vectors}
        | BRS Massive_Vector → {acc with brs_massive_vectors =
                                      wf :: acc.brs_massive_vectors}
        | BRS _ → invalid_arg "Targets.classify_wfs': not needed here")
        rest

let classify_wfs wfs = classify_wfs'
  { scalars = [];
    spinors = [];
    conjspinors = [];
    realspinors = [];
    ghostspinors = [];
    vectorspinors = [];
    vectors = [];
    ward_vectors = [];
    massive_vectors = [];
    tensors_1 = [];
    tensors_2 = [];
    brs_scalars = [];
    brs_spinors = [];
    brs_conjspinors = [];
    brs_realspinors = [];
    brs_vectorspinors = [];
    brs_vectors = [];
    brs_massive_vectors = [] } wfs
```

Sets and maps

The OVM identifies all objects via integers. Therefore, we need maps which assign the abstract object a unique ID.
I want *int list*s with less elements to come first. Used in conjunction with the int list representation of momenta, this will set the outer particles at first position and allows the OVM to set them without further instructions.

Using the Momentum module might give better performance than integer lists?

```
let rec int_lst_compare (e1 : int list) (e2 : int list) =
  match e1, e2 with
  | [], [] → 0
  | _, [] → +1
  | [], _ → −1
  | [_;_], [_] → +1
  | [_], [_;_] → −1
  | hd1 :: tl1, hd2 :: tl2 →
      let c = compare hd1 hd2 in
      if (c ≢ 0 ∧ List.length tl1 = List.length tl2) then
        c
      else
        int_lst_compare tl1 tl2
```

We need a canonical ordering for the different types of wfs. Copied, and slightly modified to order *wf*s, from `fusion.ml`.

```
let lorentz_ordering wf =
  match SCM.lorentz (F.flavor wf) with
  | Scalar → 0
  | Spinor → 1
  | ConjSpinor → 2
  | Majorana → 3
  | Vector → 4
  | Massive_Vector → 5
  | Tensor_2 → 6
  | Tensor_1 → 7
  | Vectorspinor → 8
  | BRS Scalar → 9
  | BRS Spinor → 10
  | BRS ConjSpinor → 11
  | BRS Majorana → 12
  | BRS Vector → 13
  | BRS Massive_Vector → 14
  | BRS Tensor_2 → 15
  | BRS Tensor_1 → 16
  |
  | BRS Vectorspinor → 17
  | Maj_Ghost → invalid_arg "lorentz_ordering:␣not␣implemented"
  | BRS _ → invalid_arg "lorentz_ordering:␣not␣needed"

let wf_compare (wf1, mult1) (wf2, mult2) =
  let c1 = compare (lorentz_ordering wf1) (lorentz_ordering wf2) in
  if c1 ≠ 0 then
  c1
  else
  let c2 = compare wf1 wf2 in
  if c2 ≠ 0 then
    c2
  else
    compare mult1 mult2

let amp_compare amp1 amp2 =
  let cflow a = SCM.flow (F.incoming a) (F.outgoing a) in
  let c1 = compare (cflow amp1) (cflow amp2) in
  if c1 ≠ 0 then
  c1
  else
  let process_sans_color a =
    (List.map SCM.flavor_sans_color (F.incoming a),
     List.map SCM.flavor_sans_color (F.outgoing a)) in
  compare (process_sans_color amp1) (process_sans_color amp2)

let level_compare (f1, amp1) (f2, amp2) =
  let p1 = F.momentum_list (F.lhs f1)
  and p2 = F.momentum_list (F.lhs f2) in
  let c1 = int_lst_compare p1 p2 in
  if c1 ≠ 0 then
  c1
  else
  let c2 = compare f1 f2 in
  if c2 ≠ 0 then
    c2
  else
    amp_compare amp1 amp2

module ISet = Set.Make (struct type t = int list
                              let compare = int_lst_compare end)

module WFSet = Set.Make (struct type t = CF.wf × int
                               let compare = wf_compare end)

module CSet = Set.Make (struct type t = CM.constant
                              let compare = compare end)

module FSet = Set.Make (struct type t = F.fusion × F.amplitude
                              let compare = level_compare end)
```

It might be preferable to use a *PMap* which maps mom to int, instead of this way. More standard functions like *mem* could be used. Also, *get_ID* would be faster, $\mathcal{O}(\log N)$ instead of $\mathcal{O}(N)$, and simpler. For 8 gluons: N=127 momenta. Minor performance issue.

```
  module IMap = Map.Make(Int)
```

For *wf*s it is crucial for the performance to use a different type of *Map*s.

```
  module WFMap = Map.Make (struct type t = CF.wf × int
                                 let compare = wf_compare end)

  type lookups = { pmap : int list IMap.t;
                   wfmap : int WFMap.t;
                   cmap : CM.constant IMap.t × CM.constant IMap.t;
                   amap : F.amplitude IMap.t;
```

```
                    n_wfs : int list;
                    amplitudes : CF.amplitudes;
                    dict : F.amplitude → F.wf → int }

let largest_key imap =
  if (IMap.is_empty imap) then
    failwith "largest_key:␣Map␣is␣empty!"
  else
    fst (IMap.max_binding imap)
```

OCaml's *compare* from pervasives cannot compare functional types, e.g. for type *amplitude*, if no specific equality function is given ("equal: functional value"). Therefore, we allow to specify the ordering.

```
let get_ID' comp map elt : int =
  let smallmap = IMap.filter (fun _ x → (comp x elt) = 0 ) map in
  if IMap.is_empty smallmap then
    raise Not_found
  else
    fst (IMap.min_binding smallmap)
```

Trying to curry *map* here leads to type errors of the polymorphic function *get_ID*?

```
let get_ID map = match map with
  | map → get_ID' compare map

let get_const_ID map x = match map with
  | (map1, map2) → try get_ID' compare map1 x with
                     _ → try get_ID' compare map2 x with
                     _ → failwith "Impossible"
```

Creating an integer map of a list with an optional argument that indicates where the map should start counting.

```
let map_of_list ?start : (st = 1) lst =
  let g (ind, map) wf = (succ ind, IMap.add ind wf map) in
  lst |> List.fold_left g (st, IMap.empty) |> snd

let wf_map_of_list ?start : (st = 1) lst =
  let g (ind, map) wf = (succ ind, WFMap.add wf ind map) in
  lst |> List.fold_left g (st, WFMap.empty) |> snd
```

Header

Bijan : It would be nice to save the creation date as comment. However, the Unix module doesn't seem to be loaded on default.

```
let version =
  String.concat "␣" [Config.version; Config.status; Config.date]
let model_name =
  let basename = Filename.basename Sys.executable_name in
  try
    Filename.chop_extension basename
  with
  | _ → basename

let print_description cmdline =
  printf "Model␣%s\n" model_name;
  printf "OVM␣%s\n" version;
  printf "@\nBytecode␣file␣generated␣automatically␣by␣O'Mega␣for␣OVM";
  printf "@\nDo␣not␣delete␣any␣lines.␣You␣called␣O'Mega␣with";
  printf "@\n␣␣%s" cmdline;
        printf "@\n"

let num_classified_wfs wfs =
  let wfs' = classify_wfs wfs in
```

```
    List.map List.length
      [ wfs'.scalars @ wfs'.brs_scalars;
        wfs'.spinors @ wfs'.brs_spinors;
        wfs'.conjspinors @ wfs'.brs_conjspinors;
        wfs'.realspinors @ wfs'.brs_realspinors @ wfs'.ghostspinors;
        wfs'.vectors @ wfs'.massive_vectors @ wfs'.brs_vectors
          @ wfs'.brs_massive_vectors @ wfs'.ward_vectors;
        wfs'.tensors_2;
        wfs'.tensors_1;
        wfs'.vectorspinors ]

  let description_classified_wfs =
    [ "N_scalars";
      "N_spinors";
      "N_conjspinors";
      "N_bispinors";
      "N_vectors";
      "N_tensors_2";
      "N_tensors_1";
      "N_vectorspinors" ]

  let num_particles_in amp =
    match CF.flavors amp with
    | [] → 0
    | (fin, _) :: _ → List.length fin

  let num_particles_out amp =
    match CF.flavors amp with
    | [] → 0
    | (_, fout) :: _ → List.length fout

  let num_particles amp =
    match CF.flavors amp with
    | [] → 0
    | (fin, fout) :: _ → List.length fin + List.length fout

  let num_color_indices_default = 2 (* Standard model and non-color-exotica *)

  let num_color_indices amp =
    try CFlow.rank (List.hd (CF.color_flows amp)) with
    _ → num_color_indices_default

  let num_color_factors amp =
    let table = CF.color_factors amp in
    let n_cflow = Array.length table
    and n_cfactors = ref 0 in
    for c1 = 0 to pred n_cflow do
      for c2 = 0 to pred n_cflow do
        if c1 ≤ c2 then begin
          match table.(c1).(c2) with
          | [] → ()
          | _ → incr n_cfactors
        end
      done
    done;
    !n_cfactors

  let num_helicities amp = amp |> CF.helicities |> List.length

  let num_flavors amp = amp |> CF.flavors |> List.length

  let num_ks amp = amp |> CF.processes |> List.length

  let num_color_flows amp = amp |> CF.color_flows |> List.length
```

Use *fst* since $WFSet.t = F.wf \times int$.

```
  let num_wfs wfset = wfset |> WFSet.elements |> List.map fst
```

```
                                  |> num_classified_wfs
```

largest_key gives the number of momenta if applied to *pmap*.

```
let num_lst lookups wfset =
  [ largest_key lookups.pmap;
    num_particles lookups.amplitudes;
    num_particles_in lookups.amplitudes;
    num_particles_out lookups.amplitudes;
    num_ks lookups.amplitudes;
    num_helicities lookups.amplitudes;
    num_color_flows lookups.amplitudes;
    num_color_indices lookups.amplitudes;
    num_flavors lookups.amplitudes;
    num_color_factors lookups.amplitudes ] @ num_wfs wfset

let description_lst =
  [ "N_momenta";
    "N_particles";
    "N_prt_in";
    "N_prt_out";
    "N_amplitudes";
    "N_helicities";
    "N_col_flows";
    "N_col_indices";
    "N_flavors";
    "N_col_factors" ] @ description_classified_wfs

let print_header' numbers =
  let chopped_num_lst = ThoList.chopn inst_length numbers
  and chopped_desc_lst = ThoList.chopn inst_length description_lst
  and printer a b = print_str_lst a; print_int_lst b in
  List.iter2 printer chopped_desc_lst chopped_num_lst

let print_header lookups wfset = print_header' (num_lst lookups wfset)

let print_zero_header () =
  let rec zero_list' j =
    if j < 1 then []
    else 0 :: zero_list' (j - 1) in
  let zero_list i = zero_list' (i + 1) in
  description_lst |> List.length |> zero_list |> print_header'
```

Tables

```
let print_spin_table' tuples =
  match tuples with
  | [] → ()
  | _ → tuples |> List.iter ( fun (tuple1, tuple2) →
       tuple1 @ tuple2 |> List.map (Printf.sprintf "%d␣")
                        |> String.concat "" |> printf "@\n%s" )

let print_spin_table amplitudes =
  printf "@\nSpin␣states␣table";
  print_spin_table' @@ CF.helicities amplitudes

let print_flavor_table tuples =
  match tuples with
  | [] → ()
  | _ → List.iter ( fun tuple → tuple
                      |> List.map (fun f → Printf.sprintf "%d␣" @@ M.pdg f)
                      |> String.concat "" |> printf "@\n%s"
                    ) tuples

let print_flavor_tables amplitudes =
  printf "@\nFlavor␣states␣table";
  print_flavor_table @@ List.map (fun (fin, fout) → fin @ fout)
                        @@ CF.flavors amplitudes

let print_color_flows_table' tuple =
    match CFlow.to_lists tuple with
    | [] → ()
    | cfs → printf "@\n%s" @@ String.concat "" @@ List.map
               ( fun cf → cf |> List.map (Printf.sprintf "%d␣")
                             |> String.concat ""
               ) cfs

let print_color_flows_table tuples =
  match tuples with
  | [] → ()
  | _ → List.iter print_color_flows_table' tuples

let print_ghost_flags_table tuples =
  match tuples with
  | [] → ()
  | _ →
    List.iter (fun tuple →
    match CFlow.ghost_flags tuple with
        | [] → ()
        | gfs → printf "@\n"; List.iter (fun gf → printf "%s␣"
          (if gf then "1" else "0") ) gfs
    ) tuples

let format_power
  { CFlow.num = num; CFlow.den = den; CFlow.power = pwr } =
  match num, den, pwr with
  | _, 0, _ → invalid_arg "targets.format_power:␣zero␣denominator"
  | n, d, p → [n; d; p]

let format_powers = function
  | [] → [0]
  | powers → List.flatten (List.map format_power powers)
```

Straightforward iteration gives a great speedup compared to the fancier approach which only collects nonzero colorfactors.

```
  let print_color_factor_table table =
    let n_cflow = Array.length table in
    if n_cflow > 0 then begin
      for c1 = 0 to pred n_cflow do
        for c2 = 0 to pred n_cflow do
          if c1 ≤ c2 then begin
            match table.(c1).(c2) with
            | [] → ()
            | cf → printf "@\n"; List.iter (printf "%9d")
              ([succ c1; succ c2] @ (format_powers cf));
          end
        done
      done
    end

  let option_to_binary = function
    | Some _ → "1"
    | None → "0"

  let print_flavor_color_table n_flv n_cflow table =
    if n_flv > 0 then begin
      for c = 0 to pred n_cflow do
        printf "@\n";
        for f = 0 to pred n_flv do
          printf "%s␣" (option_to_binary table.(f).(c))
        done;
      done;
    end

  let print_color_tables amplitudes =
    let cflows = CF.color_flows amplitudes
    and cfactors = CF.color_factors amplitudes in
    printf "@\nColor␣flows␣table:␣[␣(i,␣j)␣(k,␣l)␣->␣(m,␣n)␣...]";
    print_color_flows_table cflows;
    printf "@\nColor␣ghost␣flags␣table:";
    print_ghost_flags_table cflows;
    printf "@\nColor␣factors␣table:␣[␣i,␣j:␣num␣den␣power],␣%s"
      "i,␣j␣are␣indexed␣color␣flows";
    print_color_factor_table cfactors;
    printf "@\nFlavor␣color␣combination␣is␣allowed:";
    print_flavor_color_table (num_flavors amplitudes) (List.length
      (CF.color_flows amplitudes)) (CF.process_table amplitudes)
```

Momenta

Add the momenta of a WFSet to a Iset. For now, we are throwing away the information to which amplitude the momentum belongs. This could be optimized for random color flow computations.

```
  let momenta_set wfset =
    let get_mom wf = wf |> fst |> F.momentum_list in
    let momenta = List.map get_mom (WFSet.elements wfset) in
    momenta |> List.fold_left (fun set x → set |> ISet.add x) ISet.empty

  let chop_in_3 lst =
    let ceil_div i j = if (i mod j = 0) then i/j else i/j + 1 in
    ThoList.chopn (ceil_div (List.length lst) 3) lst
```

Assign momenta via instruction code. External momenta [_] are already set by the OVM. To avoid unnecessary look-ups of IDs we seperate two cases. If we have more, we split up in two or three parts.

```
  let add_mom p pmap =
    let print_mom lhs rhs1 rhs2 rhs3 = if (rhs1 ≢ 0) then
      printi ~lhs : lhs ~rhs1 : rhs1 ~rhs2 : rhs2 ~rhs3 : rhs3 ovm_ADD_MOMENTA in
    let get_p_ID = get_ID pmap in
    match p with
    | [] | [_] → print_mom 0 0 0 0
    | [rhs1; rhs2] → print_mom (get_p_ID [rhs1; rhs2]) rhs1 rhs2 0
    | [rhs1; rhs2; rhs3] → print_mom (get_p_ID [rhs1; rhs2; rhs3]) rhs1 rhs2 rhs3
    | more →
        let ids = List.map get_p_ID (chop_in_3 more) in
        if (List.length ids = 3) then
          print_mom (get_p_ID more) (List.nth ids 0) (List.nth ids 1)
            (List.nth ids 2)
        else
          print_mom (get_p_ID more) (List.nth ids 0) (List.nth ids 1) 0
```

Hand through the current level and print level seperators if necessary.

```
  let add_all_mom lookups pset =
    let add_all′ level p =
      let level′ = List.length p in
      if (level′ > level ∧ level′ > 3) then break ();
      add_mom p lookups.pmap; level′
    in
    ignore (pset |> ISet.elements |> List.fold_left add_all′ 1)
```

Expand a set of momenta to contain all needed momenta for the computation in the OVM. For this, we create a list of sets which contains the chopped momenta and unify them afterwards. If the set has become larger, we expand again.

```
  let rec expand_pset p =
    let momlst = ISet.elements p in
    let pset_of lst = List.fold_left (fun s x → ISet.add x s) ISet.empty
      lst in
    let sets = List.map (fun x → pset_of (chop_in_3 x) ) momlst in
    let bigset = List.fold_left ISet.union ISet.empty sets in
    let biggerset = ISet.union bigset p in
    if (List.length momlst < List.length (ISet.elements biggerset) ) then
      expand_pset biggerset
    else
      biggerset

  let mom_ID pmap wf = get_ID pmap (F.momentum_list wf)
```

Wavefunctions and externals

mult_wf is needed because the *wf* with same combination of flavor and momentum can have different dependencies and content.

```
  let mult_wf dict amplitude wf =
    try
      wf, dict amplitude wf
    with
      | Not_found → wf, 0
```

Build the union of all *wf*s of all amplitudes and a map of the amplitudes.

```
  let wfset_amps amplitudes =
    let amap = amplitudes |> CF.processes |> List.sort amp_compare
                          |> map_of_list
    and dict = CF.dictionary amplitudes in
    let wfset_amp amp =
      let f = mult_wf dict amp in
      let lst = List.map f ((F.externals amp) @ (F.variables amp)) in
      lst |> List.fold_left (fun s x → WFSet.add x s) WFSet.empty in
    let list_of_sets = amplitudes |> CF.processes |> List.map wfset_amp in
      List.fold_left WFSet.union WFSet.empty list_of_sets, amap
```

To obtain the Fortran index, we substract the number of precedent wave functions.

```
  let lorentz_ordering_reduced wf =
    match SCM.lorentz (F.flavor wf) with
    | Scalar | BRS Scalar → 0
    | Spinor | BRS Spinor → 1
    | ConjSpinor | BRS ConjSpinor → 2
    | Majorana | BRS Majorana → 3
    | Vector | BRS Vector | Massive_Vector | BRS Massive_Vector → 4
    | Tensor_2 | BRS Tensor_2 → 5
    | Tensor_1 | BRS Tensor_1 → 6
    | Vectorspinor | BRS Vectorspinor → 7
    | Maj_Ghost → invalid_arg "lorentz_ordering:␣not␣implemented"
    | BRS _ → invalid_arg "lorentz_ordering:␣not␣needed"

  let wf_index wfmap num_lst (wf, i) =
    let wf_ID = WFMap.find (wf, i) wfmap
    and sum lst = List.fold_left (fun x y → x + y) 0 lst in
      wf_ID − sum (ThoList.hdn (lorentz_ordering_reduced wf) num_lst)

  let print_ext lookups amp_ID inc (wf, i) =
    let mom = (F.momentum_list wf) in
    let outer_index = if List.length mom = 1 then List.hd mom else
      failwith "targets.print_ext:␣called␣with␣non-external␣particle"
    and f = F.flavor wf in
    let pdg = SCM.pdg f
    and wf_code =
      match SCM.lorentz f with
      | Scalar → ovm_LOAD_SCALAR
      | BRS Scalar → ovm_LOAD_BRS_SCALAR
      | Spinor →
          if inc then ovm_LOAD_SPINOR_INC
          else ovm_LOAD_SPINOR_OUT
      | BRS Spinor →
          if inc then ovm_LOAD_BRS_SPINOR_INC
          else ovm_LOAD_BRS_SPINOR_OUT
      | ConjSpinor →
          if inc then ovm_LOAD_CONJSPINOR_INC
          else ovm_LOAD_CONJSPINOR_OUT
      | BRS ConjSpinor →
          if inc then ovm_LOAD_BRS_CONJSPINOR_INC
          else ovm_LOAD_BRS_CONJSPINOR_OUT
      | Vector | Massive_Vector →
          if inc then ovm_LOAD_VECTOR_INC
          else ovm_LOAD_VECTOR_OUT
      | BRS Vector | BRS Massive_Vector →
          if inc then ovm_LOAD_BRS_VECTOR_INC
          else ovm_LOAD_BRS_VECTOR_OUT
      | Tensor_2 →
          if inc then ovm_LOAD_TENSOR2_INC
          else ovm_LOAD_TENSOR2_OUT
      | Vectorspinor | BRS Vectorspinor →
          if inc then ovm_LOAD_VECTORSPINOR_INC
          else ovm_LOAD_VECTORSPINOR_OUT
      | Majorana →
          if inc then ovm_LOAD_MAJORANA_INC
          else ovm_LOAD_MAJORANA_OUT
      | BRS Majorana →
          if inc then ovm_LOAD_BRS_MAJORANA_INC
          else ovm_LOAD_BRS_MAJORANA_OUT
      | Maj_Ghost →
          if inc then ovm_LOAD_MAJORANA_GHOST_INC
          else ovm_LOAD_MAJORANA_GHOST_OUT
      | Tensor_1 →
          invalid_arg "targets.print_ext:␣Tensor_1␣only␣internal"
      | BRS _ →
          failwith "targets.print_ext:␣Not␣implemented"
    and wf_ind = wf_index lookups.wfmap lookups.n_wfs (wf, i)
    in
      printi wf_code ~lhs : wf_ind ~coupl : (abs(pdg)) ~rhs1 : outer_index ~rhs4 : amp_ID

  let print_ext_amp lookups amplitude =
    let incoming = (List.map (fun _ → true) (F.incoming amplitude) @
                    List.map (fun _ → false) (F.outgoing amplitude))
    and amp_ID = get_ID′ amp_compare lookups.amap amplitude in
    let wf_tpl wf = mult_wf lookups.dict amplitude wf in
    let print_ext_wf inc wf = wf |> wf_tpl |> print_ext lookups amp_ID inc in
      List.iter2 print_ext_wf incoming (F.externals amplitude)

  let print_externals lookups seen_wfs amplitude =
    let externals =
      List.combine
        (F.externals amplitude)
        (List.map (fun _ → true) (F.incoming amplitude) @
         List.map (fun _ → false) (F.outgoing amplitude)) in
    List.fold_left (fun seen (wf, incoming) →
      let amp_ID = get_ID′ amp_compare lookups.amap amplitude in
      let wf_tpl = mult_wf lookups.dict amplitude wf in
      if ¬ (WFSet.mem wf_tpl seen) then begin
        wf_tpl |> print_ext lookups amp_ID incoming
      end;
      WFSet.add wf_tpl seen) seen_wfs externals
```

print_externals and *print_ext_amp* do in principle the same thing but *print_externals* filters out dublicate external wave functions. Even with *print_externals* the same (numerically) external wave function will be loaded if it belongs to a different color flow, just as in the native Fortran code. For color MC, *print_ext_amp* has to be used (redundant instructions but only one flow is computed) and the filtering of duplicate fusions has to be disabled.

```
let print_ext_amps lookups =
  let print_external_amp s x = print_externals lookups s x in
  ignore (
    List.fold_left print_external_amp WFSet.empty
      (CF.processes lookups.amplitudes)
    )
```

(*

Currents

*)

Parallelization issues: All fusions have to be completed before the propagation takes place. Preferably each fusion and propagation is done by one thread. Solution: All fusions are subinstructions, i.e. if they are read by the main loop they are skipped. If a propagation occurs, all fusions have to be computed first. The additional control bit is the sign of the first int of an instruction.

```
let print_fermion_current code_a code_b code_c coeff lhs c wf1 wf2 fusion =
  let printc code r1 r2 = printi code ~lhs : lhs ~coupl : c ~coeff : coeff
    ~rhs1 : r1 ~rhs2 : r2 in
  match fusion with
  | F13 → printc code_a wf1 wf2
  | F31 → printc code_a wf2 wf1
  | F23 → printc code_b wf1 wf2
  | F32 → printc code_b wf2 wf1
  | F12 → printc code_c wf1 wf2
  | F21 → printc code_c wf2 wf1

  let ferm_print_current = function
    | coeff, Psibar, V, Psi → print_fermion_current
      ovm_FUSE_V_FF ovm_FUSE_F_VF ovm_FUSE_F_FV coeff
    | coeff, Psibar, VA, Psi → print_fermion_current
      ovm_FUSE_VA_FF ovm_FUSE_F_VAF ovm_FUSE_F_FVA coeff
    | coeff, Psibar, VA2, Psi → print_fermion_current
      ovm_FUSE_VA2_FF ovm_FUSE_F_VA2F ovm_FUSE_F_FVA2 coeff
    | coeff, Psibar, A, Psi → print_fermion_current
      ovm_FUSE_A_FF ovm_FUSE_F_AF ovm_FUSE_F_FA coeff
    | coeff, Psibar, VL, Psi → print_fermion_current
      ovm_FUSE_VL_FF ovm_FUSE_F_VLF ovm_FUSE_F_FVL coeff
    | coeff, Psibar, VR, Psi → print_fermion_current
      ovm_FUSE_VR_FF ovm_FUSE_F_VRF ovm_FUSE_F_FVR coeff
    | coeff, Psibar, VLR, Psi → print_fermion_current
      ovm_FUSE_VLR_FF ovm_FUSE_F_VLRF ovm_FUSE_F_FVLR coeff
    | coeff, Psibar, SP, Psi → print_fermion_current
      ovm_FUSE_SP_FF ovm_FUSE_F_SPF ovm_FUSE_F_FSP coeff
    | coeff, Psibar, S, Psi → print_fermion_current
      ovm_FUSE_S_FF ovm_FUSE_F_SF ovm_FUSE_F_FS coeff
    | coeff, Psibar, P, Psi → print_fermion_current
      ovm_FUSE_P_FF ovm_FUSE_F_PF ovm_FUSE_F_FP coeff
    | coeff, Psibar, SL, Psi → print_fermion_current
      ovm_FUSE_SL_FF ovm_FUSE_F_SLF ovm_FUSE_F_FSL coeff
    | coeff, Psibar, SR, Psi → print_fermion_current
      ovm_FUSE_SR_FF ovm_FUSE_F_SRF ovm_FUSE_F_FSR coeff
    | coeff, Psibar, SLR, Psi → print_fermion_current
```

```
          ovm_FUSE_SLR_FF ovm_FUSE_F_SLRF ovm_FUSE_F_FSLR coeff
      | _, Psibar, _, Psi → invalid_arg
          "Targets.Fortran.VM:␣no␣superpotential␣here"
      | _, Chibar, _, _ | _, _, _, Chi → invalid_arg
          "Targets.Fortran.VM:␣Majorana␣spinors␣not␣handled"
      | _, Gravbar, _, _ | _, _, _, Grav → invalid_arg
          "Targets.Fortran.VM:␣Gravitinos␣not␣handled"

let children2 rhs =
  match F.children rhs with
  | [wf1; wf2] → (wf1, wf2)
  | _ → failwith "Targets.children2:␣can't␣happen"

let children3 rhs =
  match F.children rhs with
  | [wf1; wf2; wf3] → (wf1, wf2, wf3)
  | _ → invalid_arg "Targets.children3:␣can't␣happen"

let print_vector4 c lhs wf1 wf2 wf3 fusion (coeff, contraction) =
  let printc r1 r2 r3 = printi ovm_FUSE_V_VVV ~lhs : lhs ~coupl : c
      ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3 in
  match contraction, fusion with
  | C_12_34, (F341 | F431 | F342 | F432 | F123 | F213 | F124 | F214)
  | C_13_42, (F241 | F421 | F243 | F423 | F132 | F312 | F134 | F314)
  | C_14_23, (F231 | F321 | F234 | F324 | F142 | F412 | F143 | F413) →
      printc wf1 wf2 wf3
  | C_12_34, (F134 | F143 | F234 | F243 | F312 | F321 | F412 | F421)
  | C_13_42, (F124 | F142 | F324 | F342 | F213 | F231 | F413 | F431)
  | C_14_23, (F123 | F132 | F423 | F432 | F214 | F241 | F314 | F341) →
      printc wf2 wf3 wf1
  | C_12_34, (F314 | F413 | F324 | F423 | F132 | F231 | F142 | F241)
  | C_13_42, (F214 | F412 | F234 | F432 | F123 | F321 | F143 | F341)
  | C_14_23, (F213 | F312 | F243 | F342 | F124 | F421 | F134 | F431) →
      printc wf1 wf3 wf2

let print_current lookups lhs amplitude rhs =
  let f = mult_wf lookups.dict amplitude in
  match F.coupling rhs with
  | V3 (vertex, fusion, constant) →
      let ch1, ch2 = children2 rhs in
      let wf1 = wf_index lookups.wfmap lookups.n_wfs (f ch1)
      and wf2 = wf_index lookups.wfmap lookups.n_wfs (f ch2)
      and p1 = mom_ID lookups.pmap ch1
      and p2 = mom_ID lookups.pmap ch2
      and const_ID = get_const_ID lookups.cmap constant in
      let c = if (F.sign rhs) < 0 then - const_ID else const_ID in
      begin match vertex with
      | FBF (coeff, fb, b, f) →
          begin match coeff, fb, b, f with
          | _, Psibar, VLRM, Psi | _, Psibar, SPM, Psi
          | _, Psibar, TVA, Psi | _, Psibar, TVAM, Psi
          | _, Psibar, TLR, Psi | _, Psibar, TLRM, Psi
          | _, Psibar, TRL, Psi | _, Psibar, TRLM, Psi → failwith
    "print_current:␣V3:␣Momentum␣dependent␣fermion␣couplings␣not␣implemented"
          | _, _, _, _ →
              ferm_print_current (coeff, fb, b, f) lhs c wf1 wf2 fusion
          end
      | PBP (_, _, _, _) →
          failwith "print_current:␣V3:␣PBP␣not␣implemented"
      | BBB (_, _, _, _) →
          failwith "print_current:␣V3:␣BBB␣not␣implemented"
      | GBG (_, _, _, _) →
          failwith "print_current:␣V3:␣GBG␣not␣implemented"
```

```
| Gauge_Gauge_Gauge coeff →
    let printc r1 r2 r3 r4 = printi ovm_FUSE_G_GG
      ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3
      ~rhs4 : r4 in
    begin match fusion with
    | (F23 | F31 | F12) → printc wf1 p1 wf2 p2
    | (F32 | F13 | F21) → printc wf2 p2 wf1 p1
    end

| I_Gauge_Gauge_Gauge _ →
    failwith "print_current:␣I_Gauge_Gauge_Gauge:␣not␣implemented"

| Scalar_Vector_Vector coeff →
    let printc code r1 r2 = printi code
      ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 in
    begin match fusion with
    | (F23 | F32) → printc ovm_FUSE_S_VV wf1 wf2
    | (F12 | F13) → printc ovm_FUSE_V_SV wf1 wf2
    | (F21 | F31) → printc ovm_FUSE_V_SV wf2 wf1
    end

| Scalar_Scalar_Scalar coeff →
    printi ovm_FUSE_S_SS ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : wf1 ~rhs2 : wf2

| Vector_Scalar_Scalar coeff →
    let printc code ?flip : (f = 1) r1 r2 r3 r4 = printi code
      ~lhs : lhs ~coupl : (c × f) ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3
      ~rhs4 : r4 in
    begin match fusion with
    | F23 → printc ovm_FUSE_V_SS wf1 p1 wf2 p2
    | F32 → printc ovm_FUSE_V_SS wf2 p2 wf1 p1
    | F12 → printc ovm_FUSE_S_VS wf1 p1 wf2 p2
    | F21 → printc ovm_FUSE_S_VS wf2 p2 wf1 p1
    | F13 → printc ovm_FUSE_S_VS wf1 p1 wf2 p2 ~flip : (−1)
    | F31 → printc ovm_FUSE_S_VS wf2 p2 wf1 p1 ~flip : (−1)
    end

| Aux_Vector_Vector _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Aux_Scalar_Scalar _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Aux_Scalar_Vector _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Graviton_Scalar_Scalar _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Graviton_Vector_Vector _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Graviton_Spinor_Spinor _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Dim4_Vector_Vector_Vector_T _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Dim4_Vector_Vector_Vector_L _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Dim6_Gauge_Gauge_Gauge _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Dim4_Vector_Vector_Vector_T5 _ →
    failwith "print_current:␣V3:␣not␣implemented"

| Dim4_Vector_Vector_Vector_L5 _ →
    failwith "print_current:␣V3:␣not␣implemented"
```

```
| Dim6_Gauge_Gauge_Gauge_5 _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Aux_DScalar_DScalar _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Aux_Vector_DScalar _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Scalar_Gauge2 coeff →
    let printc code r1 r2 r3 r4 = printi code
      ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3
      ~rhs4 : r4 in
    begin match fusion with
    | (F23 | F32) → printc ovm_FUSE_S_G2 wf1 p1 wf2 p2
    | (F12 | F13) → printc ovm_FUSE_G_SG wf1 p1 wf2 p2
    | (F21 | F31) → printc ovm_FUSE_G_GS wf2 p2 wf1 p1
    end
| Dim5_Scalar_Gauge2_Skew coeff →
    let printc code ?flip : (f = 1) r1 r2 r3 r4 = printi code
      ~lhs : lhs ~coupl : (c × f) ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3
      ~rhs4 : r4 in
    begin match fusion with
    | (F23 | F32) → printc ovm_FUSE_S_G2_SKEW wf1 p1 wf2 p2
    | (F12 | F13) → printc ovm_FUSE_G_SG_SKEW wf1 p1 wf2 p2
    | (F21 | F31) → printc ovm_FUSE_G_GS_SKEW wf2 p1 wf1 p2 ~flip : (−1)
    end
| Dim5_Scalar_Vector_Vector_T _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Scalar_Vector_Vector_U _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Scalar_Scalar2 _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim6_Vector_Vector_Vector_T _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Tensor_2_Vector_Vector _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Tensor_2_Scalar_Scalar _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Tensor_2_Vector_Vector_1 _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Tensor_2_Vector_Vector_2 _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim7_Tensor_2_Vector_Vector_T _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Dim5_Scalar_Vector_Vector_TU _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Scalar_Vector_Vector_t _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Tensor_2_Vector_Vector_cf _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Tensor_2_Scalar_Scalar_cf _ →
    failwith "print_current:␣V3:␣not␣implemented"
| Tensor_2_Vector_Vector_1 _ →
    failwith "print_current:␣V3:␣not␣implemented"
    | Tensor_2_Vector_Vector_t _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorVector_Vector_Vector _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorVector_Vector_Vector_cf _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorVector_Scalar_Scalar _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorVector_Scalar_Scalar_cf _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorScalar_Vector_Vector _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorScalar_Vector_Vector_cf _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorScalar_Scalar_Scalar _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | TensorScalar_Scalar_Scalar_cf _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_Scalar_Vector_Vector_D _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_Scalar_Vector_Vector_DP _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_HAZ_D _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_HAZ_DP _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_HHH _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_Gauge_Gauge_Gauge_i _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Gauge_Gauge_Gauge_i _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_GGG _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_AWW_DP _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_AWW_DW _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_WWZ_DPWDW _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_WWZ_DW _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Dim6_WWZ_D _ →
      failwith "print_current:␣V3:␣not␣implemented"
    | Aux_Gauge_Gauge _ →
      failwith "print_current:␣V3␣(Aux_Gauge_Gauge):␣not␣implemented"
end
```

Flip the sign in c to account for the i^2 relative to diagrams with only cubic couplings.

```
  | V4 (vertex, fusion, constant) →
```

```
let ch1, ch2, ch3 = children3 rhs in
let wf1 = wf_index lookups.wfmap lookups.n_wfs (f ch1)
and wf2 = wf_index lookups.wfmap lookups.n_wfs (f ch2)
and wf3 = wf_index lookups.wfmap lookups.n_wfs (f ch3)
          and const_ID = get_const_ID lookups.cmap constant in
let c =
  if (F.sign rhs) < 0 then const_ID else - const_ID in
begin match vertex with
| Scalar4 coeff →
    printi ovm_FUSE_S_SSS ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : wf1
      ~rhs2 : wf2 ~rhs3 : wf3
| Scalar2_Vector2 coeff →
    let printc code r1 r2 r3 = printi code
      ~lhs : lhs ~coupl : c ~coeff : coeff ~rhs1 : r1 ~rhs2 : r2 ~rhs3 : r3 in
    begin match fusion with
    | F134 | F143 | F234 | F243 →
        printc ovm_FUSE_S_SVV wf1 wf2 wf3
    | F314 | F413 | F324 | F423 →
        printc ovm_FUSE_S_SVV wf2 wf1 wf3
    | F341 | F431 | F342 | F432 →
        printc ovm_FUSE_S_SVV wf3 wf1 wf2
    | F312 | F321 | F412 | F421 →
        printc ovm_FUSE_V_SSV wf2 wf3 wf1
    | F231 | F132 | F241 | F142 →
        printc ovm_FUSE_V_SSV wf1 wf3 wf2
    | F123 | F213 | F124 | F214 →
        printc ovm_FUSE_V_SSV wf1 wf2 wf3
    end

| Vector4 contractions →
    List.iter (print_vector4 c lhs wf1 wf2 wf3 fusion) contractions

| Vector4_K_Matrix_tho _
| Vector4_K_Matrix_jr _
| Vector4_K_Matrix_cf_t0 _
| Vector4_K_Matrix_cf_t1 _
| Vector4_K_Matrix_cf_t2 _
| Vector4_K_Matrix_cf_t_rsi _
| Vector4_K_Matrix_cf_m0 _
| Vector4_K_Matrix_cf_m1 _
| Vector4_K_Matrix_cf_m7 _
| DScalar2_Vector2_K_Matrix_ms _
| DScalar2_Vector2_m_0_K_Matrix_cf _
| DScalar2_Vector2_m_1_K_Matrix_cf _
| DScalar2_Vector2_m_7_K_Matrix_cf _
| DScalar4_K_Matrix_ms _ →
    failwith "print_current:␣V4:␣K_Matrix␣not␣implemented"
| Dim8_Scalar2_Vector2_1 _
| Dim8_Scalar2_Vector2_2 _
| Dim8_Scalar2_Vector2_m_0 _
| Dim8_Scalar2_Vector2_m_1 _
| Dim8_Scalar2_Vector2_m_7 _
| Dim8_Scalar4 _ →
    failwith "print_current:␣V4:␣not␣implemented"
| Dim8_Vector4_t_0 _ →
    failwith "print_current:␣V4:␣not␣implemented"
| Dim8_Vector4_t_1 _ →
    failwith "print_current:␣V4:␣not␣implemented"
| Dim8_Vector4_t_2 _ →
    failwith "print_current:␣V4:␣not␣implemented"
| Dim8_Vector4_m_0 _ →
    failwith "print_current:␣V4:␣not␣implemented"
```

```
          | Dim8_Vector4_m_1 _ →
              failwith "print_current: V4: not implemented"
          | Dim8_Vector4_m_7 _ →
              failwith "print_current: V4: not implemented"
          | GBBG _ →
              failwith "print_current: V4: GBBG not implemented"
          | DScalar4 _
          | DScalar2_Vector2 _ →
              failwith "print_current: V4: DScalars not implemented"
          | Dim6_H4_P2 _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHWW_DPB _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHWW_DPW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHWW_DW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_Vector4_DW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_Vector4_W _ →
              failwith "print_current: V4: not implemented"
          | Dim6_Scalar2_Vector2_D _ →
              failwith "print_current: V4: not implemented"
          | Dim6_Scalar2_Vector2_DP _ →
              failwith "print_current: V4: not implemented"
          | Dim6_HWWZ_DW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_HWWZ_DPB _ →
              failwith "print_current: V4: not implemented"
          | Dim6_HWWZ_DDPW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_HWWZ_DPW _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHHZ_D _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHHZ_DP _ →
              failwith "print_current: V4: not implemented"
          | Dim6_AHHZ_PB _ →
              failwith "print_current: V4: not implemented"
          | Dim6_Scalar2_Vector2_PB _ →
              failwith "print_current: V4: not implemented"
          | Dim6_HHZZ_T _ →
              failwith "print_current: V4: not implemented"
          end
      | Vn (_, _, _) → invalid_arg "Targets.print_current: n-ary fusion."
```

Fusions

```
  let print_fusion lookups lhs_momID fusion amplitude =
    if F.on_shell amplitude (F.lhs fusion) then
      failwith "print_fusion: on_shell projectors not implemented!";
    if F.is_gauss amplitude (F.lhs fusion) then
      failwith "print_fusion: gauss amplitudes not implemented!";
    let lhs_wf = mult_wf lookups.dict amplitude (F.lhs fusion) in
    let lhs_wfID = wf_index lookups.wfmap lookups.n_wfs lhs_wf in
    let f = F.flavor (F.lhs fusion) in
    let pdg = SCM.pdg f in
    let w =
      begin match SCM.width f with
      | Vanishing | Fudged → 0
      | Constant → 1
      | Timelike → 2
      | Complex_Mass → 3
      | Running → 4
      | Custom _ → failwith "Targets.VM:␣custom␣width␣not␣available"
      end
    in
    let propagate code = printi code ~lhs : lhs_wfID ~rhs1 : lhs_momID
      ~coupl : (abs(pdg)) ~coeff : w ~rhs4 : (get_ID′ amp_compare lookups.amap amplitude)
    in
    begin match SCM.propagator f with
    | Prop_Scalar →
        propagate ovm_PROPAGATE_SCALAR
    | Prop_Col_Scalar →
        propagate ovm_PROPAGATE_COL_SCALAR
    | Prop_Ghost →
        propagate ovm_PROPAGATE_GHOST
    | Prop_Spinor →
        propagate ovm_PROPAGATE_SPINOR
    | Prop_ConjSpinor →
        propagate ovm_PROPAGATE_CONJSPINOR
    | Prop_Majorana →
        propagate ovm_PROPAGATE_MAJORANA
    | Prop_Col_Majorana →
        propagate ovm_PROPAGATE_COL_MAJORANA
    | Prop_Unitarity →
        propagate ovm_PROPAGATE_UNITARITY
    | Prop_Col_Unitarity →
        propagate ovm_PROPAGATE_COL_UNITARITY
    | Prop_Feynman →
        propagate ovm_PROPAGATE_FEYNMAN
    | Prop_Col_Feynman →
        propagate ovm_PROPAGATE_COL_FEYNMAN
    | Prop_Vectorspinor →
        propagate ovm_PROPAGATE_VECTORSPINOR
    | Prop_Tensor_2 →
        propagate ovm_PROPAGATE_TENSOR2
    | Aux_Col_Scalar | Aux_Col_Vector | Aux_Col_Tensor_1 →
        failwith "print_fusion:␣Aux_Col_*␣not␣implemented!"
    | Aux_Vector | Aux_Tensor_1 | Aux_Scalar | Aux_Spinor | Aux_ConjSpinor
    | Aux_Majorana | Only_Insertion →
        propagate ovm_PROPAGATE_NONE
    | Prop_Gauge _ →
        failwith "print_fusion:␣Prop_Gauge␣not␣implemented!"
    | Prop_Tensor_pure →
        failwith "print_fusion:␣Prop_Tensor_pure␣not␣implemented!"
    | Prop_Vector_pure →
        failwith "print_fusion:␣Prop_Vector_pure␣not␣implemented!"
    | Prop_Rxi _ →
        failwith "print_fusion:␣Prop_Rxi␣not␣implemented!"
    | Prop_UFO _ →
        failwith "print_fusion:␣Prop_UFO␣not␣implemented!"
    end;
```

Since the OVM knows that we want to propagate a wf, we can send the necessary fusions now.

```
  List.iter (print_current lookups lhs_wfID amplitude) (F.rhs fusion)

let print_all_fusions lookups =
  let fusions = CF.fusions lookups.amplitudes in
  let fset = List.fold_left (fun s x → FSet.add x s) FSet.empty fusions in
  ignore (List.fold_left (fun level (f, amplitude) →
      let wf = F.lhs f in
      let lhs_momID = mom_ID lookups.pmap wf in
      let level' = List.length (F.momentum_list wf) in
      if (level' > level ∧ level' > 2) then break ();
      print_fusion lookups lhs_momID f amplitude;
      level')
    1 (FSet.elements fset) )
```

Brakets

```
let print_braket lookups amplitude braket =
  let bra = F.bra braket
  and ket = F.ket braket in
  let braID = wf_index lookups.wfmap lookups.n_wfs
    (mult_wf lookups.dict amplitude bra) in
  List.iter (print_current lookups braID amplitude) ket
```

$$\mathrm{i}T = \mathrm{i}^{\#\text{vertices}}\mathrm{i}^{\#\text{propagators}}\cdots = \mathrm{i}^{n-2}\mathrm{i}^{n-3}\cdots = -\mathrm{i}(-1)^n\cdots \tag{20.3}$$

All brakets for one cflow amplitude should be calculated by one thread to avoid multiple access on the same memory (amplitude).

```
let print_brakets lookups (amplitude, i) =
  let n = List.length (F.externals amplitude) in
  let sign = if n mod 2 = 0 then -1 else 1
  and sym = F.symmetry amplitude in
  printi ovm_CALC_BRAKET ~lhs : i ~rhs1 : sym ~coupl : sign;
  match F.brakets amplitude with
  | [([], brakets)] → List.iter (print_braket lookups amplitude) brakets
  | _ → failwith "Targets.VM().print_brakets:␣coupling␣order␣slices␣not␣supported␣yet"
```

Fortran arrays/OCaml lists start on 1/0. The amplitude list is sorted by *amp_compare* according to their color flows. In this way the amp array is sorted in the same way as *table_color_factors*.

```
let print_all_brakets lookups =
  let g i elt = print_brakets lookups (elt, i + 1) in
  lookups.amplitudes |> CF.processes |> List.sort amp_compare
                     |> ThoList.iteri g 0
```

Couplings

For now we only care to catch the arrays *gncneu*, *gnclep*, *gncup* and *gncdown* of the SM. This will need an overhaul when it is clear how we store the type information of coupling constants.

```
let strip_array_tag = function
  | Real_Array x → x
  | Complex_Array x → x

let array_constants_list =
  let params = M.parameters()
  and strip_to_constant (lhs, _) = strip_array_tag lhs in
    List.map strip_to_constant params.derived_arrays

let is_array x = List.mem x array_constants_list

let constants_map =
  let first = fun (x, _, _) → x in
  let second = fun (_, y, _) → y in
  let third = fun (_, _, z) → z in
  let v3 = List.map third (first (M.vertices () ))
  and v4 = List.map third (second (M.vertices () )) in
  let set = List.fold_left (fun s x → CSet.add x s) CSet.empty (v3 @ v4) in
  let (arrays, singles) = CSet.partition is_array set in
    (singles |> CSet.elements |> map_of_list,
     arrays |> CSet.elements |> map_of_list)
```

Output calls

```
let amplitudes_to_channel (cmdline : string) (oc : out_channel)
  (diagnostics : (diagnostic × bool) list ) (amplitudes : CF.amplitudes) =

  set_formatter_out_channel oc;
  if (num_particles amplitudes = 0) then begin
    print_description cmdline;
    print_zero_header (); nl ()
  end else begin
    let (wfset, amap) = wfset_amps amplitudes in
    let pset = expand_pset (momenta_set wfset)
    and n_wfs = num_wfs wfset in
    let wfmap = wf_map_of_list (WFSet.elements wfset)
    and pmap = map_of_list (ISet.elements pset)
    and cmap = constants_map in

    let lookups = {pmap = pmap; wfmap = wfmap; cmap = cmap; amap = amap;
      n_wfs = n_wfs; amplitudes = amplitudes;
      dict = CF.dictionary amplitudes} in

    print_description cmdline;
    print_header lookups wfset;
    print_spin_table amplitudes;
    print_flavor_tables amplitudes;
    print_color_tables amplitudes;
    printf "@\n%s" ("OVM␣instructions␣for␣momenta␣addition," ^
                    "␣fusions␣and␣brakets␣start␣here:␣");
    break ();
    add_all_mom lookups pset;
    print_ext_amps lookups;
    break ();
    print_all_fusions lookups;
    break ();
    print_all_brakets lookups;
    break (); nl ();
    print_flush ()
  end

let parameters_to_fortran oc _ =
       set_formatter_out_channel oc;
  let arrays_to_set = ¬ (IMap.is_empty (snd constants_map)) in
  let set_coupl ty dim cmap = IMap.iter (fun key elt →
    printf "␣␣␣␣%s(%s%d)␣=␣%s" ty dim key (M.constant_symbol elt);
    nl () ) cmap in
  let declarations () =
    printf "␣␣complex(%s),␣dimension(%d)␣::␣ovm_coupl_cmplx"
      !kind (constants_map |> fst |> largest_key); nl ();
    if arrays_to_set then
      printf "␣␣complex(%s),␣dimension(2,␣%d)␣::␣ovm_coupl_cmplx2"
        !kind (constants_map |> snd |> largest_key); nl () in
  let print_line str = printf "%s" str; nl() in
  let print_md5sum = function
      | Some s →
        print_line "␣␣function␣md5sum␣()";
        print_line "␣␣␣␣character(len=32)␣::␣md5sum";
        print_line ("␣␣␣␣bytecode_file␣=␣'" ^ !bytecode_file ^ "'");
        print_line "␣␣␣␣call␣initialize_vm␣(vm,␣bytecode_file)";
        print_line "␣␣␣␣!␣DON'T␣EVEN␣THINK␣of␣modifying␣the␣following␣line!";
        print_line ("␣␣␣␣md5sum␣=␣'" ^ s ^ "'");
        print_line "␣␣end␣function␣md5sum";
      | None → ()
  in
```

```
let print_inquiry_function_openmp () = begin
  print_line "␣␣pure␣function␣openmp_supported␣()␣result␣(status)";
  print_line "␣␣␣␣logical␣::␣status";
  print_line ("␣␣␣␣status␣=␣" ^ (if !openmp then ".true." else ".false."));
  print_line "␣␣end␣function␣openmp_supported";
  nl ()
end in
let print_interface whizard =
if whizard then begin
  print_line "␣␣subroutine␣init␣(par,␣scheme)";
  print_line "␣␣␣␣real(kind=default),␣dimension(*),␣intent(in)␣::␣par";
  print_line "␣␣␣␣integer,␣intent(in)␣::␣scheme";
  print_line ("␣␣␣␣␣bytecode_file␣=␣'" ^ !bytecode_file ^ "'");
  print_line "␣␣␣␣call␣import_from_whizard␣(par,␣scheme)";
  print_line "␣␣␣␣call␣initialize_vm␣(vm,␣bytecode_file)";
  print_line "␣␣end␣subroutine␣init";
  nl ();
  print_line "␣␣subroutine␣final␣()";
  print_line "␣␣␣␣call␣vm%final␣()";
  print_line "␣␣end␣subroutine␣final";
  nl ();
  print_line "␣␣subroutine␣update_alpha_s␣(alpha_s)";
  print_line ("␣␣␣␣real(kind=" ^ !kind ^ "),␣intent(in)␣::␣alpha_s");
  print_line "␣␣␣␣call␣model_update_alpha_s␣(alpha_s)";
  print_line "␣␣end␣subroutine␣update_alpha_s";
  nl ()
end
else begin
  print_line "␣␣subroutine␣init␣()";
  print_line ("␣␣␣␣␣bytecode_file␣=␣'" ^ !bytecode_file ^ "'");
  print_line "␣␣␣␣␣call␣init_parameters␣()";
  print_line "␣␣␣␣␣call␣initialize_vm␣(vm,␣bytecode_file)";
  print_line "␣␣end␣subroutine"
end in
let print_lookup_functions () = begin
  print_line "␣␣pure␣function␣number_particles_in␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_particles_in␣()";
  print_line "␣␣end␣function␣number_particles_in";
  nl();
  print_line "␣␣pure␣function␣number_particles_out␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_particles_out␣()";
  print_line "␣␣end␣function␣number_particles_out";
  nl();
  print_line "␣␣pure␣function␣number_spin_states␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_spin_states␣()";
  print_line "␣␣end␣function␣number_spin_states";
  nl();
  print_line "␣␣pure␣subroutine␣spin_states␣(a)";
  print_line "␣␣␣␣integer,␣dimension(:,:),␣intent(out)␣::␣a";
  print_line "␣␣␣␣call␣vm%spin_states␣(a)";
  print_line "␣␣end␣subroutine␣spin_states";
  nl();
  print_line "␣␣pure␣function␣number_flavor_states␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_flavor_states␣()";
  print_line "␣␣end␣function␣number_flavor_states";
  nl();
  print_line "␣␣pure␣subroutine␣flavor_states␣(a)";
  print_line "␣␣␣␣integer,␣dimension(:,:),␣intent(out)␣::␣a";
  print_line "␣␣␣␣call␣vm%flavor_states␣(a)";
  print_line "␣␣end␣subroutine␣flavor_states";
  nl();
  print_line "␣␣pure␣function␣number_color_indices␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_color_indices␣()";
  print_line "␣␣end␣function␣number_color_indices";
  nl();
  print_line "␣␣pure␣function␣number_color_flows␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_color_flows␣()";
  print_line "␣␣end␣function␣number_color_flows";
  nl();
  print_line "␣␣pure␣subroutine␣color_flows␣(a,␣g)";
  print_line "␣␣␣␣integer,␣dimension(:,:,:),␣intent(out)␣::␣a";
  print_line "␣␣␣␣logical,␣dimension(:,:),␣intent(out)␣::␣g";
  print_line "␣␣␣␣call␣vm%color_flows␣(a,␣g)";
  print_line "␣␣end␣subroutine␣color_flows";
  nl();
  print_line "␣␣pure␣function␣number_color_factors␣()␣result␣(n)";
  print_line "␣␣␣␣integer␣::␣n";
  print_line "␣␣␣␣n␣=␣vm%number_color_factors␣()";
  print_line "␣␣end␣function␣number_color_factors";
  nl();
  print_line "␣␣pure␣subroutine␣color_factors␣(cf)";
  print_line "␣␣␣␣use␣omega_color";
  print_line "␣␣␣␣type(omega_color_factor),␣dimension(:),␣intent(out)␣::␣cf";
  print_line "␣␣␣␣call␣vm%color_factors␣(cf)";
  print_line "␣␣end␣subroutine␣color_factors";
  nl();
  print_line "␣␣!pure␣unless␣OpenMP";
  print_line "␣␣!pure␣function␣color_sum␣(flv,␣hel)␣result␣(amp2)";
  print_line "␣␣function␣color_sum␣(flv,␣hel)␣result␣(amp2)";
  print_line "␣␣␣␣use␣kinds";
  print_line "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel";
  print_line "␣␣␣␣real(kind=default)␣::␣amp2";
  print_line "␣␣␣␣amp2␣=␣vm%color_sum␣(flv,␣hel)";
  print_line "␣␣end␣function␣color_sum";
  nl();
  print_line "␣␣subroutine␣new_event␣(p)";
  print_line "␣␣␣␣use␣kinds";
  print_line "␣␣␣␣real(kind=default),␣dimension(0:3,*),␣intent(in)␣::␣p";
  print_line "␣␣␣␣call␣vm%new_event␣(p)";
  print_line "␣␣end␣subroutine␣new_event";
  nl();
  print_line "␣␣subroutine␣reset_helicity_selection␣(threshold,␣cutoff)";
  print_line "␣␣␣␣use␣kinds";
  print_line "␣␣␣␣real(kind=default),␣intent(in)␣::␣threshold";
  print_line "␣␣␣␣integer,␣intent(in)␣::␣cutoff";
  print_line "␣␣␣␣call␣vm%reset_helicity_selection␣(threshold,␣cutoff)";
  print_line "␣␣end␣subroutine␣reset_helicity_selection";
  nl();
  print_line "␣␣pure␣function␣is_allowed␣(flv,␣hel,␣col)␣result␣(yorn)";
  print_line "␣␣␣␣logical␣::␣yorn";
  print_line "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel,␣col";
  print_line "␣␣␣␣yorn␣=␣vm%is_allowed␣(flv,␣hel,␣col)";
  print_line "␣␣end␣function␣is_allowed";
  nl();
  print_line "␣␣pure␣function␣get_amplitude␣(flv,␣hel,␣col)␣result␣(amp_result)";
  print_line "␣␣␣␣use␣kinds";
```

```
    print_line "␣␣␣␣complex(kind=default)␣::␣amp_result";
    print_line "␣␣␣␣integer,␣intent(in)␣::␣flv,␣hel,␣col";
    print_line "␣␣␣␣amp_result␣=␣vm%get_amplitude(flv,␣hel,␣col)";
    print_line "␣␣end␣function␣get_amplitude";
    nl();
  end in
  print_line ("module␣" ^ !wrapper_module);
  print_line ("␣␣use␣" ^ !parameter_module_external);
  print_line "␣␣use␣iso_varying_string,␣string_t␣=>␣varying_string";
  print_line "␣␣use␣kinds";
  print_line "␣␣use␣omegavm95";
  print_line "␣␣implicit␣none";
  print_line "␣␣private";
  print_line "␣␣type(vm_t)␣::␣vm";
  print_line "␣␣type(string_t)␣::␣bytecode_file";
  print_line ("␣␣public␣::␣number_particles_in,␣number_particles_out," ^
      "␣number_spin_states,␣&");
  print_line ("␣␣␣␣spin_states,␣number_flavor_states,␣flavor_states," ^
      "␣number_color_indices,␣&");
  print_line ("␣␣␣␣number_color_flows,␣color_flows," ^
      "␣number_color_factors,␣color_factors,␣&");
  print_line ("␣␣␣␣color_sum,␣new_event,␣reset_helicity_selection," ^
      "␣is_allowed,␣get_amplitude,␣&");
  print_line ("␣␣␣␣init,␣" ^
      (match !md5sum with Some _ → "md5sum,␣"
                        | None → "") ^ "openmp_supported");
  if !whizard then
    print_line ("␣␣public␣::␣final,␣update_alpha_s")
  else
    print_line ("␣␣public␣::␣initialize_vm");
  declarations ();
  print_line "contains";

  print_line "␣␣subroutine␣setup_couplings␣()";
  set_coupl "ovm_coupl_cmplx" "" (fst constants_map);
  if arrays_to_set then
    set_coupl "ovm_coupl_cmplx2" ":," (snd constants_map);
  print_line "␣␣end␣subroutine␣setup_couplings";
  print_line "␣␣subroutine␣initialize_vm␣(vm,␣bytecode_file)";
  print_line "␣␣␣␣class(vm_t),␣intent(out)␣::␣vm";
  print_line "␣␣␣␣type(string_t),␣intent(in)␣::␣bytecode_file";
  print_line "␣␣␣␣type(string_t)␣::␣version";
  print_line "␣␣␣␣type(string_t)␣::␣model";
  print_line ("␣␣␣␣version␣=␣'OVM␣" ^ version ^ "'");
  print_line ("␣␣␣␣model␣=␣'Model␣" ^ model_name ^ "'");
  print_line "␣␣␣␣call␣setup_couplings␣()";
  print_line "␣␣␣␣call␣vm%init␣(bytecode_file,␣version,␣model,␣verbose=.False.,␣&";
  print_line "␣␣␣␣␣␣coupl_cmplx=ovm_coupl_cmplx,␣&";
  if arrays_to_set then
    print_line "␣␣␣␣␣␣coupl_cmplx2=ovm_coupl_cmplx2,␣&";
  print_line ("␣␣␣␣␣␣mass=mass,␣width=width,␣openmp=" ^ (if !openmp then
    ".true." else ".false.") ^ ")");
  print_line "␣␣end␣subroutine␣initialize_vm";
  nl();
  print_md5sum !md5sum;
  print_inquiry_function_openmp ();
  print_interface !whizard;
  print_lookup_functions ();

  print_line ("end␣module␣" ^ !wrapper_module)

let parameters_to_channel oc =
  parameters_to_fortran oc (SCM.parameters ())
```

```
end
```

—21— Phase Space

21.1 Interface of Phasespace

```
module type T =
  sig
    type momentum

    type α t
    type α decay
```

Sort individual decays and complete phasespaces in a canonical order to determine topological equivalence classes.

```
    val sort : (α → α → int) → α t → α t
    val sort_decay : (α → α → int) → α decay → α decay
```

Functionals:

```
    val map : (α → β) → α t → β t
    val map_decay : (α → β) → α decay → β decay

    val eval : (α → β) → (α → β) → (α → β → β → β) → α t → β t
    val eval_decay : (α → β) → (α → β → β → β) → α decay → β decay
```

of_momenta f1 f2 plist constructs the phasespace parameterization for a process $f_1 f_2 \to X$ with flavor decoration from pairs of outgoing momenta and flavors *plist* and initial flavors $f1$ and $f2$

```
    val of_momenta : α → α → (momentum × α) list → (momentum × α) t
    val decay_of_momenta : (momentum × α) list → (momentum × α) decay

    exception Duplicate of momentum
    exception Unordered of momentum
    exception Incomplete of momentum

  end
```

```
module Make (M : Momentum.T) : T with type momentum = M.t
```

21.2 Implementation of Phasespace

21.2.1 Tools

These are candidates for *ThoList* and not specific to phase space.

```
let rec first_match' mismatch f = function
  | [] → None
  | x :: rest →
      if f x then
        Some (x, List.rev_append mismatch rest)
      else
        first_match' (x :: mismatch) f rest
```

Returns $(x, X \setminus \{x\})$ if $\exists x \in X : f(x)$.

```
let first_match f l = first_match' [] f l
let rec first_pair' mismatch1 f l1 l2 =
  match l1 with
  | [] → None
  | x1 :: rest1 →
      begin match first_match (f x1) l2 with
      | None → first_pair' (x1 :: mismatch1) f rest1 l2
      | Some (x2, rest2) →
          Some ((x1, x2), (List.rev_append mismatch1 rest1, rest2))
      end
```

Returns $((x,y),(X\setminus\{x\},Y\setminus\{y\}))$ if $\exists x\in X:\exists y\in Y:f(x,y)$.

```
let first_pair f l1 l2 = first_pair' [] f l1 l2
```

21.2.2 Phase Space Parameterization Trees

```
module type T =
  sig
    type momentum
    type α t
    type α decay
    val sort : (α → α → int) → α t → α t
    val sort_decay : (α → α → int) → α decay → α decay
    val map : (α → β) → α t → β t
    val map_decay : (α → β) → α decay → β decay
    val eval : (α → β) → (α → β) → (α → β → β → β) → α t → β t
    val eval_decay : (α → β) → (α → β → β → β) → α decay → β decay
    val of_momenta : α → α → (momentum × α) list → (momentum × α) t
    val decay_of_momenta : (momentum × α) list → (momentum × α) decay
    exception Duplicate of momentum
    exception Unordered of momentum
    exception Incomplete of momentum
  end
```

```
module Make (M : Momentum.T) =
  struct

    type momentum = M.t
```

Finally, we came back to binary trees ...

Cascade Decays

```
    type α decay =
      | Leaf of α
      | Branch of α × α decay × α decay
```

Trees of type (*momentum* × α *option*) *decay* can be build easily and mapped to (*momentum* × α) *decay* later, once all the α slots are filled. A more elegant functor operating on β *decay* directly (with *Momentum* style functions defined for β) would not allow holes in the β *decay* during the construction.

```
    let label = function
      | Leaf p → p
      | Branch (p, _, _) → p

    let rec sort_decay cmp = function
      | Leaf _ as l → l
      | Branch (p, d1, d2) →
          let d1' = sort_decay cmp d1
          and d2' = sort_decay cmp d2 in
          if cmp (label d1') (label d2') ≤ 0 then
```

```
        Branch (p, d1', d2')
      else
        Branch (p, d2', d1')

let rec map_decay f = function
  | Leaf p → Leaf (f p)
  | Branch (p, d1, d2) → Branch (f p, map_decay f d1, map_decay f d2)

let rec eval_decay fl fb = function
  | Leaf p → Leaf (fl p)
  | Branch (p, d1, d2) →
      let d1' = eval_decay fl fb d1
      and d2' = eval_decay fl fb d2 in
      Branch (fb p (label d1') (label d2'), d1', d2')
```

Assuming that $p > p_D \lor p = p_D \lor p < p_D$, where p_D is the overall momentum of a decay tree D, we can add p to D at the top or somewhere in the middle. Note that '<' is not a total ordering and the operation can fail (raise exceptions) if the set of momenta does not correspond to a tree. Also note that a momentum can already be present without flavor as a complement in a branching entered earlier.

```
exception Duplicate of momentum
exception Unordered of momentum

let rec embed_in_decay (p, f as pf) = function
  | Leaf (p', f' as pf') as d' →
      if M.less p' p then
        Branch ((p, Some f), d', Leaf (M.sub p p', None))
      else if M.less p p' then
        Branch (pf', Leaf (p, Some f), Leaf (M.sub p' p, None))
      else if p = p' then
        begin match f' with
        | None → Leaf (p, Some f)
        | Some _ → raise (Duplicate p)
        end
      else
        raise (Unordered p)
  | Branch ((p', f' as pf'), d1, d2) as d' →
      let p1, _ = label d1
      and p2, _ = label d2 in
      if M.less p' p then
        Branch ((p, Some f), d', Leaf (M.sub p p', None))
      else if M.lesseq p p1 then
        Branch (pf', embed_in_decay pf d1, d2)
      else if M.lesseq p p2 then
        Branch (pf', d1, embed_in_decay pf d2)
      else if p = p' then
        begin match f' with
        | None → Branch ((p, Some f), d1, d2)
        | Some _ → raise (Duplicate p)
        end
      else
        raise (Unordered p)
```

Note that both *embed_in_decay* and *embed_in_decays* below do *not* commute, and should process 'bigger' momenta first, because disjoint sub-momenta will create disjoint subtrees in the latter and raise exceptions in the former.

```
exception Incomplete of momentum

let finalize1 = function
  | p, Some f → (p, f)
  | p, None → raise (Incomplete p)

let finalize_decay t = map_decay finalize1 t
```

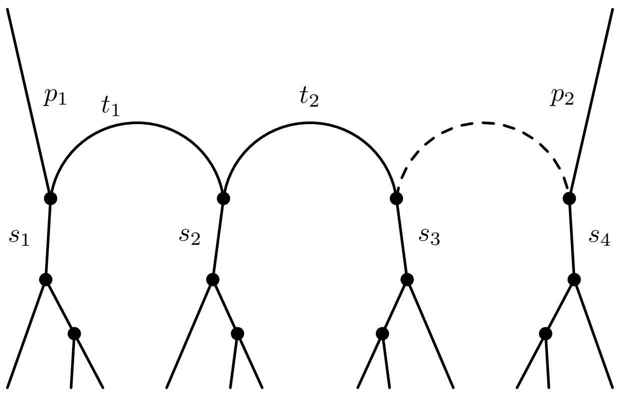


Figure 21.1: Phasespace parameterization for $2 \to n$ scattering by a sequence of cascade decays.

Process the momenta starting in with the highest $M.rank$:

```
let sort_momenta plist =
  List.sort (fun (p1, _) (p2, _) → M.compare p1 p2) plist

let decay_of_momenta plist =
  match sort_momenta plist with
  | (p, f) :: rest →
      finalize_decay (List.fold_right embed_in_decay rest (Leaf (p, Some f)))
  | [] → invalid_arg "Phasespace.decay_of_momenta:␣empty"
```

$2 \to n$ Scattering

A general $2 \to n$ scattering process can be parameterized by a sequence of cascade decays. The most symmetric representation is a little bit redundant and enters each t-channel momentum twice.

```
type α t = (α × α decay × α) list
```

let *topology* = *map snd* has type (*momentum* × α) *t* → α *t* and can be used to define topological equivalence classes "up to permutations of momenta," which are useful for calculating Whizard "groves"[1] [11].

```
let sort cmp = List.map (fun (l, d, r) → (l, sort_decay cmp d, r))
let map f = List.map (fun (l, d, r) → (f l, map_decay f d, f r))
let eval ft fl fb = List.map (fun (l, d, r) → (ft l, eval_decay fl fb d, ft r))
```

Find a tree with a defined ordering relation with respect to p or create a new one at the end of the list.

```
let rec embed_in_decays (p, f as pf) = function
  | [] → [Leaf (p, Some f)]
  | d′ :: rest →
      let p′, _ = label d′ in
      if M.lesseq p′ p ∨ M.less p p′ then
        embed_in_decay pf d′ :: rest
      else
        d′ :: embed_in_decays pf rest
```

Collecting Ingredients

```
type α unfinished_decays =
    { n : int;
      t_channel : (momentum × α option) list;
      decays : (momentum × α option) decay list }

let empty n = { n = n; t_channel = []; decays = [] }
```

[1]Not to be confused with gauge invariant classes of Feynman diagrams [13].

```
let insert_in_unfinished_decays (p, f as pf) d =
  if M.Scattering.spacelike p then
    { d with t_channel = (p, Some f) :: d.t_channel }
  else
    { d with decays = embed_in_decays pf d.decays }

let flip_incoming plist =
  List.map (fun (p', f') → (M.Scattering.flip_s_channel_in p', f')) plist

let unfinished_decays_of_momenta n f2 p =
  List.fold_right insert_in_unfinished_decays
    (sort_momenta (flip_incoming ((M.of_ints n [2], f2) :: p))) (empty n)
```

Assembling Ingredients

```
let sort3 compare x y z =
  let a = [| x; y; z |] in
  Array.sort compare a;
  (a.(0), a.(1), a.(2))
```

Take advantage of the fact that sorting with *M.compare* sorts with *rising* values of *M.rank*:

```
let allows_momentum_fusion (p, _) (p1, _) (p2, _) =
  let p2', p1', p' = sort3 M.compare p p1 p2 in
  match M.try_fusion p' p1' p2' with
  | Some _ → true
  | None → false

let allows_fusion p1 p2 d = allows_momentum_fusion (label d) p1 p2

let rec thread_unfinished_decays' p acc tlist dlist =
  match first_pair (allows_fusion p) tlist dlist with
  | None → (p, acc, tlist, dlist)
  | Some ((t, _ as td), (tlist', dlist')) →
      thread_unfinished_decays' t (td :: acc) tlist' dlist'

let thread_unfinished_decays p c =
  match thread_unfinished_decays' p [] c.t_channel c.decays with
  | _, pairs, [], [] → pairs
  | _ → failwith "thread_unfinished_decays"

let rec combine_decays = function
  | [] → []
  | ((t, f as tf), d) :: rest →
      let p, _ = label d in
      begin match M.try_sub t p with
      | Some p' → (tf, d, (p', f)) :: combine_decays rest
      | None → (tf, d, (M.sub (M.neg t) p, f)) :: combine_decays rest
      end

let finalize t = map finalize1 t

let of_momenta f1 f2 = function
  | (p, _) :: _ as l →
      let n = M.dim p in
      finalize (combine_decays
                  (thread_unfinished_decays (M.of_ints n [1], Some f1)
                     (unfinished_decays_of_momenta n f2 l)))
  | [] → []
```

Diagnostics

```
let p_to_string p =
  String.concat "" (List.map string_of_int (M.to_ints (M.abs p)))
let rec to_string1 = function
  | Leaf p → "(" ^ p_to_string p ^ ")"
  | Branch (_, d1, d2) → "(" ^ to_string1 d1 ^ to_string1 d2 ^ ")"

let to_string ps =
  String.concat "/"
    (List.map (fun (p1, d, p2) →
      p_to_string p1 ^ to_string1 d ^ p_to_string p2) ps)
```

Examples

```
  let try_thread_unfinished_decays p c =
    thread_unfinished_decays' p [] c.t_channel c.decays

  let try_of_momenta f = function
    | (p, _) :: _ as l →
        let n = M.dim p in
        try_thread_unfinished_decays
          (M.of_ints n [1], None) (unfinished_decays_of_momenta n f l)
    | [] → invalid_arg "try_of_momenta"

end
```

—22— Whizard

Talk to [11].

22.1 Interface of Whizard

```
module type T =
  sig
    type t
    type amplitude
    val trees : amplitude → t
    val merge : t → t
    val write : out_channel → string → t → unit
   end
module Make (FM : Fusion.Maker) (P : Momentum.T)
    (PW : Momentum.Whizard with type t = P.t) (M : Model.T) :
    T with type amplitude = FM(P)(M).amplitude
val write_interface : out_channel → string list → unit
val write_makefile : out_channel → α → unit
val write_makefile_processes : out_channel → string list → unit
```

22.2 Implementation of Whizard

```
open Printf
module type T =
  sig
    type t
    type amplitude
    val trees : amplitude → t
    val merge : t → t
    val write : out_channel → string → t → unit
   end
module Make (FM : Fusion.Maker) (P : Momentum.T)
    (PW : Momentum.Whizard with type t = P.t) (M : Model.T) =
  struct
    module F = FM(P)(M)
    type tree = (P.t × F.flavor list) list
    module Poles = Map.Make
        (struct
          type t = int × int
          let compare (s1, t1) (s2, t2) =
            let c = compare s2 s1 in
            if c ≠ 0 then
              c
            else
              compare t1 t2
        end)

    let add_tree maps tree trees =
      Poles.add maps
        (try tree :: (Poles.find maps trees) with Not_found → [tree]) trees

    type t =
        { in1 : F.flavor;
          in2 : F.flavor;
          out : F.flavor list;
          trees : tree list Poles.t }

    type amplitude = F.amplitude
```

22.2.1 Building Trees

A singularity is to be mapped if it is timelike and not the overall *s*-channel.

```
let timelike_map c = P.Scattering.timelike c ∧ ¬ (P.Scattering.s_channel c)

let count_maps n clist =
  List.fold_left (fun (s, t as cnt) (c, _) →
    if timelike_map c then
      (succ s, t)
    else if P.Scattering.spacelike c then
      (s, succ t)
    else
      cnt) (0, 0) clist

let poles_to_whizard n trees poles =
  let tree = List.map (fun wf →
    (P.Scattering.flip_s_channel_in (F.momentum wf), [F.flavor wf])) poles in
  add_tree (count_maps n tree) tree trees
```

I must reinstate the *conjugate* eventually!

```
let trees a =
  match F.externals a with
  | in1 :: in2 :: out →
      let n = List.length out + 2 in
      { in1 = F.flavor in1;
        in2 = F.flavor in2;
        out = List.map (fun f → (* M.conjugate *) (F.flavor f)) out;
        trees = List.fold_left
          (poles_to_whizard n) Poles.empty (F.poles a) }
  | _ → invalid_arg "Whizard().trees"
```

22.2.2 Merging Homomorphic Trees

```
module Pole_Map =
  Map.Make (struct type t = P.t list let compare = compare end)
module Flavor_Set =
  Set.Make (struct type t = F.flavor let compare = compare end)

let add_flavors flist fset =
  List.fold_right Flavor_Set.add flist fset

let set_of_flavors flist =
  List.fold_right Flavor_Set.add flist Flavor_Set.empty

let pack_tree map t =
  let c, f =
    List.split (List.sort (fun (c1, _) (c2, _) →
      compare (PW.of_momentum c2) (PW.of_momentum c1)) t) in
  let f′ =
    try
      List.map2 add_flavors f (Pole_Map.find c map)
    with
    | Not_found → List.map set_of_flavors f in
  Pole_Map.add c f′ map

let pack_map trees = List.fold_left pack_tree Pole_Map.empty trees

let merge_sets clist flist =
  List.map2 (fun c f → (c, Flavor_Set.elements f)) clist flist

let unpack_map map =
  Pole_Map.fold (fun c f l → (merge_sets c f) :: l) map []
```

If a singularity is to be mapped (i. e. if it is timelike and not the overall s-channel), expand merged particles again:

```
  let unfold1 (c, f) =
    if timelike_map c then
      List.map (fun f′ → (c, [f′])) f
    else
      [(c,f)]

  let unfold_tree tree = Product.list (fun x → x) (List.map unfold1 tree)

  let unfold trees = ThoList.flatmap unfold_tree trees

  let merge t =
    { t with trees = Poles.map
        (fun t′ → unfold (unpack_map (pack_map t′))) t.trees }
```

22.2.3 Printing Trees

```
  let flavors_to_string f =
    String.concat "/" (List.map M.flavor_to_string f)

  let whizard_tree t =
    "tree␣" ^
    (String.concat "␣" (List.rev_map (fun (c, _) →
      (string_of_int (PW.of_momentum c))) t)) ^
    "␣!␣" ^
    (String.concat ",␣" (List.rev_map (fun (_, f) → flavors_to_string f) t))

  let whizard_tree_debug t =
    "tree␣" ^
    (String.concat "␣" (List.rev_map (fun (c, _) →
      ("[" ^ (String.concat "+" (List.map string_of_int (P.to_ints c))) ^ "]"))
                                (List.sort (fun (t1,_) (t2,_) →
                                  let c =
                                    compare
                                      (List.length (P.to_ints t2))
                                      (List.length (P.to_ints t1)) in
                                  if c ≠ 0 then
                                    c
                                  else
                                    compare t1 t2) t))) ^
    "␣!␣" ^
    (String.concat ",␣" (List.rev_map (fun (_, f) → flavors_to_string f) t))

  let format_maps = function
    | (0, 0) → "neither␣mapped␣timelike␣nor␣spacelike␣poles"
    | (0, 1) → "no␣mapped␣timelike␣poles,␣one␣spacelike␣pole"
    | (0, n) → "no␣mapped␣timelike␣poles,␣" ^
```

```
          string_of_int n ^ "␣spacelike␣poles"
    | (1, 0) → "one␣mapped␣timelike␣pole,␣no␣spacelike␣pole"
    | (1, 1) → "one␣mapped␣timelike␣and␣spacelike␣pole␣each"
    | (1, n) → "one␣mapped␣timelike␣and␣" ^
          string_of_int n ^ "␣spacelike␣poles"
    | (n, 0) → string_of_int n ^
          "␣mapped␣timelike␣poles␣and␣no␣spacelike␣pole"
    | (n, 1) → string_of_int n ^
          "␣mapped␣timelike␣poles␣and␣one␣spacelike␣pole"
    | (n, n') → string_of_int n ^ "␣mapped␣timelike␣and␣" ^
          string_of_int n' ^ "␣spacelike␣poles"

let format_flavor f =
    match flavors_to_string f with
    | "d" → "d" | "dbar" → "D"
    | "u" → "u" | "ubar" → "U"
    | "s" → "s" | "sbar" → "S"
    | "c" → "c" | "cbar" → "C"
    | "b" → "b" | "bbar" → "B"
    | "t" → "t" | "tbar" → "T"
    | "e-" → "e1" | "e+" → "E1"
    | "nue" → "n1" | "nuebar" → "N1"
    | "mu-" → "e2" | "mu+" → "E2"
    | "numu" → "n2" | "numubar" → "N2"
    | "tau-" → "e3" | "tau+" → "E3"
    | "nutau" → "n3" | "nutaubar" → "N3"
    | "g" → "G" | "A" → "A" | "Z" → "Z"
    | "W+" → "W+" | "W-" → "W-"
    | "H" → "H"
    | s → s ^ "␣(not␣translated)"

module Mappable = Sets.String
let mappable =
    List.fold_right Mappable.add
        [ "T"; "Z"; "W+"; "W-"; "H" ] Mappable.empty

let analyze_tree ch t =
    List.iter (fun (c, f) →
        let f' = format_flavor f
        and c' = PW.of_momentum c in
        if P.Scattering.timelike c then begin
            if P.Scattering.s_channel c then
                fprintf ch "␣␣␣␣␣␣!␣overall␣s-channel␣%d␣%s␣not␣mapped\n" c' f'
            else if Mappable.mem f' mappable then
                fprintf ch "␣␣␣␣␣␣map␣%d␣s-channel␣%s\n" c' f'
            else
                fprintf ch
                    "␣␣␣␣␣␣!␣%d␣s-channel␣%s␣can't␣be␣mapped␣by␣whizard\n"
                    c' f'
        end else
            fprintf ch "␣␣␣␣␣␣!␣t-channel␣%d␣%s␣not␣mapped\n" c' f') t

let write ch pid t =
    failwith "Whizard.Make().write:␣incomplete"
    fprintf ch "process␣%s\n" pid;
    Poles.iter (fun maps ds →
        fprintf ch "\n␣␣␣␣!␣%d␣times␣%s:\n"
            (List.length ds) (format_maps maps);
        List.iter (fun d →
            fprintf ch "\n␣␣␣␣grove\n";
            fprintf ch "␣␣␣␣%s\n" (whizard_tree d);
            analyze_tree ch d) ds) t.trees;
    fprintf ch "\n"
```

```
i × )
  end
```

22.2.4 Process Dispatcher

```
let arguments = function
  | [] → ("", "")
  | args →
      let arg_list = String.concat ", " (List.map snd args) in
      (arg_list, ", " ^ arg_list)

let import_prefixed ch pid name =
  fprintf ch "    use %s, only: %s_%s => %s !NODEP!\n"
    pid pid name name

let declare_argument ch (arg_type, arg) =
  fprintf ch "    %s, intent(in) :: %s\n" arg_type arg

let call_function ch pid result name args =
  fprintf ch "       case (pr_%s)\n" pid;
  fprintf ch "          %s = %s_%s (%s)\n" result pid name args

let default_function ch result default =
  fprintf ch "       case default\n";
  fprintf ch "          call invalid_process (pid)\n";
  fprintf ch "          %s = %s\n" result default

let call_subroutine ch pid name args =
  fprintf ch "       case (pr_%s)\n" pid;
  fprintf ch "          call %s_%s (%s)\n" pid name args

let default_subroutine ch =
  fprintf ch "       case default\n";
  fprintf ch "          call invalid_process (pid)\n"

let write_interface_subroutine ch wrapper name args processes =
  let arg_list, arg_list' = arguments args in
  fprintf ch "  subroutine %s (pid%s)\n" wrapper arg_list';
  List.iter (fun p → import_prefixed ch p name) processes;
  List.iter (declare_argument ch) (("character(len=*)", "pid") :: args);
  fprintf ch "    select case (pid)\n";
  List.iter (fun p → call_subroutine ch p name arg_list) processes;
  default_subroutine ch;
  fprintf ch "    end select\n";
  fprintf ch "  end subroutine %s\n" wrapper

let write_interface_function ch wrapper name
    (result_type, result, default) args processes =
  let arg_list, arg_list' = arguments args in
  fprintf ch "  function %s (pid%s) result (%s)\n" wrapper arg_list' result;
  List.iter (fun p → import_prefixed ch p name) processes;
  List.iter (declare_argument ch) (("character(len=*)", "pid") :: args);
  fprintf ch "    %s :: %s\n" result_type result;
  fprintf ch "    select case (pid)\n";
  List.iter (fun p → call_function ch p result name arg_list) processes;
  default_function ch result default;
  fprintf ch "    end select\n";
  fprintf ch "  end function %s\n" wrapper

let write_other_interface_functions ch =
  fprintf ch "  subroutine invalid_process (pid)\n";
  fprintf ch "    character(len=*), intent(in) :: pid\n";
  fprintf ch "    print *, \"PANIC:";
  fprintf ch " process `\"//trim(pid)//\"' not available!\"\n";
  fprintf ch "␣␣end␣subroutine␣invalid_process\n";
  fprintf ch "␣␣function␣n_tot␣(pid)␣result␣(n)\n";
  fprintf ch "␣␣␣␣character(len=*),␣intent(in)␣::␣pid\n";
  fprintf ch "␣␣␣␣integer␣::␣n\n";
  fprintf ch "␣␣␣␣n␣=␣n_in(pid)␣+␣n_out(pid)\n";
  fprintf ch "␣␣end␣function␣n_tot\n"

let write_other_declarations ch =
  fprintf ch "␣␣public␣::␣n_in,␣n_out,␣n_tot,␣pdg_code\n";
  fprintf ch "␣␣public␣::␣allow_helicities\n";
  fprintf ch "␣␣public␣::␣create,␣destroy\n";
  fprintf ch "␣␣public␣::␣set_const,␣sqme\n";
  fprintf ch "␣␣interface␣create\n";
  fprintf ch "␣␣␣␣␣module␣procedure␣process_create\n";
  fprintf ch "␣␣end␣interface\n";
  fprintf ch "␣␣interface␣destroy\n";
  fprintf ch "␣␣␣␣␣module␣procedure␣process_destroy\n";
  fprintf ch "␣␣end␣interface\n";
  fprintf ch "␣␣interface␣set_const\n";
  fprintf ch "␣␣␣␣␣module␣procedure␣process_set_const\n";
  fprintf ch "␣␣end␣interface\n";
  fprintf ch "␣␣interface␣sqme\n";
  fprintf ch "␣␣␣␣␣module␣procedure␣process_sqme\n";
  fprintf ch "␣␣end␣interface\n"

let write_interface ch names =
  fprintf ch "module␣process_interface\n";
  fprintf ch "␣␣use␣kinds,␣only:␣default␣␣!NODEP!\n";
  fprintf ch "␣␣use␣parameters,␣only:␣parameter_set\n";
  fprintf ch "␣␣implicit␣none\n";
  fprintf ch "␣␣private\n";
  List.iter (fun p →
    fprintf ch
      "␣␣character(len=*),␣parameter,␣public␣::␣pr_%s␣=␣\"%s\"\n" p p)
    names;
  write_other_declarations ch;
  fprintf ch "contains\n";
  write_interface_function ch "n_in" "n_in" ("integer", "n", "0") [] names;
  write_interface_function ch "n_out" "n_out" ("integer", "n", "0") [] names;
  write_interface_function ch "pdg_code" "pdg_code"
    ("integer", "n", "0") [ "integer", "i" ] names;
  write_interface_function ch "allow_helicities" "allow_helicities"
    ("logical", "yorn", ".false.") [] names;
  write_interface_subroutine ch "process_create" "create" [] names;
  write_interface_subroutine ch "process_destroy" "destroy" [] names;
  write_interface_subroutine ch "process_set_const" "set_const"
    [ "type(parameter_set)", "par"] names;
  write_interface_function ch "process_sqme" "sqme"
    ("real(kind=default)", "sqme", "0")
    [ "real(kind=default),␣dimension(0:,:)", "p";
      "integer,␣dimension(:),␣optional", "h" ] names;
  write_other_interface_functions ch;
  fprintf ch "end␣module␣process_interface\n"
```

22.2.5 Makefile

```
let write_makefile ch names =
  fprintf ch "KINDS␣=␣../@KINDS@\n";
  fprintf ch "HELAS␣=␣../@HELAS@\n";
  fprintf ch "F90␣=␣@F90@\n";
  fprintf ch "F90FLAGS␣=␣@F90FLAGS@\n";
  fprintf ch "F90INCL␣=␣-I$(KINDS)␣-I$(HELAS)\n";
  fprintf ch "F90COMMON␣=␣omega_bundle_whizard.f90";
  fprintf ch "␣file_utils.f90␣process_interface.f90\n";
  fprintf ch "include␣Makefile.processes\n";
  fprintf ch "F90SRC␣=␣$(F90COMMON)␣$(F90PROCESSES)\n";
  fprintf ch "OBJ␣=␣$(F90SRC:.f90=.o)\n";
  fprintf ch "MOD␣=␣$(F90SRC:.f90=.mod)\n";
  fprintf ch "archive:␣processes.a\n";
  fprintf ch "processes.a:␣$(OBJ)\n";
  fprintf ch "\t$(AR)␣r␣$@␣$(OBJ)\n";
  fprintf ch "\t@RANLIB@␣$@\n";
  fprintf ch "clean:\n";
  fprintf ch "\trm␣-f␣$(OBJ)\n";
  fprintf ch "realclean:\n";
  fprintf ch "\trm␣-f␣processes.a\n";
  fprintf ch "parameters.o:␣file_utils.o\n";
  fprintf ch "omega_bundle_whizard.o:␣parameters.o\n";
  fprintf ch "process_interface.o:␣parameters.o\n";
  fprintf ch "%%.o:␣%%.f90␣$(KINDS)/kinds.f90\n";
  fprintf ch "\t$(F90)␣$(F90FLAGS)␣$(F90INCL)␣-c␣$<\n"

let write_makefile_processes ch names =
  fprintf ch "F90PROCESSES␣=";
  List.iter (fun f → fprintf ch "␣\\\n␣␣%s.f90" f) names;
  fprintf ch "\n";
  List.iter (fun f →
    fprintf ch "%s.o:␣omega_bundle_whizard.o␣parameters.o\n" f;
    fprintf ch "process_interface.o:␣%s.o\n" f) names
```

—23—
FeynMP, née FeynMF

Talk to [12].

23.1 Interface of Feynmp

```
module type T =
  sig
    type amplitudes

    val amplitudes_to_channel : bool → amplitudes → out_channel → unit
    val amplitudes_sans_color_to_channel : bool → amplitudes → out_channel → unit
    val amplitudes_color_only_to_channel : bool → amplitudes → out_channel → unit
```

Backward compatibility:

These can only be retired, if Whizard can deal with `"\\jobname-fmf.mp"` as metapost files!

```
    val amplitudes : bool → string → amplitudes → unit
    val amplitudes_sans_color : bool → string → amplitudes → unit
    val amplitudes_color_only : bool → string → amplitudes → unit

  end

module Make (FM : Fusion.Maker) (P : Momentum.T) (M : Model.T) : T
      with type amplitudes = Fusion.Multi(FM)(P)(M).amplitudes
```

23.2 Implementation of Feynmp

```
module type T =
  sig
    type amplitudes
    val amplitudes_to_channel : bool → amplitudes → out_channel → unit
    val amplitudes_sans_color_to_channel : bool → amplitudes → out_channel → unit
    val amplitudes_color_only_to_channel : bool → amplitudes → out_channel → unit
    val amplitudes : bool → string → amplitudes → unit
    val amplitudes_sans_color : bool → string → amplitudes → unit
    val amplitudes_color_only : bool → string → amplitudes → unit
  end

let (<<) f g x = f (g x)
let (>>) f g x = g (f x)

module Make (FM : Fusion.Maker) (P : Momentum.T) (M : Model.T) : T
      with type amplitudes = Fusion.Multi(FM)(P)(M).amplitudes =
  struct

    module F = FM(P)(M)
    module CF = Fusion.Multi(FM)(P)(M)
    module SCM = Orders.Slice(Colorize.It(M))

    type amplitudes = CF.amplitudes
```

```
let opt_array_to_list a =
  let rec opt_array_to_list' acc i a =
    if i < 0 then
      acc
    else
      begin match a.(i) with
      | None → opt_array_to_list' acc (pred i) a
      | Some x → opt_array_to_list' (x :: acc) (pred i) a
      end in
  opt_array_to_list' [] (Array.length a − 1) a

let amplitudes_by_flavor amplitudes =
  List.map opt_array_to_list (Array.to_list (CF.process_table amplitudes))
```

Take a *CF.amplitude list* assumed to correspond to the same external states after stripping the color and return a pair of the list of external particles and the corresponding Feynman diagrams without color.

```
let wf1 amplitude =
  match F.externals amplitude with
  | wf :: _ → wf
  | [] → failwith "Omega.forest_sans_color:␣no␣external␣particles"

let uniq l =
  ThoList.uniq (List.sort compare l)

let forest_sans_color = function
  | amplitude :: _ as amplitudes →
    let externals = F.externals amplitude in
    let prune_color wf =
      (F.flavor_sans_color wf, F.momentum_list wf) in
    let prune_color_and_couplings (wf, c) =
      (prune_color wf, None) in
    (List.map prune_color externals,
     uniq
       (List.map
          (fun t →
             Tree.canonicalize
               (Tree.map prune_color_and_couplings prune_color t))
          (ThoList.flatmap (fun a → F.forest (wf1 a) a) amplitudes)))
  | [] → ([], [])

let dag_sans_color = function
  | amplitude :: _ as amplitudes →
    let prune a = a in
    List.map prune amplitudes
  | [] → []

let p2s p =
  if p ≥ 0 ∧ p ≤ 9 then
    string_of_int p
  else if p ≤ 36 then
    String.make 1 (Char.chr (Char.code 'A' + p − 10))
  else
    "_"

let format_p wf =
  String.concat "" (List.map p2s (F.momentum_list wf))

let variable wf =
  M.flavor_to_string (F.flavor_sans_color wf) ^ "[" ^ format_p wf ^ "]"

let variable' wf =
  SCM.flavor_to_TeX (F.flavor wf) ^ "(" ^ format_p wf ^ ")"

let feynmf_style tex propagator color =
  { Tree.style =
      begin match propagator with
```

```
        | Coupling.Prop_Feynman
        | Coupling.Prop_Gauge _ →
           begin match color with
           | Color.AdjSUN _ → Some ("gluon", tex)
           | _ → Some ("boson", tex)
           end
        | Coupling.Prop_Col_Feynman → Some ("gluon", tex)
        | Coupling.Prop_Unitarity
        | Coupling.Prop_Rxi _ → Some ("dbl_wiggly", tex)
        | Coupling.Prop_Spinor
        | Coupling.Prop_ConjSpinor → Some ("fermion", tex)
        | _ → None
        end;
     Tree.rev =
        begin match propagator with
        | Coupling.Prop_Spinor → true
        | Coupling.Prop_ConjSpinor → false
        | _ → false
        end;
     Tree.label = None;
     Tree.tension = None }

let header incoming outgoing =
  "$␣" ^
  String.concat "␣"
    (List.map (SCM.flavor_to_TeX << F.flavor) incoming) ^
  "␣\\to␣" ^
  String.concat "␣"
    (List.map (SCM.flavor_to_TeX << SCM.conjugate << F.flavor) outgoing) ^
  "␣$"

let header_sans_color incoming outgoing =
  "$␣" ^
  String.concat "␣"
    (List.map (M.flavor_to_TeX << fst) incoming) ^
  "␣\\to␣" ^
  String.concat "␣"
    (List.map (M.flavor_to_TeX << M.conjugate << fst) outgoing) ^
  "␣$"

let diagram incoming tree =
  let fmf wf =
    let f = F.flavor wf in
    feynmf_style "" (SCM.propagator f) (SCM.color f) in
  Tree.map
    (fun (n, _) →
      let n' = fmf n in
      if List.mem n incoming then
        { n' with Tree.rev = ¬ n'.Tree.rev }
      else
        n')
    (fun l →
      if List.mem l incoming then
        l
      else
        F.conjugate l)
    tree

let diagram_sans_color incoming tree =
  let fmf (f, p) =
    feynmf_style "" (M.propagator f) (M.color f) in
  Tree.map
    (fun (n, c) →
          let n' = fmf n in
          if List.mem n incoming then
            { n' with Tree.rev = ¬ n'.Tree.rev }
          else
            n')
        (fun (f, p) →
          if List.mem (f, p) incoming then
            (f, p)
          else
            (M.conjugate f, p))
        tree

  let feynmf_set amplitude =
    match F.externals amplitude with
    | wf1 :: wf2 :: wfs →
        let incoming = [wf1; wf2] in
        { Tree.header = header incoming wfs;
          Tree.incoming = incoming;
          Tree.diagrams =
            List.map (diagram incoming) (F.forest wf1 amplitude) }
    | _ → failwith "less␣than␣two␣external␣particles"

  let feynmf_set_sans_color (externals, trees) =
    match externals with
    | wf1 :: wf2 :: wfs →
        let incoming = [wf1; wf2] in
        { Tree.header = header_sans_color incoming wfs;
          Tree.incoming = incoming;
          Tree.diagrams =
            List.map (diagram_sans_color incoming) trees }
    | _ → failwith "less␣than␣two␣external␣particles"

  let feynmf_set_sans_color_empty (externals, trees) =
    match externals with
    | wf1 :: wf2 :: wfs →
        let incoming = [wf1; wf2] in
        { Tree.header = header_sans_color incoming wfs;
          Tree.incoming = incoming;
          Tree.diagrams = [] }
    | _ → failwith "less␣than␣two␣external␣particles"

  let uncolored_colored amplitudes =
    { Tree.outer = feynmf_set_sans_color (forest_sans_color amplitudes);
      Tree.inner = List.map feynmf_set amplitudes }

  let uncolored_only amplitudes =
    { Tree.outer = feynmf_set_sans_color (forest_sans_color amplitudes);
      Tree.inner = [] }

  let colored_only amplitudes =
    { Tree.outer = feynmf_set_sans_color_empty (forest_sans_color amplitudes);
      Tree.inner = List.map feynmf_set amplitudes }

  let momentum_to_TeX (_, p) =
    String.concat "" (List.map p2s p)

  let wf_to_TeX (f, _ as wf) =
    M.flavor_to_TeX f ^ "(" ^ momentum_to_TeX wf ^ ")"

  let amplitudes latex name amplitudes =
      Tree.feynmf_sets_wrapped latex name
        wf_to_TeX momentum_to_TeX variable' format_p
        (List.map uncolored_colored (amplitudes_by_flavor amplitudes))

  let amplitudes_sans_color latex name amplitudes =
      Tree.feynmf_sets_wrapped latex name
```

```
          wf_to_TeX momentum_to_TeX variable′ format_p
          (List.map uncolored_only (amplitudes_by_flavor amplitudes))

  let amplitudes_color_only latex name amplitudes =
        Tree.feynmf_sets_wrapped latex name
          wf_to_TeX momentum_to_TeX variable′ format_p
          (List.map colored_only (amplitudes_by_flavor amplitudes))

  let amplitudes_to_channel latex amplitudes channel =
        Tree.feynmf_sets_wrapped_to_channel latex channel
          wf_to_TeX momentum_to_TeX variable′ format_p
          (List.map uncolored_colored (amplitudes_by_flavor amplitudes))

  let amplitudes_sans_color_to_channel latex amplitudes channel =
        Tree.feynmf_sets_wrapped_to_channel latex channel
          wf_to_TeX momentum_to_TeX variable′ format_p
          (List.map uncolored_only (amplitudes_by_flavor amplitudes))

  let amplitudes_color_only_to_channel latex amplitudes channel =
        Tree.feynmf_sets_wrapped_to_channel latex channel
          wf_to_TeX momentum_to_TeX variable′ format_p
          (List.map colored_only (amplitudes_by_flavor amplitudes))

end
```

—24— Applications

24.1 Interface of Omega

```
module type T =
  sig
    val main : ?current :int ref → ?argv :string array → unit → unit
```

This used to be only intended for debugging O'Giga, but might live longer …

```
    type flavor
    val diagrams : flavor → flavor → flavor list →
      ((flavor × Momentum.Default.t) ×
         (flavor × Momentum.Default.t,
          flavor × Momentum.Default.t) Tree.t) list
  end
```

Wrap the two instances of *Fusion.Maker* for amplitudes and phase space into a single functor to make sure that the Dirac and Majorana versions match. Don't export the slightly unsafe module *Make* (*FM* : *Fusion.Maker*) (*PM* : *Fusion.*

```
module Binary (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor
module Binary_Majorana (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor

module Mixed23 (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor
module Mixed23_Majorana (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor
module Mixed23_Majorana_vintage (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor

module Nary (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor
module Nary_Majorana (TM : Target.Maker) (M : Model.T) : T with type flavor = M.flavor
```

24.2 Implementation of Omega

```
module P = Momentum.Default
module P_Whizard = Momentum.DefaultW

module type T =
  sig
    val main : ?current :int ref → ?argv :string array → unit → unit
    type flavor
    val diagrams : flavor → flavor → flavor list →
      ((flavor × Momentum.Default.t) ×
         (flavor × Momentum.Default.t,
          flavor × Momentum.Default.t) Tree.t) list
  end

module Make (Fusion_Maker : Fusion.Maker) (PHS_Maker : Fusion.Maker)
         (Target_Maker : Target.Maker) (M : Model.T) =
  struct

    module CM = Colorize.It(M)
    module SCM = Orders.Slice(Colorize.It(M))
```

```
type flavor = M.flavor

module Proc = Process.Make(M)

module Coupling_Orders = Orders.Conditions(Colorize.It(M))
```

We must have initialized the vertices *before* applying *Fusion_Maker*, at least if we want to continue using the vertex cache!

NB: this causes the constant initializers in *Fusion_Maker* more than once. Such side effects must be avoided if the initializers involve expensive computations. *Relying on the fact that the functor will be called only once is not a good idea!*

```
module F = Fusion_Maker(P)(M)
module CF = Fusion.Multi(Fusion_Maker)(P)(M)
module T = Target_Maker(Fusion_Maker)(P)(M)
module W = Whizard.Make(Fusion_Maker)(P)(P_Whizard)(M)
module C = Cascade.Make(M)(P)

module VSet =
  Set.Make (struct type t = F.constant Coupling.t let compare = compare end)
```

For the phase space, we need asymmetric DAGs.

Since we will not use this to compute amplitudes, there's no need to supply the proper statistics module and we may always use Majorana fermions to be as general as possible. In principle, we could expose in *Fusion.T* the *Fusion.Stat_Maker* used by *Fusion_Maker* to construct it, but that is just not worth the effort.

For the phase space, we should be able to work on the uncolored model.

```
module MT = Modeltools.Topology3(M)
module PHS = PHS_Maker(P)(MT)
module CT = Cascade.Make(MT)(P)
```

Form a α *list* from a α *option array*, containing the elements that are not *None* in order.

```
let opt_array_to_list a =
  let rec opt_array_to_list' acc i a =
    if i < 0 then
      acc
    else
      begin match a.(i) with
      | None → opt_array_to_list' acc (pred i) a
      | Some x → opt_array_to_list' (x :: acc) (pred i) a
      end in
  opt_array_to_list' [] (Array.length a − 1) a
```

Return a list of *CF.amplitude list*s, corresponig to the diagrams for a specific color flow for each flavor combination.

```
let amplitudes_by_flavor amplitudes =
  List.map opt_array_to_list (Array.to_list (CF.process_table amplitudes))
```

If we plan to distiguish different couplings later on, we can no long map all instances of *coupling option* in the tree to *None*. In this case, we will need to normalize different fusion orders *Coupling.fuse2*, *Coupling.fuse3* or *Coupling.fusen*, because they would otherwise lead to inequivalent diagrams.

The *Tree.canonicalize* below should be necessary to remove topologically equivalent duplicates.

```
let p2s p =
  if p ≥ 0 ∧ p ≤ 9 then
    string_of_int p
  else if p ≤ 36 then
    String.make 1 (Char.chr (Char.code 'A' + p − 10))
  else
```

```
    "_"

let format_p wf =
  String.concat "" (List.map p2s (F.momentum_list wf))

let variable wf =
  M.flavor_to_string (F.flavor_sans_color wf) ^ "[" ^ format_p wf ^ "]"

let debug (str, descr, opt, var) =
  [ "-warning:" ^ str, Arg.Unit (fun () → var := (opt, false) :: !var),
    "␣␣␣␣␣␣␣␣␣check␣" ^ descr ^ "␣and␣print␣warning␣on␣error";
    "-error:" ^ str, Arg.Unit (fun () → var := (opt, true) :: !var),
    "␣␣␣␣␣␣␣␣␣␣␣check␣" ^ descr ^ "␣and␣terminate␣on␣error" ]

let rec include_goldstones = function
  | [] → false
  | (T.Gauge, _) :: _ → true
  | _ :: rest → include_goldstones rest

let read_lines_rev file =
  let ic = open_in file in
  let rev_lines = ref [] in
  let rec slurp () =
    rev_lines := input_line ic :: !rev_lines;
    slurp () in
  try
    slurp ()
  with
  | End_of_file →
      close_in ic;
      !rev_lines

let read_lines file =
  List.rev (read_lines_rev file)

let unphysical_polarization = ref None

module FMP = Feynmp.Make(Fusion_Maker)(P)(M)
```

24.2.1 Main Program

```
let main ?current ?argv () =
  (* Delay evaluation of M.external_flavors ()! *)
  let usage () =
    "usage:␣" ^ Sys.argv.(0) ^
    "␣[options]␣[" ^
      String.concat "|" (List.map M.flavor_to_string
                                  (ThoList.flatmap snd
                                     (M.external_flavors ()))) ^ "]"
  and rev_scatterings = ref []
  and rev_decays = ref []
  and cascades = ref []
  and orders = ref []
  and checks = ref []
  and output_file = ref None
  and print_forest = ref false
  and template = ref false
  and diagrams_all = ref None
  and diagrams_sans_color = ref None
  and diagrams_color_only = ref None
  and diagrams_LaTeX = ref false
  and quiet = ref false
  and write = ref true
  and params = ref false
  and poles = ref false
  and dag_out = ref None
  and dag0_out = ref None
  and phase_space_out = ref None in
  Options.parse ?current ?argv
    (Options.cmdline "-target:" T.options @
     Options.cmdline "-model:" M.options @
     Options.cmdline "-fusion:" CF.options @
     ThoList.flatmap debug
       ["a", "arguments", T.All, checks;
        "n", "#␣of␣input␣arguments", T.Arguments, checks;
        "m", "input␣momenta", T.Momenta, checks;
        "g", "internal␣Ward␣identities", T.Gauge, checks] @
     [("-o", Arg.String (fun s → output_file := Some s),
       "file␣␣␣␣␣␣␣␣␣␣␣␣␣write␣to␣given␣file␣instead␣of␣/dev/stdout");
      ("-scatter",
       Arg.String (fun s → rev_scatterings := s :: !rev_scatterings),
       "expr␣␣␣␣␣␣␣␣in1␣in2␣->␣out1␣out2␣...");
      ("-scatter_file",
       Arg.String (fun s → rev_scatterings := read_lines_rev s @ !rev_scatterings),
       "name␣␣each␣line:␣in1␣in2␣->␣out1␣out2␣...");
      ("-decay", Arg.String (fun s → rev_decays := s :: !rev_decays),
       "expr␣␣␣␣␣␣␣␣␣␣in␣->␣out1␣out2␣...");
      ("-decay_file",
       Arg.String (fun s → rev_decays := read_lines_rev s @ !rev_decays),
       "name␣␣␣␣each␣line:␣in␣->␣out1␣out2␣...");
      ("-cascade", Arg.String (fun s → cascades := s :: !cascades),
       "expr␣␣␣␣␣␣␣␣select␣diagrams");
      ("-orders", Arg.String (fun s → orders := s :: !orders),
       "expr␣␣␣␣␣␣␣␣select␣coupling␣orders");
      ("-unphysical", Arg.Int (fun i → unphysical_polarization := Some i),
       "n␣␣␣␣␣␣␣␣use␣unphysical␣polarization␣for␣n-th␣particle␣/␣test␣WIs");
      ("-template", Arg.Set template,
       "␣␣␣␣␣␣␣␣␣␣write␣a␣template␣for␣handwritten␣amplitudes");
      ("-forest", Arg.Set print_forest,
       "␣␣␣␣␣␣␣␣␣␣␣␣Diagrammatic␣expansion");
      ("-diagrams", Arg.String (fun s → diagrams_sans_color := Some s),
       "file␣␣␣␣␣␣produce␣FeynMP␣output␣for␣Feynman␣diagrams");
      ("-diagrams:c", Arg.String (fun s → diagrams_color_only := Some s),
       "file␣␣␣␣produce␣FeynMP␣output␣for␣color␣flow␣diagrams");
      ("-diagrams:C", Arg.String (fun s → diagrams_all := Some s),
       "file␣␣␣␣produce␣FeynMP␣output␣for␣Feynman␣and␣color␣flow␣diagrams");
      ("-diagrams_LaTeX", Arg.Set diagrams_LaTeX,
       "␣␣␣␣enclose␣FeynMP␣output␣in␣LaTeX␣wrapper");
      ("-quiet", Arg.Set quiet,
       "␣␣␣␣␣␣␣␣␣␣␣␣␣don't␣print␣a␣summary");
      ("-summary", Arg.Clear write,
       "␣␣␣␣␣␣␣␣␣␣␣print␣only␣a␣summary");
      ("-params", Arg.Set params,
       "␣␣␣␣␣␣␣␣␣␣␣␣print␣the␣model␣parameters");
      ("-poles", Arg.Set poles,
       "␣␣␣␣␣␣␣␣␣␣␣␣␣print␣the␣Monte␣Carlo␣poles");
      ("-dag", Arg.String (fun s → dag_out := Some s),
       "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print␣minimal␣DAG");
      ("-full_dag", Arg.String (fun s → dag0_out := Some s),
       "␣␣␣␣␣␣␣␣␣␣print␣complete␣DAG");
      ("-phase_space", Arg.String (fun s → phase_space_out := Some s),
       "␣␣␣␣␣␣␣print␣minimal␣DAG␣for␣phase␣space")])
    (fun _ → prerr_endline (usage ()); exit 1)
    usage;
```

```
let cmdline =
  String.concat "␣" (List.map ThoString.quote (Array.to_list Sys.argv)) in

let output_channel, close_output_channel =
  match !output_file with
  | None →
     (stdout, fun () → ())
  | Some name →
     let oc = open_out name in
     (oc, fun () → close_out oc) in

let processes =
  try
    ThoList.uniq
      (List.sort compare
          (match List.rev !rev_scatterings, List.rev !rev_decays with
          | [], [] → []
          | scatterings, [] →
              Proc.expand_scatterings (List.map Proc.parse_scattering scatterings)
          | [], decays →
              Proc.expand_decays (List.map Proc.parse_decay decays)
          | scatterings, decays →
              invalid_arg "mixed␣scattering␣and␣decay!"))
  with
  | Invalid_argument s →
      begin
        Printf.eprintf "O'Mega:␣invalid␣process␣specification:␣%s!\n" s;
        flush stderr;
        []
      end in
```

This is still crude. Eventually, we want to catch *all* exceptions and write an empty (but compilable) amplitude unless one of the special options is selected.

```
begin match processes, !params with
| _, true →
    if !write then
      T.parameters_to_channel output_channel;
    exit 0
| [], false →
   if !write then
     T.amplitudes_to_channel cmdline output_channel !checks CF.empty;
   exit 0
| _, false →

  let selectors =
    let fin, fout = List.hd processes in
    C.to_selectors (C.of_string_list (List.length fin + List.length fout) !cascades) in

  let orders =
    match !orders with
    | [] → None
    | strings → Some (Coupling_Orders.of_strings (List.rev strings)) in

  let amplitudes =
    try
      begin match F.check_charges () with
      | [] → ()
      | violators →
          let violator_strings =
            String.concat ",␣"
              (List.map
                  (fun flist →
                    "(" ^ String.concat "," (List.map M.flavor_to_string flist) ^ ")")
                  violators) in
          failwith ("charge␣violating␣vertices:␣" ^ violator_strings)
      end;
      CF.amplitudes (include_goldstones !checks) !unphysical_polarization selectors orders processes
    with
    | Fusion.Majorana →
        begin
          Printf.eprintf
            "O'Mega:␣found␣Majorana␣fermions,␣switching␣representation!\n";
          flush stderr;
          close_output_channel ();
          Arg.current := 0;
          raise Fusion.Majorana
        end
    | exc →
         begin
           Printf.eprintf
             "O'Mega:␣exception␣%s␣in␣amplitude␣construction!\n"
             (Printexc.to_string exc);
           flush stderr;
           CF.empty;
         end in

  if !write then
    T.amplitudes_to_channel cmdline output_channel !checks amplitudes;

  if ¬ !quiet then begin
    List.iter
      (fun amplitude →
        Printf.eprintf "SUMMARY:␣%d␣fusions,␣%d␣propagators"
          (F.count_fusions amplitude) (F.count_propagators amplitude);
        flush stderr;
        Printf.eprintf ",␣%d␣diagrams" (F.count_diagrams amplitude);
        Printf.eprintf "\n")
      (CF.processes amplitudes);
    let couplings =
      List.fold_left
        (fun acc p →
          let brakets = ThoList.flatmap snd (F.brakets p) in
          let fusions = ThoList.flatmap F.rhs (F.fusions p)
          and brakets = ThoList.flatmap F.ket brakets in
          let couplings = VSet.of_list (List.map F.coupling (fusions @ brakets)) in
          VSet.union acc couplings)
        VSet.empty (CF.processes amplitudes) in
    Printf.eprintf "SUMMARY:␣%d␣vertices\n" (VSet.cardinal couplings);
    let ufo_couplings =
      VSet.fold
        (fun v acc →
          match v with
          | Coupling.Vn (Coupling.UFO (_, v, _, _, _), _, _) →
             Sets.String.add v acc
          | _ → acc)
        couplings Sets.String.empty in
    if ¬ (Sets.String.is_empty ufo_couplings) then
      Printf.eprintf
        "SUMMARY:␣%d␣UFO␣vertices:␣%s\n"
        (Sets.String.cardinal ufo_couplings)
        (String.concat ",␣" (Sets.String.elements ufo_couplings))
  end;

  if !poles then begin
    List.iter
      (fun amplitude →
        W.write output_channel "omega" (W.merge (W.trees amplitude)))
      (CF.processes amplitudes)
  end;

  begin match !dag0_out with
  | Some name →
      let ch = open_out name in
      List.iter (F.tower_to_dot ch) (CF.processes amplitudes);
      close_out ch
  | None → ()
  end;

  begin match !dag_out with
  | Some name →
      let ch = open_out name in
      List.iter (F.amplitude_to_dot ch) (CF.processes amplitudes);
      close_out ch
  | None → ()
  end;

  begin match !phase_space_out with
  | Some name →
     let selectors =
        let fin, fout = List.hd processes in
        CT.to_selectors (CT.of_string_list (List.length fin + List.length fout) !cascades) in
     let ch = open_out name in
     begin try
        List.iter
           (fun (fin, fout) →
              Printf.fprintf
             ch "%s␣->␣%s␣::\n"
             (String.concat "␣" (List.map M.flavor_to_string fin))
             (String.concat "␣" (List.map M.flavor_to_string fout));
              match fin with
              | [] →
                  failwith "Omega():␣phase␣space:␣no␣incoming␣particles"
              | [f] →
                  PHS.phase_space_channels ch (PHS.amplitude_sans_color false selectors fin fout)
              | [f1; f2] →
                  PHS.phase_space_channels ch (PHS.amplitude_sans_color false selectors fin fout);
                  PHS.phase_space_channels_flipped ch (PHS.amplitude_sans_color false selectors [f2; f1] fout)
              | _ →
                  failwith "Omega():␣phase␣space:␣3␣or␣more␣incoming␣particles")
           processes;
        close_out ch
     with
     | exc →
        begin
          close_out ch;
          Printf.eprintf
             "O'Mega:␣exception␣%s␣in␣phase␣space␣construction!\n"
             (Printexc.to_string exc);
          flush stderr
        end
     end
  | None → ()
  end;

  if !print_forest then
    List.iter
      (fun amplitude →
        List.iter (fun t → Printf.eprintf "%s\n"
             (Tree.to_string
                (Tree.map (fun (wf, _) → variable wf) (fun _ → "") t)))
          (F.forest (List.hd (F.externals amplitude)) amplitude))
      (CF.processes amplitudes);

  begin match !diagrams_all with
  | Some name → FMP.amplitudes !diagrams_LaTeX name amplitudes
  | None → ()
  end;

  begin match !diagrams_sans_color with
  | Some name → FMP.amplitudes_sans_color !diagrams_LaTeX name amplitudes
  | None → ()
  end;

  begin match !diagrams_color_only with
  | Some name → FMP.amplitudes_color_only !diagrams_LaTeX name amplitudes
  | None → ()
  end;

  close_output_channel ();

  exit 0

end
```

This was only intended for debugging O'Giga …

```
    let decode wf =
      (F.flavor wf, (F.momentum wf : Momentum.Default.t))

    let diagrams in1 in2 out =
      match F.amplitudes false C.no_cascades None [in1; in2] out with
      | a :: _ →
          let wf1 = List.hd (F.externals a)
          and wf2 = List.hd (List.tl (F.externals a)) in
          let wf2 = decode wf2 in
          List.map (fun t →
            (wf2,
             Tree.map (fun (wf, _) → decode wf) decode t))
            (F.forest wf1 a)
      | [] → []

    let diagrams in1 in2 out =
      failwith "Omega().diagrams:␣disabled"

  end

module Binary (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Binary)(Fusion.Helac_Binary)(TM)(M)
module Binary_Majorana (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Binary_Majorana)(Fusion.Helac_Binary_Majorana)(TM)(M)
module Mixed23 (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Mixed23)(Fusion.Helac_Mixed23)(TM)(M)
module Mixed23_Majorana (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Mixed23_Majorana)(Fusion.Helac_Mixed23_Majorana)(TM)(M)
module Mixed23_Majorana_vintage (TM : Target.Maker) (M : Model.T) =
  Make(Fusion_vintage.Mixed23_Majorana)(Fusion.Helac_Mixed23_Majorana)(TM)(M)

module Bound (M : Model.T) : Tuple.Bound =
  struct
    (*
```

Above $max\_degree = 6$, the performance drops *dramatically*!

```
*)
    let max_arity () =
      pred (M.max_degree ())
  end

module Nary (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Nary(Bound(M)))(Fusion.Helac(Bound(M)))(TM)(M)
module Nary_Majorana (TM : Target.Maker) (M : Model.T) =
  Make(Fusion.Nary_Majorana(Bound(M)))(Fusion.Helac_Majorana(Bound(M)))(TM)(M)
```

24.3 Interface of Omega_cli

Next generation command line interface.

Ideally, I would have liked to use `cmdliner` (https://erratique.ch/software/cmdliner), but more recent versions of this require ocaml 4.08. Also, building it without `dune` might be a challenge.

Neverthess, I will take inspiration from `cmdliner`.

```
module Models : sig
  type t
  val of_list : (string × string × (module Model.T)) list → t
  val by_name_opt : t → string → (module Model.T) option
  val names : t → (string × string) list
end
```

Since there are only very few implementations of *Target.Maker* that are actively maintained and only *Targets.Fortran_Majorar* can currently deal with Majorana fermions, we don't implement a lookup table but select them explicitey according to the command line options.

```
module type T =
  sig
    val main : ?current :int ref → ?argv :string array → unit → unit
  end

module Make (F : Fusion.Maker) (P : Fusion.Maker) (T : Target.Maker) (M : Model.Mutable) : T
```

24.4 Implementation of Omega_cli

24.4.1 Model Collection

```
module SMap = Map.Make(String)

module Models =
  struct

    type t = (string × string × (module Model.T)) SMap.t

    let normalize = String.lowercase_ascii

    let of_list model_list =
      List.fold_left
        (fun acc (name, _, _ as model) →
          let key = normalize name in
          begin match SMap.find_opt key acc with
          | None → ()
          | Some (clash, _, _) →
             invalid_arg
               (Printf.sprintf "Omega_cli.Models.of_list:␣ambiguous␣model␣names␣'%s'␣~␣'%s'!" name clash)
          end;
          SMap.add key model acc)
        SMap.empty model_list

    let by_name_opt models name =
      match SMap.find_opt (normalize name) models with
      | None → None
      | Some (_, _, model) → Some model
    let names models =
      List.map (fun (_, (name, description, _)) → (name, description)) (SMap.bindings models)

  end
```

24.4.2 Output Files

```
type filename_components =
  { stem : string;
    extension : string }

type filename =
  | Components of filename_components
  | Full of string
  | Stdout

let open_output_channel name =
  let oc = open_out name in
  let close () = close_out oc in
  (oc, close, name)

let standard_output_channel =
  (stdout, (fun () → flush stdout), "/dev/stdout")

let prefix_directory directory_opt name =
  match directory_opt with
  | None → name
  | Some dir →
     if Filename.is_relative name ∧ Filename.is_implicit name then
       Filename.concat dir name
     else
       name

let output_channel directory_opt prefix = function
  | Stdout → standard_output_channel
  | Full name → open_output_channel (prefix_directory directory_opt name)
  | Components { stem; extension } →
     begin match prefix, stem, extension with
     | "", "", "" → standard_output_channel
     | _, _, _ →
        let suffix =
          if stem = "" ∨ extension = "" then
            stem ^ extension
          else
            stem ^ "." ^ extension in
        let name =
          if prefix = "" ∨ suffix = "" then
            prefix ^ suffix
          else
            prefix ^ "_" ^ suffix in
        open_output_channel (prefix_directory directory_opt name)
     end

let with_output_channel ?logging directory_opt prefix file f =
  let channel, close, name = output_channel directory_opt prefix file in
  begin match logging with
  | None → f channel
  | Some product →
     Printf.eprintf "Omega_cli:␣writing␣%s␣to␣'%s'␣..." product name;
     f channel;
     Printf.eprintf "␣done.\n"
  end;
  close ()
```

24.4.3 Output File Options

```
module type Output =
  sig
    type t = { write : bool; file : filename }
    val default : t
    val specs : t ref → (Arg.key × Arg.spec × Arg.doc) list
  end

module type File =
  sig
    val write : bool
    val opt : string
    val stem : string
    val ext : string
  end

module Output (F : File) : Output =
  struct

    type t = { write : bool; file : filename }

    let default =
      { write = F.write;
        file = Components { stem = F.stem; extension = F.ext } }

    let write output yorn =
      output := { !output with write = yorn }

    let stdout output () =
      output := { write = true; file = Stdout }

    let name output file =
      output := { write = true; file = Full file }

    let warn_component component ignored name =
      Printf.eprintf
        "omega3:␣new␣%s␣file␣%s␣'%s'␣ignored,␣full␣name␣'%s'␣already␣set!\n"
        F.opt component ignored name

    let stem output stem =
      match !output.file with
      | Components components → output := { !output with file = Components { components with stem } }
      | Full name → warn_component "stem" stem name
      | _ → ()

    let extension output extension =
      match !output.file with
      | Components components → output := { !output with file = Components { components with extension } }
      | Full name → warn_component "extension" extension name
      | _ → ()

    let specs output =
      let open Printf in
      [ ("--" ^ F.opt, Arg.Bool (write output),
          sprintf "true|false␣write␣%s␣file␣(default:␣%b)" F.opt F.write);
        ("--" ^ F.opt ^ "_stdout", Arg.Unit (stdout output),
          sprintf "␣write␣%s␣file␣to␣/dev/stdout" F.opt);
        ("--" ^ F.opt ^ "_name", Arg.String (name output),
          sprintf "name␣set␣%s␣file␣name" F.opt);
        ("--" ^ F.opt ^ "_stem", Arg.String (stem output),
          sprintf "stem␣set␣%s␣file␣stem␣(default='%s')" F.opt F.stem);
        ("--" ^ F.opt ^ "_extension", Arg.String (extension output),
          sprintf "ext␣set␣%s␣file␣extension␣(default='%s')" F.opt F.ext) ]

  end

module Amplitude = Output (struct let write = true let opt = "amplitude" let stem = opt let ext = "f90" end)
module Log = Output (struct let write = true let opt = "log" let stem = "amplitude" let ext = "log" end)
module Parameters = Output (struct let write = false let opt = "parameters" let stem = opt let ext = "f90" end)
module Phasespace = Output (struct let write = false let opt = "phasespace" let stem = opt let ext = "phs" end)
module Poles = Output (struct let write = false let opt = "poles" let stem = opt let ext = "poles" end)
module Whizard_Model = Output (struct let write = false let opt = "whizard" let stem = opt let ext = "mdl" end)
module Forest = Output (struct let write = false let opt = "forest" let stem = opt let ext = "out" end)
module Diagrams = Output (struct let write = false let opt = "diagrams" let stem = opt let ext = "tex" end)
module Colorflows = Output (struct let write = false let opt = "colorflows" let stem = opt let ext = "tex" end)
module DAG = Output (struct let write = false let opt = "dag" let stem = opt let ext = "dot" end)
module Full_DAG = Output (struct let write = false let opt = "full_dag" let stem = opt let ext = "dot" end)
```

24.4.4 Command Line Parsing

```
type processes =
  | Scatterings of string list
  | Decays of string list

type command_ref =
  { processes_ref : processes ref;
    restrictions_rev_ref : string list ref;
    orders_rev_ref : string list ref;
    orders2_ref : bool ref;
    unphysical_ref : int option ref;
    directory_ref : string option ref;
    prefix_ref : string ref;
    amplitude_ref : Amplitude.t ref;
    log_ref : Log.t ref;
    parameters_ref : Parameters.t ref;
    phasespace_ref : Phasespace.t ref;
    poles_ref : Poles.t ref;
    whizard_ref : Whizard_Model.t ref;
    forest_ref : Forest.t ref;
    diagrams_ref : Diagrams.t ref;
    colorflows_ref : Colorflows.t ref;
    latex_ref : bool ref;
    dag_ref : DAG.t ref;
    full_dag_ref : Full_DAG.t ref;
    template_ref : bool ref }

let default_ref =
  { processes_ref = ref (Scatterings []);
    restrictions_rev_ref = ref [];
    orders_rev_ref = ref [];
    orders2_ref = ref false;
    unphysical_ref = ref None;
    directory_ref = ref None;
    prefix_ref = ref "omega";
    amplitude_ref = ref Amplitude.default;
    log_ref = ref Log.default;
    parameters_ref = ref Parameters.default;
    phasespace_ref = ref Phasespace.default;
    poles_ref = ref Poles.default;
    whizard_ref = ref Whizard_Model.default;
    diagrams_ref = ref Diagrams.default;
    forest_ref = ref Forest.default;
    colorflows_ref = ref Colorflows.default;
    latex_ref = ref false;
    dag_ref = ref DAG.default;
    full_dag_ref = ref Full_DAG.default;
    template_ref = ref false }

let add_scatterings command lines =
  let processes =
    match !(command.processes_ref) with
    | Scatterings rev_lines → Scatterings (lines @ rev_lines)
    | Decays [] → Scatterings lines
    | Decays _ → invalid_arg "Omega_cli.add_scattering:␣mixing␣-scatter␣and␣-decay" in
  command.processes_ref := processes

let add_decays command lines =
  let processes =
    match !(command.processes_ref) with
    | Decays rev_lines → Decays (lines @ rev_lines)
    | Scatterings [] → Decays lines
    | Scatterings _ → invalid_arg "Omega_cli.add_decay:␣mixing␣-scatter␣and␣-decay" in
  command.processes_ref := processes

let add_restrictions command lines =
  command.restrictions_rev_ref := lines @ !(command.restrictions_rev_ref)

let add_orders command lines =
  command.orders_rev_ref := lines @ !(command.orders_rev_ref)

let set_orders2 command yorn =
  command.orders2_ref := yorn

let set_unphysical command n =
  command.unphysical_ref := Some n

let set_directory command directory =
  command.directory_ref := Some directory

let set_prefix command prefix =
  command.prefix_ref := prefix

type command =
  { processes : processes;
    restrictions_rev : string list;
    orders_rev : string list;
    orders2 : bool;
    unphysical : int option;
    directory : string option;
    prefix : string;
    amplitude : Amplitude.t;
    log : Log.t;
    parameters : Parameters.t;
    phasespace : Phasespace.t;
    poles : Poles.t;
    whizard : Whizard_Model.t;
    forest : Forest.t;
    diagrams : Diagrams.t;
    colorflows : Colorflows.t;
    latex : bool;
    dag : DAG.t;
    full_dag : Full_DAG.t;
    template : bool }

let command_of_ref command =
  { processes = !(command.processes_ref);
    restrictions_rev = !(command.restrictions_rev_ref);
    orders_rev = !(command.orders_rev_ref);
    orders2 = !(command.orders2_ref);
    unphysical = !(command.unphysical_ref);
    directory = !(command.directory_ref);
    prefix = !(command.prefix_ref);
    amplitude = !(command.amplitude_ref);
    log = !(command.log_ref);
    parameters = !(command.parameters_ref);
    poles = !(command.poles_ref);
    phasespace = !(command.phasespace_ref);
    whizard = !(command.whizard_ref);
    forest = !(command.forest_ref);
    diagrams = !(command.diagrams_ref);
    colorflows = !(command.colorflows_ref);
    latex = !(command.latex_ref);
    dag = !(command.dag_ref);
    full_dag = !(command.full_dag_ref);
    template = !(command.template_ref) }
```

24.4.5 Combining Target, Topology, and Model.T

The *Target* module is not in the `omega_core` library and we can not reference implementations here, only interfaces like *Target_Maker*.

```
module type T =
  sig
    val main : ?current :int ref → ?argv :string array → unit → unit
  end

module P = Momentum.Default
module P_Whizard = Momentum.DefaultW

let obsolete_UFO_options =
  Sets.String.of_list [ "UFO_dir"; "Majorana"; "dump"; "write_WHIZARD"; "exec" ]

let purge_ufo_options options =
  Options.exclude (fun o → Sets.String.mem o obsolete_UFO_options) options

module Make (FM : Fusion.Maker) (PHS_Maker : Fusion.Maker) (TM : Target.Maker) (M : Model.Mutable) =
  struct

    type flavor = M.flavor

    module Proc = Process.Make(M)
    module C = Cascade.Make(M)(P)
    module Coupling_Orders = Orders.Conditions(Colorize.It(M))

    module CM = Colorize.It(M)
    module SCM = Orders.Slice(Colorize.It(M))

    module F = FM(P)(M)
    module CF = Fusion.Multi(FM)(P)(M)
    module T = TM(FM)(P)(M)

    module VSet = Set.Make (struct type t = F.constant Coupling.t let compare = compare end)
    module W = Whizard.Make(FM)(P)(P_Whizard)(M)
    module MT = Modeltools.Topology3(M)
    module PHS = PHS_Maker(P)(MT)
    module CT = Cascade.Make(MT)(P)
    module FMP = Feynmp.Make(FM)(P)(M)

    let parse_processes processes =
      try
        ThoList.uniq
          (List.sort compare
             (match processes with
              | Scatterings lines → Proc.expand_scatterings (List.rev_map Proc.parse_scattering lines)
              | Decays lines → Proc.expand_decays (List.rev_map Proc.parse_decay lines)))
      with
      | Invalid_argument s →
          invalid_arg (Printf.sprintf "Omega_cli:␣invalid␣process␣specification:␣%s!\n" s)

    let parse_restrictions processes restrictions =
      match processes with
      | [] → C.no_cascades
      | (fin, fout) :: _ →
          begin match restrictions with
          | [] → C.no_cascades
          | restrictions →
              C.to_selectors (C.of_string_list (List.length fin + List.length fout) restrictions)
          end
```

Once more with only triple vertices for the phasespace. This could be functorized over CT:

```
  let parse_restrictions_phs processes restrictions =
    match processes with
    | [] → CT.no_cascades
    | (fin, fout) :: _ →
        begin match restrictions with
        | [] → CT.no_cascades
        | restrictions →
            CT.to_selectors (CT.of_string_list (List.length fin + List.length fout) restrictions)
        end

  let parse_orders = function
    | [] → None
    | lines → Some (Coupling_Orders.of_strings lines)

  let flavors_to_string_all_orders flavors =
    String.concat "␣" (List.map (fun f → CM.flavor_to_string (SCM.flavor_all_orders f)) flavors)

  let process_to_string_all_orders amplitude =
    flavors_to_string_all_orders (F.incoming amplitude) ^ "␣->␣" ^
    flavors_to_string_all_orders (F.outgoing amplitude)

  let log_to_channel cmdline amplitudes channel =
    let open Printf in
    fprintf channel "%s\n" cmdline;
    List.iter
      (fun amplitude →
        fprintf channel "%s:␣%d␣fusions,␣%d␣propagators,␣%d␣diagrams\n"
          (process_to_string_all_orders amplitude)
          (F.count_fusions amplitude)
          (F.count_propagators amplitude)
          (F.count_diagrams amplitude))
      (CF.processes amplitudes);
    let couplings =
      List.fold_left
        (fun acc p →
          let brakets = ThoList.flatmap snd (F.brakets p) in
          let fusions = ThoList.flatmap F.rhs (F.fusions p)
          and brakets = ThoList.flatmap F.ket brakets in
          let couplings = VSet.of_list (List.map F.coupling (fusions @ brakets)) in
          VSet.union acc couplings)
        VSet.empty (CF.processes amplitudes) in
    fprintf channel "%d␣vertices\n" (VSet.cardinal couplings);
    let ufo_couplings =
      VSet.fold
        (fun v acc →
          match v with
          | Coupling.Vn (Coupling.UFO (_, v, _, _, _), _, _) → Sets.String.add v acc
          | _ → acc)
        couplings Sets.String.empty in
    if ¬ (Sets.String.is_empty ufo_couplings) then
      fprintf channel "%d␣UFO␣vertices:␣%s\n"
        (Sets.String.cardinal ufo_couplings)
        (String.concat ",␣" (Sets.String.elements ufo_couplings))
```

```
let phasespace_to_channel restrictions processes channel =
  let selectors = parse_restrictions_phs processes restrictions in
  List.iter
    (fun (fin, fout) →
      Printf.fprintf channel "%s␣->␣%s␣::\n"
        (String.concat "␣" (List.map M.flavor_to_string fin))
        (String.concat "␣" (List.map M.flavor_to_string fout));
      match fin with
      | [_] →
          PHS.phase_space_channels channel (PHS.amplitude_sans_color false selectors fin fout)
      | [f1; f2] →
          PHS.phase_space_channels channel (PHS.amplitude_sans_color false selectors fin fout);
          PHS.phase_space_channels_flipped channel (PHS.amplitude_sans_color false selectors [f2; f1] fout)
      | _ →
          failwith (Printf.sprintf "Omega_cli.phasespace_to_channel:␣impossible:␣%␣incoming␣particles"
                      (List.length fin)))
    processes
```

Whizard.Make().write has been disabled for a while now. Don't show this option.

```
let poles_to_channel amplitudes channel =
  List.iter
    (fun amplitude → W.write channel "omega" (W.merge (W.trees amplitude)))
    (CF.processes amplitudes)

let p2s p =
  if p ≥ 0 ∧ p ≤ 9 then
    string_of_int p
  else if p ≤ 36 then
    String.make 1 (Char.chr (Char.code 'A' + p − 10))
  else
    "_"

let format_p wf =
  String.concat "" (List.map p2s (F.momentum_list wf))

let variable wf =
  M.flavor_to_string (F.flavor_sans_color wf) ^ "[" ^ format_p wf ^ "]"

let forest_to_channel amplitudes channel =
  List.iter
    (fun amplitude →
      List.iter
        (fun tree →
          Printf.fprintf channel "%s\n"
            (Tree.to_string (Tree.map (fun (wf, _) → variable wf) (fun _ → "") tree)))
        (F.forest (List.hd (F.externals amplitude)) amplitude))
    (CF.processes amplitudes)

let debug (str, descr, opt, var) =
  [ "--warning:" ^ str, Arg.Unit (fun () → var := (opt, false) :: !var),
    "␣check␣" ^ descr ^ "␣and␣warn";
    "--error:" ^ str, Arg.Unit (fun () → var := (opt, true) :: !var),
    "␣check␣" ^ descr ^ "␣and␣terminate" ]

let rec include_goldstones = function
  | [] → false
  | (T.Gauge, _) :: _ → true
  | _ :: rest → include_goldstones rest

let read_lines_rev file =
  let ic = open_in file in
  let rev_lines = ref [] in
  let rec slurp () =
      rev_lines := input_line ic :: !rev_lines;
      slurp () in
    try
      slurp ()
    with
    | End_of_file →
        close_in ic;
        !rev_lines

let read_lines file =
  List.rev (read_lines_rev file)

let list_flavors () =
  List.iter
    (fun (group, flavors) →
      Printf.printf "%s:\n" group;
      List.iter (fun f → Printf.printf "␣␣%s\n" (M.flavor_to_string f)) flavors)
    (M.external_flavors ())

let main ?current ?(argv = Sys.argv) () =
  let my_name = Filename.basename argv.(0)
  and cmdline = String.concat "␣" (List.map ThoString.quote (Array.to_list Sys.argv)) in
  let usage = Printf.sprintf "usage:␣%s␣[-help]␣[options]" my_name in
  let command = default_ref
  and arg_head_rev = ref [] in
  let checks = ref [] in

  let specs_lists =
    [ [ ("-f", Arg.Unit (fun () → list_flavors (); exit 0),
         "␣list␣all␣flavors␣and␣exit");
        ("--flavors", Arg.Unit (fun () → list_flavors (); exit 0),
         "␣list␣all␣flavors␣and␣exit");

        ("-s", Arg.String (fun s → add_scatterings command [s]),
         "process␣add␣a␣scattering␣'i1␣i2␣->␣o1␣o2␣...'");
        ("--scatter", Arg.String (fun s → add_scatterings command [s]),
         "process␣add␣a␣scattering␣'i1␣i2␣->␣o1␣o2␣...'");
        ("--scatter_file", Arg.String (fun s → add_scatterings command (read_lines_rev s)),
         "name␣add␣scattering␣lines␣'i1␣i2␣->␣o1␣o2␣...'");

        ("-d", Arg.String (fun s → add_decays command [s]),
         "process␣add␣a␣decay␣'i␣->␣o1␣o2␣...'");
        ("--decay", Arg.String (fun s → add_decays command [s]),
         "process␣add␣a␣decay␣'i␣->␣o1␣o2␣...'");
        ("--decay_file", Arg.String (fun s → add_decays command (read_lines_rev s)),
         "name␣add␣decay␣lines␣'i␣->␣o1␣o2␣...'");

        ("-r", Arg.String (fun s → add_restrictions command [s]),
         "restriction␣add␣a␣restriction");
        ("--restrictions", Arg.String (fun s → add_restrictions command [s]),
         "restriction␣add␣a␣restriction");
        ("--restrictions_file", Arg.String (fun s → add_restrictions command (read_lines_rev s)),
         "name␣add␣restrictions");

        ("-o", Arg.String (fun s → add_orders command [s]),
         "condition␣add␣a␣coupling␣order␣condition␣on␣amplitude");
        ("--orders", Arg.String (fun s → add_orders command [s]),
         "condition␣add␣a␣coupling␣order␣condition␣on␣amplitude");
        ("--orders_file", Arg.String (fun s → add_orders command (read_lines_rev s)),
         "name␣add␣coupling␣order␣conditions␣on␣amplitude");

        ("-O", Arg.Bool (set_orders2 command),
         "true|false␣coupling␣orders␣of␣|M|^2␣(default=" ^ string_of_bool !(command.orders2_ref) ^ ")");
        ("--orders2", Arg.Bool (set_orders2 command),
         "true|false␣coupling␣orders␣of␣|M|^2␣(default=" ^ string_of_bool !(command.orders2_ref) ^ ")");
```

```
          ("-u", Arg.Int (set_unphysical command),
           "n␣unphysical␣polarization␣vector␣for␣particle␣n");
          ("--unphysical", Arg.Int (set_unphysical command),
           "n␣unphysical␣polarization␣vector␣for␣particle␣n");
          ("-p", Arg.String (set_prefix command),
           "pfx␣prefix␣for␣output␣files␣(default='" ^ !(command.prefix_ref) ^ "')");
          ("--prefix", Arg.String (set_prefix command),
           "pfx␣prefix␣for␣output␣files␣(default='" ^ !(command.prefix_ref) ^ "')");
          ("--directory", Arg.String (set_directory command),
           "dir␣directory␣for␣output␣files␣(default='" ^ Filename.current_dir_name ^ "')") ];
        Amplitude.specs command.amplitude_ref;
        Options.cmdline "--model:" (purge_ufo_options M.options);
        Options.cmdline "--fusion:" CF.options;
        Options.cmdline "--target:" T.options;
        ThoList.flatmap debug
          [ ("a", "arguments", T.All, checks);
            ("n", "#␣of␣input␣arguments", T.Arguments, checks);
            ("m", "input␣momenta", T.Momenta, checks);
            ("g", "internal␣Ward␣identities", T.Gauge, checks) ];
        Log.specs command.log_ref;
        Parameters.specs command.parameters_ref;
        Phasespace.specs command.phasespace_ref;
Poles.specs command.poles_ref;
        Whizard_Model.specs command.whizard_ref;
        Forest.specs command.forest_ref;
        Diagrams.specs command.diagrams_ref;
        Colorflows.specs command.colorflows_ref;
        [ ("--latex", Arg.Set command.latex_ref, "␣wrap␣diagrams␣in␣minimal␣LaTeX") ];
        DAG.specs command.dag_ref;
        Full_DAG.specs command.full_dag_ref;
 [ ("--template", Arg.Set command.template_ref, "␣empty␣wrapper␣for␣a␣handcoded␣amplitudes")]
  ] in
```

There is no default action if the command line is empty after *Omega3* has consumed the model loading options.

```
    if Array.length argv ≤ 1 then
      begin
        prerr_endline usage;
        exit 2
      end;
```

Parse the command line.

```
    begin
      try
        Arg.parse_argv ?current argv
          (Arg.align (List.concat specs_lists))
          (fun s → arg_head_rev := s :: !arg_head_rev)
          usage
      with
      | Arg.Bad msg →
          prerr_endline msg;
          exit 2
      | Arg.Help msg →
          print_endline msg;
          exit 0
    end;
```

Collect options.

```
    let command = command_of_ref command in

    let to_output_channel ?logging write file f =
      if write then
        with_output_channel ?logging command.directory command.prefix file f in
```

Process dependent outputs make only sense if the list of processes is not empty.

```
    begin match parse_processes command.processes with
    | [] → ()
    | processes →

       let selectors = parse_restrictions processes (List.rev command.restrictions_rev)
       and orders = parse_orders (List.rev command.orders_rev) in

       let amplitudes =
         CF.amplitudes (include_goldstones !checks) command.unphysical selectors orders processes in

       to_output_channel command.amplitude.write command.amplitude.file
         (fun channel → T.amplitudes_to_channel cmdline channel !checks amplitudes);

       to_output_channel command.log.write command.log.file
         (log_to_channel cmdline amplitudes);

       to_output_channel command.phasespace.write command.phasespace.file
          (phasespace_to_channel (List.rev command.restrictions_rev) processes);

       to_output_channel command.poles.write command.poles.file
         (poles_to_channel amplitudes);

       to_output_channel command.forest.write command.forest.file
         (forest_to_channel amplitudes);

       to_output_channel command.diagrams.write command.diagrams.file
         (FMP.amplitudes_sans_color_to_channel command.latex amplitudes);

       to_output_channel command.colorflows.write command.colorflows.file
         (FMP.amplitudes_color_only_to_channel command.latex amplitudes);

       to_output_channel command.dag.write command.dag.file
         (fun channel → List.iter (F.amplitude_to_dot channel) (CF.processes amplitudes));

       to_output_channel command.full_dag.write command.full_dag.file
         (fun channel → List.iter (F.tower_to_dot channel) (CF.processes amplitudes))

    end;
```

The model dependent outputs can be written in any case.

```
    to_output_channel command.parameters.write command.parameters.file T.parameters_to_channel;
    to_output_channel command.whizard.write command.whizard.file M.write_whizard;
    ()

  end
```

24.5 Implementation of Omega3

Next generation single executable.

Ω^3: only healthy fatty acids included!

Playground for first class modules.

The following static models are still missing

- model defined in the compilation unit of the executable: `CQED`, `Littlest_Zprime`, `SM_top`, `SYM`.

```
let static_models_SM : (string × string × (module Model.T)) list =
  let open Modellib_SM in
  let pfx = "from␣Modellib_SM:␣" in
  [ ("QED", "Quantum␣Electro␣Dynamics", (module QED));
    ("QCD", "Quantum␣Chromo␣Dynamics", (module QCD));
    ("SM", "Standard␣Model␣(minimal,␣no␣CKM)", (module SM(SM_no_anomalous)));
    ("SM_CKM", pfx ^ "SM(SM_no_anomalous_ckm)", (module SM(SM_no_anomalous_ckm)));
    ("SM_Higgs", pfx ^ "SM(SM_Higgs)", (module SM(SM_Higgs)));
    ("SM_Higgs_CKM", pfx ^ "SM(SM_Higgs_CKM)", (module SM(SM_Higgs_CKM)));
    ("SM_ac", pfx ^ "SM(SM_anomalous)", (module SM(SM_anomalous)));
    ("SM_ac_CKM", pfx ^ "SM(SM_anomalous_ckm)", (module SM(SM_anomalous_ckm)));
    ("SM_top_anom", pfx ^ "SM(SM_anomalous_top)", (module SM(SM_anomalous_top)));
    ("SM_dim6", pfx ^ "SM(SM_dim6)", (module SM(SM_dim6)));
    ("SM_tt_threshold", pfx ^ "SM(SM_tt_threshold)", (module SM(SM_tt_threshold)));
    ("SM_ul", pfx ^ "SM(SM_k_matrix)", (module SM(SM_k_matrix)));
    ("SM_rx", pfx ^ "SM(SM_k_matrix)␣=␣SM_ul␣with␣fewer␣parameters␣in␣Whizard", (module SM(SM_k_matrix)));
    ("SM_Rxi", pfx ^ "SM_Rxi", (module SM_Rxi));
    ("SM_clones", pfx ^ "SM_clones", (module SM_clones));
    ("Phi3", "phi^3␣toy␣model", (module Phi3));
    ("Phi4", "phi^3␣+␣phi^4␣toy␣model", (module Phi4)) ]

let static_models_BSM : (string × string × (module Model.T)) list =
  let open Modellib_BSM in
  let pfx = "from␣Modellib_BSM:␣" in
  [ ("THDM", pfx ^ "TwoHiggsDoublet(THDM)", (module TwoHiggsDoublet(THDM)));
    ("THDM_CKM", pfx ^ "TwoHiggsDoublet(THDM_CKM)", (module TwoHiggsDoublet(THDM_CKM)));
    ("GravTest", pfx ^ "GravTest(BSM_bsm)", (module GravTest(BSM_bsm)));
    ("HSExt", pfx ^ "HSExt(BSM_bsm)", (module HSExt(BSM_bsm)));
    ("Littlest", pfx ^ "Littlest(BSM_bsm)", (module Littlest(BSM_bsm)));
    ("Littlest_Eta", pfx ^ "Littlest(BSM_ungauged)", (module Littlest(BSM_ungauged)));
    ("Littlest_Tpar", pfx ^ "(Littlest_Tpar(BSM_bsm))", (module (Littlest_Tpar(BSM_bsm))));
    ("Simplest", pfx ^ "Simplest(BSM_bsm)", (module Simplest(BSM_bsm)));
    ("Simplest_univ", pfx ^ "Simplest(BSM_anom)", (module Simplest(BSM_anom)));
    ("SSC", pfx ^ "SSC(SSC_kmatrix)", (module SSC(SSC_kmatrix)));
    ("SSC_2", pfx ^ "SSC(SSC_kmatrix_2)", (module SSC(SSC_kmatrix_2)));
    ("SSC_AltT", pfx ^ "SSC_AltT(SSC_kmatrix_2)", (module SSC_AltT(SSC_kmatrix_2)));
    ("Template", pfx ^ "Template(BSM_bsm)", (module Template(BSM_bsm)));
    ("Threeshl", pfx ^ "Threeshl(Threeshl_no_ckm)", (module Threeshl(Threeshl_no_ckm)));
    ("Threeshl_nohf", pfx ^ "Threeshl(Threeshl_no_ckm_no_hf)", (module Threeshl(Threeshl_no_ckm_no_hf)));
    ("UED", pfx ^ "UED(BSM_bsm)", (module UED(BSM_bsm)));
    ("Xdim", pfx ^ "Xdim(BSM_bsm)", (module Xdim(BSM_bsm))) ]

let static_models_MSSM : (string × string × (module Model.T)) list =
  let open Modellib_MSSM in
  let pfx = "from␣Modellib_MSSM:␣" in
  [ ("MSSM", pfx ^ "MSSM(MSSM_no_4)", (module MSSM(MSSM_no_4)));
    ("MSSM_CKM", pfx ^ "MSSM(MSSM_no_4_ckm)", (module MSSM(MSSM_no_4_ckm)));
    ("MSSM_Hgg", pfx ^ "MSSM(MSSM_Hgg)", (module MSSM(MSSM_Hgg)));
    ("MSSM_Grav", pfx ^ "MSSM(MSSM_Grav)", (module MSSM(MSSM_Grav))) ]

let static_models_NMSSM : (string × string × (module Model.T)) list =
  let open Modellib_NMSSM in
  let pfx = "from␣Modellib_NMSSM:␣" in
  [ ("NMSSM", pfx ^ "NMSSM_func(NMSSM)", (module NMSSM_func(NMSSM)));
    ("NMSSM_CKM", pfx ^ "NMSSM_func(NMSSM_CKM)", (module NMSSM_func(NMSSM_CKM)));
    ("NMSSM_Hgg", pfx ^ "NMSSM_func(NMSSM_Hgg)", (module NMSSM_func(NMSSM_Hgg))) ]

let static_models_NoH : (string × string × (module Model.T)) list =
  let open Modellib_NoH in
  let pfx = "from␣Modellib_NoH:␣" in
  [ ("AltH", pfx ^ "AltH(NoH_k_matrix)", (module AltH(NoH_k_matrix)));
    ("NoH_rx", pfx ^ "NoH(NoH_k_matrix)", (module NoH(NoH_k_matrix))) ]
```

```
let static_models_other : (string × string × (module Model.T)) list =
  let module Zprime = Modellib_Zprime in
  let module PSSSM = Modellib_PSSSM in
  let module WZW = Modellib_WZW in
  let pfx s = "from␣Modellib_" ^ s ^ ":␣" in
  [ ("Zprime", pfx "Zprime"^"Zprime.Zprime(Zprime.SM_no_anomalous)", (module Zprime.Zprime(Zprime.SM_no_anom
    ("PSSSM", pfx "PSSSM"^"PSSSM.ExtMSSM(PSSSM.PSSSM)", (module PSSSM.ExtMSSM(PSSSM.PSSSM)));
    ("WZW", pfx "WZW"^"WZW.WZW(WZW.SM_no_anomalous)", (module WZW.WZW(WZW.SM_no_anomalous))) ]
```

```
let static_models =
  Omega_cli.Models.of_list
    (List.concat [ static_models_SM;
                   static_models_BSM;
                   static_models_MSSM;
                   static_models_NMSSM;
                   static_models_NoH;
                   static_models_other ])
```

```
let list_models () =
  List.iter
    (fun (name, description) → Printf.printf "%s␣:␣%s\n" name description)
    (Omega_cli.Models.names static_models)
```

```
type model =
  | Static_Model of string
  | UFO_Model of string
```

```
let load_model ?(flags = []) = function
  | Static_Model name →
      begin match Omega_cli.Models.by_name_opt static_models name with
      | Some (module S) → (module Modeltools.Static(S) : Model.Mutable)
      | None → invalid_arg (Printf.sprintf "omega:␣static␣model␣'%s'␣not␣found!" name)
      end
  | UFO_Model directory →
      let (module U) = (module UFO.Model : Model.Mutable with type init = string × string list) in
      U.init (directory, flags);
      (module U : Model.Mutable)
```

Check if the model M contains Majorana fermions. In the case of UFO, this can only be used *after* the UFO model has been loaded with *M.init dir*, of course!

```
let needs_majorana (module M : Model.T) =
  List.exists (fun f → M.fermion f = 2) (M.flavors ())
```

Interface to the old CLI module *Omega* for testing the first class modules code before implementing the new *Omega_cli*.

```
module Legacy =
  struct
```

Match a model without Majorana fermions and a target to a topology.

```
    let dirac (module T : Target.Maker) (module M : Model.Mutable) =
      let n = M.max_degree () in
      if n > 4 then
        (module (Omega.Nary(T)(M)) : Omega_cli.T)
      else if n = 4 then
        (module (Omega.Mixed23(T)(M)) : Omega_cli.T)
      else if n = 3 then
        (module (Omega.Binary(T)(M)) : Omega_cli.T)
      else
        invalid_arg "Omega3.Legacy.dirac:␣max_degree␣<␣3"
```

Match a model containing Majorana fermions and a target to a topology.

```
    let majorana (module T : Target.Maker) (module M : Model.Mutable) =
      let n = M.max_degree () in
      if n > 4 then
        (module (Omega.Nary_Majorana(T)(M)) : Omega_cli.T)
      else if n = 4 then
        (module (Omega.Mixed23_Majorana(T)(M)) : Omega_cli.T)
      else if n = 3 then
        (module (Omega.Binary_Majorana(T)(M)) : Omega_cli.T)
      else
        invalid_arg "Omega3.Legacy.majorana:␣max_degree␣<␣3"
```

Match a model containing Majorana fermions and a target to a topology using the old implementation.

```
  let vintage_majorana (module T : Target.Maker) (module M : Model.Mutable) =
    let n = M.max_degree () in
    if n = 3 ∨ n = 4 then
      (module (Omega.Mixed23_Majorana_vintage(T)(M)) : Omega_cli.T)
    else if n > 4 then
      invalid_arg "Omega3.Legacy.vintage_majorana:␣max_degree␣>␣4"
    else
      invalid_arg "Omega3.Legacy.vintage_majorana:␣max_degree␣<␣3"

  let fortran ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
    if force_vintage_majorana then
      vintage_majorana (module Target_Fortran.Make_Majorana) (module M)
    else if force_majorana ∨ needs_majorana (module M) then
      majorana (module Target_Fortran.Make_Majorana) (module M)
    else
      dirac (module Target_Fortran.Make) (module M)

  let vm ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
    if force_vintage_majorana ∨ force_majorana ∨ needs_majorana (module M) then
      invalid_arg "Omega3.Legacy.vm:␣Majorana␣fermions␣not␣yet␣supported␣by␣the␣virtual␣machine"
    else
      dirac (module Target_VM.Make) (module M)

  let adjoin_target ?force_majorana ?force_vintage_majorana (module M : Model.Mutable) name =
    match String.lowercase_ascii name with
    | "fortran" → fortran ?force_majorana ?force_vintage_majorana (module M)
    | "vm" → vm ?force_majorana ?force_vintage_majorana (module M)
    | _ → invalid_arg (Printf.sprintf "omega:␣target␣'%s'␣not␣found!" name)

  let load_omega ?flags ?force_majorana ?force_vintage_majorana target model =
    adjoin_target ?force_majorana ?force_vintage_majorana (load_model ?flags model) target

end

module Bound (M : Model.T) : Tuple.Bound =
  struct
    let max_arity () = pred (M.max_degree ())
  end

module V3 =
  struct

    module CLI = Omega_cli.Make
```

Match a model without Majorana fermions and a target to a topology.

```
    let dirac (module T : Target.Maker) (module M : Model.Mutable) =
      let open Fusion in
      let n = M.max_degree () in
      if n > 4 then
        (module (CLI(Nary(Bound(M)))(Helac(Bound(M)))(T)(M)) : Omega_cli.T)
      else if n = 4 then
        (module (CLI(Mixed23)(Helac_Mixed23)(T)(M)) : Omega_cli.T)
      else if n = 3 then
        (module (CLI(Binary)(Helac_Binary)(T)(M)) : Omega_cli.T)
      else
        invalid_arg "Omega3.V3.dirac:␣max_degree␣<␣3"

    let dirac_helac (module T : Target.Maker) (module M : Model.Mutable) =
      let open Fusion in
      let n = M.max_degree () in
      if n > 4 then
        (module (CLI(Helac(Bound(M)))(Helac(Bound(M)))(T)(M)) : Omega_cli.T)
      else if n = 4 then
        (module (CLI(Helac_Mixed23)(Helac_Mixed23)(T)(M)) : Omega_cli.T)
      else if n = 3 then
        (module (CLI(Helac_Binary)(Helac_Binary)(T)(M)) : Omega_cli.T)
      else
        invalid_arg "Omega3.V3.dirac_helac:␣max_degree␣<␣3"
```

Match a model containing Majorana fermions and a target to a topology.

```
let majorana (module T : Target.Maker) (module M : Model.Mutable) =
  let open Fusion in
  let n = M.max_degree () in
  if n > 4 then
    (module (CLI(Nary_Majorana(Bound(M)))(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else if n = 4 then
    (module (CLI(Mixed23_Majorana)(Helac_Mixed23_Majorana)(T)(M)) : Omega_cli.T)
  else if n = 3 then
    (module (CLI(Binary_Majorana)(Helac_Binary_Majorana)(T)(M)) : Omega_cli.T)
  else
    invalid_arg "Omega3.V3.majorana:␣max_degree␣<␣3"

let majorana_helac (module T : Target.Maker) (module M : Model.Mutable) =
  let open Fusion in
  let n = M.max_degree () in
  if n > 4 then
    (module (CLI(Helac_Majorana(Bound(M)))(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else if n = 4 then
    (module (CLI(Helac_Mixed23_Majorana)(Helac_Mixed23_Majorana)(T)(M)) : Omega_cli.T)
  else if n = 3 then
    (module (CLI(Helac_Binary_Majorana)(Helac_Binary_Majorana)(T)(M)) : Omega_cli.T)
  else
    invalid_arg "Omega3.V3.majorana_helac:␣max_degree␣<␣3"
```

Match a model containing Majorana fermions and a target to a topology using the old implementation.

```
let vintage_majorana (module T : Target.Maker) (module M : Model.Mutable) =
  let open Fusion_vintage in
  let n = M.max_degree () in
  if n > 4 then
    (module (CLI(Nary_Majorana(Bound(M)))(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else if n = 4 then
    (module (CLI(Mixed23_Majorana)(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else if n = 3 then
    (module (CLI(Binary_Majorana)(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else
    invalid_arg "Omega3.V3.vintage_majorana:␣max_degree␣<␣3"

let vintage_majorana_helac (module T : Target.Maker) (module M : Model.Mutable) =
  let open Fusion_vintage in
  let n = M.max_degree () in
  if n > 2 then
    (module (CLI(Nary_Majorana(Bound(M)))(Helac_Majorana(Bound(M)))(T)(M)) : Omega_cli.T)
  else
    invalid_arg "Omega3.V3.vintage_majorana_helac:␣max_degree␣<␣3"

let fortran ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
  if force_vintage_majorana then
    vintage_majorana (module Target_Fortran.Make_Majorana) (module M)
```

```
        else if force_majorana ∨ needs_majorana (module M) then
          majorana (module Target_Fortran.Make_Majorana) (module M)
        else
          dirac (module Target_Fortran.Make) (module M)

    let fortran_helac ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
      if force_vintage_majorana then
        vintage_majorana_helac (module Target_Fortran.Make_Majorana) (module M)
      else if force_majorana ∨ needs_majorana (module M) then
        majorana_helac (module Target_Fortran.Make_Majorana) (module M)
      else
        dirac_helac (module Target_Fortran.Make) (module M)

    let vm ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
      if force_vintage_majorana ∨ force_majorana ∨ needs_majorana (module M) then
        invalid_arg "Omega3.V3.vm:␣Majorana␣fermions␣not␣yet␣supported␣by␣the␣virtual␣machine"
      else
        dirac (module Target_VM.Make) (module M)

    let vm_helac ?(force_vintage_majorana =false) ?(force_majorana =false) (module M : Model.Mutable) =
      if force_vintage_majorana ∨ force_majorana ∨ needs_majorana (module M) then
        invalid_arg "Omega3.V3.vm_helac:␣Majorana␣fermions␣not␣yet␣supported␣by␣the␣virtual␣machine"
      else
        dirac_helac (module Target_VM.Make) (module M)

    let adjoin_target ?(helac =false) ?force_majorana ?force_vintage_majorana (module M : Model.Mutable) name =
      match String.lowercase_ascii name with
      | "fortran" →
          if helac then
            fortran_helac ?force_majorana ?force_vintage_majorana (module M)
          else
            fortran ?force_majorana ?force_vintage_majorana (module M)
      | "vm" →
          if helac then
            vm_helac ?force_majorana ?force_vintage_majorana (module M)
          else
            vm ?force_majorana ?force_vintage_majorana (module M)
      | _ → invalid_arg (Printf.sprintf "omega:␣target␣'%s'␣not␣found!" name)

    let load_omega ?helac ?flags ?force_majorana ?force_vintage_majorana target model =
      adjoin_target ?helac ?force_majorana ?force_vintage_majorana (load_model ?flags model) target

  end
```

This is the first part of the command line processing. Interpret the options up to "--" to load a model (static or UFO) and a target. Then dispatch the rest of the command line to the old (*Omega.Make*().*main*, selected by "--legacy") main program or the new one (*Omega_cli.Make*().*main*, selected by "--v3" or by default).

For static models, the old command line interface should work in exactly the same way as the single executables. For UFO models, some options in the old interface will not work, due to the new loading sequence.

```
let list_targets () =
  List.iter print_endline ["fortran"; "ovm"]

type mode =
  | V3
  | Legacy

let default_static_model_name = "SM"
let default_target_name = "fortran"

let _ =
  let argv0 = Sys.argv.(0) in
  let usage = "usage:␣" ^ argv0 ^ "␣[-help]␣[options]"
  and mode = ref V3
  and arg_head_rev = ref []
  and arg_tail_rev = ref []
  and ufo_debug = ref []
  and model = ref (Static_Model default_static_model_name)
  and target_name = ref default_target_name
  and force_majorana = ref None
  and force_vintage_majorana = ref None
  and helac = ref None in
  Arg.parse
    (Arg.align
       [ ( "-M", Arg.String (fun s → model := Static_Model s),
           "model␣select␣static␣model␣(default='" ^ default_static_model_name ^ "')");
         ( "--model", Arg.String (fun s → model := Static_Model s),
           "model␣select␣static␣model␣(default='" ^ default_static_model_name ^ "')");
         ( "--model_list", Arg.Unit list_models, "␣list␣all␣available␣static␣models");
         ( "-U", Arg.String (fun s → model := UFO_Model s),
           "dir␣select␣UFO␣and␣read␣from␣directory");
         ( "--ufo_directory", Arg.String (fun s → model := UFO_Model s),
           "dir␣select␣UFO␣and␣read␣from␣directory");
         ( "--ufo_debug", Arg.String (fun s → ufo_debug := s :: !ufo_debug),
           "flag␣add␣UFO␣debug␣flags␣(undocumented)");
         ( "-T", Arg.String ((:=) target_name), "target␣select␣target␣(default='" ^ !target_name ^ "')");
         ( "--target", Arg.String ((:=) target_name), "target␣select␣target␣(default='" ^ !target_name ^ "')");
         ( "--target_list", Arg.Unit list_targets, "␣list␣all␣available␣targets");
         ( "--majorana", Arg.Unit (fun () → force_majorana := Some true),
           "␣use␣Majorana␣spinors␣even␣if␣not␣needed");
         ( "--vintage_majorana", Arg.Unit (fun () → force_vintage_majorana := Some true),
           "␣use␣the␣original␣implementation␣of␣Majorana␣spinors");
         ( "--helac", Arg.Unit (fun () → helac := Some true),
           "␣use␣asymmetrical␣topologies␣like␣HELAC");
         ( "--v3", Arg.Unit (fun () → mode := V3), "␣use␣the␣new␣omega␣CLI,␣version␣3␣(default)");
         ( "--legacy", Arg.Unit (fun () → mode := Legacy), "␣use␣the␣historically␣grown␣omega␣CLI");
         ( "--", Arg.Rest (fun s → arg_tail_rev := s :: !arg_tail_rev),
           "␣pass␣remaining␣options␣to␣the␣selected␣omega␣CLI") ])
    (fun s → arg_head_rev := s :: !arg_head_rev)
    usage;
  let arg_head = List.rev !arg_head_rev
  and arg_tail = List.rev !arg_tail_rev in
  begin match arg_head with
  | [] → ()
  | args → Printf.eprintf "omega3:␣ignoring␣options␣before␣--:␣%s\n" (String.concat "␣" args)
  end;
  let force_majorana = !force_majorana
  and force_vintage_majorana = !force_vintage_majorana
  and helac = !helac
  and flags =
    match !ufo_debug with
    | [] → None
    | flags → Some flags in
  let (module O) =
    match !mode with
    | Legacy → Legacy.load_omega ?flags ?force_majorana ?force_vintage_majorana !target_name !model
    | V3 → V3.load_omega ?flags ?helac ?force_majorana ?force_vintage_majorana !target_name !model in
  let current = ref 0
  and argv = Array.of_list (argv0 :: arg_tail) in
  O.main ~current ~argv ()
```

24.6 Implementation of Omega_QED

```
module O = Omega.Binary(Target_Fortran.Make)(Modellib_SM.QED)
let _ = O.main ()
```

24.7 Implementation of Omega_SM

```
module O = Omega.Mixed23(Target_Fortran.Make)(Modellib_SM.SM(Modellib_SM.SM_no_anomalous))
let _ = O.main ()
```

24.8 Implementation of Omega_SYM

```
module SYM =
  struct

    open Coupling

    let options = Options.empty
    let caveats () = []

    let nc = 3

    type flavor =
      | Q of int | SQ of int
      | G of int | SG of int
      | Phi

    let generations = ThoList.range 1 1

    let generations_pairs =
      List.map
        (function [a; b] → (a, b)
          | _ → failwith "omega_SYM.generations_pairs")
        (Product.power 2 generations)

    let generations_triples =
      List.map
        (function [a; b; c] → (a, b, c)
          | _ → failwith "omega_SYM.generations_triples")
        (Product.power 3 generations)

    let generations_quadruples =
      List.map
        (function [a; b; c; d] → (a, b, c, d)
          | _ → failwith "omega_SYM.generations_quadruples")
        (Product.power 4 generations)

    let external_flavors () =
      [ "Quarks", List.map (fun i → Q i) generations;
        "Anti-Quarks", List.map (fun i → Q (−i)) generations;
        "SQuarks", List.map (fun i → SQ i) generations;
        "Anti-SQuarks", List.map (fun i → SQ (−i)) generations;
        "Gluons", List.map (fun i → G i) generations;
        "SGluons", List.map (fun i → SG i) generations;
        "Other", [Phi]]

    let flavors () =
      ThoList.flatmap snd (external_flavors ())

    type gauge = unit
    type constant =
      | G_saa of int × int
      | G_saaa of int × int × int
      | G3 of int × int × int
      | I_G3 of int × int × int
      | G4 of int × int × int × int

    type coupling_order = unit
    let all_coupling_orders () = [()]
    let coupling_order_to_string () = ""
    let coupling_orders = function
      | _ → failwith "Modellib_SYM.orders:␣not␣implemented␣yet!"

    let lorentz = function
      | Q i →
          if i > 0 then
            Spinor
          else if i < 0 then
            ConjSpinor
          else
            invalid_arg "SYM.lorentz␣(Q␣0)"
      | SQ _ | Phi → Scalar
      | G _ → Vector
      | SG _ → Majorana

    let color = function
      | Q i | SQ i →
          Color.SUN (if i > 0 then nc else if i < 0 then -nc else invalid_arg "SYM.color␣(Q␣0)")
      | G _ | SG _ → Color.AdjSUN nc
      | Phi → Color.Singlet

    let nc () = nc

    let propagator = function
      | Q i →
          if i > 0 then
            Prop_Spinor
          else if i < 0 then
            Prop_ConjSpinor
          else
            invalid_arg "SYM.lorentz␣(Q␣0)"
      | SQ _ | Phi → Prop_Scalar
      | G _ → Prop_Feynman
      | SG _ → Prop_Majorana

    let width _ = Timelike
    let goldstone _ = None

    let conjugate = function
      | Q i → Q (−i)
      | SQ i → SQ (−i)
      | (G _ | SG _ | Phi) as p → p

    let fermion = function
      | Q i →
          if i > 0 then
            1
          else if i < 0 then
            -1
          else
            invalid_arg "SYM.fermion␣(Q␣0)"
      | SQ _ | G _ | Phi → 0
      | SG _ → 2

    module Ch = Charges.Null
    let charges _ = ()

    module F = Modeltools.Fusions (struct
      type f = flavor
      type c = constant
      let compare = compare
      let conjugate = conjugate
    end)

    let quark_current =
      List.map
        (fun (i, j, k) →
          ((Q (-i), G j, Q k), FBF (-1, Psibar, V, Psi), G3 (i, j, k)))
        generations_triples

    let squark_current =
      List.map
        (fun (i, j, k) →
          ((G j, SQ i, SQ (-k)), Vector_Scalar_Scalar 1, G3 (i, j, k)))
        generations_triples

    let three_gluon =
      List.map
        (fun (i, j, k) →
          ((G i, G j, G k), Gauge_Gauge_Gauge 1, I_G3 (i, j, k)))
        generations_triples

    let gluon2_phi =
      List.map
        (fun (i, j) →
          ((Phi, G i, G j), Dim5_Scalar_Gauge2 1, G_saa (i, j)))
        generations_pairs

    let vertices3 =
      quark_current @ squark_current @ three_gluon @ gluon2_phi

    let gauge4 = Vector4 [(2, C_13_42); (-1, C_12_34); (-1, C_14_23)]

    let squark_seagull =
      List.map
        (fun (i, j, k, l) →
          ((SQ i, SQ (-j), G k, G l), Scalar2_Vector2 1, G4 (i, j, k, l)))
       generations_quadruples

    let four_gluon =
      List.map
        (fun (i, j, k, l) →
          ((G i, G j, G k, G l), gauge4, G4 (i, j, k, l)))
       generations_quadruples
```

We need at least a *Dim6_Scalar_Gauge3* vertex to support this.

```
  let gluon3_phi =
    []

  let vertices4 =
    squark_seagull @ four_gluon @ gluon3_phi

  let vertices () =
    (vertices3, vertices4, [])

  let table = F.of_vertices (vertices ())
  let fuse2 = F.fuse2 table
  let fuse3 = F.fuse3 table
  let fuse = F.fuse table
  let max_degree () = 4

  let parameters () = { input = []; derived = []; derived_arrays = [] }

  let invalid_flavor s =
    invalid_arg ("omega_SYM.flavor_of_string:␣" ^ s)

  let flavor_of_string s =
    let l = String.length s in
    if l < 2 then
      invalid_flavor s
    else if l = 2 then
      if String.sub s 0 1 = "q" then
        Q (int_of_string (String.sub s 1 1))
      else if String.sub s 0 1 = "Q" then
        Q (- (int_of_string (String.sub s 1 1)))
      else if String.sub s 0 1 = "g" then
        G (int_of_string (String.sub s 1 1))
      else
        invalid_flavor s
    else if l = 3 then
      if s = "phi" then
        Phi
      else if String.sub s 0 2 = "sq" then
        SQ (int_of_string (String.sub s 2 1))
      else if String.sub s 0 2 = "sQ" then
        SQ (- (int_of_string (String.sub s 2 1)))
      else if String.sub s 0 2 = "sg" then
        SG (int_of_string (String.sub s 2 1))
      else
        invalid_flavor s
    else
      invalid_flavor s
let flavor_to_string = function
  | Q i ->
      if i > 0 then
        "q" ^ string_of_int i
      else if i < 0 then
        "Q" ^ string_of_int (-i)
      else
        invalid_arg "SYM.flavor_to_string␣(Q␣0)"
  | SQ i ->
      if i > 0 then
        "sq" ^ string_of_int i
      else if i < 0 then
        "sQ" ^ string_of_int (-i)
      else
        invalid_arg "SYM.flavor_to_string␣(SQ␣0)"
  | G i -> "g" ^ string_of_int i
  | SG i -> "sg" ^ string_of_int i
  | Phi -> "phi"
let flavor_to_TeX = function
  | Q i ->
      if i > 0 then
        "q_{" ^ string_of_int i ^ "}"
      else if i < 0 then
        "{\bar␣q}_{" ^ string_of_int (-i) ^ "}"
      else
        invalid_arg "SYM.flavor_to_string␣(Q␣0)"
  | SQ i ->
      if i > 0 then
        "{\tilde␣q}_{" ^ string_of_int i ^ "}"
      else if i < 0 then
        "{\bar{\tilde␣q}}_{" ^ string_of_int (-i) ^ "}"
      else
        invalid_arg "SYM.flavor_to_string␣(SQ␣0)"
  | G i -> "g_{" ^ string_of_int i ^ "}"
  | SG i -> "{\tilde␣g}_{" ^ string_of_int i ^ "}"
  | Phi -> "phi"
let flavor_symbol = function
  | Q i ->
      if i > 0 then
        "q" ^ string_of_int i
      else if i < 0 then
```

```
            "qbar" ^ string_of_int (-i)
          else
            invalid_arg "SYM.flavor_to_string␣(Q␣0)"
      | SQ i →
          if i > 0 then
            "sq" ^ string_of_int i
          else if i < 0 then
            "sqbar" ^ string_of_int (-i)
          else
            invalid_arg "SYM.flavor_to_string␣(SQ␣0)"
      | G i → "g" ^ string_of_int i
      | SG i → "sg" ^ string_of_int i
      | Phi → "phi"

    let gauge_symbol () =
      failwith "omega_SYM.gauge_symbol:␣internal␣error"

    let pdg _ = 0
    let mass_symbol _ = "0.0_default"
    let width_symbol _ = "0.0_default"

    let string_of_int_list int_list =
      "(" ^ String.concat "," (List.map string_of_int int_list) ^ ")"

    let constant_symbol = function
      | G_saa (i, j) → "g_saa" ^ string_of_int_list [i; j]
      | G_saaa (i, j, k) → "g_saaa" ^ string_of_int_list [i; j; k]
      | G3 (i, j, k) → "g3" ^ string_of_int_list [i; j; k]
      | I_G3 (i, j, k) → "ig3" ^ string_of_int_list [i; j; k]
      | G4 (i, j, k, l) → "g4" ^ string_of_int_list [i; j; k; l]

  end

module O = Omega.Mixed23(Target_Fortran.Make_Majorana)(SYM)
let _ = O.main ()
```

Acknowledgements

We thank Mauro Moretti for fruitful discussions of the ALPHA algorithm [1], that inspired our solution of the double counting problem.

We thank Wolfgang Kilian for providing the WHIZARD environment that turns our numbers into real events with unit weight. Thanks to the ECFA/DESY workshops and their participants for providing a showcase. Thanks to Edward Boos for discussions in Kaluza-Klein gravitons.

This research is supported by Bundesministerium für Bildung und Forschung, Germany, (05 HT9RDA) and Deutsche Forschungsgemeinschaft (MA 676/6-1).

Thanks to the Caml and Objective Caml teams from INRIA for the development and the lean and mean implementation of a programming language that does not insult the programmer's intelligence.

Bibliography

[1] F. Caravaglios, M. Moretti, Z. Phys. **C74** (1997) 291.

[2] A. Kanaki, C. Papadopoulos, DEMO-HEP-2000/01, hep-ph/0002082, February 2000.

[3] Xavier Leroy, *The Objective Caml system, documentation and user's guide*, Technical Report, INRIA, 1997.

[4] Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

[5] H. Murayama, I. Watanabe, K. Hagiwara, KEK Report 91-11, January 1992.

[6] T. Stelzer, W.F. Long, Comput. Phys. Commun. **81** (1994) 357.

[7] A. Denner, H. Eck, O. Hahn and J. Küblbeck, Phys. Lett. **B291** (1992) 278; Nucl. Phys. **B387** (1992) 467.

[8] V. Barger, A. L. Stange, R. J. N. Phillips, Phys. Rev. **D45**, (1992) 1751.

[9] T. Ohl, *Lord of the Rings*, (Computer algebra library for O'Caml, unpublished).

[10] T. Ohl, *Bocages*, (Feynman diagram library for O'Caml, unpublished).

[11] W. Kilian, *WHIZARD*, University of Karlsruhe, 2000.

[12] T. Ohl, Comput. Phys. Commun. **90** (1995), 340-354 doi:10.1016/0010-4655(95)90137-S [arXiv:hep-ph/9505351 [hep-ph]].

[13] E. E. Boos, T. Ohl, Phys. Rev. Lett. **83** (1999) 480.

[14] T. Han, J. D. Lykken and R. Zhang, Phys. Rev. **D59** (1999) 105006 [hep-ph/9811350].

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Second Edition, Cambridge University Press, 1992.

[16] P. Cvitanović, Phys. Rev. **D14** (1976) 1536.

[17] W. Kilian, T. Ohl, J. Reuter and C. Speckner, JHEP **1210** (2012) 022 [arXiv:1206.3700 [hep-ph]].

[18] C. Degrande, C. Duhr, B. Fuks, D. Grellscheid, O. Mattelaer and T. Reiter, Comput. Phys. Commun. **183** (2012), 1201-1214 doi:10.1016/j.cpc.2012.01.022 [arXiv:1108.2040 [hep-ph]].

[19] Donald E. Knuth, *The Art of Computer Programming. 2: Seminumerical algorithms*

—A— Autotools

A.1 Interface of Config

val *version* : *string*
val *date* : *string*
val *status* : *string*

val *default_UFO_dir* : *string*

val *openmp* : *bool*

Implementation `config.ml` *unavailable!*

—B— Textual Options

B.1 Interface of Options

```
type t

val empty : t
val create : (string × Arg.spec × string) list → t

val exclude : (string → bool) → t → t
val extend : t → (string × Arg.spec × string) list → t

val cmdline : string → t → (string × Arg.spec × string) list
```

This is a clone of *Arg.parse* with a delayed usage string.

```
val parse : ?current :int ref → ?argv :string array →
  (string × Arg.spec × string) list →
  (string → unit) → (unit → string) → unit
```

B.2 Implementation of Options

```
module A = Map.Make(String)

type t =
    { actions : Arg.spec A.t;
      raw : (string × Arg.spec × string) list }

let empty = { actions = A.empty; raw = [] }

let extend old options =
  { actions = List.fold_left
      (fun a (s, f, _) → A.add s f a) old.actions options;
    raw = options @ old.raw }

let create = extend empty

let exclude f options =
  { actions = A.filter (fun o _ → ¬ (f o)) options.actions;
    raw = List.filter (fun (o, _, _) → ¬ (f o)) options.raw }

let cmdline prefix options =
  List.map (fun (o, f, d) → (prefix ˆ o, f, d)) options.raw
```

Starting with O'Caml version 3.12.1 we can provide a better help* option using *Arg.usage_string*. We can finally do this!

```
let parse ?current ?(argv = Sys.argv) specs anonymous usage =
  let help () =
    raise (Arg.Help (usage ())) in
  let specs =
    [("-usage", Arg.Unit help, "␣display␣the␣external␣particles");
     ("--usage", Arg.Unit help, "display␣the␣external␣particles")] @ specs in
```

```
try
  Arg.parse_argv ?current argv specs anonymous (usage ())
with
| Arg.Bad msg → Printf.eprintf "%s\n" msg; exit 2;
| Arg.Help msg → Printf.printf "%s\n" msg; exit 0
```

—C— Progress Reports

C.1 Interface of Progress

```
type t

val dummy : t
val channel : out_channel → int → t
val file : string → int → t
val open_file : string → int → t
val reset : t → int → string → unit
val begin_step : t → string → unit
val end_step : t → string → unit
val summary : t → string → unit
```

C.2 Implementation of Progress

```
type channel =
  | Channel of out_channel
  | File of string
  | Open_File of string × out_channel

type state =
    { channel : channel;
      mutable steps : int;
      mutable digits : int;
      mutable step : int;
      created : float;
      mutable last_reset : float;
      mutable last_begin : float; }

type t = state option

let digits n =
  if n > 0 then
    succ (truncate (log10 (float n)))
  else
    invalid_arg "Progress.digits:␣non-positive␣argument"

let mod_float2 a b =
  let modulus = mod_float a b in
  ((a −. modulus) /. b, modulus)

let time_to_string seconds =
  let minutes, seconds = mod_float2 seconds 60. in
  if minutes > 0.0 then
    let hours, minutes = mod_float2 minutes 60. in
    if hours > 0.0 then
      let days, hours = mod_float2 hours 24. in
      if days > 0.0 then
        Printf.sprintf "%.0f:%02.0f␣days" days hours
      else
        Printf.sprintf "%.0f:%02.0f␣hrs" hours minutes
    else
      Printf.sprintf "%.0f:%02.0f␣mins" minutes seconds
  else
    Printf.sprintf "%.2f␣secs" seconds

let create channel steps =
  let now = Sys.time () in
  Some { channel = channel;
         steps = steps;
         digits = digits steps;
         step = 0;
         created = now;
         last_reset = now;
         last_begin = now }

let dummy =
  None

let channel oc =
  create (Channel oc)

let file name =
  let oc = open_out name in
  close_out oc;
  create (File name)

let open_file name =
  let oc = open_out name in
  create (Open_File (name, oc))

let close_channel state =
  match state.channel with
  | Channel oc →
      flush oc
  | File _ → ()
  | Open_File (_, oc) →
      flush oc;
      close_out oc

let use_channel state f =
  match state.channel with
  | Channel oc | Open_File (_, oc) →
      f oc;
      flush oc
  | File name →
      let oc = open_out_gen [Open_append; Open_creat] 644₈ name in
      f oc;
      flush oc;
      close_out oc

let reset state steps msg =
  match state with
  | None → ()
  | Some state →
      let now = Sys.time () in
      state.steps ← steps;
      state.digits ← digits steps;
      state.step ← 0;
      state.last_reset ← now;
      state.last_begin ← now

let begin_step state msg =
  match state with
  | None → ()
  | Some state →
      let now = Sys.time () in
      state.step ← succ state.step;
      state.last_begin ← now;
      use_channel state (fun oc →
        Printf.fprintf oc "[%0*d/%0*d]␣%s␣..." state.digits state.step state.digits state.steps msg)

let end_step state msg =
  match state with
  | None → ()
  | Some state →
      let now = Sys.time () in
      let last = now −. state.last_begin in
      let elapsed = now −. state.last_reset in
      let estimated = float state.steps ∗. elapsed /. float state.step in
      let remaining = estimated −. elapsed in
      use_channel state (fun oc →
        Printf.fprintf oc "␣%s.␣[time:␣%s,␣total:␣%s,␣remaining:␣%s]\n" msg
          (time_to_string last) (time_to_string estimated) (time_to_string remaining))

let summary state msg =
  match state with
  | None → ()
  | Some state →
      let now = Sys.time () in
      use_channel state (fun oc →
        Printf.fprintf oc "%s.␣[total␣time:␣%s]\n" msg
          (time_to_string (now −. state.created)));
      close_channel state
```

—D— More on Filenames

D.1 Interface of ThoFilename

```
val split : string → string list
val join : string list → string

val expand_home : string → string
```

D.2 Implementation of ThoFilename

```
let rec split' acc path =
  match Filename.dirname path, Filename.basename path with
  | "/", basename → "/" :: basename :: acc
  | ".", basename → basename :: acc
  | dirname, basename → split' (basename :: acc) dirname

let split path =
  split' [] path

let join = function
  | [] → "."
  | [basename] → basename
  | dirname :: rest → List.fold_left Filename.concat dirname rest

let expand_home path =
  match split path with
  | ("~" | "$HOME" | "${HOME}") :: rest →
      join ((try Sys.getenv "HOME" with Not_found → "/tmp") :: rest)
  | _ → path
```

—E—
Cache Files

E.1 Interface of Cache

```
module type T =
  sig

    type key
    type hash = string
    type value

    type α result =
      | Hit of α
      | Miss
      | Stale of string

    exception Mismatch of string × string × string

    val hash : key → hash
    val exists : hash → string → bool
    val find : hash → string → string option
    val write : hash → string → value → unit
    val write_dir : hash → string → string → value → unit
    val read : hash → string → value
    val maybe_read : hash → string → value result

  end

module type Key =
  sig
    type t
  end

module type Value =
  sig
    type t
  end

module Make (Key : Key) (Value : Value) :
    T with type key = Key.t and type value = Value.t
```

E.2 Implementation of Cache

```
let search_path =
  [ Filename.current_dir_name ]

module type T =
  sig

    type key
    type hash = string
    type value

    type α result =
      | Hit of α
      | Miss
      | Stale of string

    exception Mismatch of string × string × string

    val hash : key → hash
    val exists : hash → string → bool
    val find : hash → string → string option
    val write : hash → string → value → unit
    val write_dir : hash → string → string → value → unit
    val read : hash → string → value
    val maybe_read : hash → string → value result

  end

module type Key =
  sig
    type t
  end

module type Value =
  sig
    type t
  end

module Make (Key : Key) (Value : Value) =
  struct

    type key = Key.t
    type hash = string
    type value = Value.t

    type tagged =
        { tag : hash;
          value : value; }

    let hash value =
      Digest.string (Marshal.to_string value [])

    let find_first path name =
      let rec find_first′ = function
        | [] → raise Not_found
        | dir :: path →
            let f = Filename.concat dir name in
            if Sys.file_exists f then
              f
            else
              find_first′ path
      in
      find_first′ path

    let find hash name =
      try Some (find_first search_path name) with Not_found → None

    let exists hash name =
      match find hash name with
      | None → false
      | Some _ → true

    let try_first f path name =
      let rec try_first′ = function
        | [] → raise Not_found
        | dir :: path →
            try (f (Filename.concat dir name), dir) with _ → try_first′ path
      in
      try_first′ path
    let open_in_bin_first = try_first open_in_bin
    let open_out_bin_last path = try_first open_out_bin (List.rev path)

    let write hash name value =
      let oc, _ = open_out_bin_last search_path name in
      Marshal.to_channel oc { tag = hash; value = value } [];
      close_out oc

    let write_dir hash dir name value =
      let oc = open_out_bin (Filename.concat dir name) in
      Marshal.to_channel oc { tag = hash; value = value } [];
      close_out oc

    type α result =
      | Hit of α
      | Miss
      | Stale of string

    exception Mismatch of string × string × string

    let read hash name =
      let ic, dir = open_in_bin_first search_path name in
      let { tag = tag; value = value } = Marshal.from_channel ic in
      close_in ic;
      if tag = hash then
        value
      else
        raise (Mismatch (Filename.concat dir name, hash, tag))

    let maybe_read hash name =
      try
        Hit (read hash name)
      with
      | Not_found → Miss
      | Mismatch (file, _, _) → Stale file

  end
```

—F—
More On Lists

F.1 Interface of ThoList

$splitn\ n\ l\ =\ (hdn\ l,\ tln\ l)$, but more efficient.

val $hdn\ :\ int \to\ \alpha\ list \to\ \alpha\ list$
val $tln\ :\ int \to\ \alpha\ list \to\ \alpha\ list$
val $splitn\ :\ int \to\ \alpha\ list \to\ \alpha\ list \times \alpha\ list$

$split\_last\ (l\ @\ [a])\ =\ (l,\ a)$

val $split\_last\ :\ \alpha\ list \to\ \alpha\ list \times \alpha$

$chop\ n\ l$ chops l into pieces of size n (except for the last one, which contains th remainder).

val $chopn\ :\ int \to\ \alpha\ list \to\ \alpha\ list\ list$

$cycle\_until\ a\ l$ finds a member a in the list l and returns the cyclically permuted list with a as head. Raises Not_found if a is not in l.

val $cycle\_until\ :\ \alpha\ \to\ \alpha\ list \to\ \alpha\ list$

$cycle\ n\ l$ cyclically permute the list l by $n\ \geq\ 0$ positions. Raises $Not\_found\ List.length\ l\ >\ n$. NB: $cycle\ n\ l\ =\ tln\ n\ l\ @\ hdn\ n\ l$, but more efficient.

val $cycle\ :\ int \to\ \alpha\ list \to\ \alpha\ list$

$of\_subarray\ n\ m\ a$ is $[a.(n); a.(n+1); \ldots; a.(m)]$. Values of n and m out of bounds are silently shifted towards these bounds.

val $of\_subarray\ :\ int \to\ int \to\ \alpha\ array \to\ \alpha\ list$

$range\ s\ n\ m$ is $[n; n+s; n+2s; \ldots; m-((m-n)\mod s)]$

val $range\ :\ ?stride\ :int \to\ int \to\ int \to\ int\ list$

$enumerate\ s\ n\ [a1; a2; ...]\ is\ [(n, a1);\ (n+s, a2);\ ...]$

val $enumerate\ :\ ?stride\ :int \to\ int \to\ \alpha\ list \to\ (int \times \alpha)\ list$

$alist\_of\_list$ ˜$predicate$ ˜$offset\ list$ takes the elements of $list$ that satisfy $predicate$ and forms a list of pairs of an offset into the original $list$ and the element with the offsets starting from $offset$. NB: the order of the returned alist is not specified! For example $alist\_of\_list$ `["a";"b";"c"]` $=$ [(2, `"c"`); (1, `"b"`); (0, `"a"`)]

val $alist\_of\_list$:
  $?predicate : (\alpha\ \to\ bool)\ \to\ ?offset\ :int \to\ \alpha\ list \to\ (int \times \alpha)\ list$

Compress identical elements in a sorted list. Identity is determined using the polymorphic equality function $Pervasives.(=)$.

val $uniq\ :\ \alpha\ list \to\ \alpha\ list$

Test if all members of a list are structurally identical (actually $homogeneous\ l$ and $List.length\ (uniq\ l)\ \leq\ 1$ are equivalent, but the former is more efficient if a mismatch comes early).

val $homogeneous\ :\ \alpha\ list \to\ bool$

If all elements of the list l appear exactly twice, $pairs\ l$ returns a sorted list with these elements appearing once. Otherwise $Invalid\_argument$ is raised.

val $pairs\ :\ \alpha\ list \to\ \alpha\ list$

compare cmp l1 l2 compare two lists *l1* and *l2* according to *cmp*. *cmp* defaults to the polymorphic *Pervasives.compare*.

val *compare* : $?cmp : (\alpha \to \alpha \to \mathit{int}) \to \alpha\ \mathit{list} \to \alpha\ \mathit{list} \to \mathit{int}$

Collect and count identical elements in a list. Identity is determined using the polymorphic equality function *Pervasives.*(=). *classify* does not assume that the list is sorted. However, it is $O(n)$ for sorted lists and $O(n^2)$ in the worst case.

val *classify* : $\alpha\ \mathit{list} \to (\mathit{int} \times \alpha)\ \mathit{list}$

Collect the second factors with a common first factor in lists.

val *factorize* : $(\alpha \times \beta)\ \mathit{list} \to (\alpha \times \beta\ \mathit{list})\ \mathit{list}$

factorize_fold op init pairs combines the second elements of the *pairs* with common first element using the binary operator *op* and initial value *init*. If *op* is not associative and commutative, the result is *not* well defined.

val *factorize_fold* : $(\beta \to \beta \to \beta) \to \beta \to (\alpha \times \beta)\ \mathit{list} \to (\alpha \times \beta)\ \mathit{list}$

flatmap f is equivalent to *flatten* $\circ$ (*map* f), but more efficient, because no intermediate lists are built. Unfortunately, it is not tail recursive.

val *flatmap* : $(\alpha \to \beta\ \mathit{list}) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$

rev_flatmap f is equivalent to *flatten* $\circ$ (*rev_map* (*rev* $\circ f$)) = *rev* $\circ$ (*flatmap* f), but more efficient, because no intermediate lists are built. It is tail recursive.

val *rev_flatmap* : $(\alpha \to \beta\ \mathit{list}) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$

clone a n builds a list from n copies of the element a.

val *clone* : $\alpha \to \mathit{int} \to \alpha\ \mathit{list}$

multiply n l concatenates n copies of the list l.

val *multiply* : $\mathit{int} \to \alpha\ \mathit{list} \to \alpha\ \mathit{list}$

filtermap f l applies f to each element of l and drops the results *None*.

This will be *List.filter_map* starting with O'Caml 4.08!

val *filtermap* : $(\alpha \to \beta\ \mathit{option}) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$

power a_list computes the list of all sublists of *a_list*, i. e. the power set. The elements of the sublists are *not* required to have been sequential in *a_list*.

val *power* : $\alpha\ \mathit{list} \to \alpha\ \mathit{list}\ \mathit{list}$

Like *List.fold_left*, but returns immediately, if the folded function returns *None*. The analogous function val *fold_right_opt* : $(\alpha \to \beta \to \beta\ \mathit{option}) \to \alpha\ \mathit{list} \to \beta \to \beta\ \mathit{option}$ has not been implemented. It makes not much sense, because the outer function evaluation can only be performed after the results of all inner evaluations are available.

val *fold_left_opt* : $(\beta \to \alpha \to \beta\ \mathit{option}) \to \beta \to \alpha\ \mathit{list} \to \beta\ \mathit{option}$

Invent other names to avoid confusions with *List.fold_left2* and *List.fold_right2*.

val *fold_right2* : $(\alpha \to \beta \to \beta) \to \alpha\ \mathit{list}\ \mathit{list} \to \beta \to \beta$
val *fold_left2* : $(\beta \to \alpha \to \beta) \to \beta \to \alpha\ \mathit{list}\ \mathit{list} \to \beta$

iteri f n $[a; b; c]$ evaluates $f\ n\ a$, $f\ (n+1)\ b$ and $f\ (n+2)\ c$.

val *iteri* : $(\mathit{int} \to \alpha \to \mathit{unit}) \to \mathit{int} \to \alpha\ \mathit{list} \to \mathit{unit}$
val *mapi* : $(\mathit{int} \to \alpha \to \beta) \to \mathit{int} \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$

iteri2 f n m $[[aa; ab]; [ba; bb]]$ evaluates $f\ n\ m\ aa$, $f\ n\ (m+1)\ ab$, $f\ (n+1)\ m\ ba$ and $f\ (n+1)\ (m+1)\ bb$. NB: the nested lists need not be rectangular.

val *iteri2* : $(\mathit{int} \to \mathit{int} \to \alpha \to \mathit{unit}) \to \mathit{int} \to \mathit{int} \to \alpha\ \mathit{list}\ \mathit{list} \to \mathit{unit}$

Just like *List.map3*:

val *map3* : $(\alpha \to \beta \to \gamma \to \delta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list} \to \gamma\ \mathit{list} \to \delta\ \mathit{list}$

Transpose a *rectangular* list of lists like a matrix.

val *transpose* : α *list list* $\rightarrow$ α *list list*

interleave f list walks through *list* and inserts the result of *f* applied to the reversed list of elements before and the list of elements after. The empty lists at the beginning and end are included!

val *interleave* : (α *list* $\rightarrow$ α *list* $\rightarrow$ α *list*) $\rightarrow$ α *list* $\rightarrow$ α *list*

interleave_nearest f list is like *interleave f list*, but *f* looks only at the nearest neighbors.

val *interleave_nearest* : (α $\rightarrow$ α $\rightarrow$ α *list*) $\rightarrow$ α *list* $\rightarrow$ α *list*

partitioned_sort cmp index_sets list sorts the sublists of *list* specified by the *index_sets* and the complement of their union. **NB:** the sorting follows to order in the lists in *index_sets*. **NB:** the indices are 0-based.

val *partitioned_sort* : (α $\rightarrow$ α $\rightarrow$ *int*) $\rightarrow$ *int list list* $\rightarrow$ α *list* $\rightarrow$ α *list*
exception *Overlapping_indices*
exception *Out_of_bounds*

ariadne_sort cmp list sorts *list* according to *cmp* (default *Pervasives.compare*) keeping track of the original order by a 0-based list of indices.

val *ariadne_sort* : ?*cmp* : (α $\rightarrow$ α $\rightarrow$ *int*) $\rightarrow$ α *list* $\rightarrow$ α *list* $\times$ *int list*

ariadne_unsort (*ariadne_sort cmp list*) returns *list*.

val *ariadne_unsort* : α *list* $\times$ *int list* $\rightarrow$ α *list*

lexicographic cmp list1 list2 compares *list1* and *list2* lexicographically.

val *lexicographic* : ?*cmp* : (α $\rightarrow$ α $\rightarrow$ *int*) $\rightarrow$ α *list* $\rightarrow$ α *list* $\rightarrow$ *int*

common l1 l2 returns the elements common to the lists *l1* and *l2*. The lists are not required to be ordered and the result will also not be ordered.

val *common* : α *list* $\rightarrow$ α *list* $\rightarrow$ α *list*

complement l1 l2 returns the list *l1* with elements of list *l2* removed. The lists are not required to be ordered. Raises *Invalid_argument* `"ThoList.complement"`, if a member of *l1* is not in *l1*.

val *complement* : α *list* $\rightarrow$ α *list* $\rightarrow$ α *list*

to_string f list formats the elements of the list with *f*, concatenates them with `";␣"` and encloses the result in brakets.

val *to_string* : (α $\rightarrow$ *string*) $\rightarrow$ α *list* $\rightarrow$ *string*

take_first_even_opt predicate list find the first element *a* in *list* with *predicate a* = true. It returns *Some* (*a*, *remainder*), where *remainder* are all other elements of *list* reordered such that *a* :: *remainder* is equal to an even permutation of *list*. It returns *None*, if the predicate is never satisfied.

For a list of 2 elements, when the second element satisfies the predicate, there are not enough elements to construct an even permutation. Therefore the function is not well defined for this input. Instead of returning *None*, it raises the exception *Invalid_argument* `"ThoList.take_first_even_opt:␣pair"`

val *take_first_even_opt* : (α $\rightarrow$ *bool*) $\rightarrow$ α *list* $\rightarrow$ (α $\times$ α *list*) *option*

merge_alist op f1 f2 l1 l2 applies *op* to the values in the association lists with matching keys and *f1* or *f2* to the others. The result will be sorted according to the keys.

val *merge_alist* : (α $\rightarrow$ β $\rightarrow$ γ) $\rightarrow$ (α $\rightarrow$ γ) $\rightarrow$ (β $\rightarrow$ γ) $\rightarrow$
(δ $\times$ α) *list* $\rightarrow$ (δ $\times$ β) *list* $\rightarrow$ (δ $\times$ γ) *list*

Like *merge_alist*, but faster since it assumes that the lists are sorted.

val *merge_sorted_alist* : (α $\rightarrow$ β $\rightarrow$ γ) $\rightarrow$ (α $\rightarrow$ γ) $\rightarrow$ (β $\rightarrow$ γ) $\rightarrow$
(δ $\times$ α) *list* $\rightarrow$ (δ $\times$ β) *list* $\rightarrow$ (δ $\times$ γ) *list*

module *Test* : sig val *suite* : *OUnit.test* end

F.2 Implementation of ThoList

```
let rec hdn n l =
  if n ≤ 0 then
    []
  else
    match l with
    | x :: rest → x :: hdn (pred n) rest
    | [] → invalid_arg "ThoList.hdn"

let rec tln n l =
  if n ≤ 0 then
    l
  else
    match l with
    | _ :: rest → tln (pred n) rest
    | [] → invalid_arg "ThoList.tln"

let rec splitn' n l1_rev l2 =
  if n ≤ 0 then
    (List.rev l1_rev, l2)
  else
    match l2 with
    | x :: l2' → splitn' (pred n) (x :: l1_rev) l2'
    | [] → invalid_arg "ThoList.splitn␣n␣>␣len"

let splitn n l =
  if n < 0 then
    invalid_arg "ThoList.splitn␣n␣<␣0"
  else
    splitn' n [] l

let split_last l =
  match List.rev l with
  | [] → invalid_arg "ThoList.split_last␣[]"
  | ln :: l12_rev → (List.rev l12_rev, ln)
```

This is *splitn′* all over again, but without the exception.

```
let rec chopn'' n l1_rev l2 =
  if n ≤ 0 then
    (List.rev l1_rev, l2)
  else
    match l2 with
    | x :: l2' → chopn'' (pred n) (x :: l1_rev) l2'
    | [] → (List.rev l1_rev, [])

let rec chopn' n ll_rev = function
  | [] → List.rev ll_rev
  | l →
      begin match chopn'' n [] l with
      | [], [] → List.rev ll_rev
      | l1, [] → List.rev (l1 :: ll_rev)
      | l1, l2 → chopn' n (l1 :: ll_rev) l2
      end

let chopn n l =
  if n ≤ 0 then
    invalid_arg "ThoList.chopn␣n␣<=␣0"
  else
    chopn' n [] l
```

Find a member a in the list l and return the cyclically permuted list with a as head.

```
let cycle_until a l =
  let rec cycle_until' acc = function
```

```
    | [] → raise Not_found
    | a' :: l' as al' →
        if a' = a then
          al' @ List.rev acc
        else
          cycle_until' (a' :: acc) l' in
  cycle_until' [] l
```

```
let rec cycle' i acc l =
  if i ≤ 0 then
    l @ List.rev acc
  else
    match l with
    | [] → invalid_arg "ThoList.cycle"
    | a' :: l' →
        cycle' (pred i) (a' :: acc) l'
```

```
let cycle n l =
  if n < 0 then
    invalid_arg "ThoList.cycle"
  else
    cycle' n [] l
```

```
let of_subarray n1 n2 a =
  let rec of_subarray' n1 n2 =
    if n1 > n2 then
      []
    else
      a.(n1) :: of_subarray' (succ n1) n2 in
  of_subarray' (max 0 n1) (min n2 (pred (Array.length a)))
```

```
let range ?(stride = 1) n1 n2 =
  if stride ≤ 0 then
    invalid_arg "ThoList.range:␣stride␣<=␣0"
  else
    let rec range' n =
      if n > n2 then
        []
      else
        n :: range' (n + stride) in
    range' n1
```

Tail recursive:

```
let enumerate ?(stride = 1) n l =
  let _, l_rev =
    List.fold_left
      (fun (i, acc) a → (i + stride, (i, a) :: acc))
      (n, []) l in
  List.rev l_rev
```

Take the elements of *list* that satisfy *predicate* and form a list of pairs of an offset into the original list and the element with the offsets starting from *offset*. NB: the order of the returned alist is not specified!

```
let alist_of_list ?(predicate = (fun _ → true)) ?(offset = 0) list =
  let _, alist =
    List.fold_left
      (fun (n, acc) x →
        (succ n, if predicate x then (n, x) :: acc else acc))
      (offset, []) list in
  alist
```

This is *not* tail recursive!

```
let rec flatmap f = function
  | [] → []
  | x :: rest → f x @ flatmap f rest
```

This is!

```
let rev_flatmap f l =
  let rec rev_flatmap' acc f = function
    | [] → acc
    | x :: rest → rev_flatmap' (List.rev_append (f x) acc) f rest in
  rev_flatmap' [] f l

let rec power = function
  | [] → [[]]
  | a :: a_list →
      let power_a_list = power a_list in
      power_a_list @ List.map (fun a_list → a :: a_list) power_a_list

let rec fold_left_opt f acc = function
  | [] → Some acc
  | a :: rest →
      begin match f acc a with
      | None → None
      | Some acc → fold_left_opt f acc rest
      end

let fold_left2 f acc lists =
  List.fold_left (List.fold_left f) acc lists

let fold_right2 f lists acc =
  List.fold_right (List.fold_right f) lists acc

let iteri f start list =
  ignore (List.fold_left (fun i a → f i a; succ i) start list)

let iteri2 f start_outer star_inner lists =
  iteri (fun j → iteri (f j) star_inner) start_outer lists

let mapi f start list =
  let next, list' =
    List.fold_left (fun (i, acc) a → (succ i, f i a :: acc)) (start, []) list in
  List.rev list'

let rec map3 f l1 l2 l3 =
  match l1, l2, l3 with
  | [], [], [] → []
  | a1 :: l1, a2 :: l2, a3 :: l3 →
      let fa123 = f a1 a2 a3 in
      fa123 :: map3 f l1 l2 l3
  | _, _, _ → invalid_arg "ThoList.map3"
```

Is there a more efficient implementation?

```
let transpose lists =
  let rec transpose' rest =
    if List.for_all ((=) []) rest then
      []
    else
      List.map List.hd rest :: transpose' (List.map List.tl rest) in
  try
    transpose' lists
  with
  | Failure s →
      if s = "tl" then
        invalid_arg "ThoList.transpose:␣not␣rectangular"
      else
        failwith ("ThoList.transpose:␣unexpected␣Failure(" ^ s ^ ")")

let compare ?(cmp = Stdlib.compare) l1 l2 =
  let rec compare' l1' l2' =
    match l1', l2' with
    | [], [] → 0
    | [], _ → −1
    | _, [] → 1
    | n1 :: r1, n2 :: r2 →
        let c = cmp n1 n2 in
        if c ≠ 0 then
          c
        else
          compare' r1 r2
  in
  compare' l1 l2

let rec uniq' x = function
  | [] → []
  | x' :: rest →
      if x' = x then
        uniq' x rest
      else
        x' :: uniq' x' rest

let uniq = function
  | [] → []
  | x :: rest → x :: uniq' x rest

let rec homogeneous = function
  | [] | [_] → true
  | a1 :: (a2 :: _ as rest) →
      if a1 ≠ a2 then
        false
      else
        homogeneous rest

let rec pairs' acc = function
  | [] → acc
  | [x] → invalid_arg "pairs:␣odd␣number␣of␣elements"
  | x :: y :: indices →
     if x ≠ y then
       invalid_arg "pairs:␣not␣in␣pairs"
     else
       begin match acc with
       | [] → pairs' [x] indices
       | x' :: _ →
          if x = x' then
            invalid_arg "pairs:␣more␣than␣twice"
          else
            pairs' (x :: acc) indices
       end

let pairs l =
  pairs' [] (List.sort Stdlib.compare l)
```

If we needed it, we could use a polymorphic version of *Set* to speed things up from $O(n^2)$ to $O(n \ln n)$. But not before it matters somewhere . . .

```
let classify l =
  let rec add_to_class a = function
    | [] → [1, a]
    | (n, a') :: rest →
        if a = a' then
          (succ n, a) :: rest
        else
          (n, a') :: add_to_class a rest
  in
  let rec classify' cl = function
```

```
    | [] → cl
    | a :: rest → classify' (add_to_class a cl) rest
  in
  classify' [] l

let rec factorize l =
  let rec add_to_class x y = function
    | [] → [(x, [y])]
    | (x', ys) :: rest →
        if x = x' then
          (x, y :: ys) :: rest
        else
          (x', ys) :: add_to_class x y rest
  in
  let rec factorize' fl = function
    | [] → fl
    | (x, y) :: rest → factorize' (add_to_class x y fl) rest
  in
  List.map (fun (x, ys) → (x, List.rev ys)) (factorize' [] l)

let factorize_fold f acc l =
  List.map
    (fun (key, values) → (key, List.fold_left f acc values))
    (factorize l)

let rec clone x n =
  if n < 0 then
    invalid_arg "ThoList.clone"
  else if n = 0 then
    []
  else
    x :: clone x (pred n)

let interleave f list =
  let rec interleave' rev_head tail =
    let rev_head' = List.rev_append (f rev_head tail) rev_head in
    match tail with
    | [] → List.rev rev_head'
    | x :: tail' → interleave' (x :: rev_head') tail'
  in
  interleave' [] list

let interleave_nearest f list =
  interleave
    (fun head tail →
      match head, tail with
      | h :: _, t :: _ → f h t
      | _ → [])
    list

let rec rev_multiply n rl l =
  if n < 0 then
    invalid_arg "ThoList.multiply"
  else if n = 0 then
    []
  else
    List.rev_append rl (rev_multiply (pred n) rl l)

let multiply n l = rev_multiply n (List.rev l) l

let filtermap f l =
  let rec rev_filtermap acc = function
    | [] → List.rev acc
    | a :: a_list →
        match f a with
        | None → rev_filtermap acc a_list
        | Some fa → rev_filtermap (fa :: acc) a_list
  in
  rev_filtermap [] l

exception Overlapping_indices
exception Out_of_bounds

let iset_list_union list =
  List.fold_right Sets.Int.union list Sets.Int.empty

let complement_index_sets n index_set_lists =
  let index_sets = List.map Sets.Int.of_list index_set_lists in
  let index_set = iset_list_union index_sets in
  let size_index_sets =
    List.fold_left (fun acc s → Sets.Int.cardinal s + acc) 0 index_sets in
  if size_index_sets ≠ Sets.Int.cardinal index_set then
    raise Overlapping_indices
  else if Sets.Int.exists (fun i → i < 0 ∨ i ≥ n) index_set then
    raise Overlapping_indices
  else
    match Sets.Int.elements
            (Sets.Int.diff (Sets.Int.of_list (range 0 (pred n))) index_set) with
    | [] → index_set_lists
    | complement → complement :: index_set_lists

let sort_section cmp array index_set =
  List.iter2
    (Array.set array)
    index_set (List.sort cmp (List.map (Array.get array) index_set))

let partitioned_sort cmp index_sets list =
  let array = Array.of_list list in
  List.fold_left
    (fun () → sort_section cmp array)
    () (complement_index_sets (List.length list) index_sets);
  Array.to_list array

let ariadne_sort ?(cmp = Stdlib.compare) list =
  let sorted =
    List.sort (fun (n1, a1) (n2, a2) → cmp a1 a2) (enumerate 0 list) in
  (List.map snd sorted, List.map fst sorted)

let ariadne_unsort (sorted, indices) =
  List.map snd
    (List.sort
       (fun (n1, a1) (n2, a2) → Stdlib.compare n1 n2)
       (List.map2 (fun n a → (n, a)) indices sorted))

let lexicographic ?(cmp = Stdlib.compare) l1 l2 =
  let rec lexicographic' = function
    | [], [] → 0
    | [], _ → −1
    | _, [] → 1
    | x1 :: rest1, x2 :: rest2 →
       let res = cmp x1 x2 in
       if res ≠ 0 then
         res
       else
         lexicographic' (rest1, rest2) in
  lexicographic' (l1, l2)
```

If there was a polymorphic *Set*, we could also say *Set.elements* (*Set.union* (*Set.of_list l1*) (*Set.of_list l2*)).

```
let common l1 l2 =
  List.fold_left
```

```
    (fun acc x1 →
      if List.mem x1 l2 then
        x1 :: acc
      else
        acc)
    [] l1

let complement l1 = function
  | [] → l1
  | l2 →
     if List.for_all (fun x → List.mem x l1) l2 then
       List.filter (fun x → ¬ (List.mem x l2)) l1
     else
       invalid_arg "ThoList.complement"

let split_first_opt predicate list =
  let rec split_first_opt' rev_head = function
    | [] → None
    | a :: tail →
       if predicate a then
         Some (List.rev rev_head, a, tail)
       else
         split_first_opt' (a :: rev_head) tail in
  split_first_opt' [] list

let take_first_even_opt predicate list =
  match split_first_opt predicate list with
  | None → None
  | Some ([], i, []) → Some (i, [])
  | Some ([_], _, []) → invalid_arg "ThoList.take_first_even_opt:␣pair"
  | Some ([], i, tail) → Some (i, tail)
  | Some (i1 :: i2 :: head, i, []) → (* [i; i1; i2] is an even permutaion of [i1; i2; i] *)
     Some (i, i1 :: head @ [i2])
  | Some (i1 :: head, i, i2 :: tail) → (* [i; i2; i1] is an even permutaion of [i1; i; i2] *)
     Some (i, head @ (i2 :: i1 :: tail))

let to_string a2s alist =
  "[" ^ String.concat ";␣" (List.map a2s alist) ^ "]"

let merge_sorted_alist op f1 f2 l1 l2 =
  let rec merge_sorted_alist' acc l1 l2 =
    match l1, l2 with
    | [], [] → List.rev acc
    | tl1, [] → List.rev_append acc (List.map (fun (k, v) → (k, f1 v)) tl1)
    | [], tl2 → List.rev_append acc (List.map (fun (k, v) → (k, f2 v)) tl2)
    | (k1, v1) :: tl1, (k2, v2) :: tl2 →
       let c = Stdlib.compare k1 k2 in
       if c = 0 then
         merge_sorted_alist' ((k1, op v1 v2) :: acc) tl1 tl2
       else if c < 0 then
         merge_sorted_alist' ((k1, f1 v1) :: acc) tl1 l2
       else
         merge_sorted_alist' ((k2, f2 v2) :: acc) l1 tl2 in
  merge_sorted_alist' [] l1 l2

let merge_alist op f1 f2 l1 l2 =
  merge_sorted_alist op f1 f2
    (List.sort (fun (k1, _) (k2, _) → Stdlib.compare k1 k2) l1)
    (List.sort (fun (k1, _) (k2, _) → Stdlib.compare k1 k2) l2)

let random_int_list imax n =
  let imax_plus = succ imax in
  Array.to_list (Array.init n (fun _ → Random.int imax_plus))

module Test =
  struct
    let id x = x
    let int_list2_to_string l2 =
      to_string (to_string string_of_int) l2
```

Inefficient, must only be used for unit tests.

```
  let compare_lists_by_size l1 l2 =
    let lengths = Stdlib.compare (List.length l1) (List.length l2) in
    if lengths = 0 then
      Stdlib.compare l1 l2
    else
      lengths

  open OUnit

  let suite_filtermap =
    "filtermap" >:::
      [ "filtermap␣Some␣[]" >::
          (fun () →
            assert_equal ~printer : (to_string string_of_int)
              [] (filtermap (fun x → Some x) []));

        "filtermap␣None␣[]" >::
          (fun () →
            assert_equal ~printer : (to_string string_of_int)
              [] (filtermap (fun x → None) []));

        "filtermap␣even_neg␣[]" >::
          (fun () →
            assert_equal ~printer : (to_string string_of_int)
              [0; − 2; − 4]
              (filtermap
                 (fun n → if n mod 2 = 0 then Some (−n) else None)
                 (range 0 5)));

        "filtermap␣odd_neg␣[]" >::
          (fun () →
            assert_equal ~printer : (to_string string_of_int)
              [−1; − 3; − 5]
              (filtermap
                 (fun n → if n mod 2 ≠ 0 then Some (−n) else None)
                 (range 0 5))) ]

  let assert_power power_a_list a_list =
    assert_equal ~printer : int_list2_to_string
      power_a_list
      (List.sort compare_lists_by_size (power a_list))

  let suite_power =
    "power" >:::
      [ "power␣[]" >::
          (fun () →
            assert_power [[]] []);

        "power␣[1]" >::
          (fun () →
            assert_power [[]; [1]] [1]);

        "power␣[1;2]" >::
          (fun () →
            assert_power [[]; [1]; [2]; [1; 2]] [1; 2]);

        "power␣[1;2;3]" >::
          (fun () →
            assert_power
```

```
              [[];
               [1]; [2]; [3];
               [1; 2]; [1; 3]; [2; 3];
               [1; 2; 3]]
              [1; 2; 3]);
        "power␣[1;2;3;4]" >::
          (fun () →
            assert_power
              [[];
               [1]; [2]; [3]; [4];
               [1; 2]; [1; 3]; [1; 4]; [2; 3]; [2; 4]; [3; 4];
               [1; 2; 3]; [1; 2; 4]; [1; 3; 4]; [2; 3; 4];
               [1; 2; 3; 4]]
              [1; 2; 3; 4]) ]

let suite_split =
  "split*" >:::
    [ "split_last␣[]" >::
          (fun () →
            assert_raises
              (Invalid_argument "ThoList.split_last␣[]")
              (fun () → split_last []));
        "split_last␣[1]" >::
          (fun () →
            assert_equal
              ([], 1)
              (split_last [1]));
        "split_last␣[2;3;1;4]" >::
          (fun () →
            assert_equal
              ([2; 3; 1], 4)
              (split_last [2; 3; 1; 4])) ]

let test_list = random_int_list 1000 100

let assert_equal_int_list =
  assert_equal ~printer : (to_string string_of_int)

let suite_cycle =
  "cycle_until" >:::
    [ "cycle␣(-1)␣[1;2;3]" >::
          (fun () →
            assert_raises
              (Invalid_argument "ThoList.cycle")
              (fun () → cycle 4 [1; 2; 3]));
        "cycle␣4␣[1;2;3]" >::
          (fun () →
            assert_raises
              (Invalid_argument "ThoList.cycle")
              (fun () → cycle 4 [1; 2; 3]));
        "cycle␣42␣[...]" >::
          (fun () →
            let n = 42 in
            assert_equal_int_list
              (tln n test_list @ hdn n test_list)
              (cycle n test_list));
        "cycle_until␣1␣[]" >::
          (fun () →
            assert_raises
              (Not_found)
              (fun () → cycle_until 1 []));
        "cycle_until␣1␣[2;3;4]" >::
          (fun () →
            assert_raises
              (Not_found)
              (fun () → cycle_until 1 [2; 3; 4]));
        "cycle_until␣1␣[1;2;3;4]" >::
          (fun () →
            assert_equal
              [1; 2; 3; 4]
              (cycle_until 1 [1; 2; 3; 4]));
        "cycle_until␣3␣[1;2;3;4]" >::
          (fun () →
            assert_equal
              [3; 4; 1; 2]
              (cycle_until 3 [3; 4; 1; 2]));
        "cycle_until␣4␣[1;2;3;4]" >::
          (fun () →
            assert_equal
              [4; 1; 2; 3]
              (cycle_until 4 [4; 1; 2; 3])) ]

let suite_alist_of_list =
  "alist_of_list" >:::
    [ "simple" >::
          (fun () →
            assert_equal
              [(46, 4); (44, 2); (42, 0)]
              (alist_of_list
                 ~predicate : (fun n → n mod 2 = 0) ~offset : 42 [0; 1; 2; 3; 4; 5])) ]

let suite_factorize_fold =
  "factorize_fold" >:::
    [ "simple" >::
          (fun () →
            assert_equal
              [(1, 21); (2, 41)]
              (factorize_fold (+) 0 [(1, 10); (2, 20); (2, 21); (1, 11)])) ]

let suite_complement =
  "complement" >:::
    [ "simple" >::
          (fun () →
            assert_equal [2; 4] (complement [1; 2; 3; 4] [1; 3]));
        "empty" >::
          (fun () →
            assert_equal [1; 2; 3; 4] (complement [1; 2; 3; 4] []));
        "failure" >::
          (fun () →
            assert_raises
              (Invalid_argument ("ThoList.complement"))
              (fun () → complement (complement [1; 2; 3; 4] [5]))) ]

let suite_merge_alist =
  "merge_alist" >:::
    [ "[]␣[]" >::
          (fun () →
            assert_equal [] (merge_alist (+) id id [] []));

        "[]␣[(a,␣1);␣(b,␣2)]" >::
          (fun () →
            assert_equal
              [("a", 1); ("b", 2)]
              (merge_alist (+) id id [] [("a", 1); ("b", 2)]));

        "[(a,␣1);␣(b,␣2)]␣[]" >::
          (fun () →
              assert_equal
                [("a", 1); ("b", 2)]
                (merge_alist (+) id id [("a", 1); ("b", 2)] []));

          "[(a,␣1);␣(b,␣2)]␣[(c,␣3);␣(b,␣2)]" >::
            (fun () →
              assert_equal
                [("a", 1); ("b", 4); ("c", 3)]
                (merge_alist (+) id id [("a", 1); ("b", 2)] [("c", 3); ("b", 2)])) ]

  let suite_take_first_even_opt =
    "take_first_even_opt" >:::
      [ "empty" >::
            (fun () →
              assert_equal None (take_first_even_opt ((=) 1) []));

          "not␣found" >::
            (fun () →
              assert_equal None (take_first_even_opt ((=) 0) [1; 2; 3]));

          "1␣[1;2;3]" >::
            (fun () →
              assert_equal (Some (1, [2; 3])) (take_first_even_opt ((=) 1) [1; 2; 3]));

          "2␣[1;2;3]" >::
            (fun () →
              assert_equal (Some (2, [3; 1])) (take_first_even_opt ((=) 2) [1; 2; 3]));

          "3␣[1;2;3]" >::
            (fun () →
              assert_equal (Some (3, [1; 2])) (take_first_even_opt ((=) 3) [1; 2; 3]));

          "1␣[1;2;3;4]" >::
            (fun () →
              assert_equal (Some (1, [2; 3; 4])) (take_first_even_opt ((=) 1) [1; 2; 3; 4]));

          "2␣[1;2;3;4]" >::
            (fun () →
              assert_equal (Some (2, [3; 1; 4])) (take_first_even_opt ((=) 2) [1; 2; 3; 4]));

          "3␣[1;2;3;4]" >::
            (fun () →
              assert_equal (Some (3, [2; 4; 1])) (take_first_even_opt ((=) 3) [1; 2; 3; 4]));

          "4␣[1;2;3;4]" >::
            (fun () →
              assert_equal (Some (4, [1; 3; 2])) (take_first_even_opt ((=) 4) [1; 2; 3; 4]));

          "pair" >::
            (fun () →
              assert_raises
                (Invalid_argument ("ThoList.take_first_even_opt:␣pair"))
                (fun () → take_first_even_opt ((=) 2) [1; 2])) ]

  let suite =
    "ThoList" >:::
      [suite_filtermap;
       suite_power;
       suite_split;
       suite_cycle;
       suite_alist_of_list;
       suite_factorize_fold;
       suite_complement;
       suite_merge_alist;
       suite_take_first_even_opt]

end
```

—G—
Non-Empty Lists

G.1 Interface of NEList

Since O'Caml 3.11, we can use private type abbreviation to enforce invariants without sacrificing any performance.
Once we have decided on an interface that avoids any partial functions, most of the implementation will be just an indirection to the standard library module.
A nonempty list $\alpha\ t$ is represented as a "normal" α *list* …

```
type α t = private α list
```

…, but there is no way to construct an empty list, since the constructors require at least one element:

```
val make : α → α list → α t
val singleton : α → α t
val cons : α → α t → α t
```

to_list l is the same as *l* :> *elt list*, without having to specify the element type *elt*. The compiler should inline this.

```
val to_list : α t → α list
```

hd never fails. We can also have a *tl* that never fails, if we allow it to return an "normal" list.

```
val hd : α t → α
val tl : α t → α list
val tl_opt : α t → α t option
```

The inverse of *cons* (uncurried): *snoc l* = (*hd l*, *tl l*) and *snoc_opt l* = (*hd l*, *tl_opt l*), but a little bit more efficient, since the list is deconstructed only once.

```
val snoc : α t → α × α list
val snoc_opt : α t → α × α t option
```

```
val map : (α → β) → α t → β t
val fold_right : (α → β → β) → α t → β → β
val sort : (α → α → int) → α t → α t
```

G.2 Implementation of NEList

The implementation is now trivial, except for the few cases where we need to avoid incomplete pattern match warnings:

```
let impossible f = failwith ("NList." ^ f ^ ":␣impossible␣[]")
```

```
type α t = α list
```

```
let make a alist = a :: alist
let singleton a = make a []
let cons = make
```

```
let to_list l = l [@@inline]
```

```
let hd = List.hd
let tl = List.tl
let tl_opt = function
  | [] → impossible "tl_opt"
  | [_] → None
  | _ :: tail → Some tail

let snoc = function
  | [] → impossible "snoc"
  | head :: tail → (head, tail)

let snoc_opt = function
  | [] → impossible "snoc_opt"
  | [head] → (head, None)
  | head :: tail → (head, Some tail)

let map = List.map
let fold_right = List.fold_right
let sort = List.sort
```

—H—
Lists with Typed Length

H.1 Interface of NList

This is inspired by an example posted on github by Izaak Meckler that in turn appears to be based on ideas well known in the Haskell community.
These types are just Peano numerals ν used as indices for $(\nu,\ \alpha)\ t$. z encodes 0 and $\alpha\ s$ the successor.

type z
type $\alpha\ s$

A $(\nu,\ \alpha)\ t$ is a list of α of length ν with ν encoded as a church numeral and must not be too large!

type $(\nu,\ \alpha)\ t$

Constructors.

val $empty\ :\ (z,\ \alpha)\ t$
val $cons\ :\ \alpha\ \rightarrow\ (\nu,\ \alpha)\ t\ \rightarrow\ (\nu\ s,\ \alpha)\ t$

Deconstructors. Note that they cannot be applied to the empty list.

val $hd\ :\ (\nu\ s,\ \alpha)\ t\ \rightarrow\ \alpha$
val $tl\ :\ (\nu\ s,\ \alpha)\ t\ \rightarrow\ (\nu,\ \alpha)\ t$

Turn the a list with typed length into an ordinary list. Note also, that we can not implement the inverse function $of\_list\ :\ \alpha\ list\rightarrow\ (\nu,\ \alpha\ t)$, because in that case the type ν depends on the list and is *not* known at compile time.

val $to\_list\ :\ (\nu,\ \alpha)\ t\ \rightarrow\ \alpha\ list$

The usual suspects.

val $map\ :\ (\alpha\ \rightarrow\ \beta)\ \rightarrow\ (\nu,\ \alpha)\ t\ \rightarrow\ (\nu,\ \beta)\ t$
val $fold\_right\ :\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \beta)\ \rightarrow\ (\nu,\ \alpha)\ t\ \rightarrow\ \beta\ \rightarrow\ \beta$

A version of *append* is complicated, since we need to compute the sum of the lengths in the type system. It can be done by introducing additional wrappers, but the result is difficult to deconstruct and we don't need it for our applications. The usual implementation of *rev* will also not work, because we need again to maintain the sum of the lengths as an invariant. Simple successor relationships are not enough.
On the other hand, *map2*, *fold_right2* etc. can be implemented easily. Here, the type shines, because it can avoid the *Invalid_argument* exception.

val $map2\ :\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \gamma)\ \rightarrow\ (\nu,\ \alpha)\ t\ \rightarrow\ (\nu,\ \beta)\ t\ \rightarrow\ (\nu,\ \gamma)\ t$

The algorithm is not suitable for long lists, but we expect the lists to be very short anyway.

val $sort\ :\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ int)\ \rightarrow\ (\nu,\ \alpha)\ t\ \rightarrow\ (\nu,\ \alpha)\ t$

H.2 Implementation of NList

The constructor *Zero* appears to be not needed, but the constructor *Successor* is required.

type $z\ =\ Zero$
type $\alpha\ s\ =\ Successor$

type $(\_,\ \_)\ t\ =$
  $|\ Nil\ :\ (z,\ \alpha)\ t$

```
  | Cons : α × (ν, α) t → (ν s, α) t

let empty = Nil

let cons : type n. α → (n, α) t → (n s, α) t =
  fun x xs →
  Cons (x, xs)

let hd : type n. (n s, α) t → α = function
  | Cons (x, _) → x

let tl : type n. (n s, α) t → (n, α) t = function
  | Cons (_, xs) → xs

let rec fold_right : type n. (α → β → β) → (n, α) t → β → β =
  fun f alist b →
  match alist with
  | Nil → b
  | Cons (a, rest) → f a (fold_right f rest b)

let rec map : type n. (α → β) → (n, α) t → (n, β) t =
  fun f →
  function
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, map f xs)

let rec to_list : type n. (n, α) t → α list = function
  | Nil → []
  | Cons (a, a_list) → a :: to_list a_list

let rec map2 : type n. (α → β → γ) → (n, α) t → (n, β) t → (n, γ) t =
  fun f a_list b_list →
  match a_list, b_list with
  | Nil, Nil → Nil
  | Cons (x, xs), Cons (y, ys) → Cons (f x y, map2 f xs ys)
```

This corresponds to a bubble sort. Don't use this for long lists! However, we expect the lists to be very short anyway and type safe reversing or concatenating two lists as required by the better performing algorithms requires to much effort for our applications.
Inner step: find an element that is out of order and push it past the adjacent lesser elements. Report whether a transposition was made.

```
let rec cycle : type n. (α → α → int) → (n, α) t → bool × (n, α) t =
  fun cmp →
  function
  | Nil → (false, Nil)
  | Cons (_, Nil) as a → (false, a)
  | Cons (a1, (Cons (a2, alist2) as alist1)) →
      if cmp a1 a2 ≤ 0 then
        let flipped, alist = cycle cmp alist1 in
        (flipped, Cons (a1, alist))
      else
        let flipped, alist = cycle cmp (Cons (a1, alist2)) in
        (true, Cons (a2, alist))
```

Repeat the inner step until no more elements are out of order.

```
let rec sort : type n. (α → α → int) → (n, α) t → (n, α) t =
  fun cmp alist →
  let flipped, cycled = cycle cmp alist in
  if flipped then
    sort cmp cycled
  else
    cycled
```

—I—
More On Arrays

I.1 Interface of ThoArray

Compressed arrays, i. e. arrays with only unique elements and an embedding that allows to recover the original array. NB: in the current implementation, compressing saves space, if *and only if* objects of type α require more storage than integers. The main use of α *compressed* is *not* for saving space, anyway, but for avoiding the repetition of hard calculations.

```
type α compressed
val uniq : α compressed → α array
val embedding : α compressed → int array
```

These two are inverses of each other:

```
val compress : α array → α compressed
val uncompress : α compressed → α array
```

One can play the same game for matrices.

```
type α compressed2
val uniq2 : α compressed2 → α array array
val embedding1 : α compressed2 → int array
val embedding2 : α compressed2 → int array
```

Again, these two are inverses of each other:

```
val compress2 : α array array → α compressed2
val uncompress2 : α compressed2 → α array array
```

compare cmp a1 a2 compare two arrays *a1* and *a2* according to *cmp*. *cmp* defaults to the polymorphic *Pervasives.compare*.

```
val compare : ?cmp : (α → α → int) → α array → α array → int
```

Searching arrays

```
val find_first : (α → bool) → α array → int
val match_first : α → α array → int
val find_all : (α → bool) → α array → int list
val match_all : α → α array → int list
```

```
val num_rows : α array array → int
val num_columns : α array array → int
```

Implement the Fisher-Yates shuffle to randomly shuffle an array in place, cf. [19], pp. 139-140.

```
val shuffle : α array → unit
```

```
val rank3 : int → int → int → α → α array array array
```

```
module Test : sig val suite : OUnit.test end
```

I.2 Implementation of ThoArray

```
type α compressed =
    { uniq : α array;
```

```
      embedding : int array }

let uniq a = a.uniq
let embedding a = a.embedding

type α compressed2 =
    { uniq2 : α array array;
      embedding1 : int array;
      embedding2 : int array }

let uniq2 a = a.uniq2
let embedding1 a = a.embedding1
let embedding2 a = a.embedding2

module PMap = Pmap.Tree

let compress a =
  let last = Array.length a − 1 in
  let embedding = Array.make (succ last) (−1) in
  let rec scan num_uniq uniq elements n =
    if n > last then
      { uniq = Array.of_list (List.rev elements);
        embedding = embedding }
    else
      match PMap.find_opt compare a.(n) uniq with
      | Some n′ →
          embedding.(n) ← n′;
          scan num_uniq uniq elements (succ n)
      | None →
          embedding.(n) ← num_uniq;
          scan
            (succ num_uniq)
            (PMap.add compare a.(n) num_uniq uniq)
            (a.(n) :: elements)
            (succ n) in
  scan 0 PMap.empty [] 0

let uncompress a =
  Array.map (Array.get a.uniq) a.embedding
```

Using *transpose* simplifies the algorithms, but can be inefficient. If this turns out to be the case, we should add special treatments for symmetric matrices.

```
let transpose a =
  let dim1 = Array.length a
  and dim2 = Array.length a.(0) in
  let a′ = Array.make_matrix dim2 dim1 a.(0).(0) in
  for i1 = 0 to pred dim1 do
    for i2 = 0 to pred dim2 do
      a′.(i2).(i1) ← a.(i1).(i2)
    done
  done;
  a′

let compress2 a =
  let c2 = compress a in
  let c12_transposed = compress (transpose c2.uniq) in
  { uniq2 = transpose c12_transposed.uniq;
    embedding1 = c12_transposed.embedding;
    embedding2 = c2.embedding }

let uncompress2 a =
  let a2 = uncompress { uniq = a.uniq2; embedding = a.embedding2 } in
  transpose (uncompress { uniq = transpose a2; embedding = a.embedding1 })
```

FIXME: not tail recursive!

```
let compare ?(cmp = Stdlib.compare) a1 a2 =
  let l1 = Array.length a1
  and l2 = Array.length a2 in
  if l1 < l2 then
    -1
  else if l1 > l2 then
    1
  else
    let rec scan i =
      if i = l1 then
        0
      else
        let c = cmp a1.(i) a2.(i) in
        if c < 0 then
          -1
        else if c > 0 then
          1
        else
          scan (succ i) in
    scan 0

let find_first f a =
  let l = Array.length a in
  let rec find_first' i =
    if i ≥ l then
      raise Not_found
    else if f (a.(i)) then
      i
    else
      find_first' (succ i)
  in
  find_first' 0

let match_first x a =
  find_first (fun x' → x = x') a

let find_all f a =
  let matches = ref [] in
  for i = Array.length a − 1 downto 0 do
    if f (a.(i)) then
      matches := i :: !matches
  done;
  !matches

let match_all x a =
  find_all (fun x' → x = x') a

let num_rows a =
  Array.length a

let num_columns a =
  match ThoList.classify (List.map Array.length (Array.to_list a)) with
  | [ (_, n) ] → n
  | _ → invalid_arg "ThoArray.num_columns:␣inhomogeneous␣array"

let shuffle a =
  for n = Array.length a − 1 downto 1 do
    let k = Random.int (succ n) in
    if k ≠ n then
      let tmp = Array.get a n in
      Array.set a n (Array.get a k);
      Array.set a k tmp
  done

let rank3 n1 n2 n3 initial =
  let a = Array.make n1 [| |] in
  for i1 = 0 to pred n1 do
    a.(i1) ← Array.make_matrix n2 n3 initial
  done;
  a

module Test =
  struct

    open OUnit

    let test_compare_empty =
      "empty" >::
        (fun () → assert_equal 0 (compare [| |] [| |]))

    let test_compare_shorter =
      "shorter" >::
        (fun () → assert_equal (−1) (compare [|0|] [|0; 1|]))

    let test_compare_longer =
      "longer" >::
        (fun () → assert_equal ( 1) (compare [|0; 1|] [|0|]))

    let test_compare_less =
      "longer" >::
        (fun () → assert_equal (−1) (compare [|0; 1|] [|0; 2|]))

    let test_compare_equal =
      "equal" >::
        (fun () → assert_equal ( 0) (compare [|0; 1|] [|0; 1|]))

    let test_compare_more =
      "more" >::
        (fun () → assert_equal ( 1) (compare [|0; 2|] [|0; 1|]))

    let suite_compare =
      "compare" >:::
        [test_compare_empty;
         test_compare_shorter;
         test_compare_longer;
         test_compare_less;
         test_compare_equal;
         test_compare_more]

    let test_find_first_not_found =
      "not␣found" >::
        (fun () →
          assert_raises Not_found
            (fun () → find_first (fun n → n mod 2 = 0) [|1; 3; 5|]))

    let test_find_first_first =
      "first" >::
        (fun () →
          assert_equal 0
            (find_first (fun n → n mod 2 = 0) [|2; 3; 4; 5|]))

    let test_find_first_not_last =
      "last" >::
        (fun () →
          assert_equal 1
            (find_first (fun n → n mod 2 = 0) [|1; 2; 3; 4|]))

    let test_find_first_last =
      "not␣last" >::
        (fun () →
          assert_equal 1
            (find_first (fun n → n mod 2 = 0) [|1; 2|]))
```

```
let suite_find_first =
  "find_first" >:::
    [test_find_first_not_found;
     test_find_first_first;
     test_find_first_not_last;
     test_find_first_last]

let test_find_all_empty =
  "empty" >::
    (fun () →
       assert_equal [ ]
         (find_all (fun n → n mod 2 = 0) [|1; 3; 5|]))

let test_find_all_first =
  "first" >::
    (fun () →
       assert_equal [0; 2]
         (find_all (fun n → n mod 2 = 0) [|2; 3; 4; 5|]))

let test_find_all_not_last =
  "last" >::
    (fun () →
       assert_equal [1; 3]
         (find_all (fun n → n mod 2 = 0) [|1; 2; 3; 4; 5|]))

let test_find_all_last =
  "not␣last" >::
    (fun () →
       assert_equal [1; 3]
         (find_all (fun n → n mod 2 = 0) [|1; 2; 3; 4|]))

let suite_find_all =
  "find_all" >:::
    [test_find_all_empty;
     test_find_all_first;
     test_find_all_last;
     test_find_all_not_last]

let test_num_columns_ok2 =
  "ok/2" >::
    (fun () →
       assert_equal 2
         (num_columns [| [| 11; 12 |];
                         [| 21; 22 |];
                         [| 31; 32 |] |]))

let test_num_columns_ok0 =
  "ok/0" >::
    (fun () →
       assert_equal 0
         (num_columns [| [| |];
                         [| |];
                         [| |] |]))

let test_num_columns_not_ok =
  "not_ok" >::
    (fun () →
       assert_raises (Invalid_argument
                        "ThoArray.num_columns:␣inhomogeneous␣array")
         (fun () → num_columns [| [| 11; 12 |];
                                  [| 21 |];
                                  [| 31; 32 |] |]))

let suite_num_columns =
  "num_columns" >:::
    [test_num_columns_ok2;
```

```
          test_num_columns_ok0;
          test_num_columns_not_ok]

    let suite =
      "ThoArrays" >:::
        [suite_compare;
         suite_find_first;
         suite_find_all;
         suite_num_columns]

  end
```

—J— Persistent Arrays

J.1 Interface of PArray

O'Caml arrays α *array* are a special case of maps *int* $\rightarrow$ α from a subset of the integers into a set, where the subset is contiguous and starts with 0.

In O'Caml, updating element of an α *array* is not pure, since the array is updated in place and all references to the array element in other parts of the code are affected. This is efficient, but complicates backtracking.

A α *PArray.t*, on the other hand is updated with pure functions, keeping the original array in place.
The type of persistent array.

type α *t*

val *empty* : α *t*
val *is_empty* : α *t* $\rightarrow$ *bool*
val *map* : (α $\rightarrow$ β) $\rightarrow$ α *t* $\rightarrow$ β *t*
val *add* : *int* $\rightarrow$ α $\rightarrow$ α *t* $\rightarrow$ α *t*
val *remove* : *int* $\rightarrow$ α *t* $\rightarrow$ α *t*
val *get_opt* : *int* $\rightarrow$ α *t* $\rightarrow$ α *option*

Create an array from a list of pairs of index and value. Note that we assume the array indices to start from 0.

val *of_pairs* : (*int* $\times$ α) *list* $\rightarrow$ α *t*
val *to_pairs* : α *t* $\rightarrow$ (*int* $\times$ α) *list*

Compute a list of all entries of the array, starting from the index 0. Entries *a* are represented by *Some a* and missing entries are represented by *None*. For example *to_option_list* [*of_pairs* [(2, 42)]] *evaluates* to [[*None*; *Some* 42]].

val *to_option_list* : α *t* $\rightarrow$ α *option list*

For debugging:

val *to_string* : (α $\rightarrow$ *string*) $\rightarrow$ α *t* $\rightarrow$ *string*

Order:

val *compare* : (α $\rightarrow$ α $\rightarrow$ *int*) $\rightarrow$ α *t* $\rightarrow$ α *t* $\rightarrow$ *int*
val *equal* : (α $\rightarrow$ α $\rightarrow$ *bool*) $\rightarrow$ α *t* $\rightarrow$ α *t* $\rightarrow$ *bool*

take_one project_opt parray tries to find one element in *parray* that is mapped to *None* by *project_opt*. Returns *Nothing projected_parray* if nothing is found and *Unique* (*key*, *value*, *projected_parray*) if there is exactly one match where *projected_parray* is *parray* with the binding for *key* removed and the function *project_opt* has been applied (with the *Some* stripped, of course). Returns *Multiple* (*key*, *value*, *parray'*) if there are multiple matches, where *parray'* is *parray* with the binding for *key* removed. In both cases, *key* is one of the matching keys and *value* the associated binding. The rationale is that we can use *take_one* to remove bindings from a map until we can replace the type of the values by a simpler type, e. g. by unboxing.

type (α, β) *taken* = private
  | *Nothing* of β *t*
  | *Single* of *int* $\times$ α $\times$ β *t*
  | *Multiple* of *int* $\times$ α $\times$ α *t*

val *take_one* : (*int* $\rightarrow$ α $\rightarrow$ β *option*) $\rightarrow$ α *t* $\rightarrow$ (α, β) *taken*

module *Test* : sig val *suite* : *OUnit.test* end

J.2 Implementation of PArray

The *Map* based implementation has the drawback that the polymorphic *compare* and (=) will occasionally report two *PArray.t* as different even if they describe the same array. Options

1. Replace *compare* by specific functions everywhere. This is the preferred approach, but can become very tedious.
2. Replace *Map* by sorted association lists.

J.2.1 Maps

```
module Maps =
  struct

    module IMap = Map.Make(Int)

    type α t = α IMap.t

    let empty = IMap.empty
    let is_empty = IMap.is_empty
    let map = IMap.map
    let add = IMap.add
    let remove = IMap.remove
    let get_opt = IMap.find_opt

    let min_key map = fst (IMap.min_binding map)
    let max_key map = fst (IMap.max_binding map)

    let index_base = 0

    let to_option_list map =
      if IMap.is_empty map then
        []
      else if min_key map < index_base then
        invalid_arg "PArray.Maps.to_option_list"
      else
        let rec to_option_list' acc n =
          if n < index_base then
            acc
          else
            to_option_list' (get_opt n map :: acc) (pred n) in
        to_option_list' [] (max_key map)

    let to_string a2s map =
      match to_option_list map with
      | [] → "[]"
      | [None] → "?"
      | [Some a] → a2s a
      | pairs → ThoList.to_string (function None → "?" | Some a → a2s a) pairs

    let of_pairs pairs =
      List.fold_right
        (fun (k, v) map →
          if k < index_base then
            invalid_arg "PArray.Maps.of_pairs"
          else
            IMap.add k v map)
        pairs IMap.empty

    let to_pairs = IMap.bindings

    let compare = IMap.compare
    let equal = IMap.equal

    type (α, β) taken =
      | Nothing of β t
      | Single of int × α × β t
      | Multiple of int × α × α t

    let take_one project_opt parray =
      let select k v =
        match project_opt k v with
        | Some _ → false
        | None → true
      and project k v =
        match project_opt k v with
        | Some v′ → v′
        | None → failwith "PArray.Maps.take_one:␣impossible" in
      let matches, other = IMap.partition select parray in
      match IMap.choose_opt matches with
      | None → Nothing (IMap.mapi project parray)
      | Some (k, v) →
          let more_matches = remove k matches in
          if is_empty more_matches then
            Single (k, v, IMap.mapi project other)
          else
            Multiple (k, v, IMap.fold IMap.add more_matches other)

  end
```

J.2.2 Association Lists

We assume that the lists are short and use non tail recursive implementations if they are faster.

```
module Alists =
  struct

    type α t = (int × α) list

    let empty = [ ]

    let is_empty = function
      | [ ] → true
      | _ → false

    let map f parray =
      List.map (fun (i, a) → (i, f a)) parray

    let rec add i a = function
      | [ ] → [(i, a)]
      | (i′, a′ as ia′) :: tail as alist →
          if i′ = i then
            (i, a) :: tail
          else if i′ > i then
            (i, a) :: alist
          else
            ia′ :: add i a tail

    let rec remove i = function
      | [ ] → [ ]
      | (i′, _ as ia′) :: tail as alist →
          if i′ = i then
            tail
          else if i′ > i then
            alist
          else
            ia′ :: remove i tail

    let rec get_opt i = function
      | [ ] → None
      | (i′, a′) :: tail →
          if i′ = i then
          Some a′
        else
          get_opt i tail

    let min_key = function
      | [] → invalid_arg "PArray.Alists.min_key"
      | (i, _) :: _ → i

    let rec max_key = function
      | [] → invalid_arg "PArray.Alists.max_key"
      | [(i, _)] → i
      | _ :: tail → max_key tail

    let index_base = 0

    let to_option_list parray =
      let rec to_option_list′ i = function
        | [] → []
        | (i′, a′) :: tail →
            (if i′ = i then Some a′ else None) :: to_option_list′ (succ i) tail in
      to_option_list′ index_base parray

    let to_string a2s map =
      match to_option_list map with
      | [] → "[]"
      | [None] → "?"
      | [Some a] → a2s a
      | pairs → ThoList.to_string (function None → "?" | Some a → a2s a) pairs

    let of_pairs pairs =
      List.fold_right
        (fun (i, a) acc →
          if i < index_base then
            invalid_arg "PArray.Alists.of_pairs"
          else
            add i a acc)
        pairs empty

    let to_pairs parray = parray

    let compare _ = compare
    let equal _ = (=)

    type (α, β) taken =
      | Nothing of β t
      | Single of int × α × β t
      | Multiple of int × α × α t

    let take_one project_opt parray =
      let select (k, v) =
        match project_opt k v with
        | Some _ → false
        | None → true
      and project (k, v) =
        match project_opt k v with
        | Some v′ → (k, v′)
        | None → failwith "PArray.Alists.take_one:␣impossible" in
      match List.partition select parray with
      | [], other → Nothing (List.map project other)
      | [(k, v)], other → Single (k, v, List.map project other)
      | (k, v) :: _, _ → Multiple (k, v, remove k parray)

  end

include Alists

module Test =
  struct
    open OUnit

    let project_single _ = function
      | [v] → Some v
      | _ → None

    let suite_take_one =
      "take_one" >:::
        [ "Nothing" >::
            (fun () →
              assert_equal
                (Nothing (of_pairs [(1, "1"); (3, "3")]))
                (take_one project_single (of_pairs [(1, ["1"]); (3, ["3"])])));

          "Single" >::
            (fun () →
              assert_equal
                (Single (2, ["2"; "2"], of_pairs [(1, "1"); (3, "3")]))
                (take_one project_single (of_pairs [(1, ["1"]); (3, ["3"]); (2, ["2"; "2"])])));

          "Multiple" >::
            (fun () →
              assert_equal
                (Multiple (2, ["2"; "2"], of_pairs [(1, ["1"]); (3, ["3"]); (4, [])]))
                (take_one project_single (of_pairs [(1, ["1"]); (3, ["3"]); (2, ["2"; "2"]); (4, [])]))) ]

    let suite =
      "PArray" >:::
        [ suite_take_one ]

  end
```

—K—
More On Strings

K.1 Interface of ThoString

This is a very simple library if string manipulation functions missing in O'Caml's standard library.
strip_prefix prefix string returns *string* with 0 or 1 occurences of a leading *prefix* removed.

```
val strip_prefix : string → string → string
```

strip_prefix_star prefix string returns *string* with any number of leading occurences of *prefix* removed.

```
val strip_prefix_star : char → string → string
```

strip_prefix prefix string returns *string* with a leading *prefix* removed, raises *Invalid_argument* if there's no match.

```
val strip_required_prefix : string → string → string
```

strip_from_first c s returns *s* with everything starting from the first *c* removed. *strip_from_last c s* returns *s* with everything starting from the last *c* removed.

```
val strip_from_first : char → string → string
val strip_from_last : char → string → string
```

index_string pattern string returns the index of the first occurence of *pattern* in *string*, if any. Raises *Not_found*, if *pattern* is not in *string*.

```
val index_string : string → string → int
```

This silently fails if the argument contains both single and double quotes!

```
val quote : string → string
```

The corresponding functions from *String* have become obsolescent with O'Caml 4.0.3. Quanrantine them here.

```
val uppercase : string → string
val lowercase : string → string
```

Ignore the case in comparisons.

```
val compare_caseless : string → string → int
```

Match the regular expression `[A-Za-z][A-Za-z0-9_]*`

```
val valid_fortran_id : string → bool
```

Replace any invalid character by `'_'` and prepend `"N_"` iff the string doesn't start with a letter.

```
val sanitize_fortran_id : string → string
```

```
module Test : sig val suite : OUnit.test end
```

K.2 Implementation of ThoString

```
let strip_prefix p s =
  let lp = String.length p
  and ls = String.length s in
  if lp > ls then
    s
  else
    let rec strip_prefix' i =
      if i ≥ lp then
        String.sub s i (ls − i)
      else if p.[i] ≠ s.[i] then
        s
      else
        strip_prefix' (succ i)
    in
    strip_prefix' 0

let strip_prefix_star p s =
  let ls = String.length s in
  if ls < 1 then
    s
  else
    let rec strip_prefix_star' i =
      if i < ls then begin
        if p ≠ s.[i] then
          String.sub s i (ls − i)
        else
          strip_prefix_star' (succ i)
      end else
        ""
    in
    strip_prefix_star' 0

let strip_required_prefix p s =
  let lp = String.length p
  and ls = String.length s in
  if lp > ls then
    invalid_arg ("strip_required_prefix:␣expected␣‘" ^ p ^ "’␣got␣‘" ^ s ^ "’")
  else
    let rec strip_prefix' i =
      if i ≥ lp then
        String.sub s i (ls − i)
      else if p.[i] ≠ s.[i] then
        invalid_arg ("strip_required_prefix:␣expected␣‘" ^ p ^ "’␣got␣‘" ^ s ^ "’")
      else
        strip_prefix' (succ i)
    in
    strip_prefix' 0

let strip_from_first c s =
  try
    String.sub s 0 (String.index s c)
  with
  | Not_found → s

let strip_from_last c s =
  try
    String.sub s 0 (String.rindex s c)
  with
  | Not_found → s

let index_string pat s =
  let lpat = String.length pat
  and ls = String.length s in
  if lpat = 0 then
    0
  else
    let rec index_string' n =
      let i = String.index_from s n pat.[0] in
      if i + lpat > ls then
```

```
        raise Not_found
      else
        if String.compare pat (String.sub s i lpat) = 0 then
          i
        else
          index_string' (succ i)
    in
    index_string' 0

let quote s =
  if String.contains s ' ' ∨ String.contains s '\n' then begin
    if String.contains s '"' then
      "'" ^ s ^ "'"
    else
      "\"" ^ s ^ "\""
  end else
    s

let uppercase = String.uppercase_ascii
let lowercase = String.lowercase_ascii

let compare_caseless s1 s2 =
  String.compare (lowercase s1) (lowercase s2)

let is_alpha c =
  ('a' ≤ c ∧ c ≤ 'z') ∨ ('A' ≤ c ∧ c ≤ 'Z')

let is_numeric c =
  '0' ≤ c ∧ c ≤ '9'

let is_alphanum c =
  is_alpha c ∨ is_numeric c ∨ c = '_'

let valid_fortran_id s =
  let rec valid_fortran_id' n =
    if n < 0 then
      false
    else if n = 0 then
      is_alpha s.[0]
    else if is_alphanum s.[n] then
      valid_fortran_id' (pred n)
    else
      false in
  valid_fortran_id' (pred (String.length s))

let sanitize_fortran_id s =
  let sanitize s =
    String.map (fun c → if is_alphanum c then c else '_') s in
  if String.length s ≤ 0 then
    invalid_arg "ThoString.sanitize_fortran_id:␣empty"
  else if is_alpha s.[0] then
    sanitize s
  else
    "N_" ^ sanitize s

module Test =
  struct

    open OUnit

    let fortran_empty =
      "empty" >::
        (fun () → assert_equal false (valid_fortran_id ""))

    let fortran_digit =
      "0" >::
        (fun () → assert_equal false (valid_fortran_id "0"))
    let fortran_digit_alpha =
      "0abc" >::
        (fun () → assert_equal false (valid_fortran_id "0abc"))

    let fortran_underscore =
      "_" >::
        (fun () → assert_equal false (valid_fortran_id "_"))

    let fortran_underscore_alpha =
      "_ABC" >::
        (fun () → assert_equal false (valid_fortran_id "_ABC"))

    let fortran_questionmark =
      "A?C" >::
        (fun () → assert_equal false (valid_fortran_id "A?C"))

    let fortran_valid =
      "A_xyz_0_" >::
        (fun () → assert_equal true (valid_fortran_id "A_xyz_0_"))

    let sanitize_digit =
      "0" >::
        (fun () → assert_equal "N_0" (sanitize_fortran_id "0"))

    let sanitize_digit_alpha =
      "0abc" >::
        (fun () → assert_equal "N_0abc" (sanitize_fortran_id "0abc"))

    let sanitize_underscore =
      "_" >::
        (fun () → assert_equal "N__" (sanitize_fortran_id "_"))

    let sanitize_underscore_alpha =
      "_ABC" >::
        (fun () → assert_equal "N__ABC" (sanitize_fortran_id "_ABC"))

    let sanitize_questionmark =
      "A?C" >::
        (fun () → assert_equal "A_C" (sanitize_fortran_id "A?C"))

    let sanitize_valid =
      "A_xyz_0_" >::
        (fun () → assert_equal "A_xyz_0_" (sanitize_fortran_id "A_xyz_0_"))

    let suite_fortran =
      "valid_fortran_id" >:::
        [fortran_empty;
         fortran_digit;
         fortran_digit_alpha;
         fortran_underscore;
         fortran_underscore_alpha;
         fortran_questionmark;
         fortran_valid]

    let suite_sanitize =
      "sanitize_fortran_id" >:::
        [sanitize_digit;
         sanitize_digit_alpha;
         sanitize_underscore;
         sanitize_underscore_alpha;
         sanitize_questionmark;
         sanitize_valid]

    let suite =
      "ThoString" >:::
        [suite_fortran;
         suite_sanitize]

  end
```

—L— Structured Maps

L.1 Interface of ThoMap

L.1.1 Maps to Sets

```
module type Buckets =
  sig

    type t
    type key
    type element
```

The empty map.

```
    val empty : t
```

Add the *element* to the set indexed by *key*. If there is no such set, create it.

```
    val add : key → element → t → t
```

Return the sets as lists of *elements*, indexed by their *key*.

```
    val to_lists : t → (key × element list) list
```

The prototypical application of this module is group all *elements* with matching *keys*. If all *elements* for a given *key* are different, *factorize* is just a more efficient implementation of *ThoList.factorize* on page 683, but the latter keeps duplicate *elements* for a *key*, while this *factorize* keeps only one copy for each *key*.

```
    val factorize : (key × element) list → (key × element list) list
```

factorize_batches is the composition of *factorize* and *List.concat*, but doesn't build the intermediate list.

```
    val factorize_batches : (key × element) list list → (key × element list) list

  end

module Buckets (Key : Map.OrderedType) (Element : Set.OrderedType) : Buckets
        with type key = Key.t and type element = Element.t

module Test : sig val suite : OUnit.test end
```

L.2 Implementation of ThoMap

L.2.1 Maps to Sets

```
module type Buckets =
  sig

    type t
    type key
    type element

    val empty : t
    val add : key → element → t → t
    val to_lists : t → (key × element list) list
    val factorize : (key × element) list → (key × element list) list
    val factorize_batches : (key × element) list list → (key × element list) list

  end

module Buckets (Key : Map.OrderedType) (Element : Set.OrderedType) : Buckets
       with type key = Key.t and type element = Element.t =
  struct

    module Keys = Map.Make(Key)
    module Elements = Set.Make(Element)
    type t = Elements.t Keys.t
    type key = Key.t
    type element = Element.t

    let empty = Keys.empty

    let lookup key map =
      match Keys.find_opt key map with
      | None → Elements.empty
      | Some set → set

    let add key element map =
      Keys.add key (Elements.add element (lookup key map)) map

    let to_lists map =
      List.map (fun (key, set) → (key, Elements.elements set)) (Keys.bindings map)

    let add_pairs initial pairs =
      List.fold_left (fun acc (key, elt) → add key elt acc) initial pairs

    let of_pairs = add_pairs empty

    let factorize pairs =
      to_lists (of_pairs pairs)

    let factorize_batches pairs_list =
      to_lists (List.fold_left add_pairs empty pairs_list)

  end

let random_int_list imax n =
  let imax = succ imax in
  let rec random_int_list' acc i =
    if i = 0 then
      List.rev acc
    else
      random_int_list' (Random.int imax :: acc) (pred i) in
  random_int_list' [] n

let shuffle l =
  let a = Array.of_list l in
  ThoArray.shuffle a;
  Array.to_list a

module Test =
  struct

    open OUnit

    module Integers = struct type t = int let compare = compare end
    module II = Buckets(Integers)(Integers)

    let compare_pair (a1, b1) (a2, b2) =
      let c = compare a1 a2 in
      if c ≠ 0 then
        c
      else
        compare b1 b2

    let ilist = ThoList.range 1 42
    let mod7 i = (i mod 7, i)
    let mod7_ilist = List.map mod7 ilist
    let mod7_ilist_batched = ThoList.chopn 10 mod7_ilist
    let mod7_factorized = List.sort compare_pair (ThoList.factorize mod7_ilist)

    let factorized_to_string l =
      ThoList.to_string
        (fun (i, ilist) → "(" ^ string_of_int i ^ ",␣" ^ ThoList.to_string string_of_int ilist ^ ")" )
        l

    let suite_factorize =
      "factorize" >:::

        [ "int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize mod7_ilist));

          "reversed␣int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize (List.rev mod7_ilist)));

          "shuffled␣int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize (shuffle mod7_ilist))) ]

    let suite_factorize_batches =
      "factorize_batches" >:::

        [ "int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize_batches mod7_ilist_batched));

          "reversed␣int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize_batches (List.rev mod7_ilist_batched)));

          "shuffled␣int␣list" >::
            (fun () →
              assert_equal ~printer : factorized_to_string
                mod7_factorized (II.factorize_batches (shuffle mod7_ilist_batched))) ]

    let suite_buckets =
      "Buckets" >:::

        [ suite_factorize;
          suite_factorize ]

    let suite =
      "ThoMap" >:::
        [ suite_buckets ]

  end
```

—M—
Polymorphic Maps

From [9].

M.1 *Interface of Pmap*

Module *Pmap*: association tables over a polymorphic type[1].

```
module type T =
  sig
    type ('key, α) t
    val empty : ('key, α) t
    val is_empty : ('key, α) t → bool
    val singleton : 'key → α → ('key, α) t
    val add : ('key → 'key → int) → 'key → α → ('key, α) t → ('key, α) t
    val update : ('key → 'key → int) → (α → α → α) →
      'key → α → ('key, α) t → ('key, α) t
    val cons : ('key → 'key → int) → (α → α → α option) →
      'key → α → ('key, α) t → ('key, α) t
    val find : ('key → 'key → int) → 'key → ('key, α) t → α
    val find_opt : ('key → 'key → int) → 'key → ('key, α) t → α option
    val choose : ('key, α) t → 'key × α
    val choose_opt : ('key, α) t → ('key × α) option
    val uncons : ('key, α) t → 'key × α × ('key, α) t
    val uncons_opt : ('key, α) t → ('key × α × ('key, α) t) option
    val elements : ('key, α) t → ('key × α) list
    val mem : ('key → 'key → int) → 'key → ('key, α) t → bool
    val remove : ('key → 'key → int) → 'key → ('key, α) t → ('key, α) t
    val union : ('key → 'key → int) → (α → α → α) →
      ('key, α) t → ('key, α) t → ('key, α) t
    val compose : ('key → 'key → int) → (α → α → α option) →
      ('key, α) t → ('key, α) t → ('key, α) t
    val iter : ('key → α → unit) → ('key, α) t → unit
    val map : (α → β) → ('key, α) t → ('key, β) t
    val mapi : ('key → α → β) → ('key, α) t → ('key, β) t
    val fold : ('key → α → β → β) → ('key, α) t → β → β
    val compare : ('key → 'key → int) → (α → α → int) →
      ('key, α) t → ('key, α) t → int
    val canonicalize : ('key → 'key → int) → ('key, α) t → ('key, α) t
  end
```

Balanced trees: logarithmic access, but representation not unique.

```
module Tree : T
```

Sorted lists: representation unique, but linear access.

```
module List : T
```

[1]Extension of code © 1996 by Xavier Leroy

M.2 Implementation of Pmap

```
module type T =
  sig
    type ('key, α) t
    val empty : ('key, α) t
    val is_empty : ('key, α) t → bool
    val singleton : 'key → α → ('key, α) t
    val add : ('key → 'key → int) → 'key → α → ('key, α) t → ('key, α) t
    val update : ('key → 'key → int) → (α → α → α) →
      'key → α → ('key, α) t → ('key, α) t
    val cons : ('key → 'key → int) → (α → α → α option) →
      'key → α → ('key, α) t → ('key, α) t
    val find : ('key → 'key → int) → 'key → ('key, α) t → α
    val find_opt : ('key → 'key → int) → 'key → ('key, α) t → α option
    val choose : ('key, α) t → 'key × α
    val choose_opt : ('key, α) t → ('key × α) option
    val uncons : ('key, α) t → 'key × α × ('key, α) t
    val uncons_opt : ('key, α) t → ('key × α × ('key, α) t) option
    val elements : ('key, α) t → ('key × α) list
    val mem : ('key → 'key → int) → 'key → ('key, α) t → bool
    val remove : ('key → 'key → int) → 'key → ('key, α) t → ('key, α) t
    val union : ('key → 'key → int) → (α → α → α) →
      ('key, α) t → ('key, α) t → ('key, α) t
    val compose : ('key → 'key → int) → (α → α → α option) →
      ('key, α) t → ('key, α) t → ('key, α) t
    val iter : ('key → α → unit) → ('key, α) t → unit
    val map : (α → β) → ('key, α) t → ('key, β) t
    val mapi : ('key → α → β) → ('key, α) t → ('key, β) t
    val fold : ('key → α → β → β) → ('key, α) t → β → β
    val compare : ('key → 'key → int) → (α → α → int) →
      ('key, α) t → ('key, α) t → int
    val canonicalize : ('key → 'key → int) → ('key, α) t → ('key, α) t
  end

module Tree =
  struct
    type ('key, α) t =
      | Empty
      | Node of ('key, α) t × 'key × α × ('key, α) t × int

    let empty = Empty

    let is_empty = function
      | Empty → true
      | _ → false

    let singleton k d =
      Node (Empty, k, d, Empty, 1)

    let height = function
      | Empty → 0
      | Node (_, _, _, _, h) → h

    let create l x d r =
      let hl = height l and hr = height r in
      Node (l, x, d, r, (if hl ≥ hr then hl + 1 else hr + 1))

    let bal l x d r =
      let hl = match l with Empty → 0 | Node (_, _, _, _, h) → h in
      let hr = match r with Empty → 0 | Node (_, _, _, _, h) → h in
      if hl > hr + 2 then begin
        match l with
        | Empty → invalid_arg "Map.bal"
        | Node (ll, lv, ld, lr, _) →
            if height ll ≥ height lr then
              create ll lv ld (create lr x d r)
            else begin
              match lr with
              | Empty → invalid_arg "Map.bal"
              | Node (lrl, lrv, lrd, lrr, _) →
                  create (create ll lv ld lrl) lrv lrd (create lrr x d r)
            end
      end else if hr > hl + 2 then begin
        match r with
        | Empty → invalid_arg "Map.bal"
        | Node (rl, rv, rd, rr, _) →
            if height rr ≥ height rl then
              create (create l x d rl) rv rd rr
            else begin
              match rl with
              | Empty → invalid_arg "Map.bal"
              | Node (rll, rlv, rld, rlr, _) →
                  create (create l x d rll) rlv rld (create rlr rv rd rr)
            end
      end else
        Node (l, x, d, r, (if hl ≥ hr then hl + 1 else hr + 1))
```

```
  let rec join l x d r =
    match bal l x d r with
    | Empty → invalid_arg "Pmap.join"
    | Node (l', x', d', r', _) as t' →
        let d = height l' − height r' in
        if d < −2 ∨ d > 2 then
          join l' x' d' r'
        else
          t'
```

Merge two trees *t1* and *t2* into one. All elements of *t1* must precede the elements of *t2*. Assumes $height\ t1 - height\ t2 \leq 2$.

```
  let rec merge t1 t2 =
    match t1, t2 with
    | Empty, t → t
    | t, Empty → t
    | Node (l1, v1, d1, r1, h1), Node (l2, v2, d2, r2, h2) →
        bal l1 v1 d1 (bal (merge r1 l2) v2 d2 r2)
```

Same as merge, but does not assume anything about *t1* and *t2*.

```
  let rec concat t1 t2 =
    match t1, t2 with
    | Empty, t → t
    | t, Empty → t
    | Node (l1, v1, d1, r1, h1), Node (l2, v2, d2, r2, h2) →
        join l1 v1 d1 (join (concat r1 l2) v2 d2 r2)
```

Splitting

```
  let rec split cmp x = function
    | Empty → (Empty, None, Empty)
    | Node (l, v, d, r, _) →
        let c = cmp x v in
        if c = 0 then
          (l, Some d, r)
        else if c < 0 then
          let ll, vl, rl = split cmp x l in
          (ll, vl, join rl v d r)
        else (* if c > 0 then *)
```

```
          let lr, vr, rr = split cmp x r in
          (join l v d lr, vr, rr)

let rec find cmp x = function
  | Empty → raise Not_found
  | Node (l, v, d, r, _) →
      let c = cmp x v in
      if c = 0 then
        d
      else if c < 0 then
        find cmp x l
      else (* if c > 0 *)
        find cmp x r

let rec find_opt cmp x = function
  | Empty → None
  | Node (l, v, d, r, _) →
      let c = cmp x v in
      if c = 0 then
        Some d
      else if c < 0 then
        find_opt cmp x l
      else (* if c > 0 *)
        find_opt cmp x r

let rec mem cmp x = function
  | Empty → false
  | Node (l, v, d, r, _) →
      let c = cmp x v in
      if c = 0 then
        true
      else if c < 0 then
        mem cmp x l
      else (* if c > 0 *)
        mem cmp x r

let choose = function
  | Empty → raise Not_found
  | Node (l, v, d, r, _) → (v, d)

let choose_opt = function
  | Empty → None
  | Node (l, v, d, r, _) → Some (v, d)

let uncons = function
  | Empty → raise Not_found
  | Node (l, v, d, r, h) → (v, d, merge l r)

let uncons_opt = function
  | Empty → None
  | Node (l, v, d, r, h) → Some (v, d, merge l r)

let rec remove cmp x = function
  | Empty → Empty
  | Node (l, v, d, r, h) →
      let c = cmp x v in
      if c = 0 then
        merge l r
      else if c < 0 then
        bal (remove cmp x l) v d r
      else (* if c > 0 *)
        bal l v d (remove cmp x r)

let rec cons cmp resolve x data' = function
  | Empty → Node (Empty, x, data', Empty, 1)
  | Node (l, v, data, r, h) →
```

```
          let c = cmp x v in
          if c = 0 then
            match resolve data' data with
            | Some data'' → Node (l, x, data'', r, h)
            | None → merge l r
          else if c < 0 then
            bal (cons cmp resolve x data' l) v data r
          else (* if c > 0 *)
            bal l v data (cons cmp resolve x data' r)

let rec update cmp resolve x data' = function
  | Empty → Node (Empty, x, data', Empty, 1)
  | Node (l, v, data, r, h) →
          let c = cmp x v in
          if c = 0 then
            Node (l, x, resolve data' data, r, h)
          else if c < 0 then
            bal (update cmp resolve x data' l) v data r
          else (* if c > 0 *)
            bal l v data (update cmp resolve x data' r)

let add cmp x data = update cmp (fun n o → n) x data

let rec compose cmp resolve s1 s2 =
  match s1, s2 with
  | Empty, t2 → t2
  | t1, Empty → t1
  | Node (l1, v1, d1, r1, h1), Node (l2, v2, d2, r2, h2) →
          if h1 ≥ h2 then
            if h2 = 1 then
              cons cmp (fun o n → resolve n o) v2 d2 s1
            else begin
              match split cmp v1 s2 with
              | l2', None, r2' →
                    join (compose cmp resolve l1 l2') v1 d1
                      (compose cmp resolve r1 r2')
              | l2', Some d, r2' →
                    begin match resolve d1 d with
                    | None →
                          concat (compose cmp resolve l1 l2')
                            (compose cmp resolve r1 r2')
                    | Some d →
                          join (compose cmp resolve l1 l2') v1 d
                            (compose cmp resolve r1 r2')
                    end
            end
          else
            if h1 = 1 then
              cons cmp resolve v1 d1 s2
            else begin
              match split cmp v2 s1 with
              | l1', None, r1' →
                    join (compose cmp resolve l1' l2) v2 d2
                      (compose cmp resolve r1' r2)
              | l1', Some d, r1' →
                    begin match resolve d d2 with
                    | None →
                          concat (compose cmp resolve l1' l2)
                            (compose cmp resolve r1' r2)
                    | Some d →
                          join (compose cmp resolve l1' l2) v2 d
                            (compose cmp resolve r1' r2)
                    end
```

```
        end
let rec union cmp resolve s1 s2 =
  match s1, s2 with
  | Empty, t2 → t2
  | t1, Empty → t1
  | Node (l1, v1, d1, r1, h1), Node (l2, v2, d2, r2, h2) →
      if h1 ≥ h2 then
        if h2 = 1 then
          update cmp (fun o n → resolve n o) v2 d2 s1
        else begin
          match split cmp v1 s2 with
          | l2', None, r2' →
              join (union cmp resolve l1 l2') v1 d1
                (union cmp resolve r1 r2')
          | l2', Some d, r2' →
              join (union cmp resolve l1 l2') v1 (resolve d1 d)
                (union cmp resolve r1 r2')
        end
      else
        if h1 = 1 then
          update cmp resolve v1 d1 s2
        else begin
          match split cmp v2 s1 with
          | l1', None, r1' →
              join (union cmp resolve l1' l2) v2 d2
                (union cmp resolve r1' r2)
          | l1', Some d, r1' →
              join (union cmp resolve l1' l2) v2 (resolve d d2)
                (union cmp resolve r1' r2)
        end
let rec iter f = function
  | Empty → ()
  | Node (l, v, d, r, _) → iter f l; f v d; iter f r
let rec map f = function
  | Empty → Empty
  | Node (l, v, d, r, h) → Node (map f l, v, f d, map f r, h)
let rec mapi f = function
  | Empty → Empty
  | Node(l, v, d, r, h) → Node (mapi f l, v, f v d, mapi f r, h)
let rec fold f m accu =
  match m with
  | Empty → accu
  | Node (l, v, d, r, _) → fold f l (f v d (fold f r accu))
let rec compare' cmp_k cmp_d l1 l2 =
  match l1, l2 with
  | [], [] → 0
  | [], _ → −1
  | _, [] → 1
  | Empty :: t1, Empty :: t2 → compare' cmp_k cmp_d t1 t2
  | Node (Empty, v1, d1, r1, _) :: t1,
      Node (Empty, v2, d2, r2, _) :: t2 →
      let cv = cmp_k v1 v2 in
      if cv ≠ 0 then begin
        cv
      end else begin
        let cd = cmp_d d1 d2 in
        if cd ≠ 0 then
          cd
```

```
            else
              compare' cmp_k cmp_d (r1 :: t1) (r2 :: t2)
          end
      | Node (l1, v1, d1, r1, _) :: t1, t2 →
          compare' cmp_k cmp_d (l1 :: Node (Empty, v1, d1, r1, 0) :: t1) t2
      | t1, Node (l2, v2, d2, r2, _) :: t2 →
          compare' cmp_k cmp_d t1 (l2 :: Node (Empty, v2, d2, r2, 0) :: t2)

    let compare cmp_k cmp_d m1 m2 = compare' cmp_k cmp_d [m1] [m2]

    let rec elements' accu = function
      | Empty → accu
      | Node (l, v, d, r, _) → elements' ((v, d) :: elements' accu r) l

    let elements s =
      elements' [] s

    let canonicalize cmp m =
      fold (add cmp) m empty

  end

module List =
  struct
    type ('key, α) t = ('key × α) list

    let empty = []

    let is_empty = function
      | [] → true
      | _ → false

    let singleton k d = [(k, d)]

    let rec cons cmp resolve k' d' = function
      | [] → [(k', d')]
      | ((k, d) as kd :: rest) as list →
          let c = cmp k' k in
          if c = 0 then
            match resolve d' d with
            | None → rest
            | Some d'' → (k', d'') :: rest
          else if c < 0 then (* k' < k *)
            (k', d') :: list
          else (* if c > 0, i.e. k < k' *)
            kd :: cons cmp resolve k' d' rest

    let rec update cmp resolve k' d' = function
      | [] → [(k', d')]
      | ((k, d) as kd :: rest) as list →
          let c = cmp k' k in
          if c = 0 then
            (k', resolve d' d) :: rest
          else if c < 0 then (* k' < k *)
            (k', d') :: list
          else (* if c > 0, i.e. k < k' *)
            kd :: update cmp resolve k' d' rest

    let add cmp k' d' list =
      update cmp (fun n o → n) k' d' list

    let rec find cmp k' = function
      | [] → raise Not_found
      | (k, d) :: rest →
          let c = cmp k' k in
          if c = 0 then
            d
          else if c < 0 then (* k' < k *)
            raise Not_found
          else (* if c > 0, i.e. k < k' *)
            find cmp k' rest

    let rec find_opt cmp k' = function
      | [] → None
      | (k, d) :: rest →
          let c = cmp k' k in
          if c = 0 then
            Some d
          else if c < 0 then (* k' < k *)
            None
          else (* if c > 0, i.e. k < k' *)
            find_opt cmp k' rest

    let choose = function
      | [] → raise Not_found
      | kd :: _ → kd

    let rec choose_opt = function
      | [] → None
      | kd :: _ → Some kd

    let uncons = function
      | [] → raise Not_found
      | (k, d) :: rest → (k, d, rest)

    let uncons_opt = function
      | [] → None
      | (k, d) :: rest → Some (k, d, rest)

    let elements list = list

    let rec mem cmp k' = function
      | [] → false
      | (k, d) :: rest →
          let c = cmp k' k in
          if c = 0 then
            true
          else if c < 0 then (* k' < k *)
            false
          else (* if c > 0, i.e. k < k' *)
            mem cmp k' rest

    let rec remove cmp k' = function
      | [] → []
      | ((k, d) as kd :: rest) as list →
          let c = cmp k' k in
          if c = 0 then
            rest
          else if c < 0 then (* k' < k *)
            list
          else (* if c > 0, i.e. k < k' *)
            kd :: remove cmp k' rest

    let rec compare cmp_k cmp_d m1 m2 =
      match m1, m2 with
      | [], [] → 0
      | [], _ → −1
      | _, [] → 1
      | (k1, d1) :: rest1, (k2, d2) :: rest2 →
          let c = cmp_k k1 k2 in
          if c = 0 then begin
            let c' = cmp_d d1 d2 in
            if c' = 0 then
            compare cmp_k cmp_d rest1 rest2
            else
              c'
          end else
            c

    let rec iter f = function
      | [] → ()
      | (k, d) :: rest → f k d; iter f rest

    let rec map f = function
      | [] → []
      | (k, d) :: rest → (k, f d) :: map f rest

    let rec mapi f = function
      | [] → []
      | (k, d) :: rest → (k, f k d) :: mapi f rest

    let rec fold f m accu =
      match m with
      | [] → accu
      | (k, d) :: rest → fold f rest (f k d accu)

    let rec compose cmp resolve m1 m2 =
      match m1, m2 with
      | [], [] → []
      | [], m → m
      | m, [] → m
      | ((k1, d1) as kd1 :: rest1), ((k2, d2) as kd2 :: rest2) →
          let c = cmp k1 k2 in
          if c = 0 then
            match resolve d1 d2 with
            | None → compose cmp resolve rest1 rest2
            | Some d → (k1, d) :: compose cmp resolve rest1 rest2
          else if c < 0 then (* k1 < k2 *)
            kd1 :: compose cmp resolve rest1 m2
          else (* if c > 0, i. e. k2 < k1 *)
            kd2 :: compose cmp resolve m1 rest2

    let rec union cmp resolve m1 m2 =
      match m1, m2 with
      | [], [] → []
      | [], m → m
      | m, [] → m
      | ((k1, d1) as kd1 :: rest1), ((k2, d2) as kd2 :: rest2) →
          let c = cmp k1 k2 in
          if c = 0 then
            (k1, resolve d1 d2) :: union cmp resolve rest1 rest2
          else if c < 0 then (* k1 < k2 *)
            kd1 :: union cmp resolve rest1 m2
          else (* if c > 0, i. e. k2 < k1 *)
            kd2 :: union cmp resolve m1 rest2

    let canonicalize cmp x = x

  end
```

M.3 Interface of Partial

Partial maps that are constructed from assoc lists.

```
module type T =
  sig
```

The domain of the map. It needs to be compatible with *Map.OrderedType.t*

```
  type domain
```

The codomain α can be anything we want.

```
    type α t
```

A list of argument-value pairs is mapped to a partial map. If an argument appears twice, the later value takes precedence.

```
    val of_list : (domain × α) list → α t
```

Two lists of arguments and values (both must have the same length) are mapped to a partial map. Again the later value takes precedence.

```
    val of_lists : domain list → α list → α t
```

If domain and codomain disagree, we must raise an exception or provide a fallback.

```
    exception Undefined of domain
    val apply : α t → domain → α
    val apply_opt : α t → domain → α option
    val apply_with_fallback : (domain → α) → α t → domain → α
```

Iff domain and codomain of the map agree, we can fall back to the identity map.

```
    val auto : domain t → domain → domain

  end

module Make : functor (D : Map.OrderedType) → T with type domain = D.t
module Test : sig val suite : OUnit.test end
```

M.4 Implementation of Partial

```
module type T =
  sig
    type domain
    type α t
    val of_list : (domain × α) list → α t
    val of_lists : domain list → α list → α t
    exception Undefined of domain
    val apply : α t → domain → α
    val apply_opt : α t → domain → α option
    val apply_with_fallback : (domain → α) → α t → domain → α
    val auto : domain t → domain → domain
  end

module Make (D : Map.OrderedType) : T with type domain = D.t =
  struct

    module M = Map.Make (D)

    type domain = D.t
    type α t = α M.t

    let of_list l =
      List.fold_left (fun m (d, v) → M.add d v m) M.empty l

    let of_lists domain values =
      of_list
        (try
           List.map2 (fun d v → (d, v)) domain values
         with
         | Invalid_argument _ (* "List.map2" *) →
             invalid_arg "Partial.of_lists:␣length␣mismatch")

    let auto partial d =
      try
        M.find d partial
      with
      | Not_found → d
    exception Undefined of domain
    let apply partial d =
      try
        M.find d partial
      with
      | Not_found → raise (Undefined d)
    let apply_opt partial d =
      try
        Some (M.find d partial)
      with
      | Not_found → None
    let apply_with_fallback fallback partial d =
      try
        M.find d partial
      with
      | Not_found → fallback d
  end
```

M.4.1 Unit Tests

```
module Test : sig val suite : OUnit.test end =
  struct
    open OUnit
    module P = Make (struct type t = int let compare = compare end)
    let apply_ok =
      "apply/ok" >::
        (fun () →
          let p = P.of_list [ (0,"a"); (1,"b"); (2,"c") ]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "b"; "c" ] (List.map (P.apply p) l))
    let apply_ok2 =
      "apply/ok2" >::
        (fun () →
          let p = P.of_lists [0; 1; 2] ["a"; "b"; "c"]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "b"; "c" ] (List.map (P.apply p) l))
    let apply_shadowed =
      "apply/shadowed" >::
        (fun () →
          let p = P.of_list [ (0,"a"); (1,"b"); (2,"c"); (1,"d") ]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "d"; "c" ] (List.map (P.apply p) l))
    let apply_shadowed2 =
      "apply/shadowed2" >::
        (fun () →
          let p = P.of_lists [0; 1; 2; 1] ["a"; "b"; "c"; "d"]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "d"; "c" ] (List.map (P.apply p) l))
    let apply_mismatch =
      "apply/mismatch" >::
        (fun () →
          assert_raises
            (Invalid_argument "Partial.of_lists:␣length␣mismatch")
            (fun () → P.of_lists [0; 1; 2] ["a"; "b"; "c"; "d"]))
    let suite_apply =
      "apply" >:::
        [apply_ok;
         apply_ok2;
         apply_shadowed;
         apply_shadowed2;
         apply_mismatch]

    let auto_ok =
      "auto/ok" >::
        (fun () →
          let p = P.of_list [ (0,10); (1,11)]
          and l = [ 0; 1; 2 ] in
          assert_equal [ 10; 11; 2 ] (List.map (P.auto p) l))

    let suite_auto =
      "auto" >:::
        [auto_ok]

    let apply_with_fallback_ok =
      "apply_with_fallback/ok" >::
        (fun () →
          let p = P.of_list [ (0,10); (1,11)]
          and l = [ 0; 1; 2 ] in
          assert_equal
            [ 10; 11; − 2 ] (List.map (P.apply_with_fallback (fun n → − n) p) l))

    let suite_apply_with_fallback =
      "apply_with_fallback" >:::
        [apply_with_fallback_ok]

    let suite =
      "Partial" >:::
        [suite_apply;
         suite_auto;
         suite_apply_with_fallback]

    let time () =
      ()

  end
```

—N— Tensor Products

From [9].

N.1 Interface of Product

N.1.1 Lists

Since April 2001, we preserve lexicographic ordering.

val $fold2$: $(\alpha \to \beta \to \gamma \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma \to \gamma$
val $fold3$: $(\alpha \to \beta \to \gamma \to \delta \to \delta) \to \alpha\ list \to \beta\ list \to \gamma\ list \to \delta \to \delta$
val $fold$: $(\alpha\ list \to \beta \to \beta) \to \alpha\ list\ list \to \beta \to \beta$

val $list2$: $(\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$
val $list3$: $(\alpha \to \beta \to \gamma \to \delta) \to \alpha\ list \to \beta\ list \to \gamma\ list \to \delta\ list$
val $list$: $(\alpha\ list \to \beta) \to \alpha\ list\ list \to \beta\ list$

Suppress all *None* in the results.

val $list2\_opt$:
  $(\alpha \to \beta \to \gamma\ option) \to \alpha\ list \to \beta\ list \to \gamma\ list$
val $list3\_opt$:
  $(\alpha \to \beta \to \gamma \to \delta\ option) \to \alpha\ list \to \beta\ list \to \gamma\ list \to \delta\ list$
val $list\_opt$:
  $(\alpha\ list \to \beta\ option) \to \alpha\ list\ list \to \beta\ list$

val $power$: $int \to \alpha\ list \to \alpha\ list\ list$

val $thread$: $\alpha\ list\ list \to \alpha\ list\ list$

N.1.2 Sets

'a_set is actually α *set* for a suitable *set*, but this relation can not be expressed polymorphically (in *set*) in O'Caml. The two sets can be of different type, but we provide a symmetric version as syntactic sugar.

type $\alpha\ set$

type $(\alpha,\ 'a\_set,\ \beta)\ fold = (\alpha \to \beta \to \beta) \to\ 'a\_set \to \beta \to \beta$
type $(\alpha,\ 'a\_set,\ \beta,\ 'b\_set,\ \gamma)\ fold2 =$
    $(\alpha \to \beta \to \gamma \to \gamma) \to\ 'a\_set \to\ 'b\_set \to \gamma \to \gamma$

val $outer$: $(\alpha,\ 'a\_set,\ \gamma)\ fold \to (\beta,\ 'b\_set,\ \gamma)\ fold \to$
  $(\alpha,\ 'a\_set,\ \beta,\ 'b\_set,\ \gamma)\ fold2$
val $outer\_self$: $(\alpha,\ 'a\_set,\ \beta)\ fold \to (\alpha,\ 'a\_set,\ \alpha,\ 'a\_set,\ \beta)\ fold2$

N.2 Implementation of Product

N.2.1 Lists

We use the tail recursive *List.fold_left* over *List.fold_right* for efficiency, but revert the argument lists in order to preserve lexicographic ordering. The argument lists are much shorter than the results, so the cost of the *List.rev* is negligible.

```
let fold2_rev f l1 l2 acc =
  List.fold_left (fun acc1 x1 →
    List.fold_left (fun acc2 x2 → f x1 x2 acc2) acc1 l2) acc l1

let fold2 f l1 l2 acc =
  fold2_rev f (List.rev l1) (List.rev l2) acc

let fold3_rev f l1 l2 l3 acc =
  List.fold_left (fun acc1 x1 → fold2 (f x1) l2 l3 acc1) acc l1

let fold3 f l1 l2 l3 acc =
  fold3_rev f (List.rev l1) (List.rev l2) (List.rev l3) acc
```

If all lists have the same type, there's also

```
let rec fold_rev f ll acc =
  match ll with
  | [] → acc
  | [l] → List.fold_left (fun acc' x → f [x] acc') acc l
  | l :: rest →
      List.fold_left (fun acc' x → fold_rev (fun xr → f (x :: xr)) rest acc') acc l

let fold f ll acc = fold_rev f (List.map List.rev ll) acc

let list2 op l1 l2 =
  fold2 (fun x1 x2 c → op x1 x2 :: c) l1 l2 []

let list3 op l1 l2 l3 =
  fold3 (fun x1 x2 x3 c → op x1 x2 x3 :: c) l1 l2 l3 []

let list op ll =
  fold (fun l c → op l :: c) ll []

let list2_opt op l1 l2 =
  fold2
    (fun x1 x2 c →
      match op x1 x2 with
      | None → c
      | Some op_x1_x2 → op_x1_x2 :: c)
    l1 l2 []

let list3_opt op l1 l2 l3 =
  fold3
    (fun x1 x2 x3 c →
      match op x1 x2 x3 with
      | None → c
      | Some op_x1_x2_x3 → op_x1_x2_x3 :: c)
    l1 l2 l3 []

let list_opt op ll =
  fold
    (fun l c →
      match op l with
      | None → c
      | Some op_l → op_l :: c)
    ll []

let power n l =
  list (fun x → x) (ThoList.clone l n)
```

Reshuffling lists:

$$[[a_1;\ldots;a_k];[b_1;\ldots;b_k];[c_1;\ldots;c_k];\ldots] \to [[a_1;b_1;c_1;\ldots];[a_2;b_2;c_2;\ldots];\ldots] \qquad \text{(N.1)}$$

tho : Is this really an optimal implementation?

```
let thread = function
  | head :: tail →
```

```
      List.map List.rev
        (List.fold_left (fun i acc → List.map2 (fun a b → b :: a) i acc)
           (List.map (fun i → [i]) head) tail)
  | [] → []
```

N.2.2 Sets

The implementation is amazingly simple:

```
type α set

type (α, 'a_set, β) fold = (α → β → β) → 'a_set → β → β
type (α, 'a_set, β, 'b_set, γ) fold2 =
    (α → β → γ → γ) → 'a_set → 'b_set → γ → γ

let outer fold1 fold2 f l1 l2 = fold1 (fun x1 → fold2 (f x1) l2) l1
let outer_self fold f l1 l2 = fold (fun x1 → fold (f x1) l2) l1
```

—O—
(Fiber) Bundles

O.1 Interface of Bundle

See figure O.1 for the geometric intuition behind the bundle structure.

Does the current implementation support faithful projections with a forgetful comparison in the base?

```
module type Elt_Base =
  sig
    type elt
    type base
    val compare_elt : elt → elt → int
    val compare_base : base → base → int
  end
```

```
module type Projection =
  sig
    include Elt_Base
    val pi : elt → base (* projection π : E → B *)
  end
```

Note that writing π^{-1} for the "inverse" is an *abuse-de-langage*, because $\pi^{-1}\circ\pi$ is *not* the identity. It does not map each element to itself but to the fiber that contains it. It is not an automorphism of E, but a map from E to its power set 2^E.

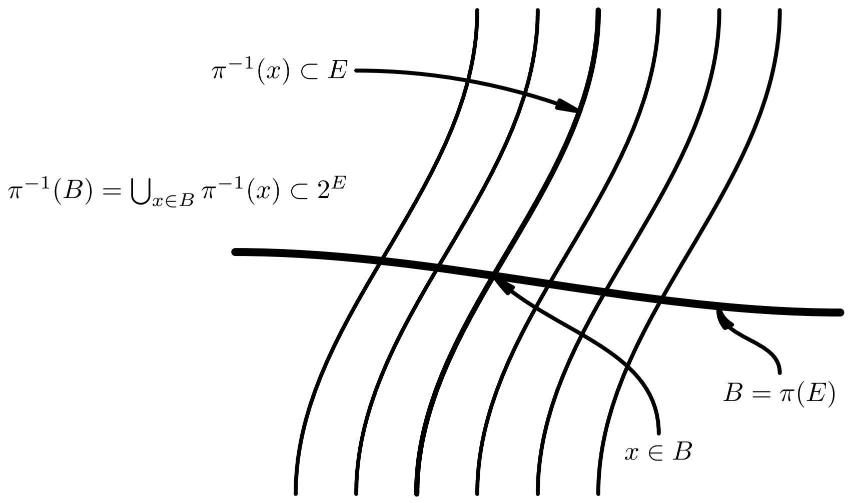


Figure O.1: The bundle structure implemented by *Bundle.T*

```
module type T =
  sig
    type t
    type elt
    type fiber = elt list
    type base
    val empty : t
    val add : t → elt → t
    val of_list : elt list → t
    val pi : elt → base (* projection $\pi : E \to B$ *)
    val inv_pi : t → base → fiber (*"inverse" projection $\pi^{-1} : B \to 2^E$*)
    val base : t → base list
    val fiber : t → elt → fiber (* $\pi^{-1} \circ \pi : E \to 2^E$ *)
    val fibers : t → (base × fiber) list
  end
```

```
module Make (P : Projection) : T with type elt = P.elt and type base = P.base
```

The same thing again, but with a projection that is not hardcoded, but passed as an argument at runtime.

```
module type Dyn =
  sig
    type t
    type elt
    type fiber = elt list
    type base
    val empty : t
    val add : (elt → base) → t → elt → t
    val of_list : (elt → base) → elt list → t
    val inv_pi : t → base → fiber
    val base : t → base list
    val fiber : (elt → base) → t → elt → fiber
    val fibers : t → (base × fiber) list
  end
```

```
module Dyn (P : Elt_Base) : Dyn with type elt = P.elt and type base = P.base
```

O.2 Implementation of Bundle

```
module type Elt_Base =
  sig
    type elt
    type base
    val compare_elt : elt → elt → int
    val compare_base : base → base → int
  end
```

```
module type Dyn =
  sig
    type t
    type elt
    type fiber = elt list
    type base
    val empty : t
    val add : (elt → base) → t → elt → t
    val of_list : (elt → base) → elt list → t
    val inv_pi : t → base → fiber
    val base : t → base list
    val fiber : (elt → base) → t → elt → fiber
    val fibers : t → (base × fiber) list
  end
```

```
module Dyn (P : Elt_Base) =
  struct

    type elt = P.elt
    type base = P.base

    type fiber = elt list

    module InvPi = Map.Make (struct type t = P.base let compare = P.compare_base end)
    module Fiber = Set.Make (struct type t = P.elt let compare = P.compare_elt end)

    type t = Fiber.t InvPi.t

    let empty = InvPi.empty

    let add pi fibers element =
      let base = pi element in
      let fiber =
        try InvPi.find base fibers with Not_found → Fiber.empty in
      InvPi.add base (Fiber.add element fiber) fibers

    let of_list pi list =
      List.fold_left (add pi) InvPi.empty list

    let fibers bundle =
      InvPi.fold (fun base fiber acc → (base, Fiber.elements fiber) :: acc) bundle []

    let base bundle =
      InvPi.fold (fun base fiber acc → base :: acc) bundle []

    let inv_pi bundle base =
      try
        Fiber.elements (InvPi.find base bundle)
      with
      | Not_found → []

    let fiber pi bundle elt =
      inv_pi bundle (pi elt)

  end

module type Projection =
  sig
    include Elt_Base
    val pi : elt → base
  end

module type T =
  sig
    type t
    type elt
    type fiber = elt list
    type base
    val empty : t
    val add : t → elt → t
    val of_list : elt list → t
    val pi : elt → base
    val inv_pi : t → base → fiber
    val base : t → base list
    val fiber : t → elt → fiber
    val fibers : t → (base × fiber) list
  end

module Make (P : Projection) =
  struct

    module D = Dyn (P)

    type elt = D.elt
    type base = D.base
    type fiber = D.fiber
    type t = D.t

    let empty = D.empty
    let pi = P.pi

    let add = D.add pi
    let of_list = D.of_list pi
    let base = D.base
    let inv_pi = D.inv_pi
    let fibers = D.fibers

    let fiber bundle elt =
      inv_pi bundle (pi elt)

  end
```

—P—
Power Sets

P.1 Interface of *PowSet*

Manipulate the power set, i. e. the set of all subsets, of an set *Ordered_Type*. The concrete order is actually irrelevant, we just need it to construct *Set.S*s in the implementation. In fact, what we are implementating is the *free semilattice* generated from the set of subsets of *Ordered_Type*, where the join operation is the set union.

The non trivial operation is *basis*, which takes a set of subsets and returns the smallest set of disjoint subsets from which the argument can be reconstructed by forming unions. It is used in O'Mega for finding coarsest partitions of sets of partiticles.

Eventually, this could be generalized from *power set* or *semi lattice* to *lattice* with a notion of subtraction.

```
module type Ordered_Type =
  sig
    type t
    val compare : t → t → int
```

Debugging ...

```
    val to_string : t → string
  end
```

```
module type T =
  sig
    type elt
    type t

    val empty : t
    val is_empty : t → bool
```

Set union (a. k. a. join).

```
    val union : t list → t
```

Construct the abstract type from a list of subsets represented as lists and the inverse operation.

```
    val of_lists : elt list list → t
    val to_lists : t → elt list list
```

The smallest set of disjoint subsets that generates the given subset.

```
    val basis : t → t
```

Debugging ...

```
    val to_string : t → string
  end
```

```
module Make (E : Ordered_Type) : T with type elt = E.t
```

P.2 Implementation of *PowSet*

```
module type Ordered_Type =
  sig
```

```
    type t
    val compare : t → t → int
    val to_string : t → string
  end

module type T =
  sig
    type elt
    type t
    val empty : t
    val is_empty : t → bool
    val union : t list → t
    val of_lists : elt list list → t
    val to_lists : t → elt list list
    val basis : t → t
    val to_string : t → string
  end

module Make (E : Ordered_Type) =
  struct

    type elt = E.t

    module ESet = Set.Make (E)
    type set = ESet.t

    module EPowSet = Set.Make (ESet)
    type t = EPowSet.t

    let empty = EPowSet.empty
    let is_empty = EPowSet.is_empty


    let union s_list =
      List.fold_right EPowSet.union s_list EPowSet.empty

    let set_to_string set =
      "{" ^ String.concat "," (List.map E.to_string (ESet.elements set)) ^ "}"

    let to_string powset =
      "{" ^ String.concat "," (List.map set_to_string (EPowSet.elements powset)) ^ "}"

    let set_of_list = ESet.of_list

    let of_lists lists =
      List.fold_right
        (fun list acc → EPowSet.add (ESet.of_list list) acc)
        lists EPowSet.empty

    let to_lists ps =
      List.map ESet.elements (EPowSet.elements ps)
```

product $(s_1, s_2) = s_1 \circ s_2 = \{s_1 \setminus s_2, s_1 \cap s_2, s_2 \setminus s_1\} \setminus \{\emptyset\}$

```
    let product s1 s2 =
      List.fold_left
        (fun pset set → if ESet.is_empty set then pset else EPowSet.add set pset)
        EPowSet.empty [ESet.diff s1 s2; ESet.inter s1 s2; ESet.diff s2 s1]

    let disjoint s1 s2 =
      ESet.is_empty (ESet.inter s1 s2)
```

In *augment_basis_overlapping* $(s, \{s_i\}_i)$, we are guaranteed that

$$\forall_i : s \cap s_i \neq \emptyset \tag{P.1a}$$

$$\forall_{i \neq j} : s_i \cap s_j = \emptyset\,. \tag{P.1b}$$

Therefore from (P.1b)

$$\forall_{i \neq j} : (s \cap s_i) \cap (s \cap s_j) = s \cap (s_i \cap s_j) = s \cap \emptyset = \emptyset \tag{P.2a}$$

$$\forall_{i\neq j} : (s_i \setminus s) \cap (s_j \setminus s) \subset s_i \cap s_j = \emptyset \tag{P.2b}$$
$$\forall_{i\neq j} : (s \setminus s_i) \cap (s_j \setminus s) \subset s \cap \bar{s} = \emptyset \tag{P.2c}$$
$$\forall_{i\neq j} : (s \cap s_i) \cap (s_j \setminus s) \subset s \cap \bar{s} = \emptyset\,, \tag{P.2d}$$

but in general

$$\exists_{i\neq j} : (s \setminus s_i) \cap (s \setminus s_j) \neq \emptyset \tag{P.3a}$$
$$\exists_{i\neq j} : (s \setminus s_i) \cap (s \cap s_j) \neq \emptyset\,, \tag{P.3b}$$

because, e. g., for $s_i = \{i\}$ and $s = \{1,2,3\}$

$$(s \setminus s_1) \cap (s \setminus s_2) = \{2,3\} \cap \{1,3\} = \{3\} \tag{P.4a}$$
$$(s \setminus s_1) \cap (s \cap s_2) = \{2,3\} \cap \{2\} = \{2\}\,. \tag{P.4b}$$

Summarizing:

$\forall_{i\neq j}:\ A_i \cap A_j$	$s_j \setminus s$	$s \cap s_j$	$s \setminus s_j$
$s_i \setminus s$	$\emptyset$	$\emptyset$	$\emptyset$
$s \cap s_i$	$\emptyset$	$\emptyset$	$\neq\emptyset$
$s \setminus s_i$	$\emptyset$	$\neq\emptyset$	$\neq\emptyset$

Fortunately, we also know from (P.1a) that

$$\forall_i : |s \setminus s_i| < |s| \tag{P.5a}$$
$$\forall_i : |s \cap s_i| < \min(|s|, |s_i|) \tag{P.5b}$$
$$\forall_i : |s_i \setminus s| < |s_i| \tag{P.5c}$$

and can call *basis* recursively without risking non-termination.

```
    let rec basis ps =
      EPowSet.fold augment_basis ps EPowSet.empty

    and augment_basis s ps =
      if EPowSet.mem s ps then
        ps
      else
        let no_overlaps, overlaps = EPowSet.partition (disjoint s) ps in
        if EPowSet.is_empty overlaps then
          EPowSet.add s ps
        else
          EPowSet.union no_overlaps (augment_basis_overlapping s overlaps)

    and augment_basis_overlapping s ps =
      basis (EPowSet.fold (fun s' → EPowSet.union (product s s')) ps EPowSet.empty)

  end
```

—Q— Combinatorics

Q.1 Interface of Combinatorics

This type is defined just for documentation. Below, most functions will construct a (possibly nested) *list* of partitions or permutations of a α *seq*.

type α *seq* = α *list*

Q.1.1 Simple Combinatorial Functions

The functions

$$\begin{aligned} \textit{factorial} &: n \to n! && \text{(Q.1a)} \\ \textit{binomial} &: (n,k) \to \binom{n}{k} = \frac{n!}{k!(n-k)!} && \text{(Q.1b)} \\ \textit{multinomial} &: [n_1; n_2; \ldots; n_k] \to \binom{n_1+n_2+\ldots+n_k}{n_1, n_2, \ldots, n_k} = \frac{(n_1+n_2+\ldots+n_k)!}{n_1! n_2! \cdots n_k!} && \text{(Q.1c)} \end{aligned}$$

have not been optimized. They can quickly run out of the range of native integers.

val *factorial* : *int* → *int*
val *binomial* : *int* → *int* → *int*
val *multinomial* : *int list* → *int*

symmetry l returns the size of the symmetric group on l, i. e. the product of the factorials of the numbers of identical elements.

val *symmetry* : α *list* → *int*

Q.1.2 Partitions

partitions $[n_1; n_2; \ldots; n_k]$ $[x_1; x_2; \ldots; x_n]$, where $n = n_1 + n_2 + \ldots + n_k$, returns all inequivalent partitions of $[x_1; x_2; \ldots; x_n]$ into parts of size n_1, n_2, ..., n_k. The order of the n_i is not respected. There are

$$\frac{1}{S(n_1, n_2, \ldots, n_k)} \binom{n_1+n_2+\ldots+n_k}{n_1, n_2, \ldots, n_k} \tag{Q.2}$$

such partitions, where the symmetry factor $S(n_1, n_2, \ldots, n_k)$ is the size of the permutation group of $[n_1; n_2; \ldots; n_k]$ as determined by the function *symmetry*.

val *partitions* : *int list* → α *seq* → α *seq list list*

ordered_partitions is identical to *partitions*, except that the order of the n_i is respected. There are

$$\binom{n_1+n_2+\ldots+n_k}{n_1, n_2, \ldots, n_k} \tag{Q.3}$$

such partitions.

val *ordered_partitions* : *int list* → α *seq* → α *seq list list*

keystones m l is equivalent to *partitions* m l, except for the special case when the length of l is even and m contains a part that has exactly half the length of l. In this case only the half of the partitions is created that has the head of l in the longest part.

val *keystones* : $int\ list \to \alpha\ seq \to \alpha\ seq\ list\ list$

It can be beneficial to factorize a common part in the partitions and keystones:

val *factorized_partitions* : $int\ list \to \alpha\ seq \to (\alpha\ seq \times \alpha\ seq\ list\ list)\ list$
val *factorized_keystones* : $int\ list \to \alpha\ seq \to (\alpha\ seq \times \alpha\ seq\ list\ list)\ list$

Special Cases

partitions is built from components that can be convenient by themselves, even thepugh they are just special cases of *partitions*.

split k l returns the list of all inequivalent splits of the list l into one part of length k and the rest. There are

$$\frac{1}{S(|l|-k,k)}\binom{|l|}{k} \tag{Q.4}$$

such splits. After replacing the pairs by two-element lists, *split k l* is equivalent to *partitions* $[k;\ length\ l\ -\ k]\ l$.

val *split* : $int \to \alpha\ seq \to (\alpha\ seq \times \alpha\ seq)\ list$

Create both equipartitions of lists of even length. There are

$$\binom{|l|}{k} \tag{Q.5}$$

such splits. After replacing the pairs by two-element lists, the result of *ordered_split k l* is equivalent to *ordered_partitions* $[k;\ length\ l\ -\ k]\ l$.

val *ordered_split* : $int \to \alpha\ seq \to (\alpha\ seq \times \alpha\ seq)\ list$

multi_split n k l returns the list of all inequivalent splits of the list l into n parts of length k and the rest.

val *multi_split* : $int \to int \to \alpha\ seq \to (\alpha\ seq\ list \times \alpha\ seq)\ list$
val *ordered_multi_split* : $int \to int \to \alpha\ seq \to (\alpha\ seq\ list \times \alpha\ seq)\ list$

Q.1.3 Choices

choose $n\,[x_1;x_2;\ldots;x_n]$ returns the list of all n-element subsets of $[x_1;x_2;\ldots;x_n]$. *choose* n is equivalent to $(map\,fst)\circ(ordered\_split\ n)$.

val *choose* : $int \to \alpha\ seq \to \alpha\ seq\ list$

multi_choose n k is equivalent to $(map\,fst)\circ(multi\_split\ n\ k)$.

val *multi_choose* : $int \to int \to \alpha\ seq \to \alpha\ seq\ list\ list$
val *ordered_multi_choose* : $int \to int \to \alpha\ seq \to \alpha\ seq\ list\ list$

Q.1.4 Permutations

val *permute* : $\alpha\ seq \to \alpha\ seq\ list$

Graded Permutations

val *permute_signed* : $\alpha\ seq \to (int \times \alpha\ seq)\ list$
val *permute_even* : $\alpha\ seq \to \alpha\ seq\ list$
val *permute_odd* : $\alpha\ seq \to \alpha\ seq\ list$
val *permute_cyclic* : $\alpha\ seq \to \alpha\ seq\ list$
val *permute_cyclic_signed* : $\alpha\ seq \to (int \times \alpha\ seq)\ list$

Tensor Products of Permutations

In other words: permutations which respect compartmentalization.

val *permute_tensor* : $\alpha\ seq\ list \to \alpha\ seq\ list\ list$
val *permute_tensor_signed* : $\alpha\ seq\ list \to (int \times \alpha\ seq\ list)\ list$
val *permute_tensor_even* : $\alpha\ seq\ list \to \alpha\ seq\ list\ list$
val *permute_tensor_odd* : $\alpha\ seq\ list \to \alpha\ seq\ list\ list$

val *sign* : $?cmp : (\alpha \to \alpha \to int) \to \alpha\ seq \to int$

Sorting

```
val sort_signed : ?cmp : (α → α → int) → α seq → int × α seq
```

Unit Tests

```
module Test : sig val suite : OUnit.test end
```

Q.2 Implementation of Combinatorics

```
type α seq = α list
```

Q.2.1 Simple Combinatorial Functions

```
let rec factorial′ fn n =
  if n < 1 then
    fn
  else
    factorial′ (n × fn) (pred n)

let factorial n =
  let result = factorial′ 1 n in
  if result < 0 then
    invalid_arg "Combinatorics.factorial␣overflow"
  else
    result
```

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} = \frac{n(n-1)\cdots(k+1)}{(n-k)(n-k-1)\cdots 1} = \begin{cases} B_{n-k+1}(n,k) & \text{for } k \le \lfloor n/2 \rfloor \\ B_{k+1}(n,n-k) & \text{for } k > \lfloor n/2 \rfloor \end{cases} \tag{Q.6}$$

where

$$B_{n_\text{min}}(n,k) = \begin{cases} nB_{n_\text{min}}(n-1,k) & \text{for } n \ge n_\text{min} \\ \frac{1}{k}B_{n_\text{min}}(n,k-1) & \text{for } k > 1 \\ 1 & \text{otherwise} \end{cases} \tag{Q.7}$$

```
let rec binomial′ n_min n k acc =
  if n ≥ n_min then
    binomial′ n_min (pred n) k (n × acc)
  else if k > 1 then
    binomial′ n_min n (pred k) (acc / k)
  else
    acc

let binomial n k =
  if k > n / 2 then
    binomial′ (k + 1) n (n − k) 1
  else
    binomial′ (n − k + 1) n k 1
```

Overflows later, but takes much more time:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{Q.8}$$

```
let rec slow_binomial n k =
  if n < 0 ∨ k < 0 then
    invalid_arg "Combinatorics.binomial"
  else if k = 0 ∨ k = n then
    1
  else
    slow_binomial (pred n) k + slow_binomial (pred n) (pred k)

let multinomial n_list =
  List.fold_left (fun acc n → acc / (factorial n))
    (factorial (List.fold_left (+) 0 n_list)) n_list

let symmetry l =
  List.fold_left (fun s (n, _) → s × factorial n) 1 (ThoList.classify l)
```

Q.2.2 Partitions

The inner steps of the recursion (i. e. $n = 1$) are expanded as follows

$$\begin{aligned}
split'(1, [p_k; p_{k-1}; \ldots; p_1], [x_l; x_{l-1}; \ldots; x_1], [x_{l+1}; x_{l+2}; \ldots; x_m]) = \\
[([p_1; \ldots; p_k; x_{l+1}], [x_1; \ldots; x_l; x_{l+2}; \ldots; x_m]); \\
([p_1; \ldots; p_k; x_{l+2}], [x_1; \ldots; x_l; x_{l+1}; x_{l+3} \ldots; x_m]); \ldots; \\
([p_1; \ldots; p_k; x_m], [x_1; \ldots; x_l; x_{l+1}; \ldots; x_{m-1}])]
\end{aligned} \tag{Q.9}$$

while the outer steps (i. e. $n > 1$) perform the same with one element moved from the last argument to the first argument. At the nth level we have

$$\begin{aligned}
split'(n, [p_k; p_{k-1}; \ldots; p_1], [x_l; x_{l-1}; \ldots; x_1], [x_{l+1}; x_{l+2}; \ldots; x_m]) = \\
[([p_1; \ldots; p_k; x_{l+1}; x_{l+2}; \ldots; x_{l+n}], [x_1; \ldots; x_l; x_{l+n+1}; \ldots; x_m]); \ldots; \\
([p_1; \ldots; p_k; x_{m-n+1}; x_{m-n+2}; \ldots; x_m], [x_1; \ldots; x_l; x_{l+1}; \ldots; x_{m-n}])]
\end{aligned} \tag{Q.10}$$

where the order of the $[x_1; x_2; \ldots; x_m]$ is maintained in the partitions. Variations on this multiple recursion idiom are used many times below.

```
let rec split' n rev_part rev_head = function
  | [] → []
  | x :: tail →
      let rev_part' = x :: rev_part
      and parts = split' n rev_part (x :: rev_head) tail in
      if n < 1 then
        failwith "Combinatorics.split': can't happen"
      else if n = 1 then
        (List.rev rev_part', List.rev_append rev_head tail) :: parts
      else
        split' (pred n) rev_part' rev_head tail @ parts
```

Kick off the recursion for $0 < n < |l|$ and handle the cases $n \in \{0, |l|\}$ explicitely. Use reflection symmetry for a small optimization.

```
let ordered_split_unsafe n abs_l l =
  let abs_l = List.length l in
  if n = 0 then
    [[], l]
  else if n = abs_l then
    [l, []]
  else if n ≤ abs_l / 2 then
    split' n [] [] l
  else
    List.rev_map (fun (a, b) → (b, a)) (split' (abs_l − n) [] [] l)
```

Check the arguments and call the workhorse:

```
let ordered_split n l =
  let abs_l = List.length l in
  if n < 0 ∨ n > abs_l then
    invalid_arg "Combinatorics.ordered_split"
  else
    ordered_split_unsafe n abs_l l
```

Handle equipartitions specially:

```
let split n l =
  let abs_l = List.length l in
  if n < 0 ∨ n > abs_l then
    invalid_arg "Combinatorics.split"
  else begin
    if 2 × n = abs_l then
      match l with
      | [] → failwith "Combinatorics.split:␣can't␣happen"
      | x :: tail →
          List.map (fun (p1, p2) → (x :: p1, p2)) (split' (pred n) [] [] tail)
    else
      ordered_split_unsafe n abs_l l
  end
```

If we chop off parts repeatedly, we can either keep permutations or suppress them. Generically, *attach_to_fst* has type

$$(\alpha \times \beta)\ list \to \alpha\ list \to (\alpha\ list \times \beta)\ list \to (\alpha\ list \times \beta)\ list$$

and semantics

$$attach\_to\_fst([(a_1,b_1),(a_2,b_2),\ldots,(a_m,b_m)],[a'_1,a'_2,\ldots]) = \\ [([a_1,a'_1,\ldots],b_1),([a_2,a'_1,\ldots],b_2),\ldots,([a_m,a'_1,\ldots],b_m)] \qquad (Q.11)$$

(where some of the result can be filtered out), assumed to be prepended to the final argument.

```
let rec multi_split' attach_to_fst n size splits =
  if n ≤ 0 then
    splits
  else
    multi_split' attach_to_fst (pred n) size
      (List.fold_left (fun acc (parts, tail) →
        attach_to_fst (ordered_split size tail) parts acc) [] splits)
```

```
let attach_to_fst_unsorted splits parts acc =
  List.fold_left (fun acc' (p, rest) → (p :: parts, rest) :: acc') acc splits
```

Similarly, if the secod argument is a list of lists:

```
let prepend_to_fst_unsorted splits parts acc =
  List.fold_left (fun acc' (p, rest) → (p @ parts, rest) :: acc') acc splits
```

```
let attach_to_fst_sorted splits parts acc =
  match parts with
  | [] → List.fold_left (fun acc' (p, rest) → ([p], rest) :: acc') acc splits
  | p :: _ as parts →
      List.fold_left (fun acc' (p', rest) →
        if p' > p then
          (p' :: parts, rest) :: acc'
        else
          acc') acc splits
```

```
let multi_split n size l =
  multi_split' attach_to_fst_sorted n size [([], l)]
```

```
let ordered_multi_split n size l =
  multi_split' attach_to_fst_unsorted n size [([], l)]
```

```
let rec partitions' splits = function
  | [] → List.map (fun (h, r) → (List.rev h, r)) splits
  | (1, size) :: more →
      partitions'
```

```
          (List.fold_left (fun acc (parts, rest) →
             attach_to_fst_unsorted (split size rest) parts acc)
              [] splits) more
  | (n, size) :: more →
        partitions′
          (List.fold_left (fun acc (parts, rest) →
             prepend_to_fst_unsorted (multi_split n size rest) parts acc)
              [] splits) more

let partitions multiplicities l =
  if List.fold_left (+) 0 multiplicities ≠ List.length l then
    invalid_arg "Combinatorics.partitions"
  else
    List.map fst (partitions′ [([], l)]
                      (ThoList.classify (List.sort compare multiplicities)))

let rec ordered_partitions′ splits = function
  | [] → List.map (fun (h, r) → (List.rev h, r)) splits
  | size :: more →
        ordered_partitions′
          (List.fold_left (fun acc (parts, rest) →
             attach_to_fst_unsorted (ordered_split size rest) parts acc)
              [] splits) more

let ordered_partitions multiplicities l =
  if List.fold_left (+) 0 multiplicities ≠ List.length l then
    invalid_arg "Combinatorics.ordered_partitions"
  else
    List.map fst (ordered_partitions′ [([], l)] multiplicities)

let hdtl = function
  | [] → invalid_arg "Combinatorics.hdtl"
  | h :: t → (h, t)

let factorized_partitions multiplicities l =
  ThoList.factorize (List.map hdtl (partitions multiplicities l))
```

In order to construct keystones (cf. chapter 3), we must eliminate reflectionsc consistently. For this to work, the lengths of the parts *must not* be reordered arbitrarily. Ordering with monotonously fallings lengths would be incorrect however, because then some remainders could fake a reflection symmetry and partitions would be dropped erroneously. Therefore we put the longest first and order the remaining with rising lengths:

```
let longest_first l =
  match ThoList.classify (List.sort (fun n1 n2 → compare n2 n1) l) with
  | [] → []
  | longest :: rest → longest :: List.rev rest

let keystones multiplicities l =
  if List.fold_left (+) 0 multiplicities ≠ List.length l then
    invalid_arg "Combinatorics.keystones"
  else
    List.map fst (partitions′ [([], l)] (longest_first multiplicities))

let factorized_keystones multiplicities l =
  ThoList.factorize (List.map hdtl (keystones multiplicities l))
```

Q.2.3 Choices

The implementation is very similar to *split*′, but here we don't have to keep track of the complements of the chosen sets.

```
let rec choose′ n rev_choice = function
  | [] → []
  | x :: tail →
      let rev_choice′ = x :: rev_choice
      and choices = choose′ n rev_choice tail in
```

```
    if $n < 1$ then
      $failwith$ "Combinatorics.choose':␣can't␣happen"
    else if $n = 1$ then
      $List.rev\ rev\_choice' :: choices$
    else
      $choose'\ (pred\ n)\ rev\_choice'\ tail\ @\ choices$
```

choose n is equivalent to $(List.map\,fst) \circ (split\_ordered\, n)$, but more efficient.

```
let $choose\ n\ l$ =
  let $abs\_l = List.length\ l$ in
  if $n < 0$ then
    $invalid\_arg$ "Combinatorics.choose"
  else if $n > abs\_l$ then
    $[\,]$
  else if $n = 0$ then
    $[[\,]]$
  else if $n = abs\_l$ then
    $[l]$
  else
    $choose'\ n\ [\,]\ l$
```

```
let $multi\_choose\ n\ size\ l$ =
  $List.map\ fst\ (multi\_split\ n\ size\ l)$
```

```
let $ordered\_multi\_choose\ n\ size\ l$ =
  $List.map\ fst\ (ordered\_multi\_split\ n\ size\ l)$
```

Q.2.4 Permutations

```
let rec $insert\ x$ = function
  | $[\,] \rightarrow [[x]]$
  | $h :: t$ as $l \rightarrow$
      $(x :: l) :: List.rev\_map$ (fun $l' \rightarrow h :: l'$) $(insert\ x\ t)$
```

```
let $permute\ l$ =
  $List.fold\_left$ (fun $acc\ x \rightarrow ThoList.rev\_flatmap\ (insert\ x)\ acc$) $[[\,]]\ l$
```

Graded Permutations

```
let rec $insert\_signed\ x$ = function
  | $(eps,\ [\,]) \rightarrow [(eps,\ [x])]$
  | $(eps,\ h :: t) \rightarrow (eps,\ x :: h :: t) ::$
      $(List.map$ (fun $(eps',\ l') \rightarrow (-eps',\ h :: l')$) $(insert\_signed\ x\ (eps,\ t)))$
```

```
let rec $permute\_signed'$ = function
  | $(eps,\ [\,]) \rightarrow [(eps,\ [\,])]$
  | $(eps,\ h :: t) \rightarrow ThoList.flatmap\ (insert\_signed\ h)\ (permute\_signed'\ (eps,\ t))$
```

```
let $permute\_signed\ l$ =
  $permute\_signed'\ (1,\ l)$
```

The following are wasting at most a factor of two and there's probably no point in improving on this . . .

```
let $filter\_sign\ s\ l$ =
  $List.map\ snd\ (List.filter$ (fun $(eps,\ \_) \rightarrow eps = s$) $l)$
```

```
let $permute\_even\ l$ =
  $filter\_sign\ 1\ (permute\_signed\ l)$
```

```
let $permute\_odd\ l$ =
  $filter\_sign\ (-1)\ (permute\_signed\ l)$
```

We have a slight inconsistency here: *permute* $[\,] = [[\,]]$, while *permute_cyclic* $[\,] = [\,]$. I don't know if it is worth fixing.

```
let permute_cyclic l =
  let rec permute_cyclic' acc before = function
    | [] → List.rev acc
    | x :: rest as after →
        permute_cyclic' ((after @ List.rev before) :: acc) (x :: before) rest
  in
  permute_cyclic' [] [] l
```

Algorithm: toggle the signs and at the end map all signs to +1, iff the last sign is positive, i. e. there's an odd number of elements.

```
let permute_cyclic_signed l =
  let rec permute_cyclic_signed' eps acc before = function
    | [] →
        if eps > 0 then
          List.rev_map (fun (_, p) → (1, p)) acc
        else
          List.rev acc
    | x :: rest as after →
        let eps' = − eps in
        permute_cyclic_signed' eps' ((eps', after @ List.rev before) :: acc) (x :: before) rest
  in
  permute_cyclic_signed' (−1) [] [] l
```

Tensor Products of Permutations

```
let permute_tensor ll =
  Product.list (fun l → l) (List.map permute ll)

let join_signs l =
  let el, pl = List.split l in
  (List.fold_left (fun acc x → x × acc) 1 el, pl)

let permute_tensor_signed ll =
  Product.list join_signs (List.map permute_signed ll)

let permute_tensor_even l =
  filter_sign 1 (permute_tensor_signed l)

let permute_tensor_odd l =
  filter_sign (−1) (permute_tensor_signed l)
```

Sorting

```
let insert_inorder_signed order x (eps, l) =
  let rec insert eps' accu = function
    | [] → (eps × eps', List.rev_append accu [x])
    | h :: t →
        if order x h = 0 then
          invalid_arg
            "Combinatorics.insert_inorder_signed:␣identical␣elements"
        else if order x h < 0 then
          (eps × eps', List.rev_append accu (x :: h :: t))
        else
          insert (−eps') (h :: accu) t
  in
  insert 1 [] l

let sort_signed ?(cmp = Stdlib.compare) l =
  List.fold_right (insert_inorder_signed cmp) l (1, [])

let sign ?(cmp = Stdlib.compare) l =
  let eps, _ = sort_signed ˜cmp l in
  eps
```

```
let sign2 ?(cmp = Stdlib.compare) l =
  let a = Array.of_list l in
  let eps = ref 1 in
  for j = 0 to Array.length a - 1 do
    for i = 0 to j - 1 do
      if cmp a.(i) a.(j) > 0 then
        eps := - !eps
    done
  done;
  !eps

module Test =
  struct

    open OUnit

    let to_string =
      ThoList.to_string (ThoList.to_string string_of_int)

    let assert_equal_perms =
      assert_equal ~printer : to_string

    let count_permutations n =
      let factorial_n = factorial n
      and range = ThoList.range 1 n in
      let sorted = List.sort compare (permute range) in
      (* Verify the count ... *)
      assert_equal factorial_n (List.length sorted);
      (* ... check that they're all different ... *)
      assert_equal factorial_n (List.length (ThoList.uniq sorted));
      (* ... make sure that they a all permutations. *)
      assert_equal_perms
        [range] (ThoList.uniq (List.map (List.sort compare) sorted))

    let suite_permute =
      "permute" >:::
        [ "permute␣[]" >::
            (fun () →
              assert_equal_perms [[]] (permute []));
          "permute␣[1]" >::
            (fun () →
              assert_equal_perms [[1]] (permute [1]));
          "permute␣[1;2;3]" >::
            (fun () →
              assert_equal_perms
                [ [2; 3; 1]; [2; 1; 3]; [3; 2; 1];
                  [1; 3; 2]; [1; 2; 3]; [3; 1; 2] ]
                (permute [1; 2; 3]));
          "permute␣[1;2;3;4]" >::
            (fun () →
              assert_equal_perms
                [ [3; 4; 1; 2]; [3; 1; 2; 4]; [3; 1; 4; 2];
                  [4; 3; 1; 2]; [1; 4; 2; 3]; [1; 2; 3; 4];
                  [1; 2; 4; 3]; [4; 1; 2; 3]; [1; 4; 3; 2];
                  [1; 3; 2; 4]; [1; 3; 4; 2]; [4; 1; 3; 2];
                  [3; 4; 2; 1]; [3; 2; 1; 4]; [3; 2; 4; 1];
                  [4; 3; 2; 1]; [2; 4; 1; 3]; [2; 1; 3; 4];
                  [2; 1; 4; 3]; [4; 2; 1; 3]; [2; 4; 3; 1];
                  [2; 3; 1; 4]; [2; 3; 4; 1]; [4; 2; 3; 1] ]
                (permute [1; 2; 3; 4]));
          "count␣permute␣5" >::
            (fun () → count_permutations 5);
          "count␣permute␣6" >::
            (fun () → count_permutations 6);
          "count␣permute␣7" >::
            (fun () → count_permutations 7);
          "count␣permute␣8" >::
            (fun () → count_permutations 8);
          "cyclic␣[]" >::
            (fun () →
              assert_equal_perms [ ] (permute_cyclic [ ]));
          "cyclic␣[1]" >::
            (fun () →
              assert_equal_perms [[1]] (permute_cyclic [1]));
          "cyclic␣[1;2;3]" >::
            (fun () →
              assert_equal_perms
                [[1; 2; 3]; [2; 3; 1]; [3; 1; 2]]
                (permute_cyclic [1; 2; 3]));
          "cyclic␣[1;2;3;4]" >::
            (fun () →
              assert_equal_perms
                [[1; 2; 3; 4]; [2; 3; 4; 1]; [3; 4; 1; 2]; [4; 1; 2; 3]]
                (permute_cyclic [1; 2; 3; 4]));
          "cyclic␣[1;2;3]␣signed" >::
            (fun () →
              assert_equal
                [(1, [1; 2; 3]); (1, [2; 3; 1]); (1, [3; 1; 2])]
                (permute_cyclic_signed [1; 2; 3]));
          "cyclic␣[1;2;3;4]␣signed" >::
            (fun () →
              assert_equal
                [(1, [1; 2; 3; 4]); (−1, [2; 3; 4; 1]); (1, [3; 4; 1; 2]); (−1, [4; 1; 2; 3])]
                (permute_cyclic_signed [1; 2; 3; 4]))]

    let sort_signed_not_unique =
      "not␣unique" >::
        (fun () →
          assert_raises
            (Invalid_argument
               "Combinatorics.insert_inorder_signed:␣identical␣elements")
            (fun () → sort_signed [1; 2; 3; 4; 2]))

    let sort_signed_even =
      "even" >::
        (fun () →
          assert_equal (1, [1; 2; 3; 4; 5; 6])
            (sort_signed [1; 2; 4; 3; 6; 5]))

    let sort_signed_odd =
      "odd" >::
        (fun () →
          assert_equal (−1, [1; 2; 3; 4; 5; 6])
            (sort_signed [2; 3; 1; 5; 4; 6]))

    let sort_signed_all =
      "all" >::
      (fun () →
        let l = ThoList.range 1 8 in
        assert_bool "all␣signed␣permutations"
          (List.for_all
             (fun (eps, p) →
               let eps', p' = sort_signed p in
               eps' = eps ∧ p' = l)
             (permute_signed l)))

    let sign_sign2 =
      "sign/sign2" >::
      (fun () →
         let l = ThoList.range 1 8 in
            assert_bool "all␣permutations"
            (List.for_all
               (fun p → sign p = sign2 p)
               (permute l)))

   let suite_sort_signed =
      "sort_signed" >:::
         [sort_signed_not_unique;
          sort_signed_even;
          sort_signed_odd;
          sort_signed_all;
          sign_sign2]

   let suite =
      "Combinatorics" >:::
         [suite_permute;
          suite_sort_signed]

  end
```

Q.3 *Interface of Permutation*

```
module type T =
  sig

   type t
```

The argument list $[p_1; \ldots; p_n]$ must contain every integer from 0 to $n-1$ exactly once.

```
   val of_list : int list → t
   val of_array : int array → t
```

list (*of_lists* l l') l = l'

```
   val of_lists : α list → α list → t

   val inverse : t → t
   val compose : t → t → t
```

compose_inv p q = *compose* p (*inverse* q), but more efficient.

```
   val compose_inv : t → t → t
```

If p is *of_list* $[p_1; \ldots; p_n]$, then *list* p $[a_1; \ldots; a_n]$ reorders the list $[a_1; \ldots; a_n]$ in the sequence given by $[p_1; \ldots; p_n]$. Thus the $[p_1; \ldots; p_n]$ are *not* used as a map of the indices reshuffling an array. Instead they denote the new positions of the elements of $[a_1; \ldots; a_n]$. However *list* (*inverse* p) $[a_1; \ldots; a_n]$ is $[a_{p_1}; \ldots; a_{p_n}]$, by duality.

```
   val list : t → α list → α list
   val array : t → α array → α array

   val all : int → t list
   val even : int → t list
   val odd : int → t list
   val cyclic : int → t list
   val signed : int → (int × t) list
```

Assuming fewer than 10 elements!

```
   val to_string : t → string

  end

module Using_Lists : T
module Using_Arrays : T

module Default : T

module Test : functor (P : T) →
  sig val suite : OUnit.test val time : unit → unit end
```

Q.4 Implementation of Permutation

```
module type T =
  sig
    type t
    val of_list : int list → t
    val of_array : int array → t
    val of_lists : α list → α list → t
    val inverse : t → t
    val compose : t → t → t
    val compose_inv : t → t → t
    val list : t → α list → α list
    val array : t → α array → α array
    val all : int → t list
    val even : int → t list
    val odd : int → t list
    val cyclic : int → t list
    val signed : int → (int × t) list
    val to_string : t → string
  end
```

```
let same_elements l1 l2 =
  List.sort compare l1 = List.sort compare l2
```

```
module PM = Pmap.Tree
```

```
let offset_map l =
  let _, offsets =
    List.fold_left
      (fun (i, map) a → (succ i, PM.add compare a i map))
      (0, PM.empty) l in
  offsets
```

TODO: this algorithm fails if the lists contain duplicate elements.

```
let of_lists_list l l′ =
  if same_elements l l′ then
    let offsets′ = offset_map l′ in
    let _, p_rev =
      List.fold_left
        (fun (i, acc) a → (succ i, PM.find compare a offsets′ :: acc))
        (0, []) l in
    List.rev p_rev
  else
    invalid_arg "Permutation.of_lists:␣incompatible␣lists"
```

```
module Using_Lists : T =
  struct

    type t = int list

    let of_list p =
      if List.sort compare p ≠ (ThoList.range 0 (List.length p − 1)) then
        invalid_arg "Permutation.of_list"
      else
        p

    let of_array p =
      try
        of_list (Array.to_list p)
      with
      | Invalid_argument s →
         if s = "Permutation.of_list" then
           invalid_arg "Permutation.of_array"
         else
```

```
          failwith ("Permutation.of_array: unexpected Invalid_argument(" ^
                    s ^ ")")

    let of_lists = of_lists_list

    let inverse p = snd (ThoList.ariadne_sort p)

    let list p l =
      List.map snd
        (List.sort (fun (i, _) (j, _) → compare i j)
          (try
            List.rev_map2 (fun i x → (i, x)) p l
           with
           | Invalid_argument s →
              if s = "List.rev_map2" then
                invalid_arg "Permutation.list: length mismatch"
              else
                failwith ("Permutation.list: unexpected Invalid_argument(" ^
                          s ^ ")")))

    let array p a =
      try
        Array.of_list (list p (Array.to_list a))
      with
      | Invalid_argument s →
         if s = "Permutation.list: length mismatch" then
           invalid_arg "Permutation.array: length mismatch"
         else
           failwith ("Permutation.array: unexpected Invalid_argument(" ^ s ^ ")")

    let compose_inv p q =
      list q p
```

Probably not optimal (or really inefficient), but correct by associativity.

```
    let compose p q =
      list (inverse q) p

    let all n =
      List.map of_list (Combinatorics.permute (ThoList.range 0 (pred n)))

    let even n =
      List.map of_list (Combinatorics.permute_even (ThoList.range 0 (pred n)))

    let odd n =
      List.map of_list (Combinatorics.permute_odd (ThoList.range 0 (pred n)))

    let cyclic n =
      List.map of_list (Combinatorics.permute_cyclic (ThoList.range 0 (pred n)))

    let signed n =
      List.map
        (fun (eps, l) → (eps, of_list l))
        (Combinatorics.permute_signed (ThoList.range 0 (pred n)))

    let to_string p =
      String.concat "" (List.map string_of_int p)

  end

module Using_Arrays : T =
  struct

    type t = int array

    let of_list p =
      if List.sort compare p ≠ (ThoList.range 0 (List.length p − 1)) then
        invalid_arg "Permutation.of_list"
      else
        Array.of_list p
```

```
let of_array p =
  try
    of_list (Array.to_list p)
  with
  | Invalid_argument s →
     if s = "Permutation.of_list" then
       invalid_arg "Permutation.of_array"
     else
       failwith ("Permutation.of_array:␣unexpected␣Invalid_argument(" ^
                  s ^ ")")

let of_lists l l′ =
  Array.of_list (of_lists_list l l′)

let inverse p =
  let len_p = Array.length p in
  let p′ = Array.make len_p p.(0) in
  for i = 0 to pred len_p do
    p′.(p.(i)) ← i
  done;
  p′

let array p a =
  let len_a = Array.length a
  and len_p = Array.length p in
  if len_a ≠ len_p then
    invalid_arg "Permutation.array:␣length␣mismatch";
  let a′ = Array.make len_a a.(0) in
  for i = 0 to pred len_a do
    a′.(p.(i)) ← a.(i)
  done;
  a′

let list p l =
  try
    Array.to_list (array p (Array.of_list l))
  with
  | Invalid_argument s →
     if s = "Permutation.array:␣length␣mismatch" then
       invalid_arg "Permutation.list:␣length␣mismatch"
     else
       failwith ("Permutation.list:␣unexpected␣Invalid_argument(" ^ s ^ ")")

let compose_inv p q =
  array q p

let compose p q =
  array (inverse q) p

let all n =
  List.map of_list (Combinatorics.permute (ThoList.range 0 (pred n)))

let even n =
  List.map of_list (Combinatorics.permute_even (ThoList.range 0 (pred n)))

let odd n =
  List.map of_list (Combinatorics.permute_odd (ThoList.range 0 (pred n)))

let cyclic n =
  List.map of_list (Combinatorics.permute_cyclic (ThoList.range 0 (pred n)))

let signed n =
  List.map
    (fun (eps, l) → (eps, of_list l))
    (Combinatorics.permute_signed (ThoList.range 0 (pred n)))

let to_string p =
      String.concat "" (List.map string_of_int (Array.to_list p))
  end

module Default = Using_Arrays

let shuffle l =
  let a = Array.of_list l in
  ThoArray.shuffle a;
  Array.to_list a

let time f x =
  let start = Sys.time () in
  let f_x = f x in
  let stop = Sys.time () in
  (f_x, stop -. start)

let print_time msg f x =
  let f_x, seconds = time f x in
  Printf.printf "%s took %10.2f ms\n" msg (seconds *. 1000.);
  f_x

let random_int_list imax n =
  let imax_plus = succ imax in
  Array.to_list (Array.init n (fun _ → Random.int imax_plus))

module Test (P : T) : sig val suite : OUnit.test val time : unit → unit end =
  struct

    open OUnit
    open P

    let of_list_overlap =
      "overlap" >::
        (fun () →
          assert_raises (Invalid_argument "Permutation.of_list")
            (fun () →
              of_list [0; 1; 2; 2]))

    let of_list_gap =
      "gap" >::
        (fun () →
          assert_raises (Invalid_argument "Permutation.of_list")
            (fun () →
              of_list [0; 1; 2; 4; 5]))

    let of_list_ok =
      "ok" >::
        (fun () →
          let l = ThoList.range 0 10 in
          assert_equal (of_list l) (of_list l))

    let suite_of_list =
      "of_list" >:::
        [of_list_overlap;
         of_list_gap;
         of_list_ok]

    let suite_of_lists =
      "of_lists" >:::
        [ "ok" >::
            (fun () →
              for i = 1 to 10 do
                let l = random_int_list 1000000 100 in
                let l' = shuffle l in
                assert_equal
                  ~printer : (ThoList.to_string string_of_int)
                  l' (list (of_lists l l') l)
              done) ]

    let apply_invalid_lengths =
      "invalid/lengths" >::
        (fun () →
          assert_raises
            (Invalid_argument "Permutation.list:␣length␣mismatch")
            (fun () →
              list (of_list [0; 1; 2; 3; 4]) [0; 1; 2; 3]))

    let apply_ok =
      "ok" >::
        (fun () →
          assert_equal [2; 0; 1; 3; 5; 4]
            (list (of_list [1; 2; 0; 3; 5; 4]) [0; 1; 2; 3; 4; 5]))

    let suite_apply =
      "apply" >:::
        [apply_invalid_lengths;
         apply_ok]

    let inverse_ok =
      "ok" >::
        (fun () →
          let l = shuffle (ThoList.range 0 1000) in
          let p = of_list (shuffle l) in
          assert_equal l (list (inverse p) (list p l)))

    let suite_inverse =
      "inverse" >:::
        [inverse_ok]

    let compose_ok =
      "ok" >::
        (fun () →
          let id = ThoList.range 0 1000 in
          let p = of_list (shuffle id)
          and q = of_list (shuffle id)
          and l = id in
          assert_equal (list p (list q l)) (list (compose p q) l))

    let compose_inverse_ok =
      "inverse/ok" >::
        (fun () →
          let id = ThoList.range 0 1000 in
          let p = of_list (shuffle id)
          and q = of_list (shuffle id) in
          assert_equal
            (compose (inverse p) (inverse q))
            (inverse (compose q p)))

    let suite_compose =
      "compose" >:::
        [compose_ok;
         compose_inverse_ok]

    let suite =
      "Permutations" >:::
        [suite_of_list;
         suite_of_lists;
         suite_apply;
         suite_inverse;
         suite_compose]

    let repeat repetitions size =
      let id = ThoList.range 0 size in
```

```
      let p = of_list (shuffle id)
      and l = shuffle (List.map string_of_int id) in
      print_time (Printf.sprintf "reps=%d,␣len=%d" repetitions size)
        (fun () →
          for i = 1 to repetitions do
            ignore (P.list p l)
          done)
        ()

    let time () =
      repeat 100000 10;
      repeat 10000 100;
      repeat 1000 1000;
      repeat 100 10000;
      repeat 10 100000;
      ()

  end
```

—R— Partitions

R.1 Interface of Partition

pairs n n1 n2 returns all (unordered) pairs of integers with the sum n in the range from *n1* to *n2*.

```
val pairs : int → int → int → (int × int) list
val triples : int → int → int → (int × int × int) list
```

tuples d n n_min n_max returns all $[n_1; n_2; \ldots; n_d]$ with $n_{\min} \le n_1 \le n_2 \le \ldots \le n_d \le n_{\max}$ and

$$\sum_{i=1}^{d} n_i = n \tag{R.1}$$

```
val tuples : int → int → int → int → int list list
```

R.2 Implementation of Partition

All unordered pairs of integers with the same sum n in a given range $\{n_1, \ldots, n_2\}$:

$$\mathit{pairs} : (n, n_1, n_2) \to \big\{(i,j) \,|\, i + j = n \land n_1 \le i \le j \le n_2\big\} \tag{R.2}$$

```
let rec pairs' acc n1 n2 =
  if n1 > n2 then
    List.rev acc
  else
    pairs' ((n1, n2) :: acc) (succ n1) (pred n2)

let pairs sum min_n1 max_n2 =
  let n1 = max min_n1 (sum − max_n2) in
  let n2 = sum − n1 in
  if n2 ≤ max_n2 then
    pairs' [] n1 n2
  else
    []

let rec tuples d sum n_min n_max =
  if d ≤ 0 then
    invalid_arg "tuples"
  else if d > 1 then
    tuples' d sum n_min n_max n_min
  else if sum ≥ n_min ∧ sum ≤ n_max then
    [[sum]]
  else
    []

and tuples' d sum n_min n_max n =
  if n > n_max then
    []
  else
    List.fold_right (fun l ll → (n :: l) :: ll)
    (tuples (pred d) (sum − n) (max n_min n) n_max)
    (tuples' d sum n_min n_max (succ n))
```

When I find a little spare time, I can provide a dedicated implementation, but we *know* that *Impossible* is *never* raised and the present approach is just as good (except for a possible tiny inefficiency).

```
exception Impossible of string
let impossible name = raise (Impossible name)

let triples sum n_min n_max =
  List.map (function [n1; n2; n3] → (n1, n2, n3) | _ → impossible "triples")
    (tuples 3 sum n_min n_max)
```

—S— Young Diagrams and Tableaux

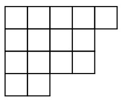
S.1 Interface of Young

Caveat: the following are not optimized for large Young diagrams and tableaux. They are straightforward implementations of the definitions, since we are unlikely to meet large diagrams.

To make matters worse, native integer arithmetic will overflow already for diagrams with more than 20 cells. Since the *Num* library has been removed from the O'Caml distribution with version 4.06, we can not use it as a shortcut. Requiring Whizard/O'Mega users to install *Num* or its successor *Zarith* is probably not worth the effort.

S.1.1 Young Diagrams

Young diagrams can be represented by a non-increasing list of positive integers, corresponding to the number of boxes in each row:



$$\Longleftrightarrow [5;4;4;2] \tag{S.1}$$

type *diagram* = *int list*

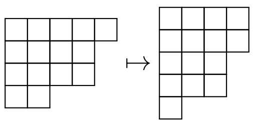
Check that the diagram is valid, i. e. the number of boxes is non-increasing from top to bottom.

val *valid_diagram* : *diagram* → *bool*

Count the number of cells.

val *num_cells_diagram* : *diagram* → *int*

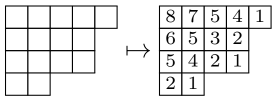
Conjugate a diagram:



(S.2)

val *conjugate_diagram* : *diagram* → *diagram*

The product of all the "hook lengths" in the diagram, e. g.



$$\mapsto 8\cdot 7\cdot 6\cdot 5^3\cdot 4^2\cdot 3\cdot 2^3 = 16128000 \tag{S.3}$$

where the intermediate step is only for illustration and does not represent a Young tableau!

val *hook_lengths_product* : *diagram* → *int*

Number of standard tableaux corresponding to the diagram. Also, the dimension of the representation of S_n described by this diagram

$$d = \frac{n!}{\prod_{i=1}^n h_i} \tag{S.4}$$

with n the number of cells and h_i the hook length of the ith cell.

val *num_standard_tableaux* : *diagram* → *int*

Normalization of the projector on the representation of GL(N) described by the diagram

$$\alpha = \frac{\prod_R |R|! \prod_C |C|!}{\prod_{i=1}^n h_i} \tag{S.5}$$

with $|R|$ and $|C|$ the lengths of the row R and column C, respectively. Returned as a pair of numerator and denominator, because it is not guaranteed to be integer.

```
val normalization : diagram → int × int
```

S.1.2 Young Tableaux

There is an obvious representation as a list of lists:

$$\begin{array}{|c|c|c|}\hline 0&2&3\\\hline 1&4\\\cline{1-2}\end{array} \Longleftrightarrow [[0;2;3];[1;4]] \tag{S.6}$$

```
type α tableau = α list list
```

Ignoring the contents of the cells of a Young tableau produces a unique corresponding Young diagram.

$$\begin{array}{|c|c|c|}\hline 0&2&3\\\hline 1&4\\\cline{1-2}\end{array} \mapsto \begin{array}{|c|c|c|}\hline \ &\ &\ \\\hline \ &\ \\\cline{1-2}\end{array} \tag{S.7}$$

```
val diagram_of_tableau : α tableau → diagram
```

The number of columns must be non-increasing. Obviously, *valid_tableau* is the composition of *diagram_of_tableau* and *valid_diagram*.

```
val valid_tableau : α tableau → bool
```

A tableau is called *semistandard*, iff the entries don't decrease along rows and strictly increase along columns. Therefore, the conjugate of a semistandard tableau is *not* necessarily semistandard.

```
val semistandard_tableau : α tableau → bool
```

A tableau is called *standard*, iff it is semistandard and the entries are an uninterrupted sequence of natural numbers. If the optional *offset* is specified, it must match the smallest of these numbers. Some authors expect *offset* = 1, but we want to be able to start from 0 as well. The conjugate of a standard tableau is again a standard tableau.

```
val standard_tableau : ?offset :int → int tableau → bool
```

The contents of the cells and their number.

```
val cells_tableau : α tableau → α list
val num_cells_tableau : α tableau → int
```

Conjugate a Young tableau

$$\begin{array}{|c|c|c|}\hline 0&2&3\\\hline 1&4\\\cline{1-2}\end{array} \mapsto \begin{array}{|c|c|}\hline 0&1\\\hline 2&4\\\hline 3\\\cline{1-1}\end{array} \tag{S.8}$$

```
val conjugate_tableau : α tableau → α tableau
```

Transform the contents cell-by-cell.

```
val map : (α → β) → α tableau → β tableau
```

Debugging and diagnostics.

```
val tableau_to_string : (α → string) → α tableau → string
```

Toplevel

```
val pp : Format.formatter → int tableau → unit
```

S.1.3 Unit Tests

```
module type Test =
  sig
    val suite : OUnit.test
    val suite_long : OUnit.test
  end

module Test : Test
```

S.2 Implementation of Young

```
type diagram = int list
type α tableau = α list list
```

Not exposed. Just for documentation.

```
type α table = α option array array
```

The following three are candidates for *ThoList*.

```
let rec sum = function
  | [] → 0
  | n :: rest → n + sum rest
```

```
let rec product = function
  | [] → 1
  | n :: rest → n × product rest
```

Test a predicate for each pair of consecutive elements of a list. Trivially true for empty and one-element lists.

```
let rec for_all_pairs predicate = function
  | [] | [_] → true
  | a1 :: (a2 :: _ as a_list) →
      if ¬ (predicate a1 a2) then
        false
      else
        for_all_pairs predicate a_list
```

```
let decreasing l = for_all_pairs (fun a1 a2 → compare a1 a2 > 0) l
let increasing l = for_all_pairs (fun a1 a2 → compare a1 a2 < 0) l
let non_increasing l = for_all_pairs (fun a1 a2 → compare a1 a2 ≥ 0) l
let non_decreasing l = for_all_pairs (fun a1 a2 → compare a1 a2 ≤ 0) l
```

```
let non_increasing_never_zero l =
  for_all_pairs (fun a1 a2 → a2 > 0 ∧ compare a1 a2 ≥ 0) l
```

```
let valid_diagram = non_increasing_never_zero
```

```
let diagram_rows d =
  List.length d
```

```
let diagram_columns = function
  | [] → 0
  | nc :: _ → nc
```

```
let take_column d =
  let rec take_column' len acc = function
    | [] → (len, List.rev acc)
    | cols :: rest →
        if cols ≤ 1 then
          take_column' (succ len) acc rest
        else
          take_column' (succ len) (pred cols :: acc) rest in
  take_column' 0 [] d
```

```
let conjugate_diagram_new d =
  let rec conjugate_diagram' rows =
    match take_column rows with
    | n, [] → [n]
    | n, rest → n :: conjugate_diagram' rest in
  conjugate_diagram' d
```

```
let tableau_rows t =
  List.length t
```

```
let tableau_columns = function
  | [] → 0
  | row :: _ → List.length row
```

```
let num_cells_diagram d =
  sum d

let cells_tableau t =
  List.flatten t

let num_cells_tableau t =
  List.fold_left (fun acc row → acc + List.length row) 0 t

let diagram_of_tableau t =
  List.map List.length t

let tableau_of_diagram cell d =
  List.map (ThoList.clone cell) d
```

Note that the first index counts the rows and the second the columns!

```
let array_of_tableau t =
  let nr = tableau_rows t
  and nc = tableau_columns t in
  let a = Array.make_matrix nr nc None in
  List.iteri
    (fun ir → List.iteri (fun ic cell → a.(ir).(ic) ← Some cell))
    t;
  a

let transpose_array a =
  let nr = Array.length a in
  if nr ≤ 0 then
    invalid_arg "Young.transpose_array"
  else
    let nc = Array.length a.(0) in
    let a′ = Array.make_matrix nc nr None in
    for ic = 0 to pred nc do
      for ir = 0 to pred nr do
        a′.(ic).(ir) ← a.(ir).(ic)
      done
    done;
    a′

let list_of_array_row a =
  let n = Array.length a in
  let rec list_of_array_row′ ic =
    if ic ≥ n then
      []
    else
      match a.(ic) with
      | None → []
      | Some cell → cell :: list_of_array_row′ (succ ic) in
  list_of_array_row′ 0

let tableau_of_array a =
  Array.fold_right (fun row acc → list_of_array_row row :: acc) a []

let conjugate_tableau t =
  array_of_tableau t |> transpose_array |> tableau_of_array

let conjugate_diagram d =
  tableau_of_diagram () d |> conjugate_tableau |> diagram_of_tableau

let valid_tableau t =
  valid_diagram (diagram_of_tableau t)

let semistandard_tableau t =
  let rows = t
  and columns = conjugate_tableau t in
  valid_tableau t
  ∧ List.for_all non_decreasing rows
  ∧ List.for_all increasing columns

let standard_tableau ?offset t =
  match List.sort compare (cells_tableau t) with
  | [] → true
  | cell :: _ as cell_list →
      (match offset with None → true | Some o → cell = o)
      ∧ for_all_pairs (fun c1 c2 → c2 = c1 + 1) cell_list
      ∧ semistandard_tableau t

let map f t =
  List.map (List.map f) t

let tableau_to_string to_string t =
  ThoList.to_string (ThoList.to_string to_string) t

let pp fmt y =
  Format.fprintf fmt "%s" (tableau_to_string string_of_int y)

let hook_lengths_table d =
  let nr = diagram_rows d
  and nc = diagram_columns d in
  if min nr nc ≤ 0 then
    invalid_arg "Young.hook_lengths_table"
  else
    let a = array_of_tableau (tableau_of_diagram 0 d) in
    let cols = Array.of_list d
    and rows = transpose_array a |> tableau_of_array
                   |> diagram_of_tableau |> Array.of_list in
    for ir = 0 to pred nr do
      for ic = 0 to pred cols.(ir) do
        a.(ir).(ic) ← Some (rows.(ic) − ir + cols.(ir) − ic − 1)
      done
    done;
    a
```

The following products and factorials can easily overflow, even if the final ratio is a smallish number. We can avoid this by representing them as lists of factors (or maps from factors to powers). The ratio can be computed by first cancelling all common factors and multiplying the remaining factors at the very end.

```
let hook_lengths_product d =
  let nr = diagram_rows d
  and nc = diagram_columns d in
  if min nr nc ≤ 0 then
    0
  else
    let cols = Array.of_list d
    and rows = Array.of_list (conjugate_diagram d) in
    let n = ref 1 in
    for ir = 0 to pred nr do
      for ic = 0 to pred cols.(ir) do
        n := !n × (rows.(ic) − ir + cols.(ir) − ic − 1)
      done
    done;
    !n

let num_standard_tableaux d =
  let num = Combinatorics.factorial (num_cells_diagram d)
  and den = hook_lengths_product d in
  if num mod den ≠ 0 then
    failwith "Young.num_standard_tableaux"
  else
    num / den
```

Note that *hook_lengths_product* calls *conjugate_diagram* and this calls it again. This is wasteful, but probably no big deal for our applications.

```
let normalization d =
  let num =
    product (List.map Combinatorics.factorial (d @ conjugate_diagram d))
  and den = hook_lengths_product d in
  (num, den)

module type Test =
  sig
    val suite : OUnit.test
    val suite_long : OUnit.test
  end

module Test =
  struct
    open OUnit

    let random_int ratio =
      truncate (Random.float ratio +. 0.5)

    let random_diagram ?(ratio = 1.0) rows =
      let rec random_diagram' acc row cols =
        if row ≥ rows then
          acc
        else
          let cols' = cols + random_int ratio in
          random_diagram' (cols' :: acc) (succ row) cols' in
      random_diagram' [] 0 (1 + random_int ratio)

    let suite_hook_lengths_product =
      "hook_lengths_product" >:::

        [ "[4;3;2]" >::
            (fun () → assert_equal 2160 (hook_lengths_product [4; 3; 2])) ]

    let suite_num_standard_tableaux =
      "num_standard_tableaux" >:::

        [ "[4;3;2]" >::
            (fun () → assert_equal 168 (num_standard_tableaux [4; 3; 2])) ]

    let suite_normalization =
      "normalization" >:::

        [ "[2;1]" >::
            (fun () → assert_equal (4, 3) (normalization [2; 1])) ]

    let suite =
      "Young" >:::
        [suite_hook_lengths_product;
         suite_num_standard_tableaux;
         suite_normalization]

    let suite_long =
      "Young␣long" >:::
        []

  end
```

—T—
Trees

From [10]: Trees with one root admit a straightforward recursive definition

$$T(N,L) = L \cup N \times T(N,L) \times T(N,L) \tag{T.1}$$

that is very well adapted to mathematical reasoning. Such recursive definitions are useful because they allow us to prove properties of elements by induction

$$\forall l \in L : p(l) \wedge (\forall n \in N : \forall t_1, t_2 \in T(N,L) : p(t_1) \wedge p(t_2) \Rightarrow p(n \times t_1 \times t_2)) \\ \Longrightarrow \forall t \in T(N,L) : p(t) \tag{T.2}$$

i. e. establishing a property for all leaves and showing that a node automatically satisfies the property if it is true for all children proves the property for *all* trees. This induction is of course modelled after standard mathematical induction

$$p(1) \wedge (\forall n \in \mathbf{N} : p(n) \Rightarrow p(n+1)) \Longrightarrow \forall n \in \mathbf{N} : p(n) \tag{T.3}$$

The recursive definition (T.1) is mirrored by the two tree construction functions[1]

$$\mathit{leaf} : \nu \times \lambda \to (\nu,\lambda)T \tag{T.4a}$$
$$\mathit{node} : \nu \times (\nu,\lambda)T \times (\nu,\lambda)T \to (\nu,\lambda)T \tag{T.4b}$$

Renaming leaves and nodes leaves the structure of the tree invariant. Therefore, morphisms $L \to L'$ and $N \to N'$ of the sets of leaves and nodes induce natural homomorphisms $T(N,L) \to T(N',L')$ of trees

$$\mathit{map} : \ (\nu \to \nu') \times (\lambda \to \lambda') \times (\nu,\lambda)T \to (\nu',\lambda')T \tag{T.5}$$

The homomorphisms constructed by *map* are trivial, but ubiquitous. More interesting are the morphisms

$$\begin{aligned} \mathit{fold} : (\nu \times \lambda \to \alpha) \times (\nu \times \alpha \times \alpha \to \alpha) \times (\nu,\lambda)T \to \alpha \\ (f_1, f_2, l \in L) \mapsto f_1(l) \\ (f_1, f_2, (n, t_1, t_2)) \mapsto f_2(n, \mathit{fold}(f_1,f_2,t_1), \mathit{fold}(f_1,f_2,t_2)) \end{aligned} \tag{T.6}$$

and

$$\begin{aligned} \mathit{fan} : (\nu \times \lambda \to \{\alpha\}) \times (\nu \times \alpha \times \alpha \to \{\alpha\}) \times (\nu,\lambda)T \to \{\alpha\} \\ (f_1, f_2, l \in L) \mapsto f_1(l) \\ (f_1, f_2, (n, t_1, t_2)) \mapsto f_2(n, \mathit{fold}(f_1,f_2,t_1) \otimes \mathit{fold}(f_1,f_2,t_2)) \end{aligned} \tag{T.7}$$

where the tensor product notation means that f_2 is applied to all combinations of list members in the argument:

$$\phi(\{x\} \otimes \{y\}) = \{\phi(x,y) | x \in \{x\} \wedge y \in \{y\}\} \tag{T.8}$$

But note that due to the recursive nature of trees, *fan* is *not* a morphism from $T(N,L)$ to $T(N \otimes N, L)$.

If we identify singleton sets with their members, *fold* could be viewed as a special case of *fan*, but that is probably more confusing than helpful. Also, using the special case $\alpha = (\nu',\lambda')T$, the homomorphism *map* can be expressed in terms of *fold* and the constructors

$$\begin{aligned} \mathit{map} : (\nu \to \nu') \times (\lambda \to \lambda') \times (\nu,\lambda)T \to (\nu',\lambda')T \\ (f,g,t) \mapsto \mathit{fold}(\mathit{leaf} \circ (f \times g), \mathit{node} \circ (f \times \mathit{id} \times \mathit{id}), t) \end{aligned} \tag{T.9}$$

[1] To make the introduction more accessible to non-experts, I avoid the 'curried' notation for functions with multiple arguments and use tuples instead. The actual implementation takes advantage of curried functions, however. Experts can read $\alpha \to \beta \to \gamma$ for $\alpha \times \beta \to \gamma$.

fold is much more versatile than *map*, because it can be used with constructors for other tree representations to translate among different representations. The target type can also be a mathematical expression. This is used extensively below for evaluating Feynman diagrams.

Using *fan* with $\alpha = (\nu', \lambda')T$ can be used to construct a multitude of homomorphic trees. In fact, below it will be used extensively to construct all Feynman diagrams $\{(\nu, \{p_1, \ldots, p_n\})T\}$ of a given topology $t \in (\emptyset, \{1, \ldots, n\})T$.

The physicist in me guesses that there is another morphism of trees that is related to *fan* like a Lie-algebra is related to the it's Lie-group. I have not been able to pin it down, but I guess that it is a generalization of *grow* below.

T.1 Interface of Tree

This module provides utilities for generic decorated trees, such as FeynMF output.

T.1.1 Abstract Data Type

```
type (ν, λ) t
```

leaf n l returns a tree consisting of a single leaf node of type *n* with a label *l*.

```
val leaf : ν → λ → (ν, λ) t
```

cons n ch returns a tree node.

```
val cons : ν → (ν, λ) t list → (ν, λ) t
```

Note that *cons node* [] constructs a terminal node, but *not* a leaf, since the latter *must* have a label!

This approach was probably tailored to Feynman diagrams, where we have external propagators as nodes with additional labels (cf. the function *to_feynmf* on page 767 below). I'm not so sure anymore that this was a good choice.

node t returns the top node of the tree *t*.

```
val node : (ν, λ) t → ν
```

leafs t returns a list of all leaf labels *in order*.

```
val leafs : (ν, λ) t → λ list
```

nodes t returns a list of all nodes that are not leafs in post-order. This guarantees that the root node can be stripped from the result by *List.tl*.

```
val nodes : (ν, λ) t → ν list
```

fuse conjg root contains_root trees joins the *trees*, using the leaf *root* in one of the trees as root of the new tree. *contains_root* guides the search for the subtree containing *root* as a leaf. fun *t* → *List.mem root* (*leafs t*) is acceptable, but more efficient solutions could be available in special circumstances.

```
val fuse : (ν → ν) → λ → ((ν, λ) t → bool) → (ν, λ) t list → (ν, λ) t
```

sort lesseq t return a sorted copy of the tree *t*: node labels are ignored and nodes are according to the supremum of the leaf labels in the corresponding subtree.

```
val sort : (λ → λ → bool) → (ν, λ) t → (ν, λ) t
val canonicalize : (ν, λ) t → (ν, λ) t
```

T.1.2 Homomorphisms

```
val map : ('n1 → 'n2) → ('l1 → 'l2) → ('n1, 'l1) t → ('n2, 'l2) t
val fold : (ν → λ → α) → (ν → α list → α) → (ν, λ) t → α
val fan : (ν → λ → α list) → (ν → α list → α list) →
  (ν, λ) t → α list
```

T.1.3 Output

```
val to_string : (string, string) t → string
```

Feynmf

style : (*string* × *string*) *option* should be replaced by *style* : *string option*; *tex_label* : *string option*

```
type feynmf =
    { style : (string × string) option;
      rev : bool;
      label : string option;
      tension : float option }
val vanilla : feynmf
val sty : (string × string) × bool × string → feynmf
```

to_feynmf file to_string incoming t write the trees in the list *t* to the file named *file*. The leaves *incoming* are used as incoming particles and *to_string* is use to convert leaf labels to LaTeX-strings.

```
type λ feynmf_set =
  { header : string;
    incoming : λ list;
    diagrams : (feynmf, λ) t list }
```

```
type (λ, μ) feynmf_sets =
  { outer : λ feynmf_set;
    inner : μ feynmf_set list }
```

```
val feynmf_sets_plain : bool → int → string →
  (λ → string) → (λ → string) →
  (μ → string) → (μ → string) → (λ, μ) feynmf_sets list → unit
```

```
val feynmf_sets_wrapped : bool → string →
  (λ → string) → (λ → string) →
  (μ → string) → (μ → string) → (λ, μ) feynmf_sets list → unit
```

```
val feynmf_sets_wrapped_to_channel : bool → out_channel →
  (λ → string) → (λ → string) →
  (μ → string) → (μ → string) → (λ, μ) feynmf_sets list → unit
```

If the diagrams at all levels are of the same type, we can recurse to arbitrary depth.

```
type λ feynmf_levels =
  { this : λ feynmf_set;
    lower : λ feynmf_levels list }
```

to_feynmf_levels_plain sections level file wf_to_TeX p_to_TeX levels ...

```
val feynmf_levels_plain : bool → int → string →
  (λ → string) → (λ → string) → λ feynmf_levels list → unit
```

to_feynmf_levels_wrapped file wf_to_TeX p_to_TeX levels ...

```
val feynmf_levels_wrapped : string →
  (λ → string) → (λ → string) → λ feynmf_levels list → unit
```

Least Squares Layout

A general graph with edges of type ε, internal nodes of type ν, and external nodes of type *'ext*.

```
type (ε, ν, 'ext) graph
val graph_of_tree : (ν → ν → ε) → (ν → ν) →
  ν → (ν, ν) t → (ε, ν, ν) graph
```

A general graph with the layout of the external nodes fixed.

```
type (ε, ν, 'ext) ext_layout
```

```
val left_to_right : int → (ε, ν, 'ext) graph → (ε, ν, 'ext) ext_layout
```

A general graph with the layout of all nodes fixed.

```
type (ε, ν, 'ext) layout
val layout : (ε, ν, 'ext) ext_layout → (ε, ν, 'ext) layout
```

```
val dump : (ε, ν, 'ext) layout → unit
val iter_edges : (ε → float × float → float × float → unit) →
  (ε, ν, 'ext) layout → unit
val iter_internal : (float × float → unit) →
  (ε, ν, 'ext) layout → unit
val iter_incoming : ('ext × float × float → unit) →
  (ε, ν, 'ext) layout → unit
val iter_outgoing : ('ext × float × float → unit) →
  (ε, ν, 'ext) layout → unit
```

T.2 Implementation of Tree

T.2.1 Abstract Data Type

```
type (ν, λ) t =
  | Leaf of ν × λ
  | Node of ν × (ν, λ) t list
```

```
let leaf n l = Leaf (n, l)
```

```
let cons n children = Node (n, children)
```

Presenting the leafs *in order* comes naturally, but will be useful below.

```
let rec leafs = function
  | Leaf (_, l) → [l]
  | Node (_, ch) → ThoList.flatmap leafs ch
```

```
let node = function
  | Leaf (n, _) → n
  | Node (n, _) → n
```

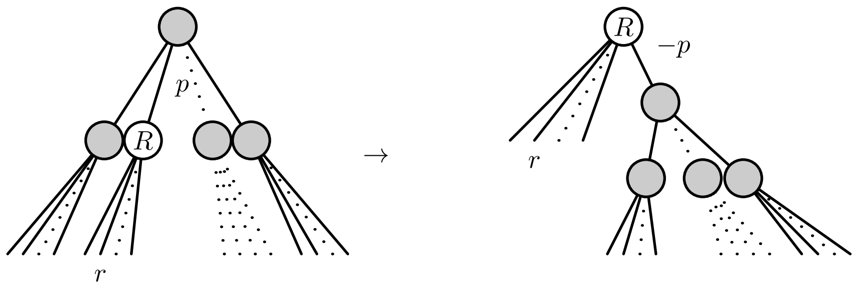
This guarantees that the root node can be stripped from the result by *List.tl*.

```
let rec nodes = function
  | Leaf _ → []
  | Node (n, ch) → n :: ThoList.flatmap nodes ch
```

first_match p list returns $(x, list')$, where x is the first element of *list* for which $p\ x$ = true and *list'* is *list* sans x.

```
let first_match p list =
  let rec first_match' no_match = function
    | [] → invalid_arg "Tree.fuse:␣prospective␣root␣not␣found"
    | t :: rest when p t → (t, List.rev_append no_match rest)
    | t :: rest → first_match' (t :: no_match) rest in
  first_match' [] list
```

One recursion step in *fuse'* rotates the topmost tree node, moving the prospective root up:



(T.10)

```
let fuse conjg root contains_root trees =
  let rec fuse' subtrees =
    match first_match contains_root subtrees with
```

If the prospective root is contained in a leaf, we have either found the root—in which case we're done—or have failed catastrophically:

```
    | Leaf (n, l), children →
        if l = root then
          Node (conjg n, children)
        else
          invalid_arg "Tree.fuse:␣root␣predicate␣inconsistent"
```

Otherwise, we perform a rotation as in (T.10) and connect all nodes that do not contain the root to a new node. For efficiency, we append the new node at the end and prevent *first_match* from searching for the root in it in vain again. Since *root_children* is probably rather short, this should be a good strategy.

```
    | Node (n, root_children), other_children →
        fuse' (root_children @ [Node (conjg n, other_children)]) in
  fuse' trees
```

Sorting is also straightforward, we only have to keep track of the suprema of the subtrees:

```
type (α, β) with_supremum = { sup : α; data : β }
```

Since the lists are rather short, *List.sort* could be replaced by an optimized version, but we're not (yet) dealing with the most important speed bottleneck here:

```
let rec sort' lesseq = function
  | Leaf (_, l) as e → { sup = l; data = e }
  | Node (n, ch) →
      let ch' = List.sort
          (fun x y → compare x.sup y.sup) (List.map (sort' lesseq) ch) in
      { sup = (List.hd (List.rev ch')).sup;
        data = Node (n, List.map (fun x → x.data) ch') }
```

finally, throw away the overall supremum:

```
let sort lesseq t = (sort' lesseq t).data
```

```
let rec canonicalize = function
  | Leaf (_, _) as l → l
  | Node (n, ch) →
    Node (n, List.sort compare (List.map canonicalize ch))
```

T.2.2 Homomorphisms

Isomophisms are simple:

```
let rec map fn fl = function
  | Leaf (n, l) → Leaf (fn n, fl l)
  | Node (n, ch) → Node (fn n, List.map (map fn fl) ch)
```

homomorphisms are not more complicated:

```
let rec fold leaf node = function
  | Leaf (n, l) → leaf n l
  | Node (n, ch) → node n (List.map (fold leaf node) ch)
```

and tensor products are fun:

```
let rec fan leaf node = function
  | Leaf (n, l) → leaf n l
  | Node (n, ch) → Product.fold
        (fun ch' t → node n ch' @ t) (List.map (fan leaf node) ch) []
```

T.2.3 Output

```
let leaf_to_string n l =
  if n = "" then
    l
  else if l = "" then
    n
  else
    n ^ "(" ^ l ^ ")"

let node_to_string n ch =
  "(" ^ (if n = "" then "" else n ^ ":") ^ (String.concat "," ch) ^ ")"

let to_string t =
  fold leaf_to_string node_to_string t
```

Feynmf

Add a value that is greater than all suprema

```
type α supremum_or_infinity = Infinity | Sup of α

type (α, β) with_supremum_or_infinity =
    { sup : α supremum_or_infinity; data : β }

let with_infinity cmp x y =
  match x.sup, y.sup with
  | Infinity, _ → 1
  | _, Infinity → −1
  | Sup x′, Sup y′ → cmp x′ y′
```

Using this, we can sort the tree in another way that guarantees that a particular leaf (*i2*) is moved as far to the end as possible. We can then flip this leaf from outgoing to incoming without introducing a crossing:

```
let rec sort_2i′ lesseq i2 = function
  | Leaf (_, l) as e →
      { sup = if l = i2 then Infinity else Sup l; data = e }
  | Node (n, ch) →
      let ch′ = List.sort (with_infinity compare)
            (List.map (sort_2i′ lesseq i2) ch) in
      { sup = (List.hd (List.rev ch′)).sup;
        data = Node (n, List.map (fun x → x.data) ch′) }
```

again, throw away the overall supremum:

```
let sort_2i lesseq i2 t = (sort_2i′ lesseq i2 t).data

type feynmf =
    { style : (string × string) option;
      rev : bool;
      label : string option;
      tension : float option }

open Printf

let style prop =
  match prop.style with
  | None → ("plain","")
  | Some s → s

let species prop = fst (style prop)
let tex_lbl prop = snd (style prop)

let leaf_label tex io leaf lab = function
  | None → fprintf tex "␣␣␣␣\\fmflabel{${%s}$}{%s%s}\n" lab io leaf
  | Some s →
      fprintf tex "␣␣␣␣\\fmflabel{${%s{}^{(%s)}}$}{%s%s}\n" s lab io leaf
```

```
let leaf_label tex io leaf lab label =
  ()
```

We try to draw diagrams more symmetrically by reducing the tension on the outgoing external lines.

This is insufficient for asymmetrical cascade decays.

```
let rec leaf_node tex to_label i2 n prop leaf =
  let io, tension, rev =
    if leaf = i2 then
      ("i", "", ¬ prop.rev)
    else
      ("o", ",tension=0.5", prop.rev) in
  leaf_label tex io (to_label leaf) (tex_lbl prop) prop.label ;
  fprintf tex "␣␣␣␣\\fmfdot{v%d}\n" n;
  if rev then
    fprintf tex "␣␣␣␣\\fmf{%s%s}{%s%s,v%d}\n"
      (species prop) tension io (to_label leaf) n
  else
    fprintf tex "␣␣␣␣\\fmf{%s%s}{v%d,%s%s}\n"
      (species prop) tension n io (to_label leaf)

and int_node tex to_label i2 n n′ prop t =
  if prop.rev then
    fprintf tex
      "␣␣␣␣\\fmf{%s,label=\\begin{scriptsize}${%s}$\\end{scriptsize}}{v%d,v%d}\n"
      (species prop) (tex_lbl prop) n′ n
  else
    fprintf tex
      "␣␣␣␣\\fmf{%s,label=\\begin{scriptsize}${%s}$\\end{scriptsize}}{v%d,v%d}\n"
      (species prop) (tex_lbl prop) n n′;
  fprintf tex "␣␣␣␣\\fmfdot{v%d,v%d}\n" n n′;
  edges_feynmf′ tex to_label i2 n′ t

and leaf_or_int_node tex to_label i2 n n′ = function
  | Leaf (prop, l) → leaf_node tex to_label i2 n prop l
  | Node (prop, _) as t → int_node tex to_label i2 n n′ prop t

and edges_feynmf′ tex to_label i2 n = function
  | Leaf (prop, l) → leaf_node tex to_label i2 n prop l
  | Node (_, ch) →
      ignore (List.fold_right
                (fun t′ n′ →
                  leaf_or_int_node tex to_label i2 n n′ t′;
                  succ n′) ch (4 × n))

let edges_feynmf tex to_label i1 i2 t =
  let n = 1 in
  begin match t with
  | Leaf _ → ()
  | Node (prop, _) →
      leaf_label tex "i" "1" (tex_lbl prop) prop.label;
      if prop.rev then
        fprintf tex "␣␣␣␣\\fmf{%s}{v%d,i%s}\n" (species prop) n (to_label i1)
      else
        fprintf tex "␣␣␣␣\\fmf{%s}{i%s,v%d}\n" (species prop) (to_label i1) n
  end;
  fprintf tex "␣␣␣␣\\fmfdot{v%d}\n" n;
  edges_feynmf′ tex to_label i2 n t

let to_feynmf_channel tex to_TeX to_label incoming t =
  match incoming with
  | i1 :: i2 :: _ →
      let t′ = sort_2i (≤ ) i2 t in
```

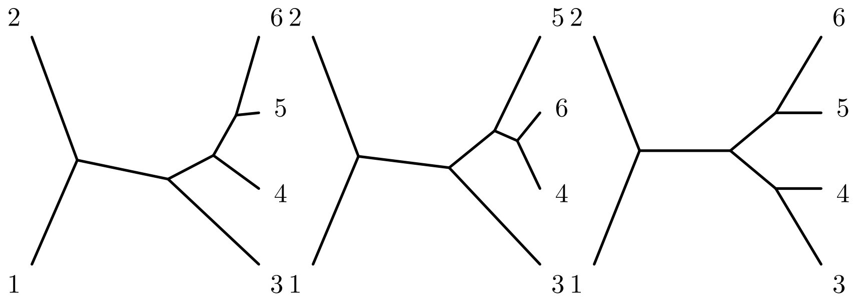


Figure T.1: Note that this is subtly different . . .

```
      let out = List.filter (fun a → i2 ≠ a) (leafs t′) in
      fprintf tex "\\fmfframe(8,7)(8,6){%%\n";
      fprintf tex "␣␣\\begin{fmfgraph*}(35,30)\n";
      fprintf tex "␣␣␣\\fmfpen{thin}\n";
      fprintf tex "␣␣␣\\fmfset{arrow_len}{2mm}\n";
      fprintf tex "␣␣␣␣\\fmfleft{i%s,i%s}\n" (to_label i1) (to_label i2);
      fprintf tex "␣␣␣␣\\fmfright{o%s}\n"
        (String.concat ",o" (List.map to_label out));
      List.iter
        (fun s →
          fprintf tex "␣␣␣␣\\fmflabel{${%s}$}{i%s}\n"
            (to_TeX s) (to_label s))
        [i1; i2];
      List.iter
        (fun s →
          fprintf tex "␣␣␣␣\\fmflabel{${%s}$}{o%s}\n"
            (to_TeX s) (to_label s))
        out;
      edges_feynmf tex to_label i1 i2 t′;
      fprintf tex "␣␣\\end{fmfgraph*}}\\hfil\\allowbreak\n"
  | _ → ()
```

```
let vanilla = { style = None; rev = false; label = None; tension = None }

let sty (s, r, l) = { vanilla with style = Some s; rev = r; label = Some l }

type λ feynmf_set =
  { header : string;
    incoming : λ list;
    diagrams : (feynmf, λ) t list }

type (λ, μ) feynmf_sets =
  { outer : λ feynmf_set;
    inner : μ feynmf_set list }

type λ feynmf_levels =
  { this : λ feynmf_set;
    lower : λ feynmf_levels list }

let latex_section = function
  | level when level < 0 → "part"
  | 0 → "chapter"
  | 1 → "section"
  | 2 → "subsection"
  | 3 → "subsubsection"
  | 4 → "paragraph"
  | _ → "subparagraph"

let rec feynmf_set tex sections level to_TeX to_label set =
  fprintf tex "%s\\%s{%s}\n"
    (if sections then "" else "%%␣")
    (latex_section level)
    set.header;
  List.iter
    (to_feynmf_channel tex to_TeX to_label set.incoming)
    set.diagrams

let feynmf_sets tex sections level
    to_TeX_outer to_label_outer to_TeX_inner to_label_inner set =
  feynmf_set tex sections level to_TeX_outer to_label_outer set.outer;
  List.iter
    (feynmf_set tex sections (succ level) to_TeX_inner to_label_inner)
    set.inner

let feynmf_sets_plain sections level file
    to_TeX_outer to_label_outer to_TeX_inner to_label_inner sets =
  let tex = open_out (file ^ ".tex") in
  List.iter
    (feynmf_sets tex sections level
       to_TeX_outer to_label_outer to_TeX_inner to_label_inner)
    sets;
  close_out tex

let feynmf_header tex file =
  fprintf tex "\\documentclass[10pt]{article}\n";
  fprintf tex "\\usepackage{ifpdf}\n";
  fprintf tex "\\usepackage[colorlinks]{hyperref}\n";
  fprintf tex "\\usepackage[a4paper,margin=1cm]{geometry}\n";
  fprintf tex "\\usepackage{feynmp}\n";
  fprintf tex "\\ifpdf\n";
  fprintf tex "␣␣␣\\DeclareGraphicsRule{*}{mps}{*}{}\n";
  fprintf tex "\\else\n";
  fprintf tex "␣␣␣\\DeclareGraphicsRule{*}{eps}{*}{}\n";
  fprintf tex "\\fi\n";
  fprintf tex "\\setlength{\\unitlength}{1mm}\n";
  fprintf tex "\\setlength{\\parindent}{0pt}\n";
  fprintf tex
    "\\renewcommand{\\mathstrut}{\\protect\\vphantom{\\hat{0123456789}}}\n";
  fprintf tex "\\begin{document}\n";
  fprintf tex "\\tableofcontents\n";
  fprintf tex "\\begin{fmffile}{%s-fmf}\n\n" file

let feynmf_footer tex =
  fprintf tex "\n";
  fprintf tex "\\end{fmffile}␣\n";
  fprintf tex "\\end{document}␣\n"

let feynmf_sets_wrapped latex file
    to_TeX_outer to_label_outer to_TeX_inner to_label_inner sets =
  let tex = open_out (file ^ ".tex") in
  if latex then feynmf_header tex file;
  List.iter
    (feynmf_sets tex latex 1
       to_TeX_outer to_label_outer to_TeX_inner to_label_inner)
    sets;
  if latex then feynmf_footer tex;
  close_out tex

let feynmf_sets_wrapped_to_channel latex channel
    to_TeX_outer to_label_outer to_TeX_inner to_label_inner sets =
  if latex then feynmf_header channel "\\jobname";
  List.iter
    (feynmf_sets channel latex 1
        to_TeX_outer to_label_outer to_TeX_inner to_label_inner)
      sets;
  if latex then feynmf_footer channel

let rec feynmf_levels tex sections level to_TeX to_label set =
  fprintf tex "%s\\%s{%s}\n"
    (if sections then "" else "%%␣")
    (latex_section level)
    set.this.header;
  List.iter
    (to_feynmf_channel tex to_TeX to_label set.this.incoming)
    set.this.diagrams;
  List.iter (feynmf_levels tex sections (succ level) to_TeX to_label) set.lower

let feynmf_levels_plain sections level file to_TeX to_label sets =
  let tex = open_out (file ^ ".tex") in
  List.iter (feynmf_levels tex sections level to_TeX to_label) sets;
  close_out tex

let feynmf_levels_wrapped file to_TeX to_label sets =
  let tex = open_out (file ^ ".tex") in
  feynmf_header tex file;
  List.iter (feynmf_levels tex true 1 to_TeX to_label) sets;
  feynmf_footer tex;
  close_out tex
```

T.2.4 Least Squares Layout

$$L = \frac{1}{2}\sum_{i\neq i'} T_{ii'}\left(x_i - x_{i'}\right)^2 + \frac{1}{2}\sum_{i,j} T'_{ij}\left(x_i - e_j\right)^2 \tag{T.11}$$

and thus

$$0 = \frac{\partial L}{\partial x_i} = \sum_{i'\neq i} T_{ii'}\left(x_i - x_{i'}\right) + \sum_{j} T'_{ij}\left(x_i - e_j\right) \tag{T.12}$$

or

$$\left(\sum_{i'\neq i} T_{ii'} + \sum_j T'_{ij}\right) x_i - \sum_{i'\neq i} T_{ii'}x_{i'} = \sum_j T'_{ij}e_j \tag{T.13}$$

where we can assume that

$$T_{ii'} = T_{i'i} \tag{T.14a}$$
$$T_{ii} = 0 \tag{T.14b}$$

```
type α node_with_tension = { node : α; tension : float }

let unit_tension t =
  map (fun n → { node = n; tension = 1.0 }) (fun l → l) t

let leafs_and_nodes i2 t =
  let t' = sort_2i (≤ ) i2 t in
  match nodes t' with
  | [] → failwith "Tree.nodes_and_leafs:␣impossible"
  | i1 :: _ as n → (i1, i2, List.filter (fun l → l ≠ i2) (leafs t'), n)
```

Not tail recursive, but they're unlikely to meet any deep trees:

```
let rec internal_edges_from n = function
  | Leaf _ → []
  | Node (n', ch) → (n', n) :: (ThoList.flatmap (internal_edges_from n') ch)
```

The root node of the tree represents a vertex (node) and an external line (leaf) of the Feynman diagram simultaneously. Thus it requires special treatment:

```
let internal_edges = function
  | Leaf _ → []
  | Node (n, ch) → ThoList.flatmap (internal_edges_from n) ch
```

```
let rec external_edges_from n = function
  | Leaf (n′, _) → [(n′, n)]
  | Node (n′, ch) → ThoList.flatmap (external_edges_from n′) ch
```

```
let external_edges = function
  | Leaf (n, _) → [(n, n)]
  | Node (n, ch) → (n, n) :: ThoList.flatmap (external_edges_from n) ch
```

```
type ('edge, 'node, 'ext) graph =
    { int_nodes : 'node array;
      ext_nodes : 'ext array;
      int_edges : ('edge × int × int) list;
      ext_edges : ('edge × int × int) list }
```

```
module M = Pmap.Tree
```

Invert an array, viewed as a map from non-negative integers into a set. The result is a map from the set to the integers: val *invert_array* : α *array* → (α, *int*) *M.t*

```
let invert_array_unsafe a =
  fst (Array.fold_left (fun (m, i) a_i →
    (M.add compare a_i i m, succ i)) (M.empty, 0) a)
```

```
exception Not_invertible
```

```
let add_unique key data map =
  if M.mem compare key map then
    raise Not_invertible
  else
    M.add compare key data map
```

```
let invert_array a =
  fst (Array.fold_left (fun (m, i) a_i →
    (add_unique a_i i m, succ i)) (M.empty, 0) a)
```

```
let graph_of_tree nodes2edge conjugate i2 t =
  let i1, i2, out, vertices = leafs_and_nodes i2 t in
  let int_nodes = Array.of_list vertices
  and ext_nodes = Array.of_list (conjugate i1 :: i2 :: out) in
  let int_nodes_index_table = invert_array int_nodes
  and ext_nodes_index_table = invert_array ext_nodes in
  let int_nodes_index n = M.find compare n int_nodes_index_table
  and ext_nodes_index n = M.find compare n ext_nodes_index_table in
  { int_nodes = int_nodes;
    ext_nodes = ext_nodes;
    int_edges = List.map
      (fun (n1, n2) →
        (nodes2edge n1 n2, int_nodes_index n1, int_nodes_index n2))
      (internal_edges t);
    ext_edges = List.map
      (fun (e, n) →
        let e′ =
          if e = i1 then
            conjugate e
          else
            e in
        (nodes2edge e′ n, ext_nodes_index e′, int_nodes_index n))
      (external_edges t) }
```

```
let int_incidence f null g =
  let n = Array.length g.int_nodes in
  let incidence = Array.make_matrix n n null in
  List.iter (fun (edge, n1, n2) →
    if n1 ≠ n2 then begin
      let edge′ = f edge g.int_nodes.(n1) g.int_nodes.(n2) in
      incidence.(n1).(n2) ← edge';
      incidence.(n2).(n1) ← edge'
    end)
    g.int_edges;
  incidence

let ext_incidence f null g =
  let n_int = Array.length g.int_nodes
  and n_ext = Array.length g.ext_nodes in
  let incidence = Array.make_matrix n_int n_ext null in
  List.iter (fun (edge, e, n) →
    incidence.(n).(e) ← f edge g.ext_nodes.(e) g.int_nodes.(n))
    g.ext_edges;
  incidence

let division n =
  if n < 0 then
    []
  else if n = 1 then
    [0.5]
  else
    let n' = pred n in
    let d = 1.0 /. (float n') in
    let rec division' i acc =
      if i < 0 then
        acc
      else
        division' (pred i) (float i *. d :: acc) in
    division' n' []

type (ε, ν, 'ext) ext_layout = (ε, ν, 'ext × float × float) graph
type (ε, ν, 'ext) layout = (ε, ν × float × float, 'ext) ext_layout

let left_to_right num_in g =
  if num_in < 1 then
    invalid_arg "left_to_right"
  else
    let num_out = Array.length g.ext_nodes − num_in in
    if num_out < 1 then
      invalid_arg "left_to_right"
    else
      let incoming =
        List.map2 (fun e y → (e, 0.0, y))
          (Array.to_list (Array.sub g.ext_nodes 0 num_in))
          (division num_in)
      and outgoing =
        List.map2 (fun e y → (e, 1.0, y))
          (Array.to_list (Array.sub g.ext_nodes num_in num_out))
          (division num_out) in
      { g with ext_nodes = Array.of_list (incoming @ outgoing) }
```

Reformulating (T.13)

$$Ax = b_x \tag{T.15a}$$
$$Ay = b_y \tag{T.15b}$$

with

$$A_{ii'} = \left(\sum_{i''\neq i} T_{ii''} + \sum_j T'_{ij}\right)\delta_{ii'} - T_{ii'} \tag{T.16a}$$
$$(b_{x/y})_i = \sum_j T'_{ij}(e_{x/y})_j \tag{T.16b}$$

```
let sum a = Array.fold_left (+.) 0.0 a

let tension_to_equation t t' e =
  let xe, ye = List.split e in
  let bx = Linalg.matmulv t' (Array.of_list xe)
  and by = Linalg.matmulv t' (Array.of_list ye)
  and a = Array.init (Array.length t)
      (fun i →
        let a_i = Array.map (~-.) t.(i) in
        a_i.(i) ← a_i.(i) +. sum t.(i) +. sum t'.(i);
        a_i) in
  (a, bx, by)

let layout g =
  let ext_nodes =
    List.map (fun (_, x, y) → (x, y)) (Array.to_list g.ext_nodes) in
  let a, bx, by =
    tension_to_equation
      (int_incidence (fun _ _ _ → 1.0) 0.0 g)
      (ext_incidence (fun _ _ _ → 1.0) 0.0 g) ext_nodes in
  match Linalg.solve_many a [bx; by] with
  | [x; y] → { g with int_nodes = Array.mapi
                  (fun i n → (n, x.(i), y.(i))) g.int_nodes }
  | _ → failwith "impossible"

let iter_edges f g =
  List.iter (fun (edge, n1, n2) →
    let _, x1, y1 = g.int_nodes.(n1)
    and _, x2, y2 = g.int_nodes.(n2) in
    f edge (x1, y1) (x2, y2)) g.int_edges;
  List.iter (fun (edge, e, n) →
    let _, x1, y1 = g.ext_nodes.(e)
    and _, x2, y2 = g.int_nodes.(n) in
    f edge (x1, y1) (x2, y2)) g.ext_edges

let iter_internal f g =
  Array.iter (fun (node, x, y) → f (x, y)) g.int_nodes

let iter_incoming f g =
  f g.ext_nodes.(0);
  f g.ext_nodes.(1)

let iter_outgoing f g =
  for i = 2 to pred (Array.length g.ext_nodes) do
    f g.ext_nodes.(i)
  done

let dump g =
  Array.iter (fun (_, x, y) → Printf.eprintf "(%g,%g)␣" x y) g.ext_nodes;
  Printf.eprintf "\n␣=>␣";
  Array.iter (fun (_, x, y) → Printf.eprintf "(%g,%g)␣" x y) g.int_nodes;
  Printf.eprintf "\n"
```

—U—
Dependency Trees

U.1 Interface of Tree2

Dependency trees for wavefunctions.

```
type (ν, ε) t
val cons : (ε × ν × (ν, ε) t list) list → (ν, ε) t
val leaf : ν → (ν, ε) t

val is_singleton : (ν, ε) t → bool
val to_string : (ν → string) → (ε → string) → (ν, ε) t → string
val to_channel :
  out_channel → (ν → string) → (ε → string) → (ν, ε) t → unit
```

U.2 Implementation of Tree2

Dependency trees for wavefunctions.

```
type (ν, ε) t =
  | Node of (ε × ν × (ν, ε) t list) list
  | Leaf of ν

let leaf node = Leaf node

let sort_children (edge, node, children) =
  (edge, node, List.sort compare children)

let cons fusions = Node (List.sort compare (List.map sort_children fusions))

let is_singleton = function
  | Leaf _ → true
  | _ → false

let rec to_string n2s e2s = function
  | Leaf n → n2s n
  | Node [children] →
      children_to_string n2s e2s children
  | Node children2 →
      "{␣" ^
        String.concat "␣|␣" (List.map (children_to_string n2s e2s) children2) ^
          "␣}"

and children_to_string n2s e2s (e, n, children) =
  "(" ^ (match e2s e with "" → "" | s → s ^ ">") ^ n2s n ^ ":" ^
    (String.concat "," (List.map (to_string n2s e2s) children)) ^ ")"

let rec to_channel ch n2s e2s = function
  | Leaf n → Printf.fprintf ch "%s" (n2s n)
  | Node [] → Printf.fprintf ch "{␣}";
  | Node [children] → children_to_channel ch n2s e2s children
  | Node (children :: children2) →
      Printf.fprintf ch "{␣";
      children_to_channel ch n2s e2s children;
    List.iter
      (fun children →
        Printf.fprintf ch "␣\\\n␣␣␣|␣";
        children_to_channel ch n2s e2s children)
      children2;
    Printf.fprintf ch "␣}"

and children_to_channel ch n2s e2s (e, n, children) =
  Printf.fprintf ch "(";
  begin match e2s e with
  | "" → ()
  | s → Printf.fprintf ch "%s>" s
  end;
  Printf.fprintf ch "%s:" (n2s n);
  begin match children with
  | [] → ()
  | [child] → to_channel ch n2s e2s child
  | child :: children →
      to_channel ch n2s e2s child;
      List.iter
        (fun child →
          Printf.fprintf ch ",";
          to_channel ch n2s e2s child)
        children
  end;
  Printf.fprintf ch ")"
```

—V—

Consistency Checks

Application `count.ml` *unavailable!*

—W— Complex Numbers

Interface `complex.mli` *unavailable!*

Implementation `complex.ml` *unavailable!*

—X— Algebra

X.1 Interface of Algebra

```
module type Test =
  sig
    val suite : OUnit.test
  end
```

X.1.1 Coefficients

For our algebra, we need coefficient rings with addition, subtraction, multiplication and the corresponding neutral elements.

```
module type CRing =
  sig
    type t
```

add null x = x = add x null

```
    val null : t
    val is_null : t → bool
    val add : t → t → t
```

neg x = sub null x and *sub x y = add x (neg y)*

```
    val neg : t → t
    val sub : t → t → t
```

mul unit x = x = mul x unit

```
    val unit : t
    val is_unit : t → bool
    val mul : t → t → t
```

Equality:

```
    val equal : t → t → bool

  end
```

Rational numbers provide a particularly important example and they come with a partial inverse:

```
module type Rational =
  sig
    include CRing
    val is_positive : t → bool
    val is_negative : t → bool
    val is_integer : t → bool
    val make : int → int → t
    val abs : t → t
    val inv : t → t
    val div : t → t → t
    val pow : t → int → t
    val sum : t list → t
    val to_ratio : t → int × int
    val to_float : t → float
    val to_integer : t → int
    (∗ Convenience: n ↦ n/1 and n ↦ 1/n ∗)
    val int : int → t
    val fraction : int → t
    (∗ Order ∗)
    val compare : t → t → int
    (∗ Tracing, debugging, toplevel and unit testing ∗)
    val to_string : t → string
    val pp : Format.formatter → t → unit
    module Test : Test
  end
```

X.1.2 Naive Rational Arithmetic

This *is* dangerous and will overflow even for simple applications. The production code will have to be linked to a library for large integer arithmetic.

```
module Small_Rational : Rational
module Q : Rational
```

X.1.3 Rational Complex Numbers

```
module type QComplex =
  sig

    include CRing

    type q
    val make : q → q → t

    val re : t → q
    val im : t → q
    val conj : t → t

    val inv : t → t
    val div : t → t → t

    val pow : t → int → t
    val sum : t list → t

    val is_positive : t → bool
    val is_negative : t → bool
    val is_integer : t → bool
    val is_real : t → bool
```

Convenience: real rationals and integers,

```
    val rational : q → t
    val int : int → t
```

$n \to 1/n$

```
    val fraction : int → t
```

$n \to n\mathrm{i}$

```
    val imag : int → t
```

Order

```
    val compare : t → t → int
```

Tracing, debugging, toplevel and unit testing

```
    val to_string : t → string
    val pp : Format.formatter → t → unit
    module Test : Test
  end
module QComplex : functor (Q' : Rational) → QComplex with type q = Q'.t
module QC : QComplex with type q = Q.t
```

X.1.4 Laurent Polynomials

Polynomials, including negative powers, in one variable. In our applications, the variable x will often be N_C, the number of colors

$$\sum_n c_n N_C^n \tag{X.1}$$

```
module type Laurent =
  sig
    include CRing
```

The type of coefficients. In the implementation below, it is *QComplex.t*: complex numbers with rational real and imaginary parts.

```
    type c
```

atom c n constructs a term cx^n, where x denotes the variable.

```
    val atom : c → int → t
```

Shortcut: *const c = atom c* 0

```
    val const : c → t
```

Elementary arithmetic

```
    val scale : c → t → t
    val sum : t list → t
    val product : t list → t
    val pow : t → int → t
```

$log(cN_C^n)$ returns $Some(c, n)$. For other terms, *log* returns *None*.

```
    val log : t → (c × int) option
```

return the corresponding list of coefficients and descending powers

```
    val to_list : t → (c × int) list
```

eval c p evaluates the polynomial p by substituting the constant c for the variable.

```
    val eval : c → t → c
```

A total ordering. Does not correspond to any mathematical order.

```
    val compare : t → t → int
```

Provide some convenience functions for constructing coefficients from integers and rationals.
Rationals coefficients (without imaginary part!) $\{(q_i, n_i)\}_n \mapsto \sum_i q_i x^{n_i}$

```
    val rationals : (Q.t × int) list → t
```

Integer coefficients $\{(k_i, n_i)\}_n \mapsto \sum_i k_i x^{n_i}$

```
    val ints : (int × int) list → t
```

For convenience, some special cases. Starting with injections

```
    val rational : Q.t → t
    val int : int → t
```

$k \mapsto 1/k = k^{-1}$

```
    val fraction : int → t
```

$k \mapsto k\mathrm{i}$

```
    val imag : int → t
```

$k \mapsto kx$

```
    val nc : int → t
```

$k \mapsto k/x = kx^{-1}$

```
    val over_nc : int → t
```

Tracing, debugging, toplevel and unit testing

```
    val to_string : string → t → string
    val pp : Format.formatter → t → unit
    module Test : Test

  end
```

Could (should?) be functorialized over *QComplex*. We had to wait until we upgraded our O'Caml requirements to 4.02, but that has been done.

```
module Laurent : Laurent with type c = QC.t
```

X.1.5 Expressions: Terms, Rings and Linear Combinations

The tensor algebra will be spanned by an abelian monoid:

```
module type Term =
  sig
    type α t
    val unit : unit → α t
    val is_unit : α t → bool
    val atom : α → α t
    val power : α t → int → α t
    val mul : α t → α t → α t
    val map : (α → β) → α t → β t
    val to_string : (α → string) → α t → string
```

The derivative of a term is *not* a term, but a sum of terms instead:

$$D(f_1^{p_1}f_2^{p_2}\cdots f_n^{p_n}) = \sum_i (Df_i)p_i f_1^{p_1}f_2^{p_2}\cdots f_i^{p_i-1}\cdots f_n^{p_n} \tag{X.2}$$

The function returns the sum as a list of triples $(Df_i, p_i, f_1^{p_1}f_2^{p_2}\cdots f_i^{p_i-1}\cdots f_n^{p_n})$. Summing the terms is left to the calling module and the Df_i are *not* guaranteed to be different. NB: The function implementating the inner derivative, is supposed to return *Some* Df_i and *None*, iff Df_i vanishes.

```
    val derive : (α → β option) → α t → (β × int × α t) list
```

convenience function

```
    val product : α t list → α t
    val atoms : α t → α list

  end
```

```
module type Ring =
  sig
    module C : Rational
    type α t
    val null : unit → α t
    val unit : unit → α t
    val is_null : α t → bool
    val is_unit : α t → bool
    val atom : α → α t
    val scale : C.t → α t → α t
    val add : α t → α t → α t
    val sub : α t → α t → α t
    val mul : α t → α t → α t
    val neg : α t → α t
```

Again

$$D(f_1^{p_1} f_2^{p_2} \cdots f_n^{p_n}) = \sum_i (Df_i) p_i f_1^{p_1} f_2^{p_2} \cdots f_i^{p_i - 1} \cdots f_n^{p_n} \tag{X.3}$$

but, iff Df_i can be identified with a f', we know how to perform the sum.

```
    val derive_inner : (α → α t) → α t → α t (* this? *)
    val derive_inner' : (α → α t option) → α t → α t (* or that? *)
```

Below, we will need partial derivatives that lead out of the ring: *derive_outer derive_atom term* returns a list of partial derivatives β with non-zero coefficients $\alpha\ t$:

```
    val derive_outer : (α → β option) → α t → (β × α t) list
```

convenience functions

```
    val sum : α t list → α t
    val product : α t list → α t
```

The list of all generators appearing in an expression:

```
    val atoms : α t → α list

    val to_string : (α → string) → α t → string

  end
```

```
module type Linear =
  sig
    module C : Ring
    type (α, γ) t
    val null : unit → (α, γ) t
    val atom : α → (α, γ) t
    val singleton : γ C.t → α → (α, γ) t
    val scale : γ C.t → (α, γ) t → (α, γ) t
    val add : (α, γ) t → (α, γ) t → (α, γ) t
    val sub : (α, γ) t → (α, γ) t → (α, γ) t
```

A partial derivative w. r. t. a vector maps from a coefficient ring to the dual vector space.

```
    val partial : (γ → (α, γ) t) → γ C.t → (α, γ) t
```

A linear combination of vectors

$$\mathit{linear}[(v_1, c_1); (v_2, c_2); \ldots; (v_n, c_n)] = \sum_{i=1}^{n} c_i \cdot v_i \tag{X.4}$$

```
    val linear : ((α, γ) t × γ C.t) list → (α, γ) t
```

Some convenience functions

```
    val map : (α → γ C.t → (β, δ) t) → (α, γ) t → (β, δ) t
    val sum : (α, γ) t list → (α, γ) t
```

The list of all generators and the list of all generators of coefficients appearing in an expression:

```
    val atoms : (α, γ) t → α list × γ list

    val to_string : (α → string) → (γ → string) → (α, γ) t → string

  end
```

```
module Term : Term

module Make_Ring (C : Rational) (T : Term) : Ring
module Make_Linear (C : Ring) : Linear with module C = C
```

X.2 Implementation of *Algebra*

```
module type Test =
  sig
    val suite : OUnit.test
  end
```

The terms will be small and there's no need to be fancy and/or efficient. It's more important to have a unique representation.

```
module PM = Pmap.List
```

X.2.1 Coefficients

```
module type CRing =
  sig
    type t
    val null : t
    val is_null : t → bool
    val add : t → t → t
    val neg : t → t
    val sub : t → t → t
    val unit : t
    val is_unit : t → bool
    val mul : t → t → t
    val equal : t → t → bool
  end
```

```
module type Rational =
  sig
    include CRing
    val is_positive : t → bool
    val is_negative : t → bool
    val is_integer : t → bool
    val make : int → int → t
    val abs : t → t
    val inv : t → t
    val div : t → t → t
    val pow : t → int → t
    val sum : t list → t
    val to_ratio : t → int × int
    val to_float : t → float
    val to_integer : t → int
    val int : int → t
    val fraction : int → t
    val compare : t → t → int
    val to_string : t → string
    val pp : Format.formatter → t → unit
    module Test : Test
  end
```

X.2.2 Naive Rational Arithmetic

This *is* dangerous and will overflow even for simple applications. The production code will have to be linked to a library for large integer arithmetic.

Anyway, here's Euclid's algorithm:

```
let rec gcd i1 i2 =
  if i2 = 0 then
    abs i1
  else
    gcd i2 (i1 mod i2)

let lcm i1 i2 = (i1 / gcd i1 i2) × i2

let abs_int = abs

module Small_Rational : Rational =
  struct

    type t = int × int

    let is_null (n, _) = (n = 0)
    let is_unit (n, d) = (n ≠ 0) ∧ (n = d)
    let is_positive (n, d) = n × d > 0
    let is_negative (n, d) = n × d < 0
    let is_integer (n, d) = (gcd n d = d)

    let null = (0, 1)
    let unit = (1, 1)

    let make n d =
      let c = gcd n d in
      (n / c, d / c)

    let abs (n, d) = (abs n, abs d)
    let inv (n, d) = (d, n)
    let mul (n1, d1) (n2, d2) = make (n1 × n2) (d1 × d2)
    let div q1 q2 = mul q1 (inv q2)
    let add (n1, d1) (n2, d2) = make (n1 × d2 + n2 × d1) (d1 × d2)
    let sub (n1, d1) (n2, d2) = make (n1 × d2 − n2 × d1) (d1 × d2)
    let neg (n, d) = (− n, d)

    let rec pow q p =
      if p = 0 then
        unit
      else if p < 0 then
        pow (inv q) (−p)
      else
        mul q (pow q (pred p))

    let sum qs =
      List.fold_right add qs null

    let to_ratio (n, d) =
      if d < 0 then
        (−n, − d)
      else
        (n, d)

    let to_float (n, d) = float n /. float d

    let to_string (n, d) =
      if abs_int d = 1 then
        Printf.sprintf "%d" (d × n)
      else
        let n, d = to_ratio (n, d) in
        Printf.sprintf "(%d/%d)" n d

    let pp fmt qc =
      Format.fprintf fmt "%s" (to_string qc)

    let to_integer (n, d) =
      if is_integer (n, d) then
        n
      else
        invalid_arg "Algebra.Small_Rational.to_integer"

    let int n = make n 1
    let fraction n = make 1 n

    let compare q1 q2 =
      let n1, d1 = to_ratio q1
      and n2, d2 = to_ratio q2 in
      compare (d2 × n1) (d1 × n2)

    let equal (n1, d1) (n2, d2) =
      d2 × n1 = d1 × n2

    module Test =
      struct
        open OUnit

        let assert_equal_rational z1 z2 =
          assert_equal ~printer : to_string ~cmp : equal z1 z2

        let suite_mul =
          "mul" >:::

            [ "1*1=1" >::
                (fun () →
                  assert_equal_rational (mul unit unit) unit) ]

        let suite =
          "Algebra.Small_Rational" >:::
            [suite_mul]
      end

  end
module Q = Small_Rational
```

X.2.3 Rational Complex Numbers

```
module type QComplex =
  sig
    include CRing
    type q
    val make : q → q → t
    val re : t → q
    val im : t → q
    val conj : t → t
    val inv : t → t
    val div : t → t → t
    val pow : t → int → t
    val sum : t list → t
    val is_positive : t → bool
    val is_negative : t → bool
    val is_integer : t → bool
    val is_real : t → bool
    val rational : q → t
    val int : int → t
    val fraction : int → t
    val imag : int → t
    val compare : t → t → int
    val to_string : t → string
    val pp : Format.formatter → t → unit
    module Test : Test
  end

module QComplex (Q : Rational) : QComplex with type q = Q.t =
  struct

    type q = Q.t
    type t = { re : q; im : q }
```

```
let make re im = { re; im }
let null = { re = Q.null; im = Q.null }
let unit = { re = Q.unit; im = Q.null }

let re z = z.re
let im z = z.im
let conj z = { re = z.re; im = Q.neg z.im }

let neg z = { re = Q.neg z.re; im = Q.neg z.im }
let add z1 z2 = { re = Q.add z1.re z2.re; im = Q.add z1.im z2.im }
let sub z1 z2 = { re = Q.sub z1.re z2.re; im = Q.sub z1.im z2.im }

let sum qs =
  List.fold_right add qs null
```

Save one multiplication with respect to the standard formula

$$(x+iy)(u+iv) = [xu - yv] + i[(x+u)(y+v) - xu - yv] \tag{X.5}$$

at the expense of one addition and two subtractions.

```
let mul z1 z2 =
  let re12 = Q.mul z1.re z2.re
  and im12 = Q.mul z1.im z2.im in
  { re = Q.sub re12 im12;
    im = Q.sub
           (Q.sub (Q.mul (Q.add z1.re z1.im) (Q.add z2.re z2.im)) re12)
           im12 }

let inv z =
  let modulus = Q.add (Q.mul z.re z.re) (Q.mul z.im z.im) in
  { re = Q.div z.re modulus;
    im = Q.div (Q.neg z.im) modulus }

let div n d =
  mul (inv d) n

let rec pow q p =
  if p = 0 then
    unit
  else if p < 0 then
    pow (inv q) (−p)
  else
    mul q (pow q (pred p))

let is_real q =
  Q.is_null q.im

let test_real test q =
  is_real q ∧ test q.re

let is_null = test_real Q.is_null
let is_unit = test_real Q.is_unit
let is_positive = test_real Q.is_positive
let is_negative = test_real Q.is_negative
let is_integer = test_real Q.is_integer

let rational q = make q Q.null
let int n = rational (Q.int n)
let fraction n = rational (Q.fraction n)
let imag n = make Q.null (Q.int n)

let compare { re = re1; im = im1 } { re = re2; im = im2 } =
  let c = compare re1 re2 in
  if c ≠ 0 then
    c
  else
    compare im1 im2
    let equal c1 c2 =
      compare c1 c2 = 0

    let q_to_string q =
      (if Q.is_negative q then "-" else " ") ^ Q.to_string (Q.abs q)

    let to_string z =
      if Q.is_null z.im then
        q_to_string z.re
      else if Q.is_null z.re then
        if Q.is_unit z.im then
          " I"
        else if Q.is_unit (Q.neg z.im) then
          "-I"
        else
          q_to_string z.im ^ "*I"
      else
        Printf.sprintf "(%s%s*I)" (Q.to_string z.re) (q_to_string z.im)

    let pp fmt qc =
      Format.fprintf fmt "%s" (to_string qc)

    module Test =
      struct
        open OUnit

        let assert_equal_complex z1 z2 =
          assert_equal ~printer : to_string ~cmp : equal z1 z2

        let suite_mul =
          "mul" >:::

            [ "1*1=1" >::
                (fun () →
                  assert_equal_complex (mul unit unit) unit) ]

        let suite =
          "Algebra.QComplex" >:::
            [suite_mul]
      end

  end

module QC = QComplex(Q)
```

X.2.4 Laurent Polynomials

```
module type Laurent =
  sig
    include CRing
    type c
    val atom : c → int → t
    val const : c → t
    val scale : c → t → t
    val sum : t list → t
    val product : t list → t
    val pow : t → int → t
    val log : t → (c × int) option
    val to_list : t → (c × int) list
    val eval : c → t → c
    val compare : t → t → int
    val rationals : (Q.t × int) list → t
    val ints : (int × int) list → t
    val rational : Q.t → t
    val int : int → t
    val fraction : int → t
    val imag : int → t
    val nc : int → t
    val over_nc : int → t
    val to_string : string → t → string
    val pp : Format.formatter → t → unit
    module Test : Test
  end
```

```
module Laurent : Laurent with type c = QC.t =
  struct

    module IMap = Map.Make(Int)

    type c = QC.t

    let qc_minus_one =
      QC.neg QC.unit

    type t = c IMap.t

    let null = IMap.empty
    let is_null l = IMap.for_all (fun _ → QC.is_null) l

    let atom qc n =
      if qc = QC.null then
        null
      else
        IMap.singleton n qc

    let const z = atom z 0
    let unit = const QC.unit
    let is_unit l = IMap.equal QC.equal l unit

    let add1 n qc l =
      try
        let qc' = QC.add qc (IMap.find n l) in
        if qc' = QC.null then
          IMap.remove n l
        else
          IMap.add n qc' l
      with
      | Not_found → IMap.add n qc l

    let add l1 l2 =
      IMap.fold add1 l1 l2

    let sum = function
      | [] → null
      | [l] → l
      | l :: l_list →
          List.fold_left add l l_list

    let scale qc l =
      IMap.map (QC.mul qc) l

    let neg l =
      IMap.map QC.neg l

    let sub l1 l2 =
      add l1 (neg l2)
```

cf. *Product.fold2_rev*

```
    let fold2 f l1 l2 acc =
      IMap.fold
        (fun n1 qc1 acc1 →
          IMap.fold
            (fun n2 qc2 acc2 → f n1 qc1 n2 qc2 acc2)
          l2 acc1)
      l1 acc

let mul l1 l2 =
  fold2
      (fun n1 qc1 n2 qc2 acc →
          add1 (n1 + n2) (QC.mul qc1 qc2) acc)
      l1 l2 null

let product = function
  | [] → unit
  | [l] → l
  | l :: l_list →
       List.fold_left mul l l_list

let poly_pow multiply one inverse x n =
  let rec pow' i x' acc =
      if i < 1 then
          acc
      else
          pow' (pred i) x' (multiply x' acc) in
  if n < 0 then
      let x' = inverse x in
      pow' (pred (−n)) x' x'
  else if n = 0 then
      one
  else
      pow' (pred n) x x

let qc_pow z n =
  poly_pow QC.mul QC.unit QC.inv z n

let pow l n =
  poly_pow mul unit (fun _ → invalid_arg "Algebra.Laurent.pow") l n

let log l =
  match IMap.bindings l with
  | [] → Some (QC.null, 0)
  | [(p, c)] → Some (c, p)
  | _ → None

let to_list l =
  List.map (fun (p, c) → (c, p)) (IMap.bindings l)

let q_to_string q =
  (if Q.is_positive q then "+" else "-") ^ Q.to_string (Q.abs q)

let qc_to_string z =
  let r = QC.re z
  and i = QC.im z in
  if Q.is_null i then
      q_to_string r
  else if Q.is_null r then
      if Q.is_unit i then
          "+I"
      else if Q.is_unit (Q.neg i) then
          "-I"
      else
          q_to_string i ^ "*I"
  else
      Printf.sprintf "(%s%s*I)" (Q.to_string r) (q_to_string i)

let to_string1 name (n, qc) =
  if n = 0 then
      qc_to_string qc
  else if n = 1 then
```

```
        if QC.is_unit qc then
          name
        else if qc = qc_minus_one then
          "-" ^ name
        else
          Printf.sprintf "%s*%s" (qc_to_string qc) name
      else if n = -1 then
        Printf.sprintf "%s/%s" (qc_to_string qc) name
      else if n > 1 then
        if QC.is_unit qc then
          Printf.sprintf "%s^%d" name n
        else if qc = qc_minus_one then
          Printf.sprintf "-%s^%d" name n
        else
          Printf.sprintf "%s*%s^%d" (qc_to_string qc) name n
      else
        Printf.sprintf "%s/%s^%d" (qc_to_string qc) name (-n)

    let to_string name l =
      match IMap.bindings l with
      | [] → "0"
      | l → String.concat "" (List.map (to_string1 name) l)

    let pp fmt l =
      Format.fprintf fmt "%s" (to_string "N" l)

    let eval v l =
      IMap.fold
        (fun n qc acc → QC.add (QC.mul qc (qc_pow v n)) acc)
        l QC.null

    let compare l1 l2 =
      IMap.compare Stdlib.compare l1 l2

    let equal l1 l2 =
      compare l1 l2 = 0
```

Laurent polynomials:

```
    let of_pairs f pairs =
      sum (List.map (fun (coeff, power) → atom (f coeff) power) pairs)

    let rationals = of_pairs QC.rational
    let ints = of_pairs QC.int

    let rational q = rationals [(q, 0)]
    let int n = ints [(n, 0)]
    let fraction n = const (QC.fraction n)
    let imag n = const (QC.imag n)
    let nc n = ints [(n, 1)]
    let over_nc n = ints [(n, -1)]

    module Test =
      struct
        open OUnit

        let assert_equal_laurent l1 l2 =
          assert_equal ~printer : (to_string "N") ~cmp : equal l1 l2

        let suite_mul =
          "mul" >:::

            [ "(1+N)(1-N)=1-N^2" >::
                (fun () →
                  assert_equal_laurent
                    (sum [unit; atom (QC.neg QC.unit) 2])
                    (product [sum [unit; atom QC.unit 1];
                              sum [unit; atom (QC.neg QC.unit) 1]]));
```

```
              "(1+N)(1-1/N)=N-1/N" >::
                (fun () →
                  assert_equal_laurent
                    (sum [atom QC.unit 1; atom (QC.neg QC.unit) (−1)])
                    (product [sum [unit; atom QC.unit 1];
                              sum [unit; atom (QC.neg QC.unit) (−1)]])); ]

        let suite =
          "Algebra.Laurent" >:::
            [suite_mul]
      end

  end
```

X.2.5 Expressions: Terms, Rings and Linear Combinations

The tensor algebra will be spanned by an abelian monoid:

```
module type Term =
  sig
    type α t
    val unit : unit → α t
    val is_unit : α t → bool
    val atom : α → α t
    val power : α t → int → α t
    val mul : α t → α t → α t
    val map : (α → β) → α t → β t
    val to_string : (α → string) → α t → string
    val derive : (α → β option) → α t → (β × int × α t) list
    val product : α t list → α t
    val atoms : α t → α list
  end

module type Ring =
  sig
    module C : Rational
    type α t
    val null : unit → α t
    val unit : unit → α t
    val is_null : α t → bool
    val is_unit : α t → bool
    val atom : α → α t
    val scale : C.t → α t → α t
    val add : α t → α t → α t
    val sub : α t → α t → α t
    val mul : α t → α t → α t
    val neg : α t → α t
    val derive_inner : (α → α t) → α t → α t (* this? *)
    val derive_inner′ : (α → α t option) → α t → α t (* or that? *)
    val derive_outer : (α → β option) → α t → (β × α t) list
    val sum : α t list → α t
    val product : α t list → α t
    val atoms : α t → α list
    val to_string : (α → string) → α t → string
  end

module type Linear =
  sig
    module C : Ring
    type (α, γ) t
    val null : unit → (α, γ) t
    val atom : α → (α, γ) t
    val singleton : γ C.t → α → (α, γ) t
    val scale : γ C.t → (α, γ) t → (α, γ) t
    val add : (α, γ) t → (α, γ) t → (α, γ) t
    val sub : (α, γ) t → (α, γ) t → (α, γ) t
    val partial : (γ → (α, γ) t) → γ C.t → (α, γ) t
    val linear : ((α, γ) t × γ C.t) list → (α, γ) t
    val map : (α → γ C.t → (β, δ) t) → (α, γ) t → (β, δ) t
    val sum : (α, γ) t list → (α, γ) t
    val atoms : (α, γ) t → α list × γ list
    val to_string : (α → string) → (γ → string) → (α, γ) t → string
  end

module Term : Term =
  struct

    module M = PM

    type α t = (α, int) M.t

    let unit () = M.empty
    let is_unit = M.is_empty

    let atom f = M.singleton f 1

    let power x p = M.map (( × ) p) x

    let insert1 binop f p term =
      let p′ = binop (try M.find compare f term with Not_found → 0) p in
      if p′ = 0 then
        M.remove compare f term
      else
        M.add compare f p′ term

    let mul1 f p term = insert1 (+) f p term
    let mul x y = M.fold mul1 x y

    let map f term = M.fold (fun t → mul1 (f t)) term M.empty

    let to_string fmt term =
      String.concat "*"
        (M.fold (fun f p acc →
          (if p = 0 then
            "1"
          else if p = 1 then
            fmt f
          else
            "[" ^ fmt f ^ "]^" ^ string_of_int p) :: acc) term [])

    let derive derive1 x =
      M.fold (fun f p dx →
        if p ≠ 0 then
          match derive1 f with
          | Some df → (df, p, mul1 f (pred p) (M.remove compare f x)) :: dx
          | None → dx
        else
          dx) x []

    let product factors =
      List.fold_left mul (unit ()) factors

    let atoms t =
      List.map fst (PM.elements t)

  end

module Make_Ring (C : Rational) (T : Term) : Ring =
  struct

    module C = C
    let one = C.unit
```

```
module M = PM

type α t = (α T.t, C.t) M.t

let null () = M.empty
let is_null = M.is_empty

let power t p = M.singleton t p
let unit () = power (T.unit ()) one

let is_unit t = unit () = t
```

The following should be correct too, but produces to many false positives instead! What's going on?

```
let broken__is_unit t =
  match M.elements t with
  | [(t, p)] → T.is_unit t ∨ C.is_null p
  | _ → false

let atom t = power (T.atom t) one

let scale c x = M.map (C.mul c) x

let insert1 binop t c sum =
  let c′ = binop (try M.find compare t sum with Not_found → C.null) c in
  if C.is_null c′ then
    M.remove compare t sum
  else
    M.add compare t c′ sum

let add x y = M.fold (insert1 C.add) x y

let sub x y = M.fold (insert1 C.sub) y x
```

One might be tempted to use *Product.outer_self M.fold* instead, but this would require us to combine *tx* and *cx* to (*tx*, *cx*).

```
let fold2 f x y =
  M.fold (fun tx cx → M.fold (f tx cx) y) x

let mul x y =
  fold2 (fun tx cx ty cy → insert1 C.add (T.mul tx ty) (C.mul cx cy))
    x y (null ())

let neg x =
  sub (null ()) x

let neg x =
  scale (C.neg C.unit) x
```

Multiply the *derivatives* by *c* and add the result to *dx*.

```
let add_derivatives derivatives c dx =
  List.fold_left (fun acc (df, dt_c, dt_t) →
    add (mul df (power dt_t (C.mul c (C.make dt_c 1)))) acc) dx derivatives

let derive_inner derive1 x =
  M.fold (fun t →
    add_derivatives (T.derive (fun f → Some (derive1 f)) t)) x (null ())

let derive_inner′ derive1 x =
  M.fold (fun t → add_derivatives (T.derive derive1 t)) x (null ())

let collect_derivatives derivatives c dx =
  List.fold_left (fun acc (df, dt_c, dt_t) →
    (df, power dt_t (C.mul c (C.make dt_c 1))) :: acc) dx derivatives

let derive_outer derive1 x =
  M.fold (fun t → collect_derivatives (T.derive derive1 t)) x []

let sum terms =
  List.fold_left add (null ()) terms
```

```
    let product factors =
      List.fold_left mul (unit ()) factors

    let atoms t =
      ThoList.uniq (List.sort compare
                        (ThoList.flatmap (fun (t, _) → T.atoms t) (PM.elements t)))

    let to_string fmt sum =
      "(" ^ String.concat "␣+␣"
              (M.fold (fun t c acc →
                if C.is_null c then
                  acc
                else if C.is_unit c then
                  T.to_string fmt t :: acc
                else if C.is_unit (C.neg c) then
                  ("(-" ^ T.to_string fmt t ^ ")") :: acc
                else
                  (C.to_string c ^ "*[" ^ T.to_string fmt t ^ "]") :: acc) sum []) ^ ")"

  end

module Make_Linear (C : Ring) : Linear with module C = C =
  struct

    module C = C

    module M = PM

    type (α, γ) t = (α, γ C.t) M.t

    let null () = M.empty
    let is_null = M.is_empty
    let atom a = M.singleton a (C.unit ())
    let singleton c a = M.singleton a c

    let scale c x = M.map (C.mul c) x

    let insert1 binop t c sum =
      let c′ = binop (try M.find compare t sum with Not_found → C.null ()) c in
      if C.is_null c′ then
        M.remove compare t sum
      else
        M.add compare t c′ sum

    let add x y = M.fold (insert1 C.add) x y
    let sub x y = M.fold (insert1 C.sub) y x

    let map f t =
      M.fold (fun a c → add (f a c)) t M.empty

    let sum terms =
      List.fold_left add (null ()) terms

    let linear terms =
      List.fold_left (fun acc (a, c) → add (scale c a) acc) (null ()) terms

    let partial derive t =
      let d t′ =
        let dt′ = derive t′ in
        if is_null dt′ then
          None
        else
          Some dt′ in
      linear (C.derive_outer d t)

    let atoms t =
      let a, c = List.split (PM.elements t) in
      (a, ThoList.uniq (List.sort compare (ThoList.flatmap C.atoms c)))

    let to_string fmt cfmt sum =
    "(" ^ String.concat "␣+␣"
              (M.fold (fun t c acc →
                 if C.is_null c then
                    acc
                 else if C.is_unit c then
                    fmt t :: acc
                 else if C.is_unit (C.neg c) then
                    ("(-" ^ fmt t ^ ")") :: acc
                 else
                    (C.to_string cfmt c ^ "*" ^ fmt t) :: acc)
                 sum []) ^ ")"

end
```

—Y— Simple Linear Algebra

Y.1 Interface of Linalg

```
exception Singular
exception Not_Square

val copy_matrix : float array array → float array array

val matmul : float array array → float array array → float array array
val matmulv : float array array → float array → float array

val lu_decompose : float array array → float array array × float array array
val solve : float array array → float array → float array
val solve_many : float array array → float array list → float array list
```

Y.2 Implementation of Linalg

This is not a functional implementations, but uses imperative array in Fotran style for maximimum speed.

```
exception Singular
exception Not_Square

let copy_matrix a =
  Array.init (Array.length a)
    (fun i → Array.copy a.(i))

let matmul a b =
  let ni = Array.length a
  and nj = Array.length b.(0)
  and n = Array.length b in
  let ab = Array.make_matrix ni nj 0.0 in
  for i = 0 to pred ni do
    for j = 0 to pred nj do
      for k = 0 to pred n do
        ab.(i).(j) ← ab.(i).(j) +. a.(i).(k) *. b.(k).(j)
      done
    done
  done;
  ab

let matmulv a v =
  let na = Array.length a in
  let nv = Array.length v in
  let v' = Array.make na 0.0 in
  for i = 0 to pred na do
    for j = 0 to pred nv do
      v'.(i) ← v'.(i) +. a.(i).(j) *. v.(j)
    done
  done;
  v'

let maxabsval a : float =
  let x = ref (abs_float a.(0)) in
  for i = 1 to Array.length a − 1 do
    x := max !x (abs_float a.(i))
  done;
  !x
```

Y.2.1 LU Decomposition

$$A = LU \tag{Y.1a}$$

In more detail

$$\begin{pmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \dots & a_{(n-1)(n-1)} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{10} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ l_{(n-1)0} & l_{(n-1)1} & \dots & 1 \end{pmatrix} \begin{pmatrix} u_{00} & u_{01} & \dots & u_{0(n-1)} \\ 0 & u_{11} & \dots & u_{1(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & u_{(n-1)(n-1)} \end{pmatrix} \tag{Y.1b}$$

Rewriting (Y.1) in block matrix notation

$$\begin{pmatrix} a_{00} & a_{0\cdot} \\ a_{\cdot 0} & A \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ l_{\cdot 0} & L \end{pmatrix} \begin{pmatrix} u_{00} & u_{0\cdot} \\ 0 & U \end{pmatrix} = \begin{pmatrix} u_{00} & u_{0\cdot} \\ l_{\cdot 0}u_{00} & l_{\cdot 0} \otimes u_{0\cdot} + LU \end{pmatrix} \tag{Y.2}$$

we can solve it easily

$$u_{00} = a_{00} \tag{Y.3a}$$
$$u_{0\cdot} = a_{0\cdot} \tag{Y.3b}$$
$$l_{\cdot 0} = \frac{a_{\cdot 0}}{a_{00}} \tag{Y.3c}$$
$$LU = A - \frac{a_{\cdot 0} \otimes a_{0\cdot}}{a_{00}} \tag{Y.3d}$$

and (Y.3c) and (Y.3d) define a simple iterative algorithm if we work from the outside in. It just remains to add pivoting.

```
let swap a i j =
  let a_i = a.(i) in
  a.(i) ← a.(j);
  a.(j) ← a_i

let pivot_column v a n =
  let n' = ref n
  and max_va = ref (v.(n) *. (abs_float a.(n).(n))) in
  for i = succ n to Array.length v − 1 do
    let va_i = v.(i) *. (abs_float a.(i).(n)) in
    if va_i > !max_va then begin
      n' := i;
      max_va := va_i
    end
  done;
  !n'

let lu_decompose_in_place a =
  let n = Array.length a in
  let eps = ref 1
  and pivots = Array.make n 0
  and v =
    try
      Array.init n (fun i →
        let a_i = a.(i) in
        if Array.length a_i ≠ n then
          raise Not_Square;
        1.0 /. (maxabsval a_i))
    with
    | Division_by_zero → raise Singular in
  for i = 0 to pred n do
    let pivot = pivot_column v a i in
    if pivot ≠ i then begin
      swap a pivot i;
      eps := − !eps;
      v.(pivot) ← v.(i)
    end;
    pivots.(i) ← pivot;
    let inv_a_ii =
      try 1.0 /. a.(i).(i) with Division_by_zero → raise Singular in
    for j = succ i to pred n do
      a.(j).(i) ← inv_a_ii *. a.(j).(i)
    done;
    for j = succ i to pred n do
      for k = succ i to pred n do
        a.(j).(k) ← a.(j).(k) −. a.(j).(i) *. a.(i).(k)
      done
    done
  done;
  (pivots, !eps)

let lu_decompose_split a pivots =
  let n = Array.length pivots in
  let l = Array.make_matrix n n 0.0 in
  let u = Array.make_matrix n n 0.0 in
  for i = 0 to pred n do
    l.(i).(i) ← 1.0;
    for j = succ i to pred n do
      l.(j).(i) ← a.(j).(i)
    done
  done;
  for i = pred n downto 0 do
    swap l i pivots.(i)
  done;
  for i = 0 to pred n do
    for j = 0 to i do
      u.(j).(i) ← a.(j).(i)
    done
  done;
  (l, u)

let lu_decompose a =
  let a = copy_matrix a in
  let pivots, _ = lu_decompose_in_place a in
  lu_decompose_split a pivots

let lu_backsubstitute a pivots b =
  let n = Array.length a in
  let nonzero = ref (−1) in
  let b = Array.copy b in
  for i = 0 to pred n do
    let ll = pivots.(i) in
    let b_i = ref (b.(ll)) in
    b.(ll) ← b.(i);
    if !nonzero ≥ 0 then
      for j = !nonzero to pred i do
        b_i := !b_i −. a.(i).(j) *. b.(j)
      done
    else if !b_i ≠ 0.0 then
      nonzero := i;
    b.(i) ← !b_i
  done;
  for i = pred n downto 0 do
    let b_i = ref (b.(i)) in
    for j = succ i to pred n do
      b_i := !b_i -. a.(i).(j) *. b.(j)
    done;
    b.(i) ← !b_i /. a.(i).(i)
  done;
  b

let solve_destructive a b =
  let pivot, _ = lu_decompose_in_place a in
  lu_backsubstitute a pivot b

let solve_many_destructive a bs =
  let pivot, _ = lu_decompose_in_place a in
  List.map (lu_backsubstitute a pivot) bs

let solve a b =
  solve_destructive (copy_matrix a) b

let solve_many a bs =
  solve_many_destructive (copy_matrix a) bs
```

—Z—

Partial Maps

Z.1 Interface of Partial

Partial maps that are constructed from assoc lists.

```
module type T =
  sig
```

The domain of the map. It needs to be compatible with *Map.OrderedType.t*

```
    type domain
```

The codomain α can be anything we want.

```
    type α t
```

A list of argument-value pairs is mapped to a partial map. If an argument appears twice, the later value takes precedence.

```
    val of_list : (domain × α) list → α t
```

Two lists of arguments and values (both must have the same length) are mapped to a partial map. Again the later value takes precedence.

```
    val of_lists : domain list → α list → α t
```

If domain and codomain disagree, we must raise an exception or provide a fallback.

```
    exception Undefined of domain
    val apply : α t → domain → α
    val apply_opt : α t → domain → α option
    val apply_with_fallback : (domain → α) → α t → domain → α
```

Iff domain and codomain of the map agree, we can fall back to the identity map.

```
    val auto : domain t → domain → domain
  end
```

```
module Make : functor (D : Map.OrderedType) → T with type domain = D.t
module Test : sig val suite : OUnit.test end
```

Z.2 Implementation of Partial

```
module type T =
  sig
    type domain
    type α t
    val of_list : (domain × α) list → α t
    val of_lists : domain list → α list → α t
    exception Undefined of domain
    val apply : α t → domain → α
    val apply_opt : α t → domain → α option
    val apply_with_fallback : (domain → α) → α t → domain → α
    val auto : domain t → domain → domain
  end

module Make (D : Map.OrderedType) : T with type domain = D.t =
  struct

    module M = Map.Make (D)

    type domain = D.t
    type α t = α M.t

    let of_list l =
      List.fold_left (fun m (d, v) → M.add d v m) M.empty l

    let of_lists domain values =
      of_list
        (try
           List.map2 (fun d v → (d, v)) domain values
         with
         | Invalid_argument _ (* "List.map2" *) →
             invalid_arg "Partial.of_lists:␣length␣mismatch")

    let auto partial d =
      try
        M.find d partial
      with
      | Not_found → d

    exception Undefined of domain

    let apply partial d =
      try
        M.find d partial
      with
      | Not_found → raise (Undefined d)

    let apply_opt partial d =
      try
        Some (M.find d partial)
      with
      | Not_found → None

    let apply_with_fallback fallback partial d =
      try
        M.find d partial
      with
      | Not_found → fallback d

  end
```

Z.2.1 Unit Tests

```
module Test : sig val suite : OUnit.test end =
  struct

    open OUnit

    module P = Make (struct type t = int let compare = compare end)

    let apply_ok =
      "apply/ok" >::
        (fun () →
          let p = P.of_list [ (0,"a"); (1,"b"); (2,"c") ]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "b"; "c" ] (List.map (P.apply p) l))

    let apply_ok2 =
      "apply/ok2" >::
        (fun () →
          let p = P.of_lists [0; 1; 2] ["a"; "b"; "c"]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "b"; "c" ] (List.map (P.apply p) l))

    let apply_shadowed =
      "apply/shadowed" >::
        (fun () →
          let p = P.of_list [ (0,"a"); (1,"b"); (2,"c"); (1,"d") ]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "d"; "c" ] (List.map (P.apply p) l))

    let apply_shadowed2 =
      "apply/shadowed2" >::
        (fun () →
          let p = P.of_lists [0; 1; 2; 1] ["a"; "b"; "c"; "d"]
          and l = [ 0; 1; 2 ] in
          assert_equal [ "a"; "d"; "c" ] (List.map (P.apply p) l))

    let apply_mismatch =
      "apply/mismatch" >::
        (fun () →
          assert_raises
            (Invalid_argument "Partial.of_lists:␣length␣mismatch")
            (fun () → P.of_lists [0; 1; 2] ["a"; "b"; "c"; "d"]))

    let suite_apply =
      "apply" >:::
        [apply_ok;
         apply_ok2;
         apply_shadowed;
         apply_shadowed2;
         apply_mismatch]

    let auto_ok =
      "auto/ok" >::
        (fun () →
          let p = P.of_list [ (0, 10); (1, 11)]
          and l = [ 0; 1; 2 ] in
          assert_equal [ 10; 11; 2 ] (List.map (P.auto p) l))

    let suite_auto =
      "auto" >:::
        [auto_ok]

    let apply_with_fallback_ok =
      "apply_with_fallback/ok" >::
        (fun () →
          let p = P.of_list [ (0, 10); (1, 11)]
          and l = [ 0; 1; 2 ] in
          assert_equal
            [ 10; 11; − 2 ] (List.map (P.apply_with_fallback (fun n → − n) p) l))

    let suite_apply_with_fallback =
      "apply_with_fallback" >:::
        [apply_with_fallback_ok]

    let suite =
      "Partial" >:::
        [suite_apply;
         suite_auto;
         suite_apply_with_fallback]

    let time () =
      ()

  end
```

—AA—
Talk To The WHiZard . . .

Talk to [11].

Temporarily disabled, until, we implement some conditional weaving. . .

—AB—
Fortran Libraries

AB.1 Trivia

⟨omega_spinors.f90⟩≡
```
  ⟨Copyleft⟩
  module omega_spinors
  use kinds
  use constants
  implicit none
  private
  public :: operator (*), operator (+), operator (-)
  public :: abs, set_zero
  ⟨intrinsic :: abs⟩
  type, public :: conjspinor
  ! private (omegalib needs access, but DON'T TOUCH IT!)
  complex(kind=default), dimension(4) :: a
  end type conjspinor
  type, public :: spinor
  ! private (omegalib needs access, but DON'T TOUCH IT!)
  complex(kind=default), dimension(4) :: a
  end type spinor
  ⟨Declaration of operations for spinors⟩
  integer, parameter, public :: omega_spinors_2010_01_A = 0
  contains
  ⟨Implementation of operations for spinors⟩
  end module omega_spinors
```

⟨intrinsic :: abs *(if working)*⟩≡
```
  intrinsic :: abs
```

⟨intrinsic :: conjg *(if working)*⟩≡
```
  intrinsic :: conjg
```

well, the Intel Fortran Compiler chokes on these with an internal error:

⟨intrinsic :: abs⟩≡

⟨intrinsic :: conjg⟩≡

To reenable the pure functions that have been removed for OpenMP, one should set this chunk to `pure &`

⟨pure *unless OpenMP*⟩≡

AB.1.1 Inner Product

⟨*Declaration of operations for spinors*⟩≡
```
  interface operator (*)
  module procedure conjspinor_spinor
  end interface
  private :: conjspinor_spinor
```

$$\bar\psi\psi' \tag{AB.1}$$

NB: `dot_product` conjugates its first argument, we can either cancel this or inline `dot_product`:

⟨*Implementation of operations for spinors*⟩≡
```
  pure function conjspinor_spinor (psibar, psi) result (psibarpsi)
```

```
complex(kind=default) :: psibarpsi
type(conjspinor), intent(in) :: psibar
type(spinor), intent(in) :: psi
psibarpsi = psibar%a(1)*psi%a(1) + psibar%a(2)*psi%a(2) &
+ psibar%a(3)*psi%a(3) + psibar%a(4)*psi%a(4)
end function conjspinor_spinor
```

AB.1.2 Spinor Vector Space

⟨*Declaration of operations for spinors*⟩+≡
```
interface set_zero
module procedure set_zero_spinor, set_zero_conjspinor
end interface
private :: set_zero_spinor, set_zero_conjspinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
elemental subroutine set_zero_spinor (x)
type(spinor), intent(out) :: x
x%a = 0
end subroutine set_zero_spinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
elemental subroutine set_zero_conjspinor (x)
type(conjspinor), intent(out) :: x
x%a = 0
end subroutine set_zero_conjspinor
```

Scalar Multiplication

⟨*Declaration of operations for spinors*⟩+≡
```
interface operator (*)
module procedure integer_spinor, spinor_integer, &
real_spinor, double_spinor, &
complex_spinor, dcomplex_spinor, &
spinor_real, spinor_double, &
spinor_complex, spinor_dcomplex
end interface
private :: integer_spinor, spinor_integer, real_spinor, &
double_spinor, complex_spinor, dcomplex_spinor, &
spinor_real, spinor_double, spinor_complex, spinor_dcomplex
```

⟨*Implementation of operations for spinors*⟩+≡
```
pure function integer_spinor (x, y) result (xy)
integer, intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function integer_spinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
pure function real_spinor (x, y) result (xy)
real(kind=single), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function real_spinor
pure function double_spinor (x, y) result (xy)
real(kind=default), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function double_spinor
pure function complex_spinor (x, y) result (xy)
complex(kind=single), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
```

```
end function complex_spinor
pure function dcomplex_spinor (x, y) result (xy)
complex(kind=default), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function dcomplex_spinor
pure function spinor_integer (y, x) result (xy)
integer, intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function spinor_integer
pure function spinor_real (y, x) result (xy)
real(kind=single), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function spinor_real
pure function spinor_double (y, x) result (xy)
real(kind=default), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function spinor_double
pure function spinor_complex (y, x) result (xy)
complex(kind=single), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function spinor_complex
pure function spinor_dcomplex (y, x) result (xy)
complex(kind=default), intent(in) :: x
type(spinor), intent(in) :: y
type(spinor) :: xy
xy%a = x * y%a
end function spinor_dcomplex
```

⟨*Declaration of operations for spinors*⟩+≡

```
interface operator (*)
module procedure integer_conjspinor, conjspinor_integer, &
real_conjspinor, double_conjspinor, &
complex_conjspinor, dcomplex_conjspinor, &
conjspinor_real, conjspinor_double, &
conjspinor_complex, conjspinor_dcomplex
end interface
private :: integer_conjspinor, conjspinor_integer, real_conjspinor, &
double_conjspinor, complex_conjspinor, dcomplex_conjspinor, &
conjspinor_real, conjspinor_double, conjspinor_complex, &
conjspinor_dcomplex
```

⟨*Implementation of operations for spinors*⟩+≡

```
pure function integer_conjspinor (x, y) result (xy)
integer, intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function integer_conjspinor
pure function real_conjspinor (x, y) result (xy)
real(kind=single), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function real_conjspinor
pure function double_conjspinor (x, y) result (xy)
real(kind=default), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function double_conjspinor
pure function complex_conjspinor (x, y) result (xy)
complex(kind=single), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function complex_conjspinor
pure function dcomplex_conjspinor (x, y) result (xy)
complex(kind=default), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function dcomplex_conjspinor
pure function conjspinor_integer (y, x) result (xy)
integer, intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function conjspinor_integer
pure function conjspinor_real (y, x) result (xy)
real(kind=single), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function conjspinor_real
pure function conjspinor_double (y, x) result (xy)
real(kind=default), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function conjspinor_double
pure function conjspinor_complex (y, x) result (xy)
complex(kind=single), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function conjspinor_complex
pure function conjspinor_dcomplex (y, x) result (xy)
complex(kind=default), intent(in) :: x
type(conjspinor), intent(in) :: y
type(conjspinor) :: xy
xy%a = x * y%a
end function conjspinor_dcomplex
```

Unary Plus and Minus

⟨*Declaration of operations for spinors*⟩+≡
```
interface operator (+)
module procedure plus_spinor, plus_conjspinor
end interface
private :: plus_spinor, plus_conjspinor
interface operator (-)
module procedure neg_spinor, neg_conjspinor
end interface
private :: neg_spinor, neg_conjspinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
pure function plus_spinor (x) result (plus_x)
type(spinor), intent(in) :: x
type(spinor) :: plus_x
plus_x%a = x%a
end function plus_spinor
pure function neg_spinor (x) result (neg_x)
type(spinor), intent(in) :: x
```

```
  type(spinor) :: neg_x
  neg_x%a = - x%a
  end function neg_spinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
  pure function plus_conjspinor (x) result (plus_x)
  type(conjspinor), intent(in) :: x
  type(conjspinor) :: plus_x
  plus_x%a = x%a
  end function plus_conjspinor
  pure function neg_conjspinor (x) result (neg_x)
  type(conjspinor), intent(in) :: x
  type(conjspinor) :: neg_x
  neg_x%a = - x%a
  end function neg_conjspinor
```

Addition and Subtraction

⟨*Declaration of operations for spinors*⟩+≡
```
  interface operator (+)
  module procedure add_spinor, add_conjspinor
  end interface
  private :: add_spinor, add_conjspinor
  interface operator (-)
  module procedure sub_spinor, sub_conjspinor
  end interface
  private :: sub_spinor, sub_conjspinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
  pure function add_spinor (x, y) result (xy)
  type(spinor), intent(in) :: x, y
  type(spinor) :: xy
  xy%a = x%a + y%a
  end function add_spinor
  pure function sub_spinor (x, y) result (xy)
  type(spinor), intent(in) :: x, y
  type(spinor) :: xy
  xy%a = x%a - y%a
  end function sub_spinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
  pure function add_conjspinor (x, y) result (xy)
  type(conjspinor), intent(in) :: x, y
  type(conjspinor) :: xy
  xy%a = x%a + y%a
  end function add_conjspinor
  pure function sub_conjspinor (x, y) result (xy)
  type(conjspinor), intent(in) :: x, y
  type(conjspinor) :: xy
  xy%a = x%a - y%a
  end function sub_conjspinor
```

AB.1.3 Norm

⟨*Declaration of operations for spinors*⟩+≡
```
  interface abs
  module procedure abs_spinor, abs_conjspinor
  end interface
  private :: abs_spinor, abs_conjspinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
  pure function abs_spinor (psi) result (x)
  type(spinor), intent(in) :: psi
  real(kind=default) :: x
  x = sqrt (real (dot_product (psi%a, psi%a)))
  end function abs_spinor
```

⟨*Implementation of operations for spinors*⟩+≡
```
pure function abs_conjspinor (psibar) result (x)
real(kind=default) :: x
type(conjspinor), intent(in) :: psibar
x = sqrt (real (dot_product (psibar%a, psibar%a)))
end function abs_conjspinor
```

AB.2 Spinors Revisited

⟨`omega_bispinors.f90`⟩≡
```
⟨Copyleft⟩
module omega_bispinors
use kinds
use constants
implicit none
private
public :: operator (*), operator (+), operator (-)
public :: abs, set_zero
type, public :: bispinor
! private (omegalib needs access, but DON'T TOUCH IT!)
complex(kind=default), dimension(4) :: a
end type bispinor
⟨Declaration of operations for bispinors⟩
integer, parameter, public :: omega_bispinors_2010_01_A = 0
contains
⟨Implementation of operations for bispinors⟩
end module omega_bispinors
```

⟨*Declaration of operations for bispinors*⟩≡
```
interface operator (*)
module procedure spinor_product
end interface
private :: spinor_product
```

$$\bar\psi\psi' \tag{AB.2}$$

NB: `dot_product` conjugates its first argument, we have to cancel this.

⟨*Implementation of operations for bispinors*⟩≡
```
pure function spinor_product (psil, psir) result (psilpsir)
complex(kind=default) :: psilpsir
type(bispinor), intent(in) :: psil, psir
type(bispinor) :: psidum
psidum%a(1) = psir%a(2)
psidum%a(2) = - psir%a(1)
psidum%a(3) = - psir%a(4)
psidum%a(4) = psir%a(3)
psilpsir = dot_product (conjg (psil%a), psidum%a)
end function spinor_product
```

AB.2.1 Spinor Vector Space

⟨*Declaration of operations for bispinors*⟩+≡
```
interface set_zero
module procedure set_zero_bispinor
end interface
private :: set_zero_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
elemental subroutine set_zero_bispinor (x)
type(bispinor), intent(out) :: x
x%a = 0
end subroutine set_zero_bispinor
```

Scalar Multiplication

⟨*Declaration of operations for bispinors*⟩+≡
```
interface operator (*)
module procedure integer_bispinor, bispinor_integer, &
real_bispinor, double_bispinor, &
complex_bispinor, dcomplex_bispinor, &
bispinor_real, bispinor_double, &
bispinor_complex, bispinor_dcomplex
end interface
private :: integer_bispinor, bispinor_integer, real_bispinor, &
double_bispinor, complex_bispinor, dcomplex_bispinor, &
bispinor_real, bispinor_double, bispinor_complex, bispinor_dcomplex
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function integer_bispinor (x, y) result (xy)
type(bispinor) :: xy
integer, intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function integer_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function real_bispinor (x, y) result (xy)
type(bispinor) :: xy
real(kind=single), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function real_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function double_bispinor (x, y) result (xy)
type(bispinor) :: xy
real(kind=default), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function double_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function complex_bispinor (x, y) result (xy)
type(bispinor) :: xy
complex(kind=single), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function complex_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function dcomplex_bispinor (x, y) result (xy)
type(bispinor) :: xy
complex(kind=default), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function dcomplex_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function bispinor_integer (y, x) result (xy)
type(bispinor) :: xy
integer, intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function bispinor_integer
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function bispinor_real (y, x) result (xy)
type(bispinor) :: xy
real(kind=single), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function bispinor_real
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function bispinor_double (y, x) result (xy)
type(bispinor) :: xy
real(kind=default), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function bispinor_double
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function bispinor_complex (y, x) result (xy)
type(bispinor) :: xy
complex(kind=single), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function bispinor_complex
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function bispinor_dcomplex (y, x) result (xy)
type(bispinor) :: xy
complex(kind=default), intent(in) :: x
type(bispinor), intent(in) :: y
xy%a = x * y%a
end function bispinor_dcomplex
```

Unary Plus and Minus

⟨*Declaration of operations for bispinors*⟩+≡
```
interface operator (+)
module procedure plus_bispinor
end interface
private :: plus_bispinor
interface operator (-)
module procedure neg_bispinor
end interface
private :: neg_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function plus_bispinor (x) result (plus_x)
type(bispinor) :: plus_x
type(bispinor), intent(in) :: x
plus_x%a = x%a
end function plus_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function neg_bispinor (x) result (neg_x)
type(bispinor) :: neg_x
type(bispinor), intent(in) :: x
neg_x%a = - x%a
end function neg_bispinor
```

Addition and Subtraction

⟨*Declaration of operations for bispinors*⟩+≡
```
interface operator (+)
module procedure add_bispinor
end interface
private :: add_bispinor
interface operator (-)
module procedure sub_bispinor
end interface
private :: sub_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function add_bispinor (x, y) result (xy)
type(bispinor) :: xy
type(bispinor), intent(in) :: x, y
xy%a = x%a + y%a
end function add_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function sub_bispinor (x, y) result (xy)
type(bispinor) :: xy
type(bispinor), intent(in) :: x, y
xy%a = x%a - y%a
end function sub_bispinor
```

AB.2.2 Norm

⟨*Declaration of operations for bispinors*⟩+≡
```
interface abs
module procedure abs_bispinor
end interface
private :: abs_bispinor
```

⟨*Implementation of operations for bispinors*⟩+≡
```
pure function abs_bispinor (psi) result (x)
real(kind=default) :: x
type(bispinor), intent(in) :: psi
x = sqrt (real (dot_product (psi%a, psi%a)))
end function abs_bispinor
```

AB.3 Vectorspinors

⟨omega_vectorspinors.f90⟩≡
```
⟨Copyleft⟩
module omega_vectorspinors
use kinds
use constants
use omega_bispinors
use omega_vectors
implicit none
private
public :: operator (*), operator (+), operator (-)
public :: abs, set_zero
type, public :: vectorspinor
! private (omegalib needs access, but DON'T TOUCH IT!)
type(bispinor), dimension(4) :: psi
end type vectorspinor
⟨Declaration of operations for vectorspinors⟩
integer, parameter, public :: omega_vectorspinors_2010_01_A = 0
contains
⟨Implementation of operations for vectorspinors⟩
end module omega_vectorspinors
```

⟨*Declaration of operations for vectorspinors*⟩≡
```
interface operator (*)
module procedure vspinor_product
end interface
private :: vspinor_product
```

$$\bar\psi^\mu\psi'_\mu \tag{AB.3}$$

⟨*Implementation of operations for vectorspinors*⟩≡
```
pure function vspinor_product (psil, psir) result (psilpsir)
complex(kind=default) :: psilpsir
type(vectorspinor), intent(in) :: psil, psir
psilpsir = psil%psi(1) * psir%psi(1) &
- psil%psi(2) * psir%psi(2) &
- psil%psi(3) * psir%psi(3) &
- psil%psi(4) * psir%psi(4)
end function vspinor_product
```

AB.3.1 Vectorspinor Vector Space

⟨*Declaration of operations for vectorspinors*⟩+≡

```
interface set_zero
module procedure set_zero_vectorspinor
end interface
private :: set_zero_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
elemental subroutine set_zero_vectorspinor (x)
type(vectorspinor), intent(out) :: x
call set_zero (x%psi)
end subroutine set_zero_vectorspinor
```

Scalar Multiplication

⟨*Declaration of operations for vectorspinors*⟩+≡
```
interface operator (*)
module procedure integer_vectorspinor, vectorspinor_integer, &
real_vectorspinor, double_vectorspinor, &
complex_vectorspinor, dcomplex_vectorspinor, &
vectorspinor_real, vectorspinor_double, &
vectorspinor_complex, vectorspinor_dcomplex, &
momentum_vectorspinor, vectorspinor_momentum
end interface
private :: integer_vectorspinor, vectorspinor_integer, real_vectorspinor, &
double_vectorspinor, complex_vectorspinor, dcomplex_vectorspinor, &
vectorspinor_real, vectorspinor_double, vectorspinor_complex, &
vectorspinor_dcomplex
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function integer_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
integer, intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = x * y%psi(k)
end do
end function integer_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function real_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
real(kind=single), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = x * y%psi(k)
end do
end function real_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function double_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
real(kind=default), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = x * y%psi(k)
end do
end function double_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function complex_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
complex(kind=single), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = x * y%psi(k)
```

```
end do
end function complex_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function dcomplex_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
complex(kind=default), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = x * y%psi(k)
end do
end function dcomplex_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function vectorspinor_integer (y, x) result (xy)
type(vectorspinor) :: xy
integer, intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = y%psi(k) * x
end do
end function vectorspinor_integer
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function vectorspinor_real (y, x) result (xy)
type(vectorspinor) :: xy
real(kind=single), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = y%psi(k) * x
end do
end function vectorspinor_real
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function vectorspinor_double (y, x) result (xy)
type(vectorspinor) :: xy
real(kind=default), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = y%psi(k) * x
end do
end function vectorspinor_double
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function vectorspinor_complex (y, x) result (xy)
type(vectorspinor) :: xy
complex(kind=single), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = y%psi(k) * x
end do
end function vectorspinor_complex
```

⟨*Implementation of operations for vectorspinors*⟩+≡

```
pure function vectorspinor_dcomplex (y, x) result (xy)
type(vectorspinor) :: xy
complex(kind=default), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%psi(k) = y%psi(k) * x
end do
end function vectorspinor_dcomplex
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function momentum_vectorspinor (y, x) result (xy)
type(bispinor) :: xy
type(momentum), intent(in) :: y
type(vectorspinor), intent(in) :: x
integer :: k
do k = 1,4
xy%a(k) = y%t    * x%psi(1)%a(k) - y%x(1) * x%psi(2)%a(k) - &
y%x(2) * x%psi(3)%a(k) - y%x(3) * x%psi(4)%a(k)
end do
end function momentum_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function vectorspinor_momentum (y, x) result (xy)
type(bispinor) :: xy
type(momentum), intent(in) :: x
type(vectorspinor), intent(in) :: y
integer :: k
do k = 1,4
xy%a(k) = x%t    * y%psi(1)%a(k) - x%x(1) * y%psi(2)%a(k) - &
x%x(2) * y%psi(3)%a(k) - x%x(3) * y%psi(4)%a(k)
end do
end function vectorspinor_momentum
```

Unary Plus and Minus

⟨*Declaration of operations for vectorspinors*⟩+≡
```
interface operator (+)
module procedure plus_vectorspinor
end interface
private :: plus_vectorspinor
interface operator (-)
module procedure neg_vectorspinor
end interface
private :: neg_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function plus_vectorspinor (x) result (plus_x)
type(vectorspinor) :: plus_x
type(vectorspinor), intent(in) :: x
integer :: k
do k = 1,4
plus_x%psi(k) = + x%psi(k)
end do
end function plus_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function neg_vectorspinor (x) result (neg_x)
type(vectorspinor) :: neg_x
type(vectorspinor), intent(in) :: x
integer :: k
do k = 1,4
neg_x%psi(k) = - x%psi(k)
end do
end function neg_vectorspinor
```

Addition and Subtraction

⟨*Declaration of operations for vectorspinors*⟩+≡
```
interface operator (+)
module procedure add_vectorspinor
end interface
private :: add_vectorspinor
interface operator (-)
module procedure sub_vectorspinor
end interface
private :: sub_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function add_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
type(vectorspinor), intent(in) :: x, y
integer :: k
do k = 1,4
xy%psi(k) = x%psi(k) + y%psi(k)
end do
end function add_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function sub_vectorspinor (x, y) result (xy)
type(vectorspinor) :: xy
type(vectorspinor), intent(in) :: x, y
integer :: k
do k = 1,4
xy%psi(k) = x%psi(k) - y%psi(k)
end do
end function sub_vectorspinor
```

AB.3.2 Norm

⟨*Declaration of operations for vectorspinors*⟩+≡
```
interface abs
module procedure abs_vectorspinor
end interface
private :: abs_vectorspinor
```

⟨*Implementation of operations for vectorspinors*⟩+≡
```
pure function abs_vectorspinor (psi) result (x)
real(kind=default) :: x
type(vectorspinor), intent(in) :: psi
x = sqrt (real (dot_product (psi%psi(1)%a, psi%psi(1)%a) &
- dot_product (psi%psi(2)%a, psi%psi(2)%a)   &
- dot_product (psi%psi(3)%a, psi%psi(3)%a)   &
- dot_product (psi%psi(4)%a, psi%psi(4)%a)))
end function abs_vectorspinor
```

AB.4 Vectors and Tensors

Condensed representation of antisymmetric rank-2 tensors:

$$\begin{pmatrix} T^{00} & T^{01} & T^{02} & T^{03} \\ T^{10} & T^{11} & T^{12} & T^{13} \\ T^{20} & T^{21} & T^{22} & T^{23} \\ T^{30} & T^{31} & T^{32} & T^{33} \end{pmatrix} = \begin{pmatrix} 0 & T_e^1 & T_e^2 & T_e^3 \\ -T_e^1 & 0 & T_b^3 & -T_b^2 \\ -T_e^2 & -T_b^3 & 0 & T_b^1 \\ -T_e^3 & T_b^2 & -T_b^1 & 0 \end{pmatrix} \tag{AB.4}$$

⟨`omega_vectors.f90`⟩≡
```
⟨Copyleft⟩
module omega_vectors
use kinds
use constants
implicit none
private
public :: assignment (=), operator(==)
public :: operator (*), operator (+), operator (-), operator (.wedge.)
public :: abs, conjg, set_zero
public :: random_momentum
⟨intrinsic :: abs⟩
⟨intrinsic :: conjg⟩
type, public :: momentum
! private (omegalib needs access, but DON'T TOUCH IT!)
real(kind=default) :: t
real(kind=default), dimension(3) :: x
end type momentum
```

```
type, public :: vector
! private (omegalib needs access, but DON'T TOUCH IT!)
complex(kind=default) :: t
complex(kind=default), dimension(3) :: x
end type vector
type, public :: tensor2odd
! private (omegalib needs access, but DON'T TOUCH IT!)
complex(kind=default), dimension(3) :: e
complex(kind=default), dimension(3) :: b
end type tensor2odd
⟨Declaration of operations for vectors⟩
integer, parameter, public :: omega_vectors_2010_01_A = 0
contains
⟨Implementation of operations for vectors⟩
end module omega_vectors
```

AB.4.1 Constructors

⟨*Declaration of operations for vectors*⟩≡

```
interface assignment (=)
module procedure momentum_of_array, vector_of_momentum, &
vector_of_array, vector_of_double_array, &
array_of_momentum, array_of_vector
end interface
private :: momentum_of_array, vector_of_momentum, vector_of_array, &
vector_of_double_array, array_of_momentum, array_of_vector
```

⟨*Implementation of operations for vectors*⟩≡

```
pure subroutine momentum_of_array (m, p)
type(momentum), intent(out) :: m
real(kind=default), dimension(0:), intent(in) :: p
m%t = p(0)
m%x = p(1:3)
end subroutine momentum_of_array
pure subroutine array_of_momentum (p, v)
real(kind=default), dimension(0:), intent(out) :: p
type(momentum), intent(in) :: v
p(0) = v%t
p(1:3) = v%x
end subroutine array_of_momentum
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure subroutine vector_of_array (v, p)
type(vector), intent(out) :: v
complex(kind=default), dimension(0:), intent(in) :: p
v%t = p(0)
v%x = p(1:3)
end subroutine vector_of_array
pure subroutine vector_of_double_array (v, p)
type(vector), intent(out) :: v
real(kind=default), dimension(0:), intent(in) :: p
v%t = p(0)
v%x = p(1:3)
end subroutine vector_of_double_array
pure subroutine array_of_vector (p, v)
complex(kind=default), dimension(0:), intent(out) :: p
type(vector), intent(in) :: v
p(0) = v%t
p(1:3) = v%x
end subroutine array_of_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure subroutine vector_of_momentum (v, p)
type(vector), intent(out) :: v
type(momentum), intent(in) :: p
v%t = p%t
v%x = p%x
end subroutine vector_of_momentum
```

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator(==)
module procedure momentum_eq
end interface
```

⟨*Implementation of operations for vectors*⟩+≡
```
elemental function momentum_eq (lhs, rhs) result (yorn)
logical :: yorn
type(momentum), intent(in) :: lhs
type(momentum), intent(in) :: rhs
yorn = all (abs(lhs%x - rhs%x) < eps0) .and. abs(lhs%t - rhs%t) < eps0
end function momentum_eq
```

AB.4.2 Inner Products

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator (*)
module procedure momentum_momentum, vector_vector, &
vector_momentum, momentum_vector, tensor2odd_tensor2odd
end interface
private :: momentum_momentum, vector_vector, vector_momentum, &
momentum_vector, tensor2odd_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function momentum_momentum (x, y) result (xy)
type(momentum), intent(in) :: x
type(momentum), intent(in) :: y
real(kind=default) :: xy
xy = x%t*y%t - x%x(1)*y%x(1) - x%x(2)*y%x(2) - x%x(3)*y%x(3)
end function momentum_momentum
pure function momentum_vector (x, y) result (xy)
type(momentum), intent(in) :: x
type(vector), intent(in) :: y
complex(kind=default) :: xy
xy = x%t*y%t - x%x(1)*y%x(1) - x%x(2)*y%x(2) - x%x(3)*y%x(3)
end function momentum_vector
pure function vector_momentum (x, y) result (xy)
type(vector), intent(in) :: x
type(momentum), intent(in) :: y
complex(kind=default) :: xy
xy = x%t*y%t - x%x(1)*y%x(1) - x%x(2)*y%x(2) - x%x(3)*y%x(3)
end function vector_momentum
pure function vector_vector (x, y) result (xy)
type(vector), intent(in) :: x
type(vector), intent(in) :: y
complex(kind=default) :: xy
xy = x%t*y%t - x%x(1)*y%x(1) - x%x(2)*y%x(2) - x%x(3)*y%x(3)
end function vector_vector
```

Just like classical electrodynamics:

$$\frac{1}{2}T_{\mu\nu}U^{\mu\nu} = \frac{1}{2}\left(-T^{0i}U^{0i} - T^{i0}U^{i0} + T^{ij}U^{ij}\right) = T_b^kU_b^k - T_e^kU_e^k \tag{AB.5}$$

⟨*Implementation of operations for vectors*⟩+≡
```
pure function tensor2odd_tensor2odd (x, y) result (xy)
type(tensor2odd), intent(in) :: x
type(tensor2odd), intent(in) :: y
complex(kind=default) :: xy
xy = x%b(1)*y%b(1) + x%b(2)*y%b(2) + x%b(3)*y%b(3) &
- x%e(1)*y%e(1) - x%e(2)*y%e(2) - x%e(3)*y%e(3)
end function tensor2odd_tensor2odd
```

AB.4.3 Not Entirely Inner Products

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator (*)
```

```
module procedure momentum_tensor2odd, tensor2odd_momentum, &
vector_tensor2odd, tensor2odd_vector
end interface
private :: momentum_tensor2odd, tensor2odd_momentum, vector_tensor2odd, &
tensor2odd_vector
```

$$y^\nu = x_\mu T^{\mu\nu} : y^0 = -x^i T^{i0} = x^i T^{0i} \tag{AB.6a}$$
$$y^1 = x^0 T^{01} - x^2 T^{21} - x^3 T^{31} \tag{AB.6b}$$
$$y^2 = x^0 T^{02} - x^1 T^{12} - x^3 T^{32} \tag{AB.6c}$$
$$y^3 = x^0 T^{03} - x^1 T^{13} - x^2 T^{23} \tag{AB.6d}$$

⟨*Implementation of operations for vectors*⟩+≡

```
pure function vector_tensor2odd (x, t2) result (xt2)
type(vector), intent(in) :: x
type(tensor2odd), intent(in) :: t2
type(vector) :: xt2
xt2%t = x%x(1)*t2%e(1) + x%x(2)*t2%e(2) + x%x(3)*t2%e(3)
xt2%x(1) = x%t*t2%e(1) + x%x(2)*t2%b(3) - x%x(3)*t2%b(2)
xt2%x(2) = x%t*t2%e(2) + x%x(3)*t2%b(1) - x%x(1)*t2%b(3)
xt2%x(3) = x%t*t2%e(3) + x%x(1)*t2%b(2) - x%x(2)*t2%b(1)
end function vector_tensor2odd
pure function momentum_tensor2odd (x, t2) result (xt2)
type(momentum), intent(in) :: x
type(tensor2odd), intent(in) :: t2
type(vector) :: xt2
xt2%t = x%x(1)*t2%e(1) + x%x(2)*t2%e(2) + x%x(3)*t2%e(3)
xt2%x(1) = x%t*t2%e(1) + x%x(2)*t2%b(3) - x%x(3)*t2%b(2)
xt2%x(2) = x%t*t2%e(2) + x%x(3)*t2%b(1) - x%x(1)*t2%b(3)
xt2%x(3) = x%t*t2%e(3) + x%x(1)*t2%b(2) - x%x(2)*t2%b(1)
end function momentum_tensor2odd
```

$$y^\mu = T^{\mu\nu} x_\nu : y^0 = -T^{0i} x^i \tag{AB.7a}$$
$$y^1 = T^{10} x^0 - T^{12} x^2 - T^{13} x^3 \tag{AB.7b}$$
$$y^2 = T^{20} x^0 - T^{21} x^1 - T^{23} x^3 \tag{AB.7c}$$
$$y^3 = T^{30} x^0 - T^{31} x^1 - T^{32} x^2 \tag{AB.7d}$$

⟨*Implementation of operations for vectors*⟩+≡

```
pure function tensor2odd_vector (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
type(vector), intent(in) :: x
type(vector) :: t2x
t2x%t = - t2%e(1)*x%x(1) - t2%e(2)*x%x(2) - t2%e(3)*x%x(3)
t2x%x(1) = - t2%e(1)*x%t + t2%b(2)*x%x(3) - t2%b(3)*x%x(2)
t2x%x(2) = - t2%e(2)*x%t + t2%b(3)*x%x(1) - t2%b(1)*x%x(3)
t2x%x(3) = - t2%e(3)*x%t + t2%b(1)*x%x(2) - t2%b(2)*x%x(1)
end function tensor2odd_vector
pure function tensor2odd_momentum (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
type(momentum), intent(in) :: x
type(vector) :: t2x
t2x%t = - t2%e(1)*x%x(1) - t2%e(2)*x%x(2) - t2%e(3)*x%x(3)
t2x%x(1) = - t2%e(1)*x%t + t2%b(2)*x%x(3) - t2%b(3)*x%x(2)
t2x%x(2) = - t2%e(2)*x%t + t2%b(3)*x%x(1) - t2%b(1)*x%x(3)
t2x%x(3) = - t2%e(3)*x%t + t2%b(1)*x%x(2) - t2%b(2)*x%x(1)
end function tensor2odd_momentum
```

AB.4.4 Outer Products

⟨*Declaration of operations for vectors*⟩+≡

```
interface operator (.wedge.)
module procedure momentum_wedge_momentum, &
```

```
momentum_wedge_vector, vector_wedge_momentum, vector_wedge_vector
end interface
private :: momentum_wedge_momentum, momentum_wedge_vector, &
vector_wedge_momentum, vector_wedge_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function momentum_wedge_momentum (x, y) result (t2)
type(momentum), intent(in) :: x
type(momentum), intent(in) :: y
type(tensor2odd) :: t2
t2%e = x%t * y%x - x%x * y%t
t2%b(1) = x%x(2) * y%x(3) - x%x(3) * y%x(2)
t2%b(2) = x%x(3) * y%x(1) - x%x(1) * y%x(3)
t2%b(3) = x%x(1) * y%x(2) - x%x(2) * y%x(1)
end function momentum_wedge_momentum
pure function momentum_wedge_vector (x, y) result (t2)
type(momentum), intent(in) :: x
type(vector), intent(in) :: y
type(tensor2odd) :: t2
t2%e = x%t * y%x - x%x * y%t
t2%b(1) = x%x(2) * y%x(3) - x%x(3) * y%x(2)
t2%b(2) = x%x(3) * y%x(1) - x%x(1) * y%x(3)
t2%b(3) = x%x(1) * y%x(2) - x%x(2) * y%x(1)
end function momentum_wedge_vector
pure function vector_wedge_momentum (x, y) result (t2)
type(vector), intent(in) :: x
type(momentum), intent(in) :: y
type(tensor2odd) :: t2
t2%e = x%t * y%x - x%x * y%t
t2%b(1) = x%x(2) * y%x(3) - x%x(3) * y%x(2)
t2%b(2) = x%x(3) * y%x(1) - x%x(1) * y%x(3)
t2%b(3) = x%x(1) * y%x(2) - x%x(2) * y%x(1)
end function vector_wedge_momentum
pure function vector_wedge_vector (x, y) result (t2)
type(vector), intent(in) :: x
type(vector), intent(in) :: y
type(tensor2odd) :: t2
t2%e = x%t * y%x - x%x * y%t
t2%b(1) = x%x(2) * y%x(3) - x%x(3) * y%x(2)
t2%b(2) = x%x(3) * y%x(1) - x%x(1) * y%x(3)
t2%b(3) = x%x(1) * y%x(2) - x%x(2) * y%x(1)
end function vector_wedge_vector
```

AB.4.5 Vector Space

⟨*Declaration of operations for vectors*⟩+≡

```
interface set_zero
module procedure set_zero_vector, set_zero_momentum, &
set_zero_tensor2odd, set_zero_real, set_zero_complex
end interface
private :: set_zero_vector, set_zero_momentum, set_zero_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡

```
elemental subroutine set_zero_vector (x)
type(vector), intent(out) :: x
x%t = 0
x%x = 0
end subroutine set_zero_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
elemental subroutine set_zero_momentum (x)
type(momentum), intent(out) :: x
x%t = 0
x%x = 0
end subroutine set_zero_momentum
```

⟨*Implementation of operations for vectors*⟩+≡

```
elemental subroutine set_zero_tensor2odd (x)
```

```
type(tensor2odd), intent(out) :: x
x%e = 0
x%b = 0
end subroutine set_zero_tensor2odd
```

Doesn't really belong here, but there is no better place ...

⟨*Implementation of operations for vectors*⟩+≡
```
elemental subroutine set_zero_real (x)
real(kind=default), intent(out) :: x
x = 0
end subroutine set_zero_real
```

⟨*Implementation of operations for vectors*⟩+≡
```
elemental subroutine set_zero_complex (x)
complex(kind=default), intent(out) :: x
x = 0
end subroutine set_zero_complex
```

Scalar Multiplication

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator (*)
module procedure integer_momentum, real_momentum, double_momentum, &
complex_momentum, dcomplex_momentum, &
integer_vector, real_vector, double_vector, &
complex_vector, dcomplex_vector, &
integer_tensor2odd, real_tensor2odd, double_tensor2odd, &
complex_tensor2odd, dcomplex_tensor2odd, &
momentum_integer, momentum_real, momentum_double, &
momentum_complex, momentum_dcomplex, &
vector_integer, vector_real, vector_double, &
vector_complex, vector_dcomplex, &
tensor2odd_integer, tensor2odd_real, tensor2odd_double, &
tensor2odd_complex, tensor2odd_dcomplex
end interface
private :: integer_momentum, real_momentum, double_momentum, &
complex_momentum, dcomplex_momentum, integer_vector, real_vector, &
double_vector, complex_vector, dcomplex_vector, &
integer_tensor2odd, real_tensor2odd, double_tensor2odd, &
complex_tensor2odd, dcomplex_tensor2odd, momentum_integer, &
momentum_real, momentum_double, momentum_complex, &
momentum_dcomplex, vector_integer, vector_real, vector_double, &
vector_complex, vector_dcomplex, tensor2odd_integer, &
tensor2odd_real, tensor2odd_double, tensor2odd_complex, &
tensor2odd_dcomplex
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function integer_momentum (x, y) result (xy)
integer, intent(in) :: x
type(momentum), intent(in) :: y
type(momentum) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function integer_momentum
pure function real_momentum (x, y) result (xy)
real(kind=single), intent(in) :: x
type(momentum), intent(in) :: y
type(momentum) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function real_momentum
pure function double_momentum (x, y) result (xy)
real(kind=default), intent(in) :: x
type(momentum), intent(in) :: y
type(momentum) :: xy
xy%t = x * y%t
xy%x = x * y%x
```

```
end function double_momentum
pure function complex_momentum (x, y) result (xy)
complex(kind=single), intent(in) :: x
type(momentum), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function complex_momentum
pure function dcomplex_momentum (x, y) result (xy)
complex(kind=default), intent(in) :: x
type(momentum), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function dcomplex_momentum
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function integer_vector (x, y) result (xy)
integer, intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function integer_vector
pure function real_vector (x, y) result (xy)
real(kind=single), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function real_vector
pure function double_vector (x, y) result (xy)
real(kind=default), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function double_vector
pure function complex_vector (x, y) result (xy)
complex(kind=single), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function complex_vector
pure function dcomplex_vector (x, y) result (xy)
complex(kind=default), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function dcomplex_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function integer_tensor2odd (x, t2) result (xt2)
integer, intent(in) :: x
type(tensor2odd), intent(in) :: t2
type(tensor2odd) :: xt2
xt2%e = x * t2%e
xt2%b = x * t2%b
end function integer_tensor2odd
pure function real_tensor2odd (x, t2) result (xt2)
real(kind=single), intent(in) :: x
type(tensor2odd), intent(in) :: t2
type(tensor2odd) :: xt2
xt2%e = x * t2%e
xt2%b = x * t2%b
```

```
  end function real_tensor2odd
  pure function double_tensor2odd (x, t2) result (xt2)
  real(kind=default), intent(in) :: x
  type(tensor2odd), intent(in) :: t2
  type(tensor2odd) :: xt2
  xt2%e = x * t2%e
  xt2%b = x * t2%b
  end function double_tensor2odd
  pure function complex_tensor2odd (x, t2) result (xt2)
  complex(kind=single), intent(in) :: x
  type(tensor2odd), intent(in) :: t2
  type(tensor2odd) :: xt2
  xt2%e = x * t2%e
  xt2%b = x * t2%b
  end function complex_tensor2odd
  pure function dcomplex_tensor2odd (x, t2) result (xt2)
  complex(kind=default), intent(in) :: x
  type(tensor2odd), intent(in) :: t2
  type(tensor2odd) :: xt2
  xt2%e = x * t2%e
  xt2%b = x * t2%b
  end function dcomplex_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡

```
  pure function momentum_integer (y, x) result (xy)
  integer, intent(in) :: x
  type(momentum), intent(in) :: y
  type(momentum) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
  end function momentum_integer
  pure function momentum_real (y, x) result (xy)
  real(kind=single), intent(in) :: x
  type(momentum), intent(in) :: y
  type(momentum) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
  end function momentum_real
  pure function momentum_double (y, x) result (xy)
  real(kind=default), intent(in) :: x
  type(momentum), intent(in) :: y
  type(momentum) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
  end function momentum_double
  pure function momentum_complex (y, x) result (xy)
  complex(kind=single), intent(in) :: x
  type(momentum), intent(in) :: y
  type(vector) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
  end function momentum_complex
  pure function momentum_dcomplex (y, x) result (xy)
  complex(kind=default), intent(in) :: x
  type(momentum), intent(in) :: y
  type(vector) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
  end function momentum_dcomplex
```

⟨*Implementation of operations for vectors*⟩+≡

```
  pure function vector_integer (y, x) result (xy)
  integer, intent(in) :: x
  type(vector), intent(in) :: y
  type(vector) :: xy
  xy%t = x * y%t
  xy%x = x * y%x
```

```
end function vector_integer
pure function vector_real (y, x) result (xy)
real(kind=single), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function vector_real
pure function vector_double (y, x) result (xy)
real(kind=default), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function vector_double
pure function vector_complex (y, x) result (xy)
complex(kind=single), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function vector_complex
pure function vector_dcomplex (y, x) result (xy)
complex(kind=default), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x * y%t
xy%x = x * y%x
end function vector_dcomplex
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function tensor2odd_integer (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
integer, intent(in) :: x
type(tensor2odd) :: t2x
t2x%e = x * t2%e
t2x%b = x * t2%b
end function tensor2odd_integer
pure function tensor2odd_real (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
real(kind=single), intent(in) :: x
type(tensor2odd) :: t2x
t2x%e = x * t2%e
t2x%b = x * t2%b
end function tensor2odd_real
pure function tensor2odd_double (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
real(kind=default), intent(in) :: x
type(tensor2odd) :: t2x
t2x%e = x * t2%e
t2x%b = x * t2%b
end function tensor2odd_double
pure function tensor2odd_complex (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
complex(kind=single), intent(in) :: x
type(tensor2odd) :: t2x
t2x%e = x * t2%e
t2x%b = x * t2%b
end function tensor2odd_complex
pure function tensor2odd_dcomplex (t2, x) result (t2x)
type(tensor2odd), intent(in) :: t2
complex(kind=default), intent(in) :: x
type(tensor2odd) :: t2x
t2x%e = x * t2%e
t2x%b = x * t2%b
end function tensor2odd_dcomplex
```

Unary Plus and Minus

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator (+)
module procedure plus_momentum, plus_vector, plus_tensor2odd
end interface
private :: plus_momentum, plus_vector, plus_tensor2odd
interface operator (-)
module procedure neg_momentum, neg_vector, neg_tensor2odd
end interface
private :: neg_momentum, neg_vector, neg_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function plus_momentum (x) result (plus_x)
type(momentum), intent(in) :: x
type(momentum) :: plus_x
plus_x = x
end function plus_momentum
pure function neg_momentum (x) result (neg_x)
type(momentum), intent(in) :: x
type(momentum) :: neg_x
neg_x%t = - x%t
neg_x%x = - x%x
end function neg_momentum
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function plus_vector (x) result (plus_x)
type(vector), intent(in) :: x
type(vector) :: plus_x
plus_x = x
end function plus_vector
pure function neg_vector (x) result (neg_x)
type(vector), intent(in) :: x
type(vector) :: neg_x
neg_x%t = - x%t
neg_x%x = - x%x
end function neg_vector
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function plus_tensor2odd (x) result (plus_x)
type(tensor2odd), intent(in) :: x
type(tensor2odd) :: plus_x
plus_x = x
end function plus_tensor2odd
pure function neg_tensor2odd (x) result (neg_x)
type(tensor2odd), intent(in) :: x
type(tensor2odd) :: neg_x
neg_x%e = - x%e
neg_x%b = - x%b
end function neg_tensor2odd
```

Addition and Subtraction

⟨*Declaration of operations for vectors*⟩+≡
```
interface operator (+)
module procedure add_momentum, add_vector, &
add_vector_momentum, add_momentum_vector, add_tensor2odd
end interface
private :: add_momentum, add_vector, add_vector_momentum, &
add_momentum_vector, add_tensor2odd
interface operator (-)
module procedure sub_momentum, sub_vector, &
sub_vector_momentum, sub_momentum_vector, sub_tensor2odd
end interface
private :: sub_momentum, sub_vector, sub_vector_momentum, &
sub_momentum_vector, sub_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function add_momentum (x, y) result (xy)
type(momentum), intent(in) :: x, y
type(momentum) :: xy
xy%t = x%t + y%t
xy%x = x%x + y%x
end function add_momentum
pure function add_vector (x, y) result (xy)
type(vector), intent(in) :: x, y
type(vector) :: xy
xy%t = x%t + y%t
xy%x = x%x + y%x
end function add_vector
pure function add_momentum_vector (x, y) result (xy)
type(momentum), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x%t + y%t
xy%x = x%x + y%x
end function add_momentum_vector
pure function add_vector_momentum (x, y) result (xy)
type(vector), intent(in) :: x
type(momentum), intent(in) :: y
type(vector) :: xy
xy%t = x%t + y%t
xy%x = x%x + y%x
end function add_vector_momentum
pure function add_tensor2odd (x, y) result (xy)
type(tensor2odd), intent(in) :: x, y
type(tensor2odd) :: xy
xy%e = x%e + y%e
xy%b = x%b + y%b
end function add_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function sub_momentum (x, y) result (xy)
type(momentum), intent(in) :: x, y
type(momentum) :: xy
xy%t = x%t - y%t
xy%x = x%x - y%x
end function sub_momentum
pure function sub_vector (x, y) result (xy)
type(vector), intent(in) :: x, y
type(vector) :: xy
xy%t = x%t - y%t
xy%x = x%x - y%x
end function sub_vector
pure function sub_momentum_vector (x, y) result (xy)
type(momentum), intent(in) :: x
type(vector), intent(in) :: y
type(vector) :: xy
xy%t = x%t - y%t
xy%x = x%x - y%x
end function sub_momentum_vector
pure function sub_vector_momentum (x, y) result (xy)
type(vector), intent(in) :: x
type(momentum), intent(in) :: y
type(vector) :: xy
xy%t = x%t - y%t
xy%x = x%x - y%x
end function sub_vector_momentum
pure function sub_tensor2odd (x, y) result (xy)
type(tensor2odd), intent(in) :: x, y
type(tensor2odd) :: xy
xy%e = x%e - y%e
xy%b = x%b - y%b
end function sub_tensor2odd
```

AB.4.6 Norm

Not the covariant length!

⟨*Declaration of operations for vectors*⟩+≡
```
interface abs
module procedure abs_momentum, abs_vector, abs_tensor2odd
end interface
private :: abs_momentum, abs_vector, abs_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function abs_momentum (x) result (absx)
type(momentum), intent(in) :: x
real(kind=default) :: absx
absx = sqrt (real (x%t*x%t + dot_product (x%x, x%x)))
end function abs_momentum
pure function abs_vector (x) result (absx)
type(vector), intent(in) :: x
real(kind=default) :: absx
absx = sqrt (real (conjg(x%t)*x%t + dot_product (x%x, x%x)))
end function abs_vector
pure function abs_tensor2odd (x) result (absx)
type(tensor2odd), intent(in) :: x
real(kind=default) :: absx
absx = sqrt (real (dot_product (x%e, x%e) + dot_product (x%b, x%b)))
end function abs_tensor2odd
```

AB.4.7 Conjugation

⟨*Declaration of operations for vectors*⟩+≡
```
interface conjg
module procedure conjg_momentum, conjg_vector, conjg_tensor2odd
end interface
private :: conjg_momentum, conjg_vector, conjg_tensor2odd
```

⟨*Implementation of operations for vectors*⟩+≡
```
pure function conjg_momentum (x) result (conjg_x)
type(momentum), intent(in) :: x
type(momentum) :: conjg_x
conjg_x = x
end function conjg_momentum
pure function conjg_vector (x) result (conjg_x)
type(vector), intent(in) :: x
type(vector) :: conjg_x
conjg_x%t = conjg (x%t)
conjg_x%x = conjg (x%x)
end function conjg_vector
pure function conjg_tensor2odd (t2) result (conjg_t2)
type(tensor2odd), intent(in) :: t2
type(tensor2odd) :: conjg_t2
conjg_t2%e = conjg (t2%e)
conjg_t2%b = conjg (t2%b)
end function conjg_tensor2odd
```

AB.4.8 ϵ-Tensors

$$\epsilon_{0123} = 1 = -\epsilon^{0123} \tag{AB.8}$$

in particular

$$\epsilon(p_1,p_2,p_3,p_4) = \epsilon_{\mu_1\mu_2\mu_3\mu_4} p_1^{\mu_1} p_2^{\mu_2} p_3^{\mu_3} p_4^{\mu_4} = p_1^0 p_2^1 p_3^2 p_4^3 \pm \ldots \tag{AB.9}$$

⟨*Declaration of operations for vectors*⟩+≡
```
interface pseudo_scalar
module procedure pseudo_scalar_momentum, pseudo_scalar_vector, &
pseudo_scalar_vec_mom
end interface
public :: pseudo_scalar
private :: pseudo_scalar_momentum, pseudo_scalar_vector
```

⟨*Implementation of operations for vectors*⟩+≡
```
  pure function pseudo_scalar_momentum (p1, p2, p3, p4) result (eps1234)
  type(momentum), intent(in) :: p1, p2, p3, p4
  real(kind=default) :: eps1234
  eps1234 = &
  p1%t    * p2%x(1) * (p3%x(2) * p4%x(3) - p3%x(3) * p4%x(2)) &
  + p1%t    * p2%x(2) * (p3%x(3) * p4%x(1) - p3%x(1) * p4%x(3)) &
  + p1%t    * p2%x(3) * (p3%x(1) * p4%x(2) - p3%x(2) * p4%x(1)) &
  - p1%x(1) * p2%x(2) * (p3%x(3) * p4%t    - p3%t    * p4%x(3)) &
  - p1%x(1) * p2%x(3) * (p3%t    * p4%x(2) - p3%x(2) * p4%t   ) &
  - p1%x(1) * p2%t    * (p3%x(2) * p4%x(3) - p3%x(3) * p4%x(2)) &
  + p1%x(2) * p2%x(3) * (p3%t    * p4%x(1) - p3%x(1) * p4%t   ) &
  + p1%x(2) * p2%t    * (p3%x(1) * p4%x(3) - p3%x(3) * p4%x(1)) &
  + p1%x(2) * p2%x(1) * (p3%x(3) * p4%t    - p3%t    * p4%x(3)) &
  - p1%x(3) * p2%t    * (p3%x(1) * p4%x(2) - p3%x(2) * p4%x(1)) &
  - p1%x(3) * p2%x(1) * (p3%x(2) * p4%t    - p3%t    * p4%x(2)) &
  - p1%x(3) * p2%x(2) * (p3%t    * p4%x(1) - p3%x(1) * p4%t   )
  end function pseudo_scalar_momentum
```
⟨*Implementation of operations for vectors*⟩+≡
```
  pure function pseudo_scalar_vector (p1, p2, p3, p4) result (eps1234)
  type(vector), intent(in) :: p1, p2, p3, p4
  complex(kind=default) :: eps1234
  eps1234 = &
  p1%t    * p2%x(1) * (p3%x(2) * p4%x(3) - p3%x(3) * p4%x(2)) &
  + p1%t    * p2%x(2) * (p3%x(3) * p4%x(1) - p3%x(1) * p4%x(3)) &
  + p1%t    * p2%x(3) * (p3%x(1) * p4%x(2) - p3%x(2) * p4%x(1)) &
  - p1%x(1) * p2%x(2) * (p3%x(3) * p4%t    - p3%t    * p4%x(3)) &
  - p1%x(1) * p2%x(3) * (p3%t    * p4%x(2) - p3%x(2) * p4%t   ) &
  - p1%x(1) * p2%t    * (p3%x(2) * p4%x(3) - p3%x(3) * p4%x(2)) &
  + p1%x(2) * p2%x(3) * (p3%t    * p4%x(1) - p3%x(1) * p4%t   ) &
  + p1%x(2) * p2%t    * (p3%x(1) * p4%x(3) - p3%x(3) * p4%x(1)) &
  + p1%x(2) * p2%x(1) * (p3%x(3) * p4%t    - p3%t    * p4%x(3)) &
  - p1%x(3) * p2%t    * (p3%x(1) * p4%x(2) - p3%x(2) * p4%x(1)) &
  - p1%x(3) * p2%x(1) * (p3%x(2) * p4%t    - p3%t    * p4%x(2)) &
  - p1%x(3) * p2%x(2) * (p3%t    * p4%x(1) - p3%x(1) * p4%t   )
  end function pseudo_scalar_vector
```
⟨*Implementation of operations for vectors*⟩+≡
```
  pure function pseudo_scalar_vec_mom (p1, v1, p2, v2) result (eps1234)
  type(momentum), intent(in)   :: p1, p2
  type(vector), intent(in) :: v1, v2
  complex(kind=default) :: eps1234
  eps1234 = &
  p1%t    * v1%x(1) * (p2%x(2) * v2%x(3) - p2%x(3) * v2%x(2)) &
  + p1%t    * v1%x(2) * (p2%x(3) * v2%x(1) - p2%x(1) * v2%x(3)) &
  + p1%t    * v1%x(3) * (p2%x(1) * v2%x(2) - p2%x(2) * v2%x(1)) &
  - p1%x(1) * v1%x(2) * (p2%x(3) * v2%t    - p2%t    * v2%x(3)) &
  - p1%x(1) * v1%x(3) * (p2%t    * v2%x(2) - p2%x(2) * v2%t   ) &
  - p1%x(1) * v1%t    * (p2%x(2) * v2%x(3) - p2%x(3) * v2%x(2)) &
  + p1%x(2) * v1%x(3) * (p2%t    * v2%x(1) - p2%x(1) * v2%t   ) &
  + p1%x(2) * v1%t    * (p2%x(1) * v2%x(3) - p2%x(3) * v2%x(1)) &
  + p1%x(2) * v1%x(1) * (p2%x(3) * v2%t    - p2%t    * v2%x(3)) &
  - p1%x(3) * v1%t    * (p2%x(1) * v2%x(2) - p2%x(2) * v2%x(1)) &
  - p1%x(3) * v1%x(1) * (p2%x(2) * v2%t    - p2%t    * v2%x(2)) &
  - p1%x(3) * v1%x(2) * (p2%t    * v2%x(1) - p2%x(1) * v2%t   )
  end function pseudo_scalar_vec_mom
```
$$\epsilon_\mu(p_1,p_2,p_3) = \epsilon_{\mu\mu_1\mu_2\mu_3} p_1^{\mu_1} p_2^{\mu_2} p_3^{\mu_3} \tag{AB.10}$$

i. e.

$$\epsilon_0(p_1,p_2,p_3) = p_1^1 p_2^2 p_3^3 \pm \ldots \tag{AB.11a}$$
$$\epsilon_1(p_1,p_2,p_3) = p_1^2 p_2^3 p_3^0 \pm \ldots \tag{AB.11b}$$
$$\epsilon_2(p_1,p_2,p_3) = -p_1^3 p_2^0 p_3^1 \pm \ldots \tag{AB.11c}$$
$$\epsilon_3(p_1,p_2,p_3) = p_1^0 p_2^1 p_3^2 \pm \ldots \tag{AB.11d}$$

⟨*Declaration of operations for vectors*⟩+≡

```
interface pseudo_vector
module procedure pseudo_vector_momentum, pseudo_vector_vector, &
pseudo_vector_vec_mom
end interface
public :: pseudo_vector
private :: pseudo_vector_momentum, pseudo_vector_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function pseudo_vector_momentum (p1, p2, p3) result (eps123)
type(momentum), intent(in) :: p1, p2, p3
type(momentum) :: eps123
eps123%t = &
+ p1%x(1) * (p2%x(2) * p3%x(3) - p2%x(3) * p3%x(2)) &
+ p1%x(2) * (p2%x(3) * p3%x(1) - p2%x(1) * p3%x(3)) &
+ p1%x(3) * (p2%x(1) * p3%x(2) - p2%x(2) * p3%x(1))
eps123%x(1) = &
+ p1%x(2) * (p2%x(3) * p3%t    - p2%t    * p3%x(3)) &
+ p1%x(3) * (p2%t    * p3%x(2) - p2%x(2) * p3%t   ) &
+ p1%t    * (p2%x(2) * p3%x(3) - p2%x(3) * p3%x(2))
eps123%x(2) = &
- p1%x(3) * (p2%t    * p3%x(1) - p2%x(1) * p3%t   ) &
- p1%t    * (p2%x(1) * p3%x(3) - p2%x(3) * p3%x(1)) &
- p1%x(1) * (p2%x(3) * p3%t    - p2%t    * p3%x(3))
eps123%x(3) =  &
+ p1%t    * (p2%x(1) * p3%x(2) - p2%x(2) * p3%x(1)) &
+ p1%x(1) * (p2%x(2) * p3%t    - p2%t    * p3%x(2)) &
+ p1%x(2) * (p2%t    * p3%x(1) - p2%x(1) * p3%t   )
end function pseudo_vector_momentum
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function pseudo_vector_vector (p1, p2, p3) result (eps123)
type(vector), intent(in) :: p1, p2, p3
type(vector) :: eps123
eps123%t = &
+ p1%x(1) * (p2%x(2) * p3%x(3) - p2%x(3) * p3%x(2)) &
+ p1%x(2) * (p2%x(3) * p3%x(1) - p2%x(1) * p3%x(3)) &
+ p1%x(3) * (p2%x(1) * p3%x(2) - p2%x(2) * p3%x(1))
eps123%x(1) = &
+ p1%x(2) * (p2%x(3) * p3%t    - p2%t    * p3%x(3)) &
+ p1%x(3) * (p2%t    * p3%x(2) - p2%x(2) * p3%t   ) &
+ p1%t    * (p2%x(2) * p3%x(3) - p2%x(3) * p3%x(2))
eps123%x(2) = &
- p1%x(3) * (p2%t    * p3%x(1) - p2%x(1) * p3%t   ) &
- p1%t    * (p2%x(1) * p3%x(3) - p2%x(3) * p3%x(1)) &
- p1%x(1) * (p2%x(3) * p3%t    - p2%t    * p3%x(3))
eps123%x(3) =  &
+ p1%t    * (p2%x(1) * p3%x(2) - p2%x(2) * p3%x(1)) &
+ p1%x(1) * (p2%x(2) * p3%t    - p2%t    * p3%x(2)) &
+ p1%x(2) * (p2%t    * p3%x(1) - p2%x(1) * p3%t   )
end function pseudo_vector_vector
```

⟨*Implementation of operations for vectors*⟩+≡

```
pure function pseudo_vector_vec_mom (p1, p2, v) result (eps123)
type(momentum), intent(in) :: p1, p2
type(vector), intent(in)   :: v
type(vector) :: eps123
eps123%t = &
+ p1%x(1) * (p2%x(2) * v%x(3) - p2%x(3) * v%x(2)) &
+ p1%x(2) * (p2%x(3) * v%x(1) - p2%x(1) * v%x(3)) &
+ p1%x(3) * (p2%x(1) * v%x(2) - p2%x(2) * v%x(1))
eps123%x(1) = &
+ p1%x(2) * (p2%x(3) * v%t    - p2%t    * v%x(3)) &
+ p1%x(3) * (p2%t    * v%x(2) - p2%x(2) * v%t   ) &
+ p1%t    * (p2%x(2) * v%x(3) - p2%x(3) * v%x(2))
eps123%x(2) = &
- p1%x(3) * (p2%t    * v%x(1) - p2%x(1) * v%t   ) &
- p1%t    * (p2%x(1) * v%x(3) - p2%x(3) * v%x(1)) &
- p1%x(1) * (p2%x(3) * v%t    - p2%t    * v%x(3))
```

```
  eps123%x(3) =  &
  + p1%t    * (p2%x(1) * v%x(2) - p2%x(2) * v%x(1)) &
  + p1%x(1) * (p2%x(2) * v%t    - p2%t    * v%x(2)) &
  + p1%x(2) * (p2%t    * v%x(1) - p2%x(1) * v%t   )
  end function pseudo_vector_vec_mom
```

AB.4.9 Utilities

⟨*Declaration of operations for vectors*⟩+≡

⟨*Implementation of operations for vectors*⟩+≡
```
  subroutine random_momentum (p, pabs, m)
  type(momentum), intent(out) :: p
  real(kind=default), intent(in) :: pabs, m
  real(kind=default), dimension(2) :: r
  real(kind=default) :: phi, cos_th
  call random_number (r)
  phi = 2*PI * r(1)
  cos_th = 2 * r(2) - 1
  p%t = sqrt (pabs**2 + m**2)
  p%x = pabs * (/ cos_th * cos(phi), cos_th * sin(phi), sqrt (1 - cos_th**2) /)
  end subroutine random_momentum
```

AB.5 Polarization vectors

⟨omega_polarizations.f90⟩≡
  ⟨*Copyleft*⟩
```
  module omega_polarizations
  use kinds
  use constants
  use omega_vectors
  implicit none
  private
```
  ⟨*Declaration of polarization vectors*⟩
```
  integer, parameter, public :: omega_polarizations_2010_01_A = 0
  contains
```
  ⟨*Implementation of polarization vectors*⟩
```
  end module omega_polarizations
```

Here we use a phase convention for the polarization vectors compatible with the angular momentum coupling to spin 3/2 and spin 2.

$$\epsilon_1^\mu(k) = \frac{1}{|\vec k|\sqrt{k_x^2+k_y^2}}\left(0; k_zk_x, k_yk_z, -k_x^2-k_y^2\right) \tag{AB.12a}$$

$$\epsilon_2^\mu(k) = \frac{1}{\sqrt{k_x^2+k_y^2}}\left(0; -k_y, k_x, 0\right) \tag{AB.12b}$$

$$\epsilon_3^\mu(k) = \frac{k_0}{m|\vec k|}\left(\vec k^2/k_0; k_x, k_y, k_z\right) \tag{AB.12c}$$

and

$$\epsilon_\pm^\mu(k) = \frac{1}{\sqrt{2}}(\epsilon_1^\mu(k) \pm \mathrm{i}\epsilon_2^\mu(k)) \tag{AB.13a}$$

$$\epsilon_0^\mu(k) = \epsilon_3^\mu(k) \tag{AB.13b}$$

i. e.

$$\epsilon_+^\mu(k) = \frac{1}{\sqrt{2}\sqrt{k_x^2+k_y^2}}\left(0; \frac{k_zk_x}{|\vec k|} - \mathrm{i}k_y, \frac{k_yk_z}{|\vec k|} + \mathrm{i}k_x, -\frac{k_x^2+k_y^2}{|\vec k|}\right) \tag{AB.14a}$$

$$\epsilon_-^\mu(k) = \frac{1}{\sqrt{2}\sqrt{k_x^2+k_y^2}}\left(0; \frac{k_zk_x}{|\vec k|} + \mathrm{i}k_y, \frac{k_yk_z}{|\vec k|} - \mathrm{i}k_x, -\frac{k_x^2+k_y^2}{|\vec k|}\right) \tag{AB.14b}$$

$$\epsilon_0^\mu(k) = \frac{k_0}{m|\vec{k}|}\left(\vec{k}^2/k_0; k_x, k_y, k_z\right) \tag{AB.14c}$$

Determining the mass from the momenta is a numerically haphazardous for light particles. Therefore, we accept some redundancy and pass the mass explicitely.

⟨*Declaration of polarization vectors*⟩≡

```
public :: eps
```

⟨*Implementation of polarization vectors*⟩≡

```
pure function eps (m, k, s) result (e)
type(vector) :: e
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
integer, intent(in) :: s
real(kind=default) :: kt, kabs, kabs2, sqrt2
sqrt2 = sqrt (2.0_default)
kabs2 = dot_product (k%x, k%x)
e%t = 0
e%x = 0
if (kabs2 > 0) then
kabs = sqrt (kabs2)
select case (s)
case (1)
kt = sqrt (k%x(1)**2 + k%x(2)**2)
if (abs(kt) <= epsilon(kt) * kabs) then
if (k%x(3) > 0) then
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0,   1, kind=default) / sqrt2
else
e%x(1) = cmplx ( - 1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0,   1, kind=default) / sqrt2
end if
else
e%x(1) = cmplx (   k%x(3)*k%x(1)/kabs, &
- k%x(2), kind=default) / kt / sqrt2
e%x(2) = cmplx (   k%x(2)*k%x(3)/kabs, &
k%x(1), kind=default) / kt / sqrt2
e%x(3) = - kt / kabs / sqrt2
end if
case (-1)
kt = sqrt (k%x(1)**2 + k%x(2)**2)
if (abs(kt) <= epsilon(kt) * kabs) then
if (k%x(3) > 0) then
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
else
e%x(1) = cmplx (  -1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
end if
else
e%x(1) = cmplx (   k%x(3)*k%x(1)/kabs, &
k%x(2), kind=default) / kt / sqrt2
e%x(2) = cmplx (   k%x(2)*k%x(3)/kabs, &
- k%x(1), kind=default) / kt / sqrt2
e%x(3) = - kt / kabs / sqrt2
end if
case (0)
if (m > 0) then
e%t = kabs / m
e%x = k%t / (m*kabs) * k%x
end if
case (3)
e = (0,1) * k
case (4)
if (m > 0) then
e = (1 / m) * k
else
```

```
e = (1 / k%t) * k
end if
end select
else   !!! for particles in their rest frame defined to be
!!! polarized along the 3-direction
select case (s)
case (1)
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0,   1, kind=default) / sqrt2
case (-1)
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
case (0)
if (m > 0) then
e%x(3) = 1
end if
case (4)
if (m > 0) then
e = (1 / m) * k
else
e = (1 / k%t) * k
end if
end select
end if
end function eps
```

AB.6 Polarization vectors revisited

⟨omega_polarizations_madgraph.f90⟩≡
```
⟨Copyleft⟩
module omega_polarizations_madgraph
use kinds
use constants
use omega_vectors
implicit none
private
⟨Declaration of polarization vectors for madgraph⟩
integer, parameter, public :: omega_pols_madgraph_2010_01_A = 0
contains
⟨Implementation of polarization vectors for madgraph⟩
end module omega_polarizations_madgraph
```

This set of polarization vectors is compatible with HELAS [5]:

$$\epsilon^\mu_1(k) = \frac{1}{|\vec k|\sqrt{k_x^2+k_y^2}} \left(0; k_z k_x, k_y k_z, -k_x^2 - k_y^2\right) \tag{AB.15a}$$

$$\epsilon^\mu_2(k) = \frac{1}{\sqrt{k_x^2+k_y^2}} \left(0; -k_y, k_x, 0\right) \tag{AB.15b}$$

$$\epsilon^\mu_3(k) = \frac{k_0}{m|\vec k|} \left(\vec k^2/k_0; k_x, k_y, k_z\right) \tag{AB.15c}$$

and

$$\epsilon^\mu_\pm(k) = \frac{1}{\sqrt{2}} (\mp\epsilon^\mu_1(k) - \mathrm{i}\epsilon^\mu_2(k)) \tag{AB.16a}$$

$$\epsilon^\mu_0(k) = \epsilon^\mu_3(k) \tag{AB.16b}$$

i. e.

$$\epsilon^\mu_+(k) = \frac{1}{\sqrt{2}\sqrt{k_x^2+k_y^2}} \left(0; -\frac{k_z k_x}{|\vec k|} + \mathrm{i}k_y, -\frac{k_y k_z}{|\vec k|} - \mathrm{i}k_x, \frac{k_x^2+k_y^2}{|\vec k|}\right) \tag{AB.17a}$$

$$\epsilon^\mu_-(k) = \frac{1}{\sqrt{2}\sqrt{k_x^2+k_y^2}} \left(0; \frac{k_z k_x}{|\vec k|} + \mathrm{i}k_y, \frac{k_y k_z}{|\vec k|} - \mathrm{i}k_x, -\frac{k_x^2+k_y^2}{|\vec k|}\right) \tag{AB.17b}$$

$$\epsilon_0^\mu(k) = \frac{k_0}{m|\vec{k}|}\left(\vec{k}^2/k_0; k_x, k_y, k_z\right) \tag{AB.17c}$$

Fortunately, for comparing with squared matrix generated by Madgraph we can also use the modified version, since the difference is only a phase and does *not* mix helicity states. Determining the mass from the momenta is a numerically haphazardous for light particles. Therefore, we accept some redundancy and pass the mass explicitely.

⟨*Declaration of polarization vectors for madgraph*⟩≡
```
public :: eps
```

⟨*Implementation of polarization vectors for madgraph*⟩≡
```
pure function eps (m, k, s) result (e)
type(vector) :: e
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
integer, intent(in) :: s
real(kind=default) :: kt, kabs, kabs2, sqrt2
sqrt2 = sqrt (2.0_default)
kabs2 = dot_product (k%x, k%x)
e%t = 0
e%x = 0
if (kabs2 > 0) then
kabs = sqrt (kabs2)
select case (s)
case (1)
kt = sqrt (k%x(1)**2 + k%x(2)**2)
if (abs(kt) <= epsilon(kt) * kabs) then
if (k%x(3) > 0) then
e%x(1) = cmplx ( - 1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
else
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
end if
else
e%x(1) = cmplx ( - k%x(3)*k%x(1)/kabs, &
k%x(2), kind=default) / kt / sqrt2
e%x(2) = cmplx ( - k%x(2)*k%x(3)/kabs, &
- k%x(1), kind=default) / kt / sqrt2
e%x(3) = kt / kabs / sqrt2
end if
case (-1)
kt = sqrt (k%x(1)**2 + k%x(2)**2)
if (abs(kt) <= epsilon(kt) * kabs) then
if (k%x(3) > 0) then
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
else
e%x(1) = cmplx (  -1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
end if
else
e%x(1) = cmplx (   k%x(3)*k%x(1)/kabs, &
k%x(2), kind=default) / kt / sqrt2
e%x(2) = cmplx (   k%x(2)*k%x(3)/kabs, &
- k%x(1), kind=default) / kt / sqrt2
e%x(3) = - kt / kabs / sqrt2
end if
case (0)
if (m > 0) then
e%t = kabs / m
e%x = k%t / (m*kabs) * k%x
end if
case (3)
e = (0,1) * k
case (4)
if (m > 0) then
```

```
e = (1 / m) * k
else
e = (1 / k%t) * k
end if
end select
else   !!! for particles in their rest frame defined to be
!!! polarized along the 3-direction
select case (s)
case (1)
e%x(1) = cmplx ( - 1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
case (-1)
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0, - 1, kind=default) / sqrt2
case (0)
if (m > 0) then
e%x(3) = 1
end if
case (4)
if (m > 0) then
e = (1 / m) * k
else
e = (1 / k%t) * k
end if
end select
end if
end function eps
```

AB.7 Symmetric Tensors

Spin-2 polarization tensors are symmetric, transversal and traceless

$$\epsilon_m^{\mu\nu}(k) = \epsilon_m^{\nu\mu}(k) \tag{AB.18a}$$
$$k_\mu \epsilon_m^{\mu\nu}(k) = k_\nu \epsilon_m^{\mu\nu}(k) = 0 \tag{AB.18b}$$
$$\epsilon_{m,\mu}^{\mu}(k) = 0 \tag{AB.18c}$$

with $m = 1, 2, 3, 4, 5$. Our current representation is redundant and does *not* enforce symmetry or tracelessness.

⟨omega_tensors.f90⟩≡
```
⟨Copyleft⟩
module omega_tensors
use kinds
use constants
use omega_vectors
implicit none
private
public :: operator (*), operator (+), operator (-), &
operator (.tprod.)
public :: abs, conjg, set_zero
⟨intrinsic :: abs⟩
⟨intrinsic :: conjg⟩
type, public :: tensor
! private (omegalib needs access, but DON'T TOUCH IT!)
complex(kind=default), dimension(0:3,0:3) :: t
end type tensor
⟨Declaration of operations for tensors⟩
integer, parameter, public :: omega_tensors_2010_01_A = 0
contains
⟨Implementation of operations for tensors⟩
end module omega_tensors
```

AB.7.1 Vector Space

⟨*Declaration of operations for tensors*⟩≡
```
interface set_zero
```

```
module procedure set_zero_tensor
end interface
private :: set_zero_tensor
```

⟨*Implementation of operations for tensors*⟩≡

```
elemental subroutine set_zero_tensor (x)
type(tensor), intent(out) :: x
x%t = 0
end subroutine set_zero_tensor
```

Scalar Multliplication

⟨*Declaration of operations for tensors*⟩+≡

```
interface operator (*)
module procedure integer_tensor, real_tensor, double_tensor, &
complex_tensor, dcomplex_tensor
end interface
private :: integer_tensor, real_tensor, double_tensor
private :: complex_tensor, dcomplex_tensor
```

⟨*Implementation of operations for tensors*⟩+≡

```
pure function integer_tensor (x, y) result (xy)
integer, intent(in) :: x
type(tensor), intent(in) :: y
type(tensor) :: xy
xy%t = x * y%t
end function integer_tensor
pure function real_tensor (x, y) result (xy)
real(kind=single), intent(in) :: x
type(tensor), intent(in) :: y
type(tensor) :: xy
xy%t = x * y%t
end function real_tensor
pure function double_tensor (x, y) result (xy)
real(kind=default), intent(in) :: x
type(tensor), intent(in) :: y
type(tensor) :: xy
xy%t = x * y%t
end function double_tensor
pure function complex_tensor (x, y) result (xy)
complex(kind=single), intent(in) :: x
type(tensor), intent(in) :: y
type(tensor) :: xy
xy%t = x * y%t
end function complex_tensor
pure function dcomplex_tensor (x, y) result (xy)
complex(kind=default), intent(in) :: x
type(tensor), intent(in) :: y
type(tensor) :: xy
xy%t = x * y%t
end function dcomplex_tensor
```

Addition and Subtraction

⟨*Declaration of operations for tensors*⟩+≡

```
interface operator (+)
module procedure plus_tensor
end interface
private :: plus_tensor
interface operator (-)
module procedure neg_tensor
end interface
private :: neg_tensor
```

⟨*Implementation of operations for tensors*⟩+≡

```
pure function plus_tensor (t1) result (t2)
type(tensor), intent(in) :: t1
type(tensor) :: t2
t2 = t1
end function plus_tensor
pure function neg_tensor (t1) result (t2)
type(tensor), intent(in) :: t1
type(tensor) :: t2
t2%t = - t1%t
end function neg_tensor
```

⟨*Declaration of operations for tensors*⟩+≡

```
interface operator (+)
module procedure add_tensor
end interface
private :: add_tensor
interface operator (-)
module procedure sub_tensor
end interface
private :: sub_tensor
```

⟨*Implementation of operations for tensors*⟩+≡

```
pure function add_tensor (x, y) result (xy)
type(tensor), intent(in) :: x, y
type(tensor) :: xy
xy%t = x%t + y%t
end function add_tensor
pure function sub_tensor (x, y) result (xy)
type(tensor), intent(in) :: x, y
type(tensor) :: xy
xy%t = x%t - y%t
end function sub_tensor
```

⟨*Declaration of operations for tensors*⟩+≡

```
interface operator (.tprod.)
module procedure out_prod_vv, out_prod_vm, &
out_prod_mv, out_prod_mm
end interface
private :: out_prod_vv, out_prod_vm, &
out_prod_mv, out_prod_mm
```

⟨*Implementation of operations for tensors*⟩+≡

```
pure function out_prod_vv (v, w) result (t)
type(tensor) :: t
type(vector), intent(in) :: v, w
integer :: i, j
t%t(0,0) = v%t * w%t
t%t(0,1:3) = v%t * w%x
t%t(1:3,0) = v%x * w%t
do i = 1, 3
do j = 1, 3
t%t(i,j) = v%x(i) * w%x(j)
end do
end do
end function out_prod_vv
```

⟨*Implementation of operations for tensors*⟩+≡

```
pure function out_prod_vm (v, m) result (t)
type(tensor) :: t
type(vector), intent(in) :: v
type(momentum), intent(in) :: m
integer :: i, j
t%t(0,0) = v%t * m%t
t%t(0,1:3) = v%t * m%x
t%t(1:3,0) = v%x * m%t
do i = 1, 3
do j = 1, 3
t%t(i,j) = v%x(i) * m%x(j)
end do
end do
end function out_prod_vm
```

⟨*Implementation of operations for tensors*⟩+≡
```
pure function out_prod_mv (m, v) result (t)
type(tensor) :: t
type(vector), intent(in) :: v
type(momentum), intent(in) :: m
integer :: i, j
t%t(0,0) = m%t * v%t
t%t(0,1:3) = m%t * v%x
t%t(1:3,0) = m%x * v%t
do i = 1, 3
do j = 1, 3
t%t(i,j) = m%x(i) * v%x(j)
end do
end do
end function out_prod_mv
```

⟨*Implementation of operations for tensors*⟩+≡
```
pure function out_prod_mm (m, n) result (t)
type(tensor) :: t
type(momentum), intent(in) :: m, n
integer :: i, j
t%t(0,0) = m%t * n%t
t%t(0,1:3) = m%t * n%x
t%t(1:3,0) = m%x * n%t
do i = 1, 3
do j = 1, 3
t%t(i,j) = m%x(i) * n%x(j)
end do
end do
end function out_prod_mm
```

⟨*Declaration of operations for tensors*⟩+≡
```
interface abs
module procedure abs_tensor
end interface
private :: abs_tensor
```

⟨*Implementation of operations for tensors*⟩+≡
```
pure function abs_tensor (t) result (abs_t)
type(tensor), intent(in) :: t
real(kind=default) :: abs_t
abs_t = sqrt (sum ((abs (t%t))**2))
end function abs_tensor
```

⟨*Declaration of operations for tensors*⟩+≡
```
interface conjg
module procedure conjg_tensor
end interface
private :: conjg_tensor
```

⟨*Implementation of operations for tensors*⟩+≡
```
pure function conjg_tensor (t) result (conjg_t)
type(tensor), intent(in) :: t
type(tensor) :: conjg_t
conjg_t%t = conjg (t%t)
end function conjg_tensor
```

⟨*Declaration of operations for tensors*⟩+≡
```
interface operator (*)
module procedure tensor_tensor, vector_tensor, tensor_vector, &
momentum_tensor, tensor_momentum
end interface
private :: tensor_tensor, vector_tensor, tensor_vector, &
momentum_tensor, tensor_momentum
```

⟨*Implementation of operations for tensors*⟩+≡
```
pure function tensor_tensor (t1, t2) result (t1t2)
type(tensor), intent(in) :: t1
type(tensor), intent(in) :: t2
complex(kind=default) :: t1t2
```

```
  integer :: i1, i2
  t1t2 = t1%t(0,0)*t2%t(0,0) &
  - dot_product (conjg (t1%t(0,1:)), t2%t(0,1:)) &
  - dot_product (conjg (t1%t(1:,0)), t2%t(1:,0))
  do i1 = 1, 3
  do i2 = 1, 3
  t1t2 = t1t2 + t1%t(i1,i2)*t2%t(i1,i2)
  end do
  end do
  end function tensor_tensor
```

⟨*Implementation of operations for tensors*⟩+≡

```
  pure function tensor_vector (t, v) result (tv)
  type(tensor), intent(in) :: t
  type(vector), intent(in) :: v
  type(vector) :: tv
  tv%t =    t%t(0,0) * v%t - dot_product (conjg (t%t(0,1:)), v%x)
  tv%x(1) = t%t(0,1) * v%t - dot_product (conjg (t%t(1,1:)), v%x)
  tv%x(2) = t%t(0,2) * v%t - dot_product (conjg (t%t(2,1:)), v%x)
  tv%x(3) = t%t(0,3) * v%t - dot_product (conjg (t%t(3,1:)), v%x)
  end function tensor_vector
```

⟨*Implementation of operations for tensors*⟩+≡

```
  pure function vector_tensor (v, t) result (vt)
  type(vector), intent(in) :: v
  type(tensor), intent(in) :: t
  type(vector) :: vt
  vt%t =    v%t * t%t(0,0) - dot_product (conjg (v%x), t%t(1:,0))
  vt%x(1) = v%t * t%t(0,1) - dot_product (conjg (v%x), t%t(1:,1))
  vt%x(2) = v%t * t%t(0,2) - dot_product (conjg (v%x), t%t(1:,2))
  vt%x(3) = v%t * t%t(0,3) - dot_product (conjg (v%x), t%t(1:,3))
  end function vector_tensor
```

⟨*Implementation of operations for tensors*⟩+≡

```
  pure function tensor_momentum (t, p) result (tp)
  type(tensor), intent(in) :: t
  type(momentum), intent(in) :: p
  type(vector) :: tp
  tp%t =    t%t(0,0) * p%t - dot_product (conjg (t%t(0,1:)), p%x)
  tp%x(1) = t%t(0,1) * p%t - dot_product (conjg (t%t(1,1:)), p%x)
  tp%x(2) = t%t(0,2) * p%t - dot_product (conjg (t%t(2,1:)), p%x)
  tp%x(3) = t%t(0,3) * p%t - dot_product (conjg (t%t(3,1:)), p%x)
  end function tensor_momentum
```

⟨*Implementation of operations for tensors*⟩+≡

```
  pure function momentum_tensor (p, t) result (pt)
  type(momentum), intent(in) :: p
  type(tensor), intent(in) :: t
  type(vector) :: pt
  pt%t =    p%t * t%t(0,0) - dot_product (p%x, t%t(1:,0))
  pt%x(1) = p%t * t%t(0,1) - dot_product (p%x, t%t(1:,1))
  pt%x(2) = p%t * t%t(0,2) - dot_product (p%x, t%t(1:,2))
  pt%x(3) = p%t * t%t(0,3) - dot_product (p%x, t%t(1:,3))
  end function momentum_tensor
```

AB.8 Symmetric Polarization Tensors

$$\epsilon^{\mu\nu}_{+2}(k) = \epsilon^{\mu}_{+}(k)\epsilon^{\nu}_{+}(k) \tag{AB.19a}$$

$$\epsilon^{\mu\nu}_{+1}(k) = \frac{1}{\sqrt{2}}\left(\epsilon^{\mu}_{+}(k)\epsilon^{\nu}_{0}(k) + \epsilon^{\mu}_{0}(k)\epsilon^{\nu}_{+}(k)\right) \tag{AB.19b}$$

$$\epsilon^{\mu\nu}_{0}(k) = \frac{1}{\sqrt{6}}\left(\epsilon^{\mu}_{+}(k)\epsilon^{\nu}_{-}(k) + \epsilon^{\mu}_{-}(k)\epsilon^{\nu}_{+}(k) - 2\epsilon^{\mu}_{0}(k)\epsilon^{\nu}_{0}(k)\right) \tag{AB.19c}$$

$$\epsilon^{\mu\nu}_{-1}(k) = \frac{1}{\sqrt{2}}\left(\epsilon^{\mu}_{-}(k)\epsilon^{\nu}_{0}(k) + \epsilon^{\mu}_{0}(k)\epsilon^{\nu}_{-}(k)\right) \tag{AB.19d}$$

$$\epsilon^{\mu\nu}_{-2}(k) = \epsilon^{\mu}_{-}(k)\epsilon^{\nu}_{-}(k) \tag{AB.19e}$$

Note that $\epsilon^{\mu}_{\pm 2,\mu}(k) = \epsilon^{\mu}_{\pm}(k)\epsilon_{\pm,\mu}(k) \propto \epsilon^{\mu}_{\pm}(k)\epsilon^{*}_{\mp,\mu}(k) = 0$ and that the sign in $\epsilon^{\mu\nu}_{0}(k)$ insures its tracelessness[1].

```
⟨omega_tensor_polarizations.f90⟩≡
  ⟨Copyleft⟩
  module omega_tensor_polarizations
  use kinds
  use constants
  use omega_vectors
  use omega_tensors
  use omega_polarizations
  implicit none
  private
  ⟨Declaration of polarization tensors⟩
  integer, parameter, public :: omega_tensor_pols_2010_01_A = 0
  contains
  ⟨Implementation of polarization tensors⟩
  end module omega_tensor_polarizations
```

```
⟨Declaration of polarization tensors⟩≡
  public :: eps2
```

```
⟨Implementation of polarization tensors⟩≡
  pure function eps2 (m, k, s) result (t)
  type(tensor) :: t
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: k
  integer, intent(in) :: s
  type(vector) :: ep, em, e0
  t%t = 0
  select case (s)
  case (2)
  ep = eps (m, k, 1)
  t = ep.tprod.ep
  case (1)
  ep = eps (m, k, 1)
  e0 = eps (m, k, 0)
  t = (1 / sqrt (2.0_default)) &
  * ((ep.tprod.e0) + (e0.tprod.ep))
  case (0)
  ep = eps (m, k, 1)
  e0 = eps (m, k, 0)
  em = eps (m, k, -1)
  t = (1 / sqrt (6.0_default)) &
  * ((ep.tprod.em) + (em.tprod.ep) - 2*(e0.tprod.e0))
  case (-1)
  e0 = eps (m, k, 0)
  em = eps (m, k, -1)
  t = (1 / sqrt (2.0_default)) &
  * ((em.tprod.e0) + (e0.tprod.em))
  case (-2)
  em = eps (m, k, -1)
  t = em.tprod.em
  end select
  end function eps2
```

AB.9 Couplings

```
⟨omega_couplings.f90⟩≡
  ⟨Copyleft⟩
  module omega_couplings
  use kinds
  use constants
```

[1] On the other hand, with the shift operator $L_{-} \left|+\right\rangle = \mathrm{e}^{\mathrm{i}\phi} \left|0\right\rangle$ and $L_{-} \left|0\right\rangle = \mathrm{e}^{\mathrm{i}\chi} \left|-\right\rangle$, we find

$$L_{-}^{2} \left|++\right\rangle = 2\mathrm{e}^{2\mathrm{i}\phi} \left|00\right\rangle + \mathrm{e}^{\mathrm{i}(\phi+\chi)}(\left|+-\right\rangle + \left|-+\right\rangle)$$

i. e. $\chi - \phi = \pi$, if we want to identify $\epsilon^{\mu}_{-,0,+}$ with $\left|-,0,+\right\rangle$.

```
  use omega_vectors
  use omega_tensors
  implicit none
  private
  ⟨Declaration of couplings⟩
  ⟨Declaration of propagators⟩
  integer, parameter, public :: omega_couplings_2010_01_A = 0
  contains
  ⟨Implementation of couplings⟩
  ⟨Implementation of propagators⟩
  end module omega_couplings
```

⟨*Declaration of propagators*⟩≡

```
  public :: wd_tl
```

⟨*Declaration of propagators*⟩+≡

```
  public :: wd_run
```

⟨*Declaration of propagators*⟩+≡

```
  public :: gauss
```

$$\Theta(p^2)\Gamma \tag{AB.20}$$

⟨*Implementation of propagators*⟩≡

```
  pure function wd_tl (p, w) result (width)
  real(kind=default) :: width
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: w
  if (p*p > 0) then
  width = w
  else
  width = 0
  end if
  end function wd_tl
```

$$\frac{p^2}{m^2}\Gamma \tag{AB.21}$$

⟨*Implementation of propagators*⟩+≡

```
  pure function wd_run (p, m, w) result (width)
  real(kind=default) :: width
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m
  real(kind=default), intent(in) :: w
  if (p*p > 0) then
  width = w * (p*p) / m**2
  else
  width = 0
  end if
  end function wd_run
```

⟨*Implementation of propagators*⟩+≡

```
  pure function gauss (x, mu, w) result (gg)
  real(kind=default) :: gg
  real(kind=default), intent(in) :: x, mu, w
  if (w > 0) then
  gg = exp(-(x - mu**2)**2/4.0_default/mu**2/w**2) * &
  sqrt(sqrt(PI/2)) / w / mu
  else
  gg = 1.0_default
  end if
  end function gauss
```

⟨*Declaration of propagators*⟩+≡

```
  public :: pr_phi, pr_unitarity, pr_feynman, pr_gauge, pr_rxi
  public :: pr_vector_pure
  public :: pj_phi, pj_unitarity
  public :: pg_phi, pg_unitarity
```

$$\frac{\mathrm{i}}{p^2-m^2+\mathrm{i}m\Gamma}\phi \tag{AB.22}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_phi (p, m, w, phi) result (pphi)
  complex(kind=default) :: pphi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  complex(kind=default), intent(in) :: phi
  pphi = (1 / cmplx (p*p - m**2, m*w, kind=default)) * phi
  end function pr_phi
```

$$\sqrt{\frac{\pi}{M\Gamma}}\phi \tag{AB.23}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pj_phi (m, w, phi) result (pphi)
  complex(kind=default) :: pphi
  real(kind=default), intent(in) :: m, w
  complex(kind=default), intent(in) :: phi
  pphi = (0, -1) * sqrt (PI / m / w) * phi
  end function pj_phi
```

⟨*Implementation of propagators*⟩+≡
```
  pure function pg_phi (p, m, w, phi) result (pphi)
  complex(kind=default) :: pphi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  complex(kind=default), intent(in) :: phi
  pphi = ((0, 1) * gauss (p*p, m, w)) * phi
  end function pg_phi
```

$$\frac{\mathrm{i}}{p^2-m^2+\mathrm{i}m\Gamma}\left(-g_{\mu\nu}+\frac{p_\mu p_\nu}{m^2}\right)\epsilon^\nu(p) \tag{AB.24}$$

NB: the explicit cast to `vector` is required here, because a specific `complex_momentum` procedure for `operator (*)` would introduce ambiguities. NB: we used to use the constructor `vector (p%t, p%x)` instead of the temporary variable, but the Intel Fortran Compiler choked on it.

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_unitarity (p, m, w, cms, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(vector), intent(in) :: e
  logical, intent(in) :: cms
  type(vector) :: pv
  complex(kind=default) :: c_mass2
  pv = p
  if (cms) then
  c_mass2 = cmplx (m**2, -m*w, kind=default)
  else
  c_mass2 = m**2
  end if
  pe = - (1 / cmplx (p*p - m**2, m*w, kind=default)) &
  * (e - (p*e / c_mass2) * pv)
  end function pr_unitarity
```

$$\sqrt{\frac{\pi}{M\Gamma}}\left(-g_{\mu\nu}+\frac{p_\mu p_\nu}{m^2}\right)\epsilon^\nu(p) \tag{AB.25}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pj_unitarity (p, m, w, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(vector), intent(in) :: e
  type(vector) :: pv
  pv = p
  pe = (0, 1) * sqrt (PI / m / w) * (e - (p*e / m**2) * pv)
  end function pj_unitarity
```

⟨*Implementation of propagators*⟩+≡
```
  pure function pg_unitarity (p, m, w, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(vector), intent(in) :: e
  type(vector) :: pv
  pv = p
  pe = - gauss (p*p, m, w) &
  * (e - (p*e / m**2) * pv)
  end function pg_unitarity
```

$$\frac{-i}{p^2}\epsilon^\nu(p) \tag{AB.26}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_feynman (p, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  type(vector), intent(in) :: e
  pe = - (1 / (p*p)) * e
  end function pr_feynman
```

$$\frac{\mathrm{i}}{p^2}\left(-g_{\mu\nu}+(1-\xi)\frac{p_\mu p_\nu}{p^2}\right)\epsilon^\nu(p) \tag{AB.27}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_gauge (p, xi, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: xi
  type(vector), intent(in) :: e
  real(kind=default) :: p2
  type(vector) :: pv
  p2 = p*p
  pv = p
  pe = - (1 / p2) * (e - ((1 - xi) * (p*e) / p2) * pv)
  end function pr_gauge
```

$$\frac{\mathrm{i}}{p^2-m^2+\mathrm{i}m\Gamma}\left(-g_{\mu\nu}+(1-\xi)\frac{p_\mu p_\nu}{p^2-\xi m^2}\right)\epsilon^\nu(p) \tag{AB.28}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_rxi (p, m, w, xi, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w, xi
  type(vector), intent(in) :: e
  real(kind=default) :: p2
  type(vector) :: pv
  p2 = p*p
  pv = p
  pe = - (1 / cmplx (p2 - m**2, m*w, kind=default)) &
  * (e - ((1 - xi) * (p*e) / (p2 - xi * m**2)) * pv)
  end function pr_rxi
```

$$\frac{\mathrm{i}}{p^2-m^2+\mathrm{i}m\Gamma}\left(-g_{\mu\nu}\right)\epsilon^\nu(p) \tag{AB.29}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_vector_pure (p, m, w, e) result (pe)
  type(vector) :: pe
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(vector), intent(in) :: e
  real(kind=default) :: p2
  type(vector) :: pv
  p2 = p*p
  pv = p
  pe = - (1 / cmplx (p2 - m**2, m*w, kind=default)) * e
  end function pr_vector_pure
```

⟨*Declaration of propagators*⟩+≡
```
  public :: pr_tensor, pr_tensor_pure
```

$$\frac{\mathrm{i}P^{\mu\nu,\rho\sigma}(p,m)}{p^2-m^2+\mathrm{i}m\Gamma}T_{\rho\sigma} \tag{AB.30a}$$

with

$$P^{\mu\nu,\rho\sigma}(p,m) = \frac{1}{2}\left(g^{\mu\rho}-\frac{p^\mu p^\nu}{m^2}\right)\left(g^{\nu\sigma}-\frac{p^\nu p^\sigma}{m^2}\right)+\frac{1}{2}\left(g^{\mu\sigma}-\frac{p^\mu p^\sigma}{m^2}\right)\left(g^{\nu\rho}-\frac{p^\nu p^\rho}{m^2}\right) \\ -\frac{1}{3}\left(g^{\mu\nu}-\frac{p^\mu p^\nu}{m^2}\right)\left(g^{\rho\sigma}-\frac{p^\rho p^\sigma}{m^2}\right) \tag{AB.30b}$$

Be careful with raising and lowering of indices:

$$g^{\mu\nu}-\frac{k^\mu k^\nu}{m^2} = \begin{pmatrix} 1-k^0k^0/m^2 & -k^0\vec{k}/m^2 \\ -\vec{k}k^0/m^2 & -\mathbf{1}-\vec{k}\otimes\vec{k}/m^2 \end{pmatrix} \tag{AB.31a}$$

$$g^{\mu}{}_{\nu}-\frac{k^\mu k_\nu}{m^2} = \begin{pmatrix} 1-k^0k^0/m^2 & k^0\vec{k}/m^2 \\ -\vec{k}k^0/m^2 & \mathbf{1}+\vec{k}\otimes\vec{k}/m^2 \end{pmatrix} \tag{AB.31b}$$

⟨*Implementation of propagators*⟩+≡
```
  pure function pr_tensor (p, m, w, t) result (pt)
  type(tensor) :: pt
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(tensor), intent(in) :: t
  complex(kind=default) :: p_dd_t
  real(kind=default), dimension(0:3,0:3) :: p_uu, p_ud, p_du, p_dd
  integer :: i, j
  p_uu(0,0) = 1 - p%t * p%t / m**2
  p_uu(0,1:3) = - p%t * p%x / m**2
  p_uu(1:3,0) = p_uu(0,1:3)
  do i = 1, 3
  do j = 1, 3
  p_uu(i,j) = - p%x(i) * p%x(j) / m**2
  end do
  end do
  do i = 1, 3
  p_uu(i,i) = - 1 + p_uu(i,i)
  end do
  p_ud(:,0) = p_uu(:,0)
  p_ud(:,1:3) = - p_uu(:,1:3)
  p_du = transpose (p_ud)
  p_dd(:,0) = p_du(:,0)
  p_dd(:,1:3) = - p_du(:,1:3)
  p_dd_t = 0
  do i = 0, 3
  do j = 0, 3
  p_dd_t = p_dd_t + p_dd(i,j) * t%t(i,j)
  end do
  end do
  pt%t = matmul (p_ud, matmul (0.5_default * (t%t + transpose (t%t)), p_du)) &
  - (p_dd_t / 3.0_default) * p_uu
  pt%t = pt%t / cmplx (p*p - m**2, m*w, kind=default)
  end function pr_tensor
```

$$\frac{\mathrm{i}P_p^{\mu\nu,\rho\sigma}}{p^2-m^2+\mathrm{i}m\Gamma}T_{\rho\sigma} \tag{AB.32a}$$

with

$$P_p^{\mu\nu,\rho\sigma} = \frac{1}{2}g^{\mu\rho}g^{\nu\sigma} + \frac{1}{2}g^{\mu\sigma}g^{\nu\rho} - \frac{1}{2}g^{\mu\nu}g^{\rho\sigma} \tag{AB.32b}$$

⟨*Implementation of propagators*⟩+≡

```
pure function pr_tensor_pure (p, m, w, t) result (pt)
type(tensor) :: pt
type(momentum), intent(in) :: p
real(kind=default), intent(in) :: m, w
type(tensor), intent(in) :: t
complex(kind=default) :: p_dd_t
real(kind=default), dimension(0:3,0:3) :: g_uu
integer :: i, j
g_uu(0,0) = 1
g_uu(0,1:3) = 0
g_uu(1:3,0) = g_uu(0,1:3)
do i = 1, 3
do j = 1, 3
g_uu(i,j) = 0
end do
end do
do i = 1, 3
g_uu(i,i) = - 1
end do
p_dd_t = t%t(0,0) - t%t(1,1) - t%t(2,2) - t%t(3,3)
pt%t =  0.5_default * ((t%t + transpose (t%t)) &
- p_dd_t * g_uu )
pt%t = pt%t / cmplx (p*p - m**2, m*w, kind=default)
end function pr_tensor_pure
```

AB.9.1 Triple Gauge Couplings

⟨*Declaration of couplings*⟩≡

```
public :: g_gg
```

According to (16.6c)

$$A^{a,\mu}(k_1+k_2) = -\mathrm{i}g\big((k_1^\mu - k_2^\mu)A^{a_1}(k_1)\cdot A^{a_2}(k_2) \\ + (2k_2+k_1)\cdot A^{a_1}(k_1)A^{a_2,\mu}(k_2) - A^{a_1,\mu}(k_1)A^{a_2}(k_2)\cdot(2k_1+k_2)\big) \tag{AB.33}$$

⟨*Implementation of couplings*⟩≡

```
pure function g_gg (g, a1, k1, a2, k2) result (a)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: a1, a2
type(momentum), intent(in) :: k1, k2
type(vector) :: a
a = (0, -1) * g * ((k1 - k2) * (a1 * a2) &
+ ((2*k2 + k1) * a1) * a2 - a1 * ((2*k1 + k2) * a2))
end function g_gg
```

AB.9.2 Quadruple Gauge Couplings

⟨*Declaration of couplings*⟩+≡

```
public :: x_gg, g_gx
```

$$T^{a,\mu\nu}(k_1+k_2) = g\big(A^{a_1,\mu}(k_1)A^{a_2,\nu}(k_2) - A^{a_1,\nu}(k_1)A^{a_2,\mu}(k_2)\big) \tag{AB.34}$$

⟨*Implementation of couplings*⟩+≡

```
pure function x_gg (g, a1, a2) result (x)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: a1, a2
type(tensor2odd) :: x
x = g * (a1 .wedge. a2)
end function x_gg
```

$$A^{a,\mu}(k_1+k_2) = gA^{a_1}_\nu(k_1)T^{a_2,\nu\mu}(k_2) \tag{AB.35}$$

⟨*Implementation of couplings*⟩+≡

```
pure function g_gx (g, a1, x) result (a)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: a1
type(tensor2odd), intent(in) :: x
type(vector) :: a
```

```
  a = g * (a1 * x)
  end function g_gx
```

AB.9.3 Scalar Current

⟨*Declaration of couplings*⟩+≡
```
  public :: v_ss, s_vs
```

$$V^\mu(k_1+k_2) = g(k_1^\mu - k_2^\mu)\phi_1(k_1)\phi_2(k_2) \tag{AB.36}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_ss (g, phi1, k1, phi2, k2) result (v)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  v = (k1 - k2) * (g * phi1 * phi2)
  end function v_ss
```

$$\phi(k_1+k_2) = g(k_1^\mu + 2k_2^\mu)V_\mu(k_1)\phi(k_2) \tag{AB.37}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function s_vs (g, v1, k1, phi2, k2) result (phi)
  complex(kind=default), intent(in) :: g, phi2
  type(vector), intent(in) :: v1
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: phi
  phi = g * ((k1 + 2*k2) * v1) * phi2
  end function s_vs
```

AB.9.4 Transversal Scalar-Vector Coupling

⟨*Declaration of couplings*⟩+≡
```
  public :: s_vv_t, v_sv_t
```

$$phi(k_1+k_2) = g((V_1(k_1)V_2(k_2))(k_1k_2) - (V_1(k_1)k_2)(V_2(k_2)k_1)) \tag{AB.38}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function s_vv_t (g, v1, k1, v2, k2) result (phi)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: phi
  phi = g * ((v1*v2) * (k1*k2) - (v1*k2) * (v2*k1))
  end function s_vv_t
```

$$V_1^\mu(k_\phi+k_V) = gphi(((k_\phi+k_V)k_V)V_2^\mu - (k_\phi+k_V)V_2)k_V^\mu) \tag{AB.39}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_sv_t (g, phi, kphi,v, kv) result (vout)
  complex(kind=default), intent(in) :: g, phi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: kv, kphi
  type(momentum) :: kout
  type(vector)  :: vout
  kout = - (kv + kphi)
  vout = g * phi * ((kout*kv) * v - (v * kout) * kv)
  end function v_sv_t
```

AB.9.5 Transversal TensorScalar-Vector Coupling

⟨*Declaration of couplings*⟩+≡
```
  public :: tphi_vv, tphi_vv_cf, v_tphiv, v_tphiv_cf
```

$$phi(k_1+k_2) = g(V_1(k_1)(k_1+k_2)) * (V_2(k_2)(k_1+k_2)) \tag{AB.40}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function tphi_vv (g, v1, k1, v2, k2) result (phi)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: phi
  type(momentum) :: k
  k = - (k1 + k2)
  phi = 2 * g * (v1*k) * (v2*k)
  end function tphi_vv
```

$$phi(k_1+k_2) = g((V_1(k_1)V_2(k_2))(k_1+k_2)^2) \tag{AB.41}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function tphi_vv_cf (g, v1, k1, v2, k2) result (phi)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: phi
  type(momentum) :: k
  k = - (k1 + k2)
  phi = - g/2 * (v1*v2) * (k*k)
  end function tphi_vv_cf
```

$$V_1^\mu(k_\phi+k_V) = gphi((k_\phi+k_V)V_2)(k_\phi+k_V)^\mu \tag{AB.42}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_tphiv (g, phi, kphi,v, kv) result (vout)
  complex(kind=default), intent(in) :: g, phi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: kv, kphi
  type(momentum) :: kout
  type(vector)  :: vout
  kout = - (kv + kphi)
  vout = 2 * g * phi * ((v * kout) * kout)
  end function v_tphiv
```

$$V_1^\mu(k_\phi+k_V) = gphi((k_\phi+k_V)(k_\phi+k_V))V_2^\mu \tag{AB.43}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_tphiv_cf (g, phi, kphi,v, kv) result (vout)
  complex(kind=default), intent(in) :: g, phi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: kv, kphi
  type(momentum) :: kout
  type(vector)  :: vout
  kout = - (kv + kphi)
  vout = -g/2 * phi * (kout*kout) * v
  end function v_tphiv_cf
```

AB.9.6 Triple Vector Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: tkv_vv, lkv_vv, tv_kvv, lv_kvv, kg_kgkg
  public :: t5kv_vv, l5kv_vv, t5v_kvv, l5v_kvv, kg5_kgkg, kg_kg5kg
  public :: dv_vv, v_dvv, dv_vv_cf, v_dvv_cf
```

$$V^\mu(k_1+k_2) = \mathrm{i}g(k_1-k_2)^\mu V_1^\nu(k_1)V_{2,\nu}(k_2) \tag{AB.44}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function tkv_vv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  v = (k1 - k2) * ((0, 1) * g * (v1*v2))
  end function tkv_vv
```

$$V^\mu(k_1+k_2) = \mathrm{i}g\epsilon^{\mu\nu\rho\sigma}(k_1-k_2)_\nu V_{1,\rho}(k_1)V_{2,\sigma}(k_2) \tag{AB.45}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function t5kv_vv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  type(vector) :: k
  k = k1 - k2
  v = (0, 1) * g * pseudo_vector (k, v1, v2)
  end function t5kv_vv
```

$$V^\mu(k_1+k_2) = \mathrm{i}g(k_1+k_2)^\mu V_1^\nu(k_1)V_{2,\nu}(k_2) \tag{AB.46}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function lkv_vv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  v = (k1 + k2) * ((0, 1) * g * (v1*v2))
  end function lkv_vv
```

$$V^\mu(k_1+k_2) = \mathrm{i}g\epsilon^{\mu\nu\rho\sigma}(k_1+k_2)_\nu V_{1,\rho}(k_1)V_{2,\sigma}(k_2) \tag{AB.47}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function l5kv_vv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  type(vector) :: k
  k = k1 + k2
  v = (0, 1) * g * pseudo_vector (k, v1, v2)
  end function l5kv_vv
```

$$V^\mu(k_1+k_2) = \mathrm{i}g(k_2-k)^\nu V_{1,\nu}(k_1)V_2^\mu(k_2) = \mathrm{i}g(2k_2+k_1)^\nu V_{1,\nu}(k_1)V_2^\mu(k_2) \tag{AB.48}$$

using $k=-k_1-k_2$

⟨*Implementation of couplings*⟩+≡

```
  pure function tv_kvv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  v = v2 * ((0, 1) * g * ((2*k2 + k1)*v1))
  end function tv_kvv
```

$$V^\mu(k_1+k_2) = \mathrm{i}g\epsilon^{\mu\nu\rho\sigma}(2k_2+k_1)_\nu V_{1,\rho}(k_1)V_{2,\sigma}(k_2) \tag{AB.49}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function t5v_kvv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  type(vector) :: k
  k = k1 + 2*k2
  v = (0, 1) * g * pseudo_vector (k, v1, v2)
  end function t5v_kvv
```

$$V^\mu(k_1+k_2) = -\mathrm{i}gk_1^\nu V_{1,\nu}(k_1)V_2^\mu(k_2) \tag{AB.50}$$

using $k=-k_1-k_2$

⟨*Implementation of couplings*⟩+≡

```
  pure function lv_kvv (g, v1, k1, v2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1
  type(vector) :: v
  v = v2 * ((0, -1) * g * (k1*v1))
  end function lv_kvv
```

$$V^\mu(k_1+k_2) = -\mathrm{i}g\epsilon^{\mu\nu\rho\sigma}k_{1,\nu}V_{1,\rho}(k_1)V_{2,\sigma}(k_2) \tag{AB.51}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function l5v_kvv (g, v1, k1, v2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1
  type(vector) :: v
  type(vector) :: k
  k = k1
  v = (0, -1) * g * pseudo_vector (k, v1, v2)
  end function l5v_kvv
```

$$A^\mu(k_1+k_2) = \mathrm{i}gk^\nu\Big(F_{1,\nu}{}^\rho(k_1)F_{2,\rho\mu}(k_2) - F_{1,\mu}{}^\rho(k_1)F_{2,\rho\nu}(k_2)\Big) \tag{AB.52}$$

with $k=-k_1-k_2$, i. e.

$$\begin{aligned}A^\mu(k_1+k_2) = -\mathrm{i}g\Big([(kk_2)(k_1A_2)-(k_1k_2)(kA_2)]A_1^\mu \\ +[(k_1k_2)(kA_1)-(kk_1)(k_2A_1)]A_2^\mu \\ +[(k_2A_1)(kA_2)-(kk_2)(A_1A_2)]k_1^\mu \\ +[(kk_1)(A_1A_2)-(kA_1)(k_1A_2)]k_2^\mu\Big)\end{aligned} \tag{AB.53}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function kg_kgkg (g, a1, k1, a2, k2) result (a)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: a1, a2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: a
  real(kind=default) :: k1k1, k2k2, k1k2, kk1, kk2
  complex(kind=default) :: a1a2, k2a1, ka1, k1a2, ka2
  k1k1 = k1 * k1
  k1k2 = k1 * k2
  k2k2 = k2 * k2
  kk1 = k1k1 + k1k2
  kk2 = k1k2 + k2k2
  k2a1 = k2 * a1
  ka1 = k2a1 + k1 * a1
  k1a2 = k1 * a2
  ka2 = k1a2 + k2 * a2
  a1a2 = a1 * a2
  a = (0, -1) * g * (   (kk2  * k1a2 - k1k2 * ka2 ) * a1 &
  + (k1k2 * ka1  - kk1  * k2a1) * a2 &
  + (ka2  * k2a1 - kk2  * a1a2) * k1 &
  + (kk1  * a1a2 - ka1  * k1a2) * k2 )
  end function kg_kgkg
```

$$A^\mu(k_1+k_2) = \mathrm{i}g\epsilon^{\mu\nu\rho\sigma}k_\nu F_{1,\rho}{}^\lambda(k_1)F_{2,\lambda\sigma}(k_2) \tag{AB.54}$$

with $k=-k_1-k_2$, i. e.

$$\begin{aligned}A^\mu(k_1+k_2) = -2\mathrm{i}g\epsilon^{\mu\nu\rho\sigma}k_\nu\Big((k_2A_1)k_{1,\rho}A_{2,\sigma}+(k_1A_2)A_{1,\rho}k_{2,\sigma} \\ -(A_1A_2)k_{1,\rho}k_{2,\sigma}-(k_1k_2)A_{1,\rho}A_{2,\sigma}\Big)\end{aligned} \tag{AB.55}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function kg5_kgkg (g, a1, k1, a2, k2) result (a)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: a1, a2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: a
  type(vector) :: kv, k1v, k2v
  kv = - k1 - k2
  k1v = k1
  k2v = k2
  a = (0, -2) * g * (   (k2*A1) * pseudo_vector (kv, k1v, a2 ) &
  + (k1*A2) * pseudo_vector (kv, A1 , k2v) &
  - (A1*A2) * pseudo_vector (kv, k1v, k2v) &
  - (k1*k2) * pseudo_vector (kv, a1 , a2 ) )
  end function kg5_kgkg
```

$$A^\mu(k_1+k_2) = \mathrm{i}gk_\nu\Big(\epsilon^{\mu\rho\lambda\sigma}F_{1,}{}^{\nu}{}_{\rho} - \epsilon^{\nu\rho\lambda\sigma}F_{1,}{}^{\mu}{}_{\rho}\Big)\frac{1}{2}F_{1,\lambda\sigma} \tag{AB.56}$$

with $k = -k_1 - k_2$, i. e.

$$A^\mu(k_1+k_2) = -\mathrm{i}g\Big(\epsilon^{\mu\rho\lambda\sigma}(kk_2)A_{2,\rho} - \epsilon^{\mu\rho\lambda\sigma}(kA_2)k_{2,\rho} - k_2^\mu\epsilon^{\nu\rho\lambda\sigma}k_nuA_{2,\rho} + A_2^\mu\epsilon^{\nu\rho\lambda\sigma}k_nuk_{2,\rho}\Big)k_{1,\lambda}A_{1,\sigma} \tag{AB.57}$$

This is not the most efficienct way of doing it: $\epsilon^{\mu\nu\rho\sigma}F_{1,\rho\sigma}$ should be cached!

⟨*Implementation of couplings*⟩+≡
```
  pure function kg_kg5kg (g, a1, k1, a2, k2) result (a)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: a1, a2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: a
  type(vector) :: kv, k1v, k2v
  kv = - k1 - k2
  k1v = k1
  k2v = k2
  a = (0, -1) * g * (   (kv*k2v) * pseudo_vector (a2 , k1v, a1) &
  - (kv*a2 ) * pseudo_vector (k2v, k1v, a1) &
  -  k2v * pseudo_scalar (kv, a2,  k1v, a1) &
  +  a2  * pseudo_scalar (kv, k2v, k1v, a1) )
  end function kg_kg5kg
```

$$V^\mu(k_1+k_2) = -g((k_1+k_2)V_1)V_2^\mu + ((k_1+k_2)V_2)V_1^\mu \tag{AB.58}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function dv_vv (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  type(vector) :: k
  k = -(k1 + k2)
  v = g * ((k * v1) * v2 + (k * v2) * v1)
  end function dv_vv
```

$$V^\mu(k_1+k_2) = \frac{g}{2}(V_1(k_1)V_2(k_2))(k_1+k_2)^\mu \tag{AB.59}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function dv_vv_cf (g, v1, k1, v2, k2) result (v)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: v
  type(vector) :: k
  k = -(k1 + k2)
  v = - g/2 * (v1 * v2) * k
  end function dv_vv_cf
```

$$V_1^\mu = g * (kV_2)V(k) + (VV_2)k \tag{AB.60}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_dvv (g, v, k, v2) result (v1)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v, v2
  type(momentum), intent(in) :: k
  type(vector) :: v1
  v1 = g * ((v * v2) * k + (k * v2) * v)
  end function v_dvv
```

$$V_1^\mu = -\frac{g}{2}(V(k)k)V_2^\mu \tag{AB.61}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_dvv_cf (g, v, k, v2) result (v1)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) ::  v, v2
  type(momentum), intent(in) :: k
  type(vector) :: v1
  v1 = - g/2 * (v * k) * v2
  end function v_dvv_cf
```

AB.10 Tensorvector - Scalar coupling

⟨*Declaration of couplings*⟩+≡
```
public :: dv_phi2,phi_dvphi, dv_phi2_cf, phi_dvphi_cf
```

$$V^\mu(k_1+k_2) = g*((k_1k_2+k_2k_2)k_1^\mu + (k_1k_2+k_1k_1)k_2^\mu)*phi_1(k_1)phi_2(k_2) \tag{AB.62}$$

⟨*Implementation of couplings*⟩+≡
```
pure function dv_phi2 (g, phi1, k1, phi2, k2) result (v)
complex(kind=default), intent(in) :: g, phi1, phi2
type(momentum), intent(in) :: k1, k2
type(vector) :: v
v = g * phi1 * phi2 * ( &
(k1 * k2 + k2 * k2 ) * k1 + &
(k1 * k2 + k1 * k1 ) * k2 )
end function dv_phi2
```

$$V^\mu(k_1+k_2) = -\frac{g}{2}*(k_1k_2)*(k_1+k_2)^\mu*phi_1(k_1)phi_2(k_2) \tag{AB.63}$$

⟨*Implementation of couplings*⟩+≡
```
pure function dv_phi2_cf (g, phi1, k1, phi2, k2) result (v)
complex(kind=default), intent(in) :: g, phi1, phi2
type(momentum), intent(in) :: k1, k2
type(vector) :: v
v = - g/2 * phi1 * phi2 * (k1 * k2) * (k1 + k2)
end function dv_phi2_cf
```

$$phi_1(k_1) = g*((k_1k_2+k_2k_2)(k_1*V(-k_1-k_2)) + (k_1k_2+k_1k_1)(k_2*V(-k_1-k_2)))*phi_2(k_2) \tag{AB.64}$$

⟨*Implementation of couplings*⟩+≡
```
pure function phi_dvphi (g, v, k, phi2, k2) result (phi1)
complex(kind=default), intent(in) :: g, phi2
type(vector), intent(in) :: v
type(momentum), intent(in) :: k, k2
complex(kind=default) :: phi1
type(momentum) :: k1
k1 = - (k + k2)
phi1 = g * phi2 * ( &
(k1 * k2 + k2 * k2 ) * ( k1 * V ) + &
(k1 * k2 + k1 * k1 ) * ( k2 * V ) )
end function phi_dvphi
```

$$phi_1(k_1) = -\frac{g}{2}*(k_1k_2)*((k_1+k_2)V(-k_1-k_2)) \tag{AB.65}$$

⟨*Implementation of couplings*⟩+≡
```
pure function phi_dvphi_cf (g, v, k, phi2, k2) result (phi1)
complex(kind=default), intent(in) :: g, phi2
type(vector), intent(in) :: v
type(momentum), intent(in) :: k, k2
complex(kind=default) :: phi1
type(momentum) :: k1
k1 = -(k + k2)
phi1 = - g/2 * phi2 * (k1 * k2)  * ((k1 + k2) * v)
end function phi_dvphi_cf
```

AB.11 Scalar-Vector Dim-5 Couplings

⟨*Declaration of couplings*⟩+≡
```
public :: phi_vv, v_phiv, phi_u_vv, v_u_phiv
```

⟨*Implementation of couplings*⟩+≡
```
pure function phi_vv (g, k1, k2, v1, v2) result (phi)
complex(kind=default), intent(in) :: g
type(momentum), intent(in) :: k1, k2
type(vector), intent(in) :: v1, v2
complex(kind=default) :: phi
phi = g * pseudo_scalar (k1, v1, k2, v2)
end function phi_vv
```

⟨*Implementation of couplings*⟩+≡
```
pure function v_phiv (g, phi, k1, k2, v) result (w)
complex(kind=default), intent(in) :: g, phi
type(vector), intent(in) :: v
type(momentum), intent(in) :: k1, k2
type(vector) :: w
w = g * phi * pseudo_vector (k1, k2, v)
end function v_phiv
```

⟨*Implementation of couplings*⟩+≡
```
pure function phi_u_vv (g, k1, k2, v1, v2) result (phi)
complex(kind=default), intent(in) :: g
type(momentum), intent(in) :: k1, k2
type(vector), intent(in) :: v1, v2
complex(kind=default) :: phi
phi = g * ((k1*v2)*((-(k1+k2))*v1) + &
(k2*v1)*((-(k1+k2))*v2) + &
(((k1+k2)*(k1+k2)) * (v1*v2)))
end function phi_u_vv
```

⟨*Implementation of couplings*⟩+≡
```
pure function v_u_phiv (g, phi, k1, k2, v) result (w)
complex(kind=default), intent(in) :: g, phi
type(vector), intent(in) :: v
type(momentum), intent(in) :: k1, k2
type(vector) :: w
w = g * phi * ((k1*v)*k2 + &
((-(k1+k2))*v)*k1 + &
((k1*k1)*v))
end function v_u_phiv
```

AB.12 Dim-6 Anoumalous Couplings with Higgs

⟨*Declaration of couplings*⟩+≡
```
public :: s_vv_6D, v_sv_6D, s_vv_6DP, v_sv_6DP, a_hz_D, h_az_D, z_ah_D, &
a_hz_DP, h_az_DP, z_ah_DP, h_hh_6
```

⟨*Implementation of couplings*⟩+≡
```
pure function s_vv_6D (g, v1, k1, v2, k2) result (phi)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
complex(kind=default) :: phi
phi =  g * (-(k1 * v1) * (k1 * v2) - (k2 * v1) * (k2 * v2) &
+ ((k1 * k1) + (k2 * k2)) * (v1 * v2))
end function s_vv_6D
```

⟨*Implementation of couplings*⟩+≡
```
pure function v_sv_6D (g, phi, kphi, v, kv) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: phi
type(vector), intent(in) :: v
type(momentum), intent(in) :: kphi, kv
type(vector) :: vout
vout = g * ( - phi * (kv * v) * kv - phi * ((kphi + kv) * v) * (kphi + kv) &
+ phi * (kv * kv) * v + phi * ((kphi + kv)*(kphi + kv)) * v)
end function v_sv_6D
```

⟨*Implementation of couplings*⟩+≡
```
pure function s_vv_6DP (g, v1, k1, v2, k2) result (phi)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
complex(kind=default) :: phi
phi = g * ( (-(k1+k2)*v1) * (k1*v2) - ((k1+k2)*v2) * (k2*v1) + &
((k1+k2)*(k1+k2))*(v1*v2) )
end function s_vv_6DP
```

⟨*Implementation of couplings*⟩+≡
```
  pure function v_sv_6DP (g, phi, kphi, v, kv) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: kphi, kv
  type(vector) :: vout
  vout = g * phi * ((-(kphi + kv)*v) * kphi + (kphi * v) * kv + &
  (kphi*kphi) * v )
  end function v_sv_6DP
```
⟨*Implementation of couplings*⟩+≡
```
  pure function a_hz_D (g, h1, k1, v2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * h1 * (((k1 + k2) * v2) * (k1 + k2) + &
  ((k1 + k2) * (k1 + k2)) * v2)
  end function a_hz_D
```
⟨*Implementation of couplings*⟩+≡
```
  pure function h_az_D (g, v1, k1, v2, k2) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: hout
  hout = g * ((k1 * v1) * (k1 * v2) + (k1 * k1) * (v1 * v2))
  end function h_az_D
```
⟨*Implementation of couplings*⟩+≡
```
  pure function z_ah_D (g, v1, k1, h2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * h2 * ((k1 * v1) * k1 + ((k1 * k1)) *v1)
  end function z_ah_D
```
⟨*Implementation of couplings*⟩+≡
```
  pure function a_hz_DP (g, h1, k1, v2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * ((- h1 * (k1 + k2) * v2) * (k1) &
  + h1 * ((k1 + k2) * (k1)) *v2)
  end function a_hz_DP
```
⟨*Implementation of couplings*⟩+≡
```
  pure function h_az_DP (g, v1, k1, v2, k2) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: hout
  hout = g * (- (k1 * v2) * ((k1 + k2) * v1) + (k1 * (k1 + k2)) * (v1 * v2))
  end function h_az_DP
```
⟨*Implementation of couplings*⟩+≡
```
  pure function z_ah_DP (g, v1, k1, h2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * h2* ((k2 * v1) * k1 - (k1 * k2) * v1)
  end function z_ah_DP
```

⟨*Implementation of couplings*⟩+≡
```
  pure function h_hh_6 (g, h1, k1, h2, k2) result (hout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1, h2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: hout
  hout =  g * ((k1* k1) + (k2 * k2) + (k1* k2)) * h1 * h2
  end function h_hh_6
```

AB.13 Dim-6 Anoumalous Couplings without Higgs

⟨*Declaration of couplings*⟩+≡
```
  public :: g_gg_13, g_gg_23, g_gg_6, kg_kgkg_i
```
⟨*Implementation of couplings*⟩+≡
```
  pure function g_gg_23 (g, v1, k1, v2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * (v1 * (-2*(k1*v2)) + v2 * (2*k2 * v1) + (k1 - k2) * (v1*v2))
  end function g_gg_23
```
⟨*Implementation of couplings*⟩+≡
```
  pure function g_gg_13 (g, v1, k1, v2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * (v1 * (2*(k1 + k2)*v2) - v2 * ((k1 + 2*k2) * v1) + 2*k2 * (v1 * v2))
  end function g_gg_13
```
⟨*Implementation of couplings*⟩+≡
```
  pure function g_gg_6 (g, v1, k1, v2, k2) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: vout
  vout = g * &
  ( k1 * ((-(k1 + k2) * v2) * (k2 * v1) + ((k1 + k2) * k2) * (v1 * v2)) &
  + k2 * (((k1 + k2) * v1) * (k1 * v2) - ((k1 + k2) * k1) * (v1 * v2)) &
  + v1 * (-((k1 + k2) * k2) * (k1 * v2) + (k1 * k2) * ((k1 + k2) * v2)) &
  + v2 * (((k1 + k2) * k1) * (k2 * v1) - (k1 * k2) * ((k1 + k2) * v1)))
  end function g_gg_6
```
⟨*Implementation of couplings*⟩+≡
```
  pure function kg_kgkg_i (g, a1, k1, a2, k2) result (a)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: a1, a2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: a
  real(kind=default) :: k1k1, k2k2, k1k2, kk1, kk2
  complex(kind=default) :: a1a2, k2a1, ka1, k1a2, ka2
  k1k1 = k1 * k1
  k1k2 = k1 * k2
  k2k2 = k2 * k2
  kk1 = k1k1 + k1k2
  kk2 = k1k2 + k2k2
  k2a1 = k2 * a1
  ka1 = k2a1 + k1 * a1
  k1a2 = k1 * a2
  ka2 = k1a2 + k2 * a2
  a1a2 = a1 * a2
  a = (-1) * g * (   (kk2  * k1a2 - k1k2 * ka2 ) * a1 &
  + (k1k2 * ka1  - kk1  * k2a1) * a2 &
  + (ka2  * k2a1 - kk2  * a1a2) * k1 &
  + (kk1  * a1a2 - ka1  * k1a2) * k2 )
  end function kg_kgkg_i
```

AB.14 Dim-6 Anoumalous Couplings with AWW

⟨*Declaration of couplings*⟩+≡
```
public ::a_ww_DP, w_aw_DP, a_ww_DW
```
⟨*Implementation of couplings*⟩+≡
```
pure function a_ww_DP (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * ( - ((k1 + k2) * v2) * v1 + ((k1 + k2) * v1) * v2)
end function a_ww_DP
```
⟨*Implementation of couplings*⟩+≡
```
pure function w_aw_DP (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * ((k1 * v2) * v1 - (v1 * v2) * k1)
end function w_aw_DP
```
⟨*Implementation of couplings*⟩+≡
```
pure function a_ww_DW (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v1 * (- (4*k1 + 2*k2) * v2) &
+ v2 * ( (2*k1 + 4*k2) * v1) &
+ (k1 - k2) * (2*v1*v2))
end function a_ww_DW
```
⟨*Declaration of couplings*⟩+≡
```
public :: w_wz_DPW, z_ww_DPW, w_wz_DW, z_ww_DW, w_wz_D, z_ww_D
```
⟨*Implementation of couplings*⟩+≡
```
pure function w_wz_DPW (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v1 * (-(k1+k2)*v2 - k1*v2) + v2 * ((k1+k2)*v1) + k1 * (v1*v2))
end function w_wz_DPW
```
⟨*Implementation of couplings*⟩+≡
```
pure function z_ww_DPW (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (k1*(v1*v2) - k2*(v1*v2) - v1*(k1*v2) + v2*(k2*v1))
end function z_ww_DPW
```
⟨*Implementation of couplings*⟩+≡
```
pure function w_wz_DW (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v2 * (v1 * k2) - k2 * (v1 * v2))
end function w_wz_DW
```
⟨*Implementation of couplings*⟩+≡
```
pure function z_ww_DW (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v1 * ((-1)*(k1+k2) * v2) + v2 * ((k1+k2) * v1))
end function z_ww_DW
```

⟨*Implementation of couplings*⟩+≡
```
pure function w_wz_D (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v2 * (k2*v1) - k2 * (v1*v2))
end function w_wz_D
```
⟨*Implementation of couplings*⟩+≡
```
pure function z_ww_D (g, v1, k1, v2, k2) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(vector) :: vout
vout = g * (v1 * (- (k1 + k2) * v2) + v2 * ((k1 + k2) * v1))
end function z_ww_D
```

AB.15 Dim-6 Quartic Couplings

⟨*Declaration of couplings*⟩+≡
```
public :: hhhh_p2, a_hww_DPB, h_aww_DPB, w_ahw_DPB, a_hww_DPW, h_aww_DPW, &
w_ahw_DPW, a_hww_DW, h_aww_DW, w3_ahw_DW, w4_ahw_DW
```
⟨*Implementation of couplings*⟩+≡
```
pure function hhhh_p2 (g, h1, k1, h2, k2, h3, k3) result (hout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1, h2, h3
type(momentum), intent(in) :: k1, k2, k3
complex(kind=default) :: hout
hout = g * h1*h2*h3* (k1*k1 + k2*k2 +k3*k3 + k1*k3 + k1*k2 + k2*k3)
end function hhhh_p2
```
⟨*Implementation of couplings*⟩+≡
```
pure function a_hww_DPB (g, h1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h1 * (v3*((k1+k2+k3)*v2) - v2*((k1+k2+k3)*v3))
end function a_hww_DPB
```
⟨*Implementation of couplings*⟩+≡
```
pure function h_aww_DPB (g, v1, k1, v2, k2, v3, k3) result (hout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
complex(kind=default) :: hout
hout = g * ((k1 * v3) * (v1 * v2) - (k1 * v2) * (v1 * v3))
end function h_aww_DPB
```
⟨*Implementation of couplings*⟩+≡
```
pure function w_ahw_DPB (g, v1, k1, h2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h2
type(vector), intent(in) :: v1, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h2 * (v1 * (k1 * v3) - k1 * (v1 * v3))
end function w_ahw_DPB
```
⟨*Implementation of couplings*⟩+≡
```
pure function a_hww_DPW (g, h1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
```

```
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * (v3 * ((2*k1+k2+k3)*v2) - v2 * ((2*k1+k2+k3)*v3))
  end function a_hww_DPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_aww_DPW (g, v1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * ((-(2*k1+k2+k3)*v2)*(v1*v3)+((2*k1+k2+k3)*v3)*(v1*v2))
  end function h_aww_DPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w_ahw_DPW (g, v1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * ((k2 - k1) * (v1 * v3) + v1 * ((k1 - k2) * v3))
  end function w_ahw_DPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function a_hww_DW (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * ( v2 * (-(3*k1 + 4*k2 + 4*k3) * v3) &
  + v3 * ((3*k1 + 2*k2 + 4*k3) * v2)  &
  + (k2 - k3) *2*(v2 * v3))
  end function a_hww_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_aww_DW (g, v1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * ((v1*v2) * ((3*k1 - k2 - k3)*v3) &
  + (v1*v3) * ((-3*k1 - k2 + k3)*v2) &
  + (v2*v3) * (2*(k2-k3)*v1))
  end function h_aww_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w3_ahw_DW (g, v1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * (v1 * ((4*k1 + k2) * v3) &
  +v3 * (-2*(k1 + k2 + 2*k3) * v1) &
  +(-2*k1 + k2 + 2*k3) * (v1*v3))
  end function w3_ahw_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w4_ahw_DW (g, v1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * (v1 * (-(4*k1 + k2 + 2*k3) * v3) &
  + v3 * (2*(k1 + k2 + 2*k3) * v1) &
  +(4*k1 + k2) * (v1*v3))
  end function w4_ahw_DW
```

⟨*Declaration of couplings*⟩+≡
```
public ::a_aww_DW, w_aaw_DW, a_aww_W, w_aaw_W
```
⟨*Implementation of couplings*⟩+≡
```
pure function a_aww_DW (g, v1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * (2*v1*(v2*v3) - v2*(v1*v3) - v3*(v1*v2))
end function a_aww_DW
pure function w_aaw_DW (g, v1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * (2*v3*(v1*v2) - v2*(v1*v3) - v1*(v2*v3))
end function w_aaw_DW
pure function a_aww_W (g, v1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
!!! Recalculated WK 2018-08-24
type(momentum) :: k4
k4 = -(k1+k2+k3)
!!! negative sign (-g) causes expected gauge cancellation
vout = (-g) * ( &
+ (k1*v3)*(k3*v2)*v1 - (k3*v2)*(v1*v3)*k1 &
- (k1*k3)*(v2*v3)*v1 + (k3*v1)*(v2*v3)*k1 &
- (k1*v3)*(v1*v2)*k3 + (k1*v2)*(v1*v3)*k3 &
+ (k1*k3)*(v1*v2)*v3 - (k3*v1)*(k1*v2)*v3 &
+ (k3*v2)*(k4*v3)*v1 - (k3*v2)*(k4*v1)*v3 &
- (k3*k4)*(v2*v3)*v1 + (k4*v1)*(v2*v3)*k3 &
- (k3*v1)*(k4*v3)*v2 + (k3*v1)*(k4*v2)*v3 &
+ (k3*k4)*(v1*v3)*v2 - (k4*v2)*(v1*v3)*k3 &
+ (k1*v2)*(k2*v3)*v1 - (k2*v3)*(v1*v2)*k1 &
- (k1*k2)*(v2*v3)*v1 + (k2*v1)*(v2*v3)*k1 &
- (k1*v2)*(v1*v3)*k2 + (k1*v3)*(v1*v2)*k2 &
+ (k1*k2)*(v1*v3)*v2 - (k2*v1)*(k1*v3)*v2 &
+ (k2*v3)*(k4*v2)*v1 - (k2*v3)*(k4*v1)*v2 &
- (k2*k4)*(v2*v3)*v1 + (k4*v1)*(v2*v3)*k2 &
- (k2*v1)*(k4*v2)*v3 + (k2*v1)*(k4*v3)*v2 &
+ (k2*k4)*(v1*v2)*v3 - (k4*v3)*(v1*v2)*k2 &
)
!!! Original Version
!   vout = g * (v1*((-(k2+k3)*v2)*(k2*v3) + (-(k2+k3)*v3)*(k3*v2)) &
!        +v2*((-((k2-k3)*v1)*(k1+k2+k3)*v3) - (k1*v3)*(k2*v1) &
!        + ((k1+k2+k3)*v1)*(k2*v3)) &
!        +v3*(((k2-k3)*v1)*((k1+k2+k3)*v2) - (k1*v2)*(k3*v1) &
!        + ((k1+k2+k3)*v1)*(k3*v2)) &
!        +(v1*v2)*(((2*k1+k2+k3)*v3)*k2 - (k2*v3)*k1 -(k1*v3)*k3) &
!        +(v1*v3)*(((2*k1+k2+k3)*v2)*k3 - (k3*v2)*k1 - (k1*v2)*k3) &
!        +(v2*v3)*((-(k1+k2+k3)*v1)*(k2+k3) + ((k2+k3)*v1)*k1) &
!        +(-(k1+k2+k3)*k3 +k1*k2)*((v1*v3)*v2 - (v2*v3)*v1) &
!        +(-(k1+k2+k3)*k2 + k1*k3)*((v1*v2)*v3 - (v2*v3)*v1))
end function a_aww_W
pure function w_aaw_W (g, v1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
!!! Recalculated WK 2018-08-25
type(momentum) :: k4
k4 = -(k1+k2+k3)
!!! negative sign (-g) causes expected gauge cancellation
vout = (-g) * ( &
+ (k3*v1)*(k1*v2)*v3 - (k1*v2)*(v3*v1)*k3 &
- (k3*k1)*(v2*v1)*v3 + (k1*v3)*(v2*v1)*k3 &
- (k3*v1)*(v3*v2)*k1 + (k3*v2)*(v3*v1)*k1 &
+ (k3*k1)*(v3*v2)*v1 - (k1*v3)*(k3*v2)*v1 &
+ (k1*v2)*(k4*v1)*v3 - (k1*v2)*(k4*v3)*v1 &
- (k1*k4)*(v2*v1)*v3 + (k4*v3)*(v2*v1)*k1 &
- (k1*v3)*(k4*v1)*v2 + (k1*v3)*(k4*v2)*v1 &
+ (k1*k4)*(v3*v1)*v2 - (k4*v2)*(v3*v1)*k1 &
+ (k3*v2)*(k2*v1)*v3 - (k2*v1)*(v3*v2)*k3 &
- (k3*k2)*(v2*v1)*v3 + (k2*v3)*(v2*v1)*k3 &
- (k3*v2)*(v3*v1)*k2 + (k3*v1)*(v3*v2)*k2 &
+ (k3*k2)*(v3*v1)*v2 - (k2*v3)*(k3*v1)*v2 &
+ (k2*v1)*(k4*v2)*v3 - (k2*v1)*(k4*v3)*v2 &
- (k2*k4)*(v2*v1)*v3 + (k4*v3)*(v2*v1)*k2 &
- (k2*v3)*(k4*v2)*v1 + (k2*v3)*(k4*v1)*v2 &
+ (k2*k4)*(v3*v2)*v1 - (k4*v1)*(v3*v2)*k2 &
)
!!! Original Version
!   vout = g * (v1*((k1*v3)*(-(k1+k2+2*k3)*v2) + (k2*v3)*((k1+k2+k3)*v2) &
!        + (k1*v2)*((k1+k2+k3)*v3)) &
!        + v2*(((k1-k2)*v3)*((k1+k2+k3)*v1) - (k2*v3)*(k3*v1) &
!        + (k2*v1)*((k1+k2+k3)*v3)) &
!        + v3*((k1*v2)*(-(k1+k2)*v1) + (k2*v1)*(-(k1+k2)*v2)) &
!        + (v1*v2)*((k1+k2)*(-(k1+k2+k3)*v3) + k3*((k1+k2)*v3))&
!        + (v1*v3)*(-k2*(k3*v2) - k3*(k1*v2) + k1*((k1+k2+2*k3)*v2)) &
!        + (v2*v3)*(-k1*(k3*v1) - k3*(k2*v1) + k2*((k1+k2+2*k3)*v1)) &
!        + (-k2*(k1+k2+k3) + k1*k3)*(v1*(v2*v3) - v3*(v1*v2)) &
!        + (-k1*(k1+k2+k3) + k2*k3)*(v2*(v1*v3) - v3*(v1*v2)) )
end function w_aaw_W
```

⟨*Declaration of couplings*⟩+≡

```
public :: h_hww_D, w_hhw_D, h_hww_DP, w_hhw_DP, h_hvv_PB, v_hhv_PB
```

⟨*Implementation of couplings*⟩+≡

```
pure function h_hww_D (g, h1, k1, v2, k2, v3, k3) result (hout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
complex(kind=default) :: hout
hout = g * h1 * ((v2*v3)*((k2*k2)+(k3*k3)) - (k2*v2)*(k2*v3) &
- (k3*v2)*(k3*v3))
end function h_hww_D
```

⟨*Implementation of couplings*⟩+≡

```
pure function w_hhw_D (g, h1, k1, h2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1, h2
type(vector), intent(in) :: v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h1 * h2 * (v3 * ((k1+k2+k3)*(k1+k2+k3)+(k3*k3)) &
- (k1+k2+k3) * ((k1+k2+k3)*v3) - k3 * (k3*v3))
end function w_hhw_D
```

⟨*Implementation of couplings*⟩+≡

```
pure function h_hww_DP (g, h1, k1, v2, k2, v3, k3) result (hout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
complex(kind=default) :: hout
hout = g * h1 * (-((k2+k3)*v2)*(k2*v3) - &
((k2+k3)*v3)*(k3*v2)+ (v2*v3)*((k2+k3)*(k2+k3)))
end function h_hww_DP
```

⟨*Implementation of couplings*⟩+≡

```
pure function w_hhw_DP (g, h1, k1, h2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
```

```
  complex(kind=default), intent(in) :: h1, h2
  type(vector), intent(in) :: v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * h2 * (k3*((k1+k2)*v3) + (k1+k2)*(-(k1+k2+k3)*v3) &
  + v3*((k1+k2)*(k1+k2)))
  end function w_hhw_DP
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_hvv_PB (g, h1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * h1 * ((k2*v3)*(k3*v2) - (k2*k3)*(v2*v3))
  end function h_hvv_PB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function v_hhv_PB (g, h1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1, h2
  type(vector), intent(in) :: v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * h2 * ((-(k1+k2+k3)*v3)*k3 + ((k1+k2+k3)*k3)*v3)
  end function v_hhv_PB
```

⟨*Declaration of couplings*⟩+≡

```
  public :: a_hhz_D, h_ahz_D, z_ahh_D, a_hhz_DP, h_ahz_DP, z_ahh_DP, &
  a_hhz_PB, h_ahz_PB, z_ahh_PB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function a_hhz_D (g, h1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1, h2
  type(vector), intent(in) :: v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * h2 * ((k1+k2+k3) * ((k1+k2+k3)*v3) &
  - v3 * ((k1+k2+k3)*(k1+k2+k3)))
  end function a_hhz_D
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_ahz_D (g, v1, k1, h2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * h2 * ((k1*v1)*(k1*v3) - (k1*k1)*(v1*v3))
  end function h_ahz_D
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_ahh_D (g, v1, k1, h2, k2, h3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1
  complex(kind=default), intent(in) :: h2, h3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * h3 * ((k1*v1)*k1 - (k1*k1)*v1)
  end function z_ahh_D
```

⟨*Implementation of couplings*⟩+≡

```
  pure function a_hhz_DP (g, h1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1, h2
  type(vector), intent(in) :: v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * h2 * ((-(k1+k2+k3)*v3)*(k1+k2) + ((k1+k2+k3)*(k1+k2))*v3)
  end function a_hhz_DP
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_ahz_DP (g, v1, k1, h2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * h2 * ( (k1*v3)*(-(k1+k3)*v1) + (k1*(k1+k3))*(v1*v3) )
  end function h_ahz_DP
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_ahh_DP (g, v1, k1, h2, k2, h3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1
  complex(kind=default), intent(in) :: h2, h3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * h3 * (k1*((k2+k3)*v1) - v1*(k1*(k2+k3)))
  end function z_ahh_DP
```

⟨*Implementation of couplings*⟩+≡

```
  pure function a_hhz_PB (g, h1, k1, h2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1, h2
  type(vector), intent(in) :: v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * h2 * (k3*((k1+k2+k3)*v3) - v3*((k1+k2+k3)*k3))
  end function a_hhz_PB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_ahz_PB (g, v1, k1, h2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h2
  type(vector), intent(in) :: v1, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * h2 * ((-k1*v3)*(k3*v1) + (k1*k3)*(v1*v3))
  end function h_ahz_PB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_ahh_PB (g, v1, k1, h2, k2, h3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1
  complex(kind=default), intent(in) :: h2, h3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h2 * h3 * (k1*((k1+k2+k3)*v1) - v1*(k1*(k1+k2+k3)))
  end function z_ahh_PB
```

⟨*Declaration of couplings*⟩+≡

```
  public :: h_wwz_DW, w_hwz_DW, z_hww_DW, h_wwz_DPB, w_hwz_DPB, z_hww_DPB
  public :: h_wwz_DDPW, w_hwz_DDPW, z_hww_DDPW, h_wwz_DPW, w_hwz_DPW, z_hww_DPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_wwz_DW (g, v1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * (((k1-k2)*v3)*(v1*v2)-((2*k1+k2)*v2)*(v1*v3) + &
  ((k1+2*k2)*v1)*(v2*v3))
  end function h_wwz_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w_hwz_DW (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * ( v2*(-(k1+2*k2+k3)*v3) + v3*((2*k1+k2+2*k3)*v2) - &
  (k1 - k2 + k3)*(v2*v3))
  end function w_hwz_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_hww_DW (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * ((k2-k3)*(v2*v3) - v2*((2*k2+k3)*v3) + v3*((k2+2*k3)*v2))
  end function z_hww_DW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_wwz_DPB (g, v1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * ((k3*v1)*(v2*v3) - (k3*v2)*(v1*v3))
  end function h_wwz_DPB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w_hwz_DPB (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * (k3*(v2*v3) - v3*(k3*v2))
  end function w_hwz_DPB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_hww_DPB (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * (((k1+k2+k3)*v3)*v2 - ((k1+k2+k3)*v2)*v3)
  end function z_hww_DPB
```

⟨*Implementation of couplings*⟩+≡

```
  pure function h_wwz_DDPW (g, v1, k1, v2, k2, v3, k3) result (hout)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  complex(kind=default) :: hout
  hout = g * (((k1-k2)*v3)*(v1*v2)-((k1-k3)*v2)*(v1*v3)+((k2-k3)*v1)*(v2*v3))
  end function h_wwz_DDPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function w_hwz_DDPW (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
  type(vector), intent(in) :: v2, v3
  type(momentum), intent(in) :: k1, k2, k3
  type(vector) :: vout
  vout = g * h1 * ((-(k1+2*k2+k3)*v3)*v2 + ((k1+k2+2*k3)*v2)*v3 + &
  (v2*v3)*(k2-k3))
  end function w_hwz_DDPW
```

⟨*Implementation of couplings*⟩+≡

```
  pure function z_hww_DDPW (g, h1, k1, v2, k2, v3, k3) result (vout)
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: h1
```

```
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h1 * ((v2*v3)*(k2-k3) - ((k1+2*k2+k3)*v3) *v2 + &
((k1+k2+2*k3)*v2)*v3 )
end function z_hww_DDPW
```

⟨*Implementation of couplings*⟩+≡

```
pure function h_wwz_DPW (g, v1, k1, v2, k2, v3, k3) result (hout)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2, v3
type(momentum), intent(in) :: k1, k2, k3
complex(kind=default) :: hout
hout = g * (((k1-k2)*v3)*(v1*v2) + (-(2*k1+k2+k3)*v2)*(v1*v3) + &
((k1+2*k2+k3)*v1)*(v2*v3))
end function h_wwz_DPW
```

⟨*Implementation of couplings*⟩+≡

```
pure function w_hwz_DPW (g, h1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h1 * ((-(k1+2*k2+k3)*v3)*v2 + ((2*k1+k2+k3)*v2)*v3 + &
(v2*v3)*(k2-k1))
end function w_hwz_DPW
```

⟨*Implementation of couplings*⟩+≡

```
pure function z_hww_DPW (g, h1, k1, v2, k2, v3, k3) result (vout)
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: h1
type(vector), intent(in) :: v2, v3
type(momentum), intent(in) :: k1, k2, k3
type(vector) :: vout
vout = g * h1 * ((v2*v3)*(k2-k3) + ((k1-k2)*v3)*v2 + ((k3-k1)*v2)*v3)
end function z_hww_DPW
```

AB.16 Scalar3 Dim-5 Couplings

⟨*Declaration of couplings*⟩+≡

```
public :: phi_dim5s2
```

$$\phi_1(k_1) = g(k_2 \cdot k_3)\phi_2(k_2)\phi_3(k_3) \tag{AB.66}$$

⟨*Implementation of couplings*⟩+≡

```
pure function phi_dim5s2 (g, phi2, k2, phi3, k3) result (phi1)
complex(kind=default), intent(in) :: g, phi2, phi3
type(momentum), intent(in) :: k2, k3
complex(kind=default) :: phi1
phi1 = g * phi2 * phi3 * (k2 * k3)
end function phi_dim5s2
```

AB.17 Tensorscalar-Scalar Couplings

⟨*Declaration of couplings*⟩+≡

```
public :: tphi_ss, tphi_ss_cf, s_tphis, s_tphis_cf
```

$$\phi(k_1 + k_2) = 2g((k_1 \cdot k_2) + (k_1 \cdot k_1))((k_1 \cdot k_2) + (k_2 \cdot k_2))\phi_1(k_1)\phi_2(k_2) \tag{AB.67}$$

⟨*Implementation of couplings*⟩+≡

```
pure function tphi_ss (g, phi1, k1, phi2, k2) result (phi)
complex(kind=default), intent(in) :: g, phi1, phi2
type(momentum), intent(in) :: k1, k2
complex(kind=default) :: phi
phi = 2 * g * phi1 * phi2 * &
```

```
  ((k1 * k2)+ (k1 * k1)) * &
  ((k1 * k2)+ (k2 * k2))
  end function tphi_ss
```

$$\phi(k_1+k_2) = -g/2(k_1\cdot k_2)((k_1+k_2)\cdot(k_1+k_2))\phi_1(k_1)\phi_2(k_2) \tag{AB.68}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function tphi_ss_cf (g, phi1, k1, phi2, k2) result (phi)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2
  complex(kind=default) :: phi
  phi = - g/2 * phi1 * phi2 * &
  (k1 * k2) * &
  ((k1 + k2) * (k1 + k2))
  end function tphi_ss_cf
```

$$\phi_1(k_1) = 2g((k_1\cdot k_2)+(k_1\cdot k_1))((k_1\cdot k_2)+(k_2\cdot k_2))\phi(k_2-k_1)\phi_2(k_2) \tag{AB.69}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function s_tphis (g, phi, k, phi2, k2) result (phi1)
  complex(kind=default), intent(in) :: g, phi, phi2
  type(momentum), intent(in) :: k, k2
  complex(kind=default) :: phi1
  type(momentum) :: k1
  k1 = - ( k + k2)
  phi1 = 2 * g * phi * phi2 * &
  ((k1 * k2)+ (k1 * k1)) * &
  ((k1 * k2)+ (k2 * k2))
  end function s_tphis
```

$$\phi_1(k_1) = -g/2(k_1\cdot k_2)((k_1+k_2)\cdot(k_1+k_2))\phi(k_2-k_1)\phi_2(k_2) \tag{AB.70}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function s_tphis_cf (g, phi, k, phi2, k2) result (phi1)
  complex(kind=default), intent(in) :: g, phi, phi2
  type(momentum), intent(in) :: k, k2
  complex(kind=default) :: phi1
  type(momentum) :: k1
  k1 = - ( k + k2)
  phi1 = - g/2 * phi * phi2 * &
  (k1 * k2) * &
  ((k1 + k2) * (k1 + k2))
  end function s_tphis_cf
```

AB.18 Scalar2-Vector2 Dim-8 Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: phi_phi2v_1, v_phi2v_1, phi_phi2v_2, v_phi2v_2
```

$$\phi_2(k_2) = g\left((k_1\cdot V_1)\,(k_2\cdot V_2) + (k_1\cdot V_1)\,(k_1\cdot V_2)\right)\phi_1(k_1) \tag{AB.71}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function phi_phi2v_1 (g, phi1, k1, v1, k_v1, v2, k_v2) result (phi2)
  complex(kind=default), intent(in) :: g, phi1
  type(momentum), intent(in) :: k1, k_v1, k_v2
  type(momentum) :: k2
  type(vector), intent(in) :: v1, v2
  complex(kind=default) :: phi2
  k2 = - k1 - k_v1 - k_v2
  phi2 = g * phi1 * &
  ( (k1 * v1) * (k2 * v2) + (k1 * v2) * (k2 * v1) )
  end function phi_phi2v_1
```

$$V_2^\mu = g\left(k_1^\mu\,(k_2\cdot V_1) + k_2^\mu\,(k_1\cdot V_1)\right)\phi_1(k_1)\phi_2(k_2) \tag{AB.72}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_phi2v_1 (g, phi1, k1, phi2, k2, v1) result (v2)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2
  type(vector), intent(in) :: v1
  type(vector) :: v2
  v2 = g * phi1 * phi2 * &
  ( k1  * (k2 * v1) + k2 * (k1 * v1) )
  end function v_phi2v_1
```

$$\phi_2(k_2) = g\,(k_1 \cdot k_2)\,(V_1 \cdot V_2)\,\phi_1(k_1) \tag{AB.73}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function phi_phi2v_2 (g, phi1, k1, v1,k_v1, v2, k_v2) result (phi2)
  complex(kind=default), intent(in) :: g, phi1
  type(momentum), intent(in) :: k1, k_v1, k_v2
  type(vector), intent(in) :: v1, v2
  type(momentum) :: k2
  complex(kind=default) :: phi2
  k2 = - k1 - k_v1 - k_v2
  phi2 = g * phi1 * (k1 * k2) * (v1 * v2)
  end function phi_phi2v_2
```

$$V_2^\mu = g V_1^\mu\,(k_1 \cdot k_2)\,\phi_1\phi_2 \tag{AB.74}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_phi2v_2 (g, phi1, k1, phi2, k2, v1) result (v2)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2
  type(vector), intent(in) :: v1
  type(vector) :: v2
  v2 = g * phi1 * phi2 * &
  ( k1  * k2 ) * v1
  end function v_phi2v_2
```

AB.19 Scalar4 Dim-8 Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: s_dim8s3
```

$$\phi(k_1) = g\,[(k_1 \cdot k_2)\,(k_3 \cdot k_4) + (k_1 \cdot k_3)\,(k_2 \cdot k_4) + (k_1 \cdot k_4)\,(k_2 \cdot k_3)]\,\phi_2(k_2)\phi_3(k_3)\phi_4(k_4) \tag{AB.75}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function s_dim8s3 (g, phi2, k2, phi3, k3, phi4, k4) result (phi1)
  complex(kind=default), intent(in) :: g, phi2, phi3, phi4
  type(momentum), intent(in) :: k2, k3, k4
  type(momentum) :: k1
  complex(kind=default) :: phi1
  k1 = - k2 - k3 - k4
  phi1 = g * ( (k1 * k2) * (k3 * k4) + (k1 * k3) * (k2 * k4) &
  + (k1 * k4) * (k2 * k3) ) * phi2 * phi3 * phi4
  end function s_dim8s3
```

AB.20 Mixed Scalar2-Vector2 Dim-8 Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: phi_phi2v_m_0, v_phi2v_m_0, phi_phi2v_m_1, v_phi2v_m_1, phi_phi2v_m_7, v_phi2v_m_7
```

$$\phi_2(k_2) = g\,((V_1 \cdot k_{V_2})\,(V_2 \cdot k_{V_1})\,(k_1 \cdot k_2) - ((V_1 \cdot V_2)\,(k_{V_1} \cdot k_{V_2})\,(k_1 \cdot k_2))\,\phi_1(k_1) \tag{AB.76}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function phi_phi2v_m_0 (g, phi1, k1, v1, k_v1, v2, k_v2) result (phi2)
  complex(kind=default), intent(in) :: g, phi1
  type(momentum), intent(in) :: k1, k_v1, k_v2
  type(momentum) :: k2
  type(vector), intent(in) :: v1, v2
  complex(kind=default) :: phi2
  k2 = - k1 - k_v1 - k_v2
  phi2 = g * phi1 * &
  ( (v1 * k_v2) * (v2 * k_v1) * (k1 * k2) &
  - (v1 * v2) * (k_v1 * k_v2) * (k1 * k2) )
  end function phi_phi2v_m_0
```

$$V_2^\mu = g\left(k_{V_1}^\mu\left(V_1 \cdot k_{V_2}\right)\left(k_1 \cdot k_2\right) - V_1^\mu\left(k_{V_1} \cdot k_{V_2}\right)\left(k_1 \cdot k_2\right)\right)\phi_1(k_1)\phi_2(k_2)) \tag{AB.77}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_phi2v_m_0 (g, phi1, k1, phi2, k2, v1, k_v1) result (v2)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2, k_v1
  type(vector), intent(in) :: v1
  type(momentum) :: k_v2
  type(vector) :: v2
  k_v2 = - k_v1 - k1 - k2
  v2 = g * phi1 * phi2 * &
  ( k_v1 * (v1 *  k_v2) * (k1 * k2) &
  - v1 * (k_v2 * k_v1) * (k1 * k2) )
  end function v_phi2v_m_0
```

$$\phi_2(k_2) = g\left(\left(V_1 \cdot V_2\right)\left(k_1 \cdot k_{V_2}\right)\left(k_2 \cdot k_{V_1}\right) + \left(\left(V_1 \cdot V_2\right)\left(k_1 \cdot k_{V_1}\right)\left(k_2 \cdot k_{V_2}\right) + \left(\left(V_1 \cdot k_2\right)\left(V_2 \cdot k_1\right)\left(k_{V_1} \cdot k_{V_2}\right) + \left(\left(V_1 \cdot k_1\right)\left(V_2 \cdot k_2\right)\left(k_V \right.\right.\right.\right.\right. \tag{AB.78}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function phi_phi2v_m_1 (g, phi1, k1, v1, k_v1, v2, k_v2) result (phi2)
  complex(kind=default), intent(in) :: g, phi1
  type(momentum), intent(in) :: k1, k_v1, k_v2
  type(momentum) :: k2
  type(vector), intent(in) :: v1, v2
  complex(kind=default) :: phi2
  k2 = - k1 - k_v1 - k_v2
  phi2 = g * phi1 * &
  ( (v1 * v2) * (k1 * k_v2) * (k2 * k_v1) &
  + (v1 * v2) * (k1 * k_v1) * (k2 * k_v2) &
  + (v1 * k2) * (v2 * k1) * (k_v1 * k_v2) &
  + (v1 * k1) * (v2 * k2) * (k_v1 * k_v2) &
  - (v1 * k_v2) * (v2 * k2) * (k1 * k_v1) &
  - (v1 * k2) * (v2 * k_v1) * (k1 * k_v2) &
  - (v1 * k_v2) * (v2 * k1) * (k2 * k_v1) &
  - (v1 * k1) * (v2 * k_v1) * (k2 * k_v2) )
  end function phi_phi2v_m_1
```

$$V_2^\mu = g\left(k_1^\mu\left(V_1 \cdot k_2\right)\left(k_{V_1} \cdot k_{V_2}\right) + k_2^\mu\left(V_1 \cdot k_1\right)\left(k_{V_1} \cdot k_{V_2}\right) + V_1^\mu\left(k_{V_1} \cdot k_1\right)\left(k_{V_2} \cdot k_2\right) + V_1^\mu\left(k_{V_1} \cdot k_2\right)\left(k_{V_2} \cdot k_1\right) - k_1^\mu\left(V_1 \cdot k_{V_2}\right)\left(k_{V_1}\right.\right. \tag{AB.79}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_phi2v_m_1 (g, phi1, k1, phi2, k2, v1, k_v1) result (v2)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2, k_v1
  type(vector), intent(in) :: v1
  type(momentum) :: k_v2
  type(vector) :: v2
  k_v2 = - k_v1 - k1 - k2
  v2 = g * phi1 * phi2 * &
  ( k1 * (v1 * k2) * (k_v1 * k_v2) &
  + k2 * (v1 * k1) * (k_v1 * k_v2) &
  + v1 * (k_v1 * k1) * (k_v2 * k2) &
  + v1 * (k_v1 * k2) * (k_v2 * k1) &
  - k1 * (v1 * k_v2) * (k_v1 * k2) &
  - k2 * (v1 * k_v2) * (k_v1 * k1) &
  - k_v1 * (v1 * k1) * (k_v2 * k2) &
  - k_v1 * (v1 * k2) * (k_v2 * k1) )
  end function v_phi2v_m_1
```

$$\phi_2(k_2) = g\left(\left(V_1 \cdot k_{V_2}\right)\left(k_1 \cdot V_2\right)\left(k_2 \cdot k_{V_1}\right) + \left(\left(V_1 \cdot k_{V_2}\right)\left(k_1 \cdot k_{V_1}\right)\left(k_2 \cdot k_{V_2}\right) + \left(\left(V_1 \cdot k_1\right)\left(V_2 \cdot k_{V_1}\right)\left(k_2 \cdot k_{V_2}\right) + \left(\left(V_1 \cdot k_2\right)\left(V_2 \cdot k_{V_1}\right)\left(\right.\right.\right.\right.\right. \tag{AB.80}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function phi_phi2v_m_7 (g, phi1, k1, v1, k_v1, v2, k_v2) result (phi2)
  complex(kind=default), intent(in) :: g, phi1
  type(momentum), intent(in) :: k1, k_v1, k_v2
  type(momentum) :: k2
  type(vector), intent(in) :: v1, v2
  complex(kind=default) :: phi2
  k2 = - k1 - k_v1 - k_v2
  phi2 = g * phi1 * &
  ( (v1 * k_v2) * (k1 * v2) * (k2 * k_v1) &
  + (v1 * k_v2) * (k1 * k_v1) * (k2 * v2) &
  + (v1 * k1) * (v2 * k_v1) * (k2 * k_v2) &
  + (v1 * k2) * (v2 * k_v1) * (k1 * k_v2) &
  - (v1 * v2) * (k1 * k_v2) * (k2 * k_v1) &
  - (v1 * v2) * (k1 * k_v1) * (k2 * k_v2) &
  - (v1 * k2) * (v2 * k1) * (k_v1 * k_v2) &
  - (v1 * k1) * (v2 * k2) * (k_v1 * k_v2) )
  end function phi_phi2v_m_7
```

$$V_2^\mu = g \left(k_1^\mu \left(V_1 \cdot k_{V_2}\right) \left(k_2 \cdot k_{V_1}\right) + k_2^\mu \left(V_1 \cdot k_{V_2}\right) \left(k_1 \cdot k_{V_1}\right) + k_{V_1}^\mu \left(V_1 \cdot k_1\right) \left(k_2 \cdot k_{V_2}\right) + k_{V_1}^\mu \left(V_1 \cdot k_2\right) \left(k_1 \cdot k_{V_2}\right) - k_1^\mu \left(V_1 \cdot k_2\right) \left(k_{V_1} \cdot \right.\right. \tag{AB.81}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_phi2v_m_7 (g, phi1, k1, phi2, k2, v1, k_v1) result (v2)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2, k_v1
  type(vector), intent(in) :: v1
  type(momentum) :: k_v2
  type(vector) :: v2
  k_v2 = - k_v1 - k1 - k2
  v2 = g * phi1 * phi2 * &
  ( k1 * (v1 * k_v2) * (k2 * k_v1) &
  + k2 * (v1 * k_v2) * (k1 * k_v1) &
  + k_v1 * (v1 * k1) * (k2 * k_v2) &
  + k_v1 * (v1 * k2) * (k1 * k_v2) &
  - k1 * (v1 * k2) * (k_v1 * k_v2) &
  - k2 * (v1 * k1) * (k_v1 * k_v2) &
  - v1 * (k1 * k_v2) * (k2 * k_v1) &
  - v1 * (k1 * k_v1) * (k2 * k_v2) )
  end function v_phi2v_m_7
```

AB.21 Transversal Gauge4 Dim-8 Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: g_dim8g3_t_0, g_dim8g3_t_1, g_dim8g3_t_2
```

$$V_1^\mu = g\left[k_2^\mu \left(k_1 \cdot V_2\right) - V_2^\mu \left(k_1 \cdot k_2\right)\right]\left[\left(k_3 \cdot V_4\right)\left(k_4 \cdot V_3\right) - \left(V_3 \cdot V_4\right)\left(k_3 \cdot k_4\right)\right] \tag{AB.82}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function g_dim8g3_t_0 (g, v2, k2, v3, k3, v4, k4) result (v1)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v2, v3, v4
  type(momentum), intent(in) :: k2, k3, k4
  type(vector) :: v1
  type(momentum) :: k1
  k1 = - k2 - k3 - k4
  v1 = g * (k2 * (k1 * v2) - v2 * (k1 * k2)) &
  * ((k3 * v4) * (k4 * v3) - (v3 * v4) * (k3 * k4))
  end function g_dim8g3_t_0
```

$$V_1^\mu = g\left[k_2^\mu \left(k_1 \cdot V_2\right) - V_2^\mu \left(k_1 \cdot k_2\right)\right]\left[\left(k_3 \cdot V_4\right)\left(k_4 \cdot V_3\right) - \left(V_3 \cdot V_4\right)\left(k_3 \cdot k_4\right)\right] \tag{AB.83}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function g_dim8g3_t_1 (g, v2, k2, v3, k3, v4, k4) result (v1)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v2, v3, v4
  type(momentum), intent(in) :: k2, k3, k4
  type(vector) :: v1
  type(momentum) :: k1
  k1 = - k2 - k3 - k4
  v1 = g * (v3 * (v2 * k4) * (k1 * k3) * (k2 * v4) &
  + v4 * (v2 * k3) * (k1 * k4) * (k2 * v3) &
  + k3 * (v2 * v4) * (k1 * v3) * (k2 * k4) &
  + k4 * (v2 * v3) * (k1 * v4) * (k2 * k3) &
  - v3 * (v2 * v4) * (k1 * k3) * (k2 * k4) &
  - v4 * (v2 * v3) * (k1 * k4) * (k2 * k3) &
  - k3 * (v2 * k4) * (k1 * v3) * (k2 * v4) &
  - k4 * (v2 * k3) * (k1 * v4) * (k2 * v3))
  end function g_dim8g3_t_1
```

$$V_1^\mu = g\left[k_2^\mu\left(V_2\cdot k_3\right)\left(V_3\cdot k_4\right)\left(V_4\cdot k_1\right) + k_3^\mu\left(V_2\cdot k_1\right)\left(V_3\cdot k_4\right)\left(V_4\cdot k_2\right) + k_2^\mu\left(V_2\cdot k_4\right)\left(V_3\cdot k_1\right)\left(V_4\cdot k_3\right) + k_4^\mu\left(V_2\cdot k_1\right)\left(V_3\cdot k_2\right)\left(V_4\right.\right. \tag{AB.84}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function g_dim8g3_t_2 (g, v2, k2, v3, k3, v4, k4) result (v1)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v2, v3, v4
  type(momentum), intent(in) :: k2, k3, k4
  type(vector) :: v1
  type(momentum) :: k1
  k1 = - k2 - k3 - k4
  v1 = g * (k2 * (v2 * k3) * (v3 * k4) * (v4 * k1) &
  + k3 * (v2 * k1) * (v3 * k4) * (v4 * k2) &
  + k2 * (v2 * k4) * (v3 * k1) * (v4 * k3) &
  + k4 * (v2 * k1) * (v3 * k2) * (v4 * k3) &
  + k4 * (v2 * k3) * (v3 * v4) * (k1 * k2) &
  + k3 * (v2 * k4) * (v3 * v4) * (k1 * k2) &
  - k3 * (v2 * v4) * (v3 * k4) * (k1 * k2) &
  - v4 * (v2 * k3) * (v3 * k4) * (k1 * k2) &
  - k4 * (v2 * v3) * (v4 * k3) * (k1 * k2) &
  - v3 * (v2 * k4) * (v4 * k3) * (k1 * k2) &
  - k2 * (v2 * k4) * (v3 * v4) * (k1 * k3) &
  + k2 * (v2 * v4) * (v3 * k4) * (k1 * k3) &
  - v2 * (v3 * k4) * (v4 * k2) * (k1 * k3) &
  - k2 * (v2 * k3) * (v3 * v4) * (k1 * k4) &
  + k2 * (v2 * v3) * (v4 * k3) * (k1 * k4) &
  - v2 * (v3 * k2) * (v4 * k3) * (k1 * k4) &
  - k4 * (v2 * k1) * (v3 * v4) * (k2 * k3) &
  + v4 * (v2 * k1) * (v3 * k4) * (k2 * k3) &
  - v2 * (v3 * k4) * (v4 * k1) * (k2 * k3) &
  + v2 * (v3 * v4) * (k1 * k4) * (k2 * k3) &
  - k3 * (v2 * k1) * (v3 * v4) * (k2 * k4) &
  + v3 * (v2 * k1) * (v4 * k3) * (k2 * k4) &
  - v2 * (v3 * k1) * (v4 * k3) * (k2 * k4) &
  + v2 * (v3 * v4) * (k1 * k3) * (k2 * k4) &
  - k2 * (v2 * v4) * (v3 * k1) * (k3 * k4) &
  - v4 * (v2 * k1) * (v3 * k2) * (k3 * k4) &
  - k2 * (v2 * v3) * (v4 * k1) * (k3 * k4) &
  + v2 * (v3 * k2) * (v4 * k1) * (k3 * k4) &
  - v3 * (v2 * k1) * (v4 * k2) * (k3 * k4) &
  + v2 * (v3 * k1) * (v4 * k2) * (k3 * k4) &
  + v4 * (v2 * v3) * (k1 * k2) * (k3 * k4) &
  + v3 * (v2 * v4) * (k1 * k2) * (k3 * k4))
  end function g_dim8g3_t_2
```

AB.22 Mixed Gauge4 Dim-8 Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: g_dim8g3_m_0, g_dim8g3_m_1, g_dim8g3_m_7
```

$$V_1^\mu = g_1\left[V_2^\mu\left(V_3\cdot V_4\right)\left(k_1\cdot k_2\right) - k_2^\mu\left(V_2\cdot k_1\right)\left(V_3\cdot V_4\right)\right] + g_2\left[V_2^\mu\left(V_3\cdot V_4\right)\left(k_3\cdot k_4\right) - V_2^\mu\left(V_3\cdot k_4\right)\left(V_4\cdot k_3\right)\right] \tag{AB.85}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function g_dim8g3_m_0 (g1, g2, v2, k2, v3, k3, v4, k4) result (v1)
  complex(kind=default), intent(in) :: g1, g2
  type(vector), intent(in) :: v2, v3, v4
  type(momentum), intent(in) :: k2, k3, k4
  type(vector) :: v1
  type(momentum) :: k1
  k1 = - k2 - k3 - k4
  v1 = g1 * (v2 * (v3 * v4) * (k1 * k2)  &
  - k2 * (v2 * k1) * (v3 * v4)) &
  + g2 * (v2 * (v3 * v4) * (k3 * k4)  &
  - v2 * (v3 * k4) * (v4 * k3))
  end function g_dim8g3_m_0
```

$$V_1^\mu = g_1 \left[k_2^\mu \left(V_2 \cdot V_4\right)\left(V_3 \cdot k_1\right) + V_4^\mu \left(V_2 \cdot k_1\right)\left(V_3 \cdot k_2\right) + k_2^\mu \left(V_2 \cdot V_3\right)\left(V_4 \cdot k_1\right) + V_3^\mu \left(V_2 \cdot k_1\right)\left(V_4 \cdot k_2\right) - V_2^\mu \left(V_3 \cdot k_2\right)\left(V_4 \cdot k_1\right) - \right. \tag{AB.86}$$

⟨*Implementation of couplings*⟩+≡

```
pure function g_dim8g3_m_1 (g1, g2, v2, k2, v3, k3, v4, k4) result (v1)
complex(kind=default), intent(in) :: g1, g2
type(vector), intent(in) :: v2, v3, v4
type(momentum), intent(in) :: k2, k3, k4
type(vector) :: v1
type(momentum) :: k1
k1 = - k2 - k3 - k4
v1 = g1 * (k2 * (v2 * v4) * (v3 * k1)  &
+ v4 * (v2 * k1) * (v3 * k2)  &
+ k2 * (v2 * v3) * (v4 * k1)  &
+ v3 * (v2 * k1) * (v4 * k2)  &
- v2 * (v3 * k2) * (v4 * k1)  &
- v2 * (v3 * k1) * (v4 * k2)  &
- v4 * (v2 * v3) * (k1 * k2)  &
- v3 * (v2 * v4) * (k1 * k2)) &
+ g2 * (k3 * (v2 * v4) * (v3 * k4)  &
- k4 * (v2 * k3) * (v3 * v4)  &
- k3 * (v2 * k4) * (v3 * v4)  &
+ v4 * (v2 * k3) * (v3 * k4)  &
+ k4 * (v2 * v3) * (v4 * k3)  &
+ v3 * (v2 * k4) * (v4 * k3)  &
- v4 * (v2 * v3) * (k3 * k4)  &
- v3 * (v2 * v4) * (k3 * k4))
end function g_dim8g3_m_1
```

$$V_1^\mu = g_1 \left[V_2^\mu \left(V_3 \cdot k_2\right)\left(V_4 \cdot k_1\right) + V_2^\mu \left(V_4 \cdot k_1\right)\left(V_4 \cdot k_2\right) + V_4^\mu \left(V_2 \cdot V_3\right)\left(k_1 \cdot k_2\right) + V_3^\mu \left(V_2 \cdot V_4\right)\left(k_1 \cdot k_2\right) - k_2^\mu \left(V_2 \cdot V_4\right)\left(V_3 \cdot k_1\right) - \right. \tag{AB.87}$$

⟨*Implementation of couplings*⟩+≡

```
pure function g_dim8g3_m_7 (g1, g2, g3, v2, k2, v3, k3, v4, k4) result (v1)
complex(kind=default), intent(in) :: g1, g2, g3
type(vector), intent(in) :: v2, v3, v4
type(momentum), intent(in) :: k2, k3, k4
type(vector) :: v1
type(momentum) :: k1
k1 = - k2 - k3 - k4
v1 = g1 * (v2 * (v3 * k2) * (v4 * k1)  &
+ v2 * (v3 * k1) * (v4 * k2)  &
+ v4 * (v2 * v3) * (k1 * k2)  &
+ v3 * (v2 * v4) * (k1 * k2)  &
- k2 * (v2 * v4) * (v3 * k1)  &
- v4 * (v2 * k1) * (v3 * k2)  &
- k2 * (v2 * v3) * (v4 * k1)  &
- v3 * (v2 * k1) * (v4 * k2)) &
+ g2 * (k3 * (v2 * k1) * (v3 * v4)  &
+ k4 * (v2 * k1) * (v3 * v4)  &
+ k2 * (v2 * k3) * (v3 * v4)  &
+ k2 * (v2 * k4) * (v3 * v4)  &
+ v4 * (v2 * k4) * (v3 * k1)  &
+ k4 * (v2 * v4) * (v3 * k2)  &
+ v3 * (v2 * k3) * (v4 * k1)  &
+ v2 * (v3 * k4) * (v4 * k1)  &
+ k3 * (v2 * v3) * (v4 * k2)  &
+ v2 * (v3 * k4) * (v4 * k2)  &
+ v2 * (v3 * k1) * (v4 * k3)  &
+ v2 * (v3 * k2) * (v4 * k3)  &
+ v4 * (v2 * v3) * (k1 * k3)  &
+ v3 * (v2 * v4) * (k1 * k4)  &
+ v3 * (v2 * v4) * (k2 * k3)  &
+ v4 * (v2 * v3) * (k2 * k4)  &
- k4 * (v2 * v4) * (v3 * k1)  &
- v4 * (v2 * k3) * (v3 * k1)  &
- k3 * (v2 * v4) * (v3 * k2)  &
- v4 * (v2 * k4) * (v3 * k2)  &
- k2 * (v2 * v4) * (v3 * k4)  &
- v4 * (v2 * k1) * (v3 * k4)  &
- k3 * (v2 * v3) * (v4 * k1)  &
- v3 * (v2 * k4) * (v4 * k1)  &
- k4 * (v2 * v3) * (v4 * k2)  &
- v3 * (v2 * k3) * (v4 * k2)  &
- k2 * (v2 * v3) * (v4 * k3)  &
- v3 * (v2 * k1) * (v4 * k3)  &
- v2 * (v3 * v4) * (k1 * k3)  &
- v2 * (v3 * v4) * (k1 * k4)  &
- v2 * (v3 * v4) * (k2 * k3)  &
- v2 * (v3 * v4) * (k2 * k4)) &
+ g3 * (k4 * (v2 * k3) * (v3 * v4)  &
+ k3 * (v2 * k4) * (v3 * v4)  &
+ v4 * (v2 * v3) * (k3 * k4)  &
+ v3 * (v2 * v4) * (k3 * k4)  &
- k3 * (v2 * v4) * (v3 * k4)  &
- v4 * (v2 * k3) * (v3 * k4)  &
- k4 * (v2 * v3) * (v4 * k3)  &
- v3 * (v2 * k4) * (v4 * k3))
end function g_dim8g3_m_7
```

AB.23 Graviton Couplings

⟨*Declaration of couplings*⟩+≡
```
public :: s_gravs, v_gravv, grav_ss, grav_vv
```

⟨*Implementation of couplings*⟩+≡
```
pure function s_gravs (g, m, k1, k2, t, s) result (phi)
complex(kind=default), intent(in) :: g, s
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k1, k2
type(tensor), intent(in) :: t
complex(kind=default) :: phi, t_tr
t_tr = t%t(0,0) - t%t(1,1) - t%t(2,2) - t%t(3,3)
phi = g * s * (((t*k1)*k2) + ((t*k2)*k1) &
- g * (m**2 + (k1*k2))*t_tr)/2.0_default
end function s_gravs
```

⟨*Implementation of couplings*⟩+≡
```
pure function grav_ss (g, m, k1, k2, s1, s2) result (t)
complex(kind=default), intent(in) :: g, s1, s2
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k1, k2
type(tensor) :: t_metric, t
t_metric%t = 0
t_metric%t(0,0) = 1.0_default
t_metric%t(1,1) = - 1.0_default
t_metric%t(2,2) = - 1.0_default
t_metric%t(3,3) = - 1.0_default
t = g*s1*s2/2.0_default * (-(m**2 + (k1*k2)) * t_metric &
+ (k1.tprod.k2) + (k2.tprod.k1))
end function grav_ss
```

⟨*Implementation of couplings*⟩+≡
```
pure function v_gravv (g, m, k1, k2, t, v) result (vec)
complex(kind=default), intent(in) :: g
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k1, k2
type(vector), intent(in) :: v
type(tensor), intent(in) :: t
complex(kind=default) :: t_tr
real(kind=default) :: xi
type(vector) :: vec
xi = 1.0_default
t_tr = t%t(0,0) - t%t(1,1) - t%t(2,2) - t%t(3,3)
```

```
  vec = (-g)/ 2.0_default * (((k1*k2) + m**2) * &
  (t*v + v*t - t_tr * v) + t_tr * (k1*v) * k2 &
  - (k1*v) * ((k2*t) + (t*k2)) &
  - ((k1*(t*v)) + (v*(t*k1))) * k2 &
  + ((k1*(t*k2)) + (k2*(t*k1))) * v)
  !!!       Unitarity gauge: xi -> Infinity
  !!!       + (1.0_default/xi) * (t_tr * ((k1*v)*k2) + &
  !!!       (k2*v)*k2 + (k2*v)*k1 - (k1*(t*v))*k1 + &
  !!!       (k2*v)*(k2*t) - (v*(t*k1))*k1 - (k2*v)*(t*k2)))
  end function v_gravv
```

⟨*Implementation of couplings*⟩+≡

```
  pure function grav_vv (g, m, k1, k2, v1, v2) result (t)
  complex(kind=default), intent(in) :: g
  type(momentum), intent(in) :: k1, k2
  real(kind=default), intent(in) :: m
  real(kind=default) :: xi
  type(vector), intent (in) :: v1, v2
  type(tensor) :: t_metric, t
  xi = 0.00001_default
  t_metric%t = 0
  t_metric%t(0,0) = 1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  t = (-g)/2.0_default * ( &
  ((k1*k2) + m**2) * ( &
  (v1.tprod.v2) +  (v2.tprod.v1) - (v1*v2) * t_metric) &
  + (v1*k2)*(v2*k1)*t_metric &
  - (k2*v1)*((v2.tprod.k1) + (k1.tprod.v2)) &
  - (k1*v2)*((v1.tprod.k2) + (k2.tprod.v1)) &
  + (v1*v2)*((k1.tprod.k2) + (k2.tprod.k1)))
  !!!       Unitarity gauge: xi -> Infinity
  !!!       + (1.0_default/xi) * ( &
  !!!       ((k1*v1)*(k1*v2) + (k2*v1)*(k2*v2) + (k1*v1)*(k2*v2))* &
  !!!       t_metric) - (k1*v1) * ((k1.tprod.v2) + (v2.tprod.k1)) &
  !!!       - (k2*v2) * ((k2.tprod.v1) + (v1.tprod.k2)))
  end function grav_vv
```

AB.24 Tensor Couplings

⟨*Declaration of couplings*⟩+≡

```
  public :: t2_vv, v_t2v, t2_vv_cf, v_t2v_cf, &
  t2_vv_1, v_t2v_1, t2_vv_t, v_t2v_t, &
  t2_phi2, phi_t2phi, t2_phi2_cf, phi_t2phi_cf
```

$$T_{\mu\nu} = g * V_{1\,\mu} V_{2\,\nu} + V_{1\,\nu} V_{2\,\mu} \tag{AB.88}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function t2_vv (g, v1, v2) result (t)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(tensor) :: t
  type(tensor) :: tmp
  tmp = v1.tprod.v2
  t%t = g * (tmp%t + transpose (tmp%t))
  end function t2_vv
```

$$V_{1\,\mu} = g * T_{\mu\nu} V_2^\nu + T_{\nu\mu} V_2^\nu \tag{AB.89}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function v_t2v (g, t, v) result (tv)
  complex(kind=default), intent(in) :: g
  type(tensor), intent(in) :: t
  type(vector), intent(in) :: v
  type(vector) :: tv
  type(tensor) :: tmp
  tmp%t = t%t + transpose (t%t)
  tv = g * (tmp * v)
  end function v_t2v
```

$$T_{\mu\nu} = -\frac{g}{2} V_1^{\rho} V_{2\,\rho} \tag{AB.90}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function t2_vv_cf (g, v1, v2) result (t)
  complex(kind=default), intent(in) :: g
  complex(kind=default) :: tmp_s
  type(vector), intent(in) :: v1, v2
  type(tensor) :: t_metric, t
  t_metric%t = 0
  t_metric%t(0,0) =   1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  tmp_s = v1 * v2
  t%t = - (g /2.0_default) * tmp_s * t_metric%t
  end function t2_vv_cf
```

$$V_{1\,\mu} = -\frac{g}{2} T_{\nu}^{\nu} V_2^{\mu} \tag{AB.91}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function v_t2v_cf (g, t, v) result (tv)
  complex(kind=default), intent(in) :: g
  type(tensor), intent(in) :: t
  type(vector), intent(in) :: v
  type(vector) :: tv, tmp_tv
  tmp_tv =  ( t%t(0,0)-t%t(1,1)-t%t(2,2)-t%t(3,3) ) * v
  tv = - ( g /2.0_default) * tmp_tv
  end function v_t2v_cf
```

$$T_{\mu\nu} = g * (k_{1\,\mu} k_{2\,\nu} + k_{1\,\nu} k_{2\,\mu})\,\phi_1\,(k_1)\,\phi_1\,(k_2) \tag{AB.92}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function t2_phi2 (g, phi1, k1, phi2, k2) result (t)
  complex(kind=default), intent(in) :: g, phi1, phi2
  type(momentum), intent(in) :: k1, k2
  type(tensor) :: t
  type(tensor) :: tmp
  tmp = k1.tprod.k2
  t%t = g * (tmp%t + transpose (tmp%t)) * phi1 * phi2
  end function t2_phi2
```

$$\phi_1(k_1) = g * (T_{\mu\nu} k_1^{\mu} k_2^{\nu} + T_{\nu\mu} k_2^{\mu} k_1^{\nu})\,\phi_2\,(k_2) \tag{AB.93}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function phi_t2phi (g, t, kt, phi2, k2) result (phi1)
  complex(kind=default), intent(in) :: g, phi2
  type(tensor), intent(in) :: t
  type(momentum), intent(in) :: kt, k2
  type(momentum) :: k1
  complex(kind=default) :: phi1
  type(tensor) :: tmp
  k1 = -kt - k2
  tmp%t = t%t + transpose (t%t)
  phi1 = g * ( (tmp * k2) * k1) * phi2
  end function phi_t2phi
```

$$T_{\mu\nu} = -\frac{g}{2} k_1^{\rho} k_{2\,\rho} \phi_1\,(k_1)\,\phi_2\,(k_2) \tag{AB.94}$$

⟨*Implementation of couplings*⟩+≡
```
  pure function t2_phi2_cf (g, phi1, k1, phi2, k2) result (t)
  complex(kind=default), intent(in) :: g, phi1, phi2
  complex(kind=default) :: tmp_s
  type(momentum), intent(in) :: k1, k2
  type(tensor) :: t_metric, t
  t_metric%t = 0
  t_metric%t(0,0) =   1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  tmp_s = (k1 * k2) * phi1 * phi2
  t%t = - (g /2.0_default) * tmp_s * t_metric%t
  end function t2_phi2_cf
```

$$\phi_1(k_1) = -\frac{g}{2} T_\nu^\nu \, (k_1 \cdot k_2) \, \phi_2(k_2) \tag{AB.95}$$

⟨*Implementation of couplings*⟩+≡

```
  pure function phi_t2phi_cf (g, t, kt, phi2, k2) result (phi1)
  complex(kind=default), intent(in) :: g, phi2
  type(tensor), intent(in) :: t
  type(momentum), intent(in) :: kt, k2
  type(momentum) :: k1
  complex(kind=default) ::  tmp_ts, phi1
  k1 = - kt - k2
  tmp_ts =  ( t%t(0,0)-t%t(1,1)-t%t(2,2)-t%t(3,3) )
  phi1 = - ( g /2.0_default) * tmp_ts * (k1 * k2) * phi2
  end function phi_t2phi_cf
```

⟨*Implementation of couplings*⟩+≡

```
  pure function t2_vv_1 (g, v1, v2) result (t)
  complex(kind=default), intent(in) :: g
  complex(kind=default) :: tmp_s
  type(vector), intent(in) :: v1, v2
  type(tensor) :: tmp
  type(tensor) :: t_metric, t
  t_metric%t = 0
  t_metric%t(0,0) =   1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  tmp = v1.tprod.v2
  tmp_s = v1 * v2
  t%t = g * (tmp%t + transpose (tmp%t) - tmp_s * t_metric%t )
  end function t2_vv_1
```

⟨*Implementation of couplings*⟩+≡

```
  pure function v_t2v_1 (g, t, v) result (tv)
  complex(kind=default), intent(in) :: g
  type(tensor), intent(in) :: t
  type(vector), intent(in) :: v
  type(vector) :: tv, tmp_tv
  type(tensor) :: tmp
  tmp_tv =  ( t%t(0,0)-t%t(1,1)-t%t(2,2)-t%t(3,3) ) * v
  tmp%t = t%t + transpose (t%t)
  tv = g * (tmp * v - tmp_tv)
  end function v_t2v_1
```

⟨*Implementation of couplings*⟩+≡

```
  pure function t2_vv_t (g, v1, k1, v2, k2) result (t)
  complex(kind=default), intent(in) :: g
  complex(kind=default) :: tmp_s
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(tensor) :: tmp, tmp_v1k2, tmp_v2k1, tmp_k1k2, tmp2
  type(tensor) :: t_metric, t
  t_metric%t = 0
  t_metric%t(0,0) =   1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  tmp = v1.tprod.v2
  tmp_s = v1 * v2
  tmp_v1k2 = (v2 * k1) * (v1.tprod.k2)
  tmp_v2k1 = (v1 * k2) * (v2.tprod.k1)
  tmp_k1k2 = tmp_s * (k1.tprod.k2)
  tmp2%t = tmp_v1k2%t + tmp_v2k1%t - tmp_k1k2%t
  t%t = g * ( (k1*k2) * (tmp%t + transpose (tmp%t) - tmp_s * t_metric%t ) &
  + ((v1 * k2) * (v2 * k1)) * t_metric%t &
  - tmp2%t - transpose(tmp2%t))
  end function t2_vv_t
```

⟨*Implementation of couplings*⟩+≡

```
pure function v_t2v_t (g, t, kt, v, kv) result (tv)
complex(kind=default), intent(in) :: g
type(tensor), intent(in) :: t
type(vector), intent(in) :: v
type(momentum), intent(in) :: kt, kv
type(momentum) :: kout
type(vector) :: tv, tmp_tv
type(tensor) :: tmp
kout = - (kt + kv)
tmp_tv =  ( t%t(0,0)-t%t(1,1)-t%t(2,2)-t%t(3,3) ) * v
tmp%t = t%t + transpose (t%t)
tv = g * ( (tmp * v - tmp_tv) * (kv * kout )&
+ ( t%t(0,0)-t%t(1,1)-t%t(2,2)-t%t(3,3) ) * (kout * v ) * kv &
- (kout * v) * ( tmp * kv) &
- (v* (t * kout) + kout * (t * v)) * kv &
+ (kout* (t * kv) + kv * (t * kout)) * v)
end function v_t2v_t
```

⟨*Declaration of couplings*⟩+≡

```
public :: t2_vv_d5_1, v_t2v_d5_1
```

⟨*Implementation of couplings*⟩+≡

```
pure function t2_vv_d5_1 (g, v1, k1, v2, k2) result (t)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(tensor) :: t
t = (g * (v1 * v2)) * (k1-k2).tprod.(k1-k2)
end function t2_vv_d5_1
```

⟨*Implementation of couplings*⟩+≡

```
pure function v_t2v_d5_1 (g, t1, k1, v2, k2) result (tv)
complex(kind=default), intent(in) :: g
type(tensor), intent(in) :: t1
type(vector), intent(in) :: v2
type(momentum), intent(in) :: k1, k2
type(vector) :: tv
tv = (g * ((k1+2*k2).tprod.(k1+2*k2) * t1)) * v2
end function v_t2v_d5_1
```

⟨*Declaration of couplings*⟩+≡

```
public :: t2_vv_d5_2, v_t2v_d5_2
```

⟨*Implementation of couplings*⟩+≡

```
pure function t2_vv_d5_2 (g, v1, k1, v2, k2) result (t)
complex(kind=default), intent(in) :: g
type(vector), intent(in) :: v1, v2
type(momentum), intent(in) :: k1, k2
type(tensor) :: t
t = (g * (k2 * v1)) * (k2-k1).tprod.v2
t%t = t%t + transpose (t%t)
end function t2_vv_d5_2
```

⟨*Implementation of couplings*⟩+≡

```
pure function v_t2v_d5_2 (g, t1, k1, v2, k2) result (tv)
complex(kind=default), intent(in) :: g
type(tensor), intent(in) :: t1
type(vector), intent(in) :: v2
type(momentum), intent(in) :: k1, k2
type(vector) :: tv
type(tensor) :: tmp
type(momentum) :: k1_k2, k1_2k2
k1_k2 = k1 + k2
k1_2k2 = k1_k2 + k2
tmp%t = t1%t + transpose (t1%t)
```

```
  tv = (g * (k1_k2 * v2)) * (k1_2k2 * tmp)
  end function v_t2v_d5_2
```

⟨*Declaration of couplings*⟩+≡
```
  public :: t2_vv_d7, v_t2v_d7
```

⟨*Implementation of couplings*⟩+≡
```
  pure function t2_vv_d7 (g, v1, k1, v2, k2) result (t)
  complex(kind=default), intent(in) :: g
  type(vector), intent(in) :: v1, v2
  type(momentum), intent(in) :: k1, k2
  type(tensor) :: t
  t = (g * (k2 * v1) * (k1 * v2)) * (k1-k2).tprod.(k1-k2)
  end function t2_vv_d7
```

⟨*Implementation of couplings*⟩+≡
```
  pure function v_t2v_d7 (g, t1, k1, v2, k2) result (tv)
  complex(kind=default), intent(in) :: g
  type(tensor), intent(in) :: t1
  type(vector), intent(in) :: v2
  type(momentum), intent(in) :: k1, k2
  type(vector) :: tv
  type(vector) :: k1_k2, k1_2k2
  k1_k2 = k1 + k2
  k1_2k2 = k1_k2 + k2
  tv = (- g * (k1_k2 * v2) * (k1_2k2.tprod.k1_2k2 * t1)) * k2
  end function v_t2v_d7
```

AB.25 Spinor Couplings

⟨omega_spinor_couplings.f90⟩≡
```
  ⟨Copyleft⟩
  module omega_spinor_couplings
  use kinds
  use constants
  use omega_spinors
  use omega_vectors
  use omega_tensors
  use omega_couplings
  implicit none
  private
  ⟨Declaration of spinor on shell wave functions⟩
  ⟨Declaration of spinor off shell wave functions⟩
  ⟨Declaration of spinor currents⟩
  ⟨Declaration of spinor propagators⟩
  integer, parameter, public :: omega_spinor_cpls_2010_01_A = 0
  contains
  ⟨Implementation of spinor on shell wave functions⟩
  ⟨Implementation of spinor off shell wave functions⟩
  ⟨Implementation of spinor currents⟩
  ⟨Implementation of spinor propagators⟩
  end module omega_spinor_couplings
```

See table AB.1 for the names of Fortran functions. We could have used long names instead, but this would increase the chance of running past continuation line limits without adding much to the legibility.

AB.25.1 Fermionic Vector and Axial Couplings

There's more than one chiral representation. This one is compatible with HELAS [5].

$$\gamma^0 = \begin{pmatrix} 0 & \mathbf{1} \\ \mathbf{1} & 0 \end{pmatrix},\ \gamma^i = \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix},\ \gamma_5 = i\gamma^0\gamma^1\gamma^2\gamma^3 = \begin{pmatrix} -\mathbf{1} & 0 \\ 0 & \mathbf{1} \end{pmatrix} \tag{AB.96}$$

Therefore

$$g_S + g_P\gamma_5 = \begin{pmatrix} g_S - g_P & 0 & 0 & 0 \\ 0 & g_S - g_P & 0 & 0 \\ 0 & 0 & g_S + g_P & 0 \\ 0 & 0 & 0 & g_S + g_P \end{pmatrix} \tag{AB.97a}$$

$\bar\psi(g_V\gamma^\mu - g_A\gamma^\mu\gamma_5)\psi$	`va_ff`$(g_V, g_A, \bar\psi, \psi)$
$g_V\bar\psi\gamma^\mu\psi$	`v_ff`$(g_V, \bar\psi, \psi)$
$g_A\bar\psi\gamma_5\gamma^\mu\psi$	`a_ff`$(g_A, \bar\psi, \psi)$
$g_L\bar\psi\gamma^\mu(1-\gamma_5)\psi$	`vl_ff`$(g_L, \bar\psi, \psi)$
$g_R\bar\psi\gamma^\mu(1+\gamma_5)\psi$	`vr_ff`$(g_R, \bar\psi, \psi)$
$\not{V}(g_V - g_A\gamma_5)\psi$	`f_vaf`(g_V, g_A, V, ψ)
$g_V\not{V}\psi$	`f_vf`(g_V, V, ψ)
$g_A\gamma_5\not{V}\psi$	`f_af`(g_A, V, ψ)
$g_L\not{V}(1-\gamma_5)\psi$	`f_vlf`(g_L, V, ψ)
$g_R\not{V}(1+\gamma_5)\psi$	`f_vrf`(g_R, V, ψ)
$\bar\psi\not{V}(g_V - g_A\gamma_5)$	`f_fva`$(g_V, g_A, \bar\psi, V)$
$g_V\bar\psi\not{V}$	`f_fv`$(g_V, \bar\psi, V)$
$g_A\bar\psi\gamma_5\not{V}$	`f_fa`$(g_A, \bar\psi, V)$
$g_L\bar\psi\not{V}(1-\gamma_5)$	`f_fvl`$(g_L, \bar\psi, V)$
$g_R\bar\psi\not{V}(1+\gamma_5)$	`f_fvr`$(g_R, \bar\psi, V)$

Table AB.1: Mnemonically abbreviated names of Fortran functions implementing fermionic vector and axial currents.

$\bar\psi(g_S + g_P\gamma_5)\psi$	`sp_ff`$(g_S, g_P, \bar\psi, \psi)$
$g_S\bar\psi\psi$	`s_ff`$(g_S, \bar\psi, \psi)$
$g_P\bar\psi\gamma_5\psi$	`p_ff`$(g_P, \bar\psi, \psi)$
$g_L\bar\psi(1-\gamma_5)\psi$	`sl_ff`$(g_L, \bar\psi, \psi)$
$g_R\bar\psi(1+\gamma_5)\psi$	`sr_ff`$(g_R, \bar\psi, \psi)$
$\phi(g_S + g_P\gamma_5)\psi$	`f_spf`(g_S, g_P, ϕ, ψ)
$g_S\phi\psi$	`f_sf`(g_S, ϕ, ψ)
$g_P\phi\gamma_5\psi$	`f_pf`(g_P, ϕ, ψ)
$g_L\phi(1-\gamma_5)\psi$	`f_slf`(g_L, ϕ, ψ)
$g_R\phi(1+\gamma_5)\psi$	`f_srf`(g_R, ϕ, ψ)
$\bar\psi\phi(g_S + g_P\gamma_5)$	`f_fsp`$(g_S, g_P, \bar\psi, \phi)$
$g_S\bar\psi\phi$	`f_fs`$(g_S, \bar\psi, \phi)$
$g_P\bar\psi\phi\gamma_5$	`f_fp`$(g_P, \bar\psi, \phi)$
$g_L\bar\psi\phi(1-\gamma_5)$	`f_fsl`$(g_L, \bar\psi, \phi)$
$g_R\bar\psi\phi(1+\gamma_5)$	`f_fsr`$(g_R, \bar\psi, \phi)$

Table AB.2: Mnemonically abbreviated names of Fortran functions implementing fermionic scalar and pseudo scalar "currents".

$$g_V\gamma^0 - g_A\gamma^0\gamma_5 = \begin{pmatrix} 0 & 0 & g_V - g_A & 0 \\ 0 & 0 & 0 & g_V - g_A \\ g_V + g_A & 0 & 0 & 0 \\ 0 & g_V + g_A & 0 & 0 \end{pmatrix} \tag{AB.97b}$$

$$g_V\gamma^1 - g_A\gamma^1\gamma_5 = \begin{pmatrix} 0 & 0 & 0 & g_V - g_A \\ 0 & 0 & g_V - g_A & 0 \\ 0 & -g_V - g_A & 0 & 0 \\ -g_V - g_A & 0 & 0 & 0 \end{pmatrix} \tag{AB.97c}$$

$$g_V\gamma^2 - g_A\gamma^2\gamma_5 = \begin{pmatrix} 0 & 0 & 0 & -\mathrm{i}(g_V - g_A) \\ 0 & 0 & \mathrm{i}(g_V - g_A) & 0 \\ 0 & \mathrm{i}(g_V + g_A) & 0 & 0 \\ -\mathrm{i}(g_V + g_A) & 0 & 0 & 0 \end{pmatrix} \tag{AB.97d}$$

$$g_V\gamma^3 - g_A\gamma^3\gamma_5 = \begin{pmatrix} 0 & 0 & g_V - g_A & 0 \\ 0 & 0 & 0 & -g_V + g_A \\ -g_V - g_A & 0 & 0 & 0 \\ 0 & g_V + g_A & 0 & 0 \end{pmatrix} \tag{AB.97e}$$

⟨*Declaration of spinor currents*⟩≡
```
  public :: va_ff, v_ff, a_ff, vl_ff, vr_ff, vlr_ff, grav_ff, va2_ff, &
  tva_ff, tlr_ff, trl_ff, tvam_ff, tlrm_ff, trlm_ff, va3_ff
```

⟨*Implementation of spinor currents*⟩≡

```
  pure function va_ff (gv, ga, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  gl = gv + ga
  gr = gv - ga
  g13 = psibar%a(1)*psi%a(3)
  g14 = psibar%a(1)*psi%a(4)
  g23 = psibar%a(2)*psi%a(3)
  g24 = psibar%a(2)*psi%a(4)
  g31 = psibar%a(3)*psi%a(1)
  g32 = psibar%a(3)*psi%a(2)
  g41 = psibar%a(4)*psi%a(1)
  g42 = psibar%a(4)*psi%a(2)
  j%t    =  gr * (   g13 + g24) + gl * (   g31 + g42)
  j%x(1) =  gr * (   g14 + g23) - gl * (   g32 + g41)
  j%x(2) = (gr * ( - g14 + g23) + gl * (   g32 - g41)) * (0, 1)
  j%x(3) =  gr * (   g13 - g24) + gl * ( - g31 + g42)
  end function va_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function va2_ff (gva, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in), dimension(2) :: gva
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  gl = gva(1) + gva(2)
  gr = gva(1) - gva(2)
  g13 = psibar%a(1)*psi%a(3)
  g14 = psibar%a(1)*psi%a(4)
  g23 = psibar%a(2)*psi%a(3)
  g24 = psibar%a(2)*psi%a(4)
  g31 = psibar%a(3)*psi%a(1)
  g32 = psibar%a(3)*psi%a(2)
  g41 = psibar%a(4)*psi%a(1)
  g42 = psibar%a(4)*psi%a(2)
  j%t    =  gr * (   g13 + g24) + gl * (   g31 + g42)
  j%x(1) =  gr * (   g14 + g23) - gl * (   g32 + g41)
  j%x(2) = (gr * ( - g14 + g23) + gl * (   g32 - g41)) * (0, 1)
  j%x(3) =  gr * (   g13 - g24) + gl * ( - g31 + g42)
  end function va2_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function va3_ff (gv, ga, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j   = va_ff (gv, ga, psibar, psi)
  j%t = 0.0_default
  end function va3_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function tva_ff (gv, ga, psibar, psi) result (t)
  type(tensor2odd) :: t
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: g12, g21, g1m2, g34, g43, g3m4
  gr     = gv + ga
  gl     = gv - ga
  g12    = psibar%a(1)*psi%a(2)
  g21    = psibar%a(2)*psi%a(1)
  g1m2   = psibar%a(1)*psi%a(1) - psibar%a(2)*psi%a(2)
  g34    = psibar%a(3)*psi%a(4)
  g43    = psibar%a(4)*psi%a(3)
  g3m4   = psibar%a(3)*psi%a(3) - psibar%a(4)*psi%a(4)
  t%e(1) = (gl * ( - g12 - g21) + gr * (   g34 + g43)) * (0, 1)
  t%e(2) =  gl * ( - g12 + g21) + gr * (   g34 - g43)
  t%e(3) = (gl * ( - g1m2     ) + gr * (   g3m4     )) * (0, 1)
  t%b(1) =  gl * (   g12 + g21) + gr * (   g34 + g43)
  t%b(2) = (gl * ( - g12 + g21) + gr * ( - g34 + g43)) * (0, 1)
  t%b(3) =  gl * (   g1m2     ) + gr * (   g3m4     )
  end function tva_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function tlr_ff (gl, gr, psibar, psi) result (t)
  type(tensor2odd) :: t
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  t = tva_ff (gr+gl, gr-gl, psibar, psi)
  end function tlr_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function trl_ff (gr, gl, psibar, psi) result (t)
  type(tensor2odd) :: t
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  t = tva_ff (gr+gl, gr-gl, psibar, psi)
  end function trl_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function tvam_ff (gv, ga, psibar, psi, p) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: p
  j = (tva_ff(gv, ga, psibar, psi) * p) * (0,1)
  end function tvam_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function tlrm_ff (gl, gr, psibar, psi, p) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: p
  j = tvam_ff (gr+gl, gr-gl, psibar, psi, p)
  end function tlrm_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function trlm_ff (gr, gl, psibar, psi, p) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: p
  j = tvam_ff (gr+gl, gr-gl, psibar, psi, p)
  end function trlm_ff
```

Special cases that avoid some multiplications

⟨*Implementation of spinor currents*⟩+≡

```
  pure function v_ff (gv, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  g13 = psibar%a(1)*psi%a(3)
  g14 = psibar%a(1)*psi%a(4)
  g23 = psibar%a(2)*psi%a(3)
  g24 = psibar%a(2)*psi%a(4)
  g31 = psibar%a(3)*psi%a(1)
  g32 = psibar%a(3)*psi%a(2)
  g41 = psibar%a(4)*psi%a(1)
  g42 = psibar%a(4)*psi%a(2)
  j%t    =   gv * (   g13 + g24 + g31 + g42)
  j%x(1) =   gv * (   g14 + g23 - g32 - g41)
  j%x(2) =   gv * ( - g14 + g23 + g32 - g41) * (0, 1)
  j%x(3) =   gv * (   g13 - g24 - g31 + g42)
  end function v_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function a_ff (ga, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: ga
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  g13 = psibar%a(1)*psi%a(3)
  g14 = psibar%a(1)*psi%a(4)
  g23 = psibar%a(2)*psi%a(3)
  g24 = psibar%a(2)*psi%a(4)
  g31 = psibar%a(3)*psi%a(1)
  g32 = psibar%a(3)*psi%a(2)
  g41 = psibar%a(4)*psi%a(1)
  g42 = psibar%a(4)*psi%a(2)
  j%t    =   ga * ( - g13 - g24 + g31 + g42)
  j%x(1) = - ga * (   g14 + g23 + g32 + g41)
  j%x(2) =   ga * (   g14 - g23 + g32 - g41) * (0, 1)
  j%x(3) =   ga * ( - g13 + g24 - g31 + g42)
  end function a_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function vl_ff (gl, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl2
  complex(kind=default) :: g31, g32, g41, g42
  gl2 = 2 * gl
  g31 = psibar%a(3)*psi%a(1)
  g32 = psibar%a(3)*psi%a(2)
  g41 = psibar%a(4)*psi%a(1)
  g42 = psibar%a(4)*psi%a(2)
  j%t    =   gl2 * (   g31 + g42)
  j%x(1) = - gl2 * (   g32 + g41)
  j%x(2) =   gl2 * (   g32 - g41) * (0, 1)
  j%x(3) =   gl2 * ( - g31 + g42)
  end function vl_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function vr_ff (gr, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gr2
  complex(kind=default) :: g13, g14, g23, g24
  gr2 = 2 * gr
  g13 = psibar%a(1)*psi%a(3)
  g14 = psibar%a(1)*psi%a(4)
  g23 = psibar%a(2)*psi%a(3)
  g24 = psibar%a(2)*psi%a(4)
  j%t    = gr2 * (   g13 + g24)
  j%x(1) = gr2 * (   g14 + g23)
  j%x(2) = gr2 * ( - g14 + g23) * (0, 1)
  j%x(3) = gr2 * (   g13 - g24)
  end function vr_ff
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function grav_ff (g, m, kb, k, psibar, psi) result (j)
  type(tensor) :: j
  complex(kind=default), intent(in) :: g
  real(kind=default), intent(in) :: m
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: kb, k
  complex(kind=default) :: g2, g8, c_dum
  type(vector) :: v_dum
  type(tensor) :: t_metric
  t_metric%t = 0
  t_metric%t(0,0) = 1.0_default
  t_metric%t(1,1) = - 1.0_default
  t_metric%t(2,2) = - 1.0_default
  t_metric%t(3,3) = - 1.0_default
  g2 = g/2.0_default
  g8 = g/8.0_default
  v_dum = v_ff(g8, psibar, psi)
  c_dum = (- m) * s_ff (g2, psibar, psi) - (kb+k)*v_dum
  j = c_dum*t_metric - (((kb+k).tprod.v_dum) + &
  (v_dum.tprod.(kb+k)))
  end function grav_ff
```

$$g_L\gamma_\mu(1-\gamma_5)+g_R\gamma_\mu(1+\gamma_5)=(g_L+g_R)\gamma_\mu-(g_L-g_R)\gamma_\mu\gamma_5=g_V\gamma_\mu-g_A\gamma_\mu\gamma_5 \tag{AB.98}$$

... give the compiler the benefit of the doubt that it will optimize the function all. If not, we could inline it ...

⟨*Implementation of spinor currents*⟩+≡

```
  pure function vlr_ff (gl, gr, psibar, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j = va_ff (gl+gr, gl-gr, psibar, psi)
  end function vlr_ff
```

and

$$\not v-\not a\gamma_5=\begin{pmatrix} 0 & 0 & v_--a_- & -v^*+a^* \\ 0 & 0 & -v+a & v_+-a_+ \\ v_++a_+ & v^*+a^* & 0 & 0 \\ v+a & v_-+a_- & 0 & 0 \end{pmatrix} \tag{AB.99}$$

with $v_\pm=v_0\pm v_3$, $a_\pm=a_0\pm a_3$, $v=v_1+\mathrm{i}v_2$, $v^*=v_1-\mathrm{i}v_2$, $a=a_1+\mathrm{i}a_2$, and $a^*=a_1-\mathrm{i}a_2$. But note that $\cdot^*$ is *not* complex conjugation for complex v_μ or a_μ.

⟨*Declaration of spinor currents*⟩+≡

```
  public :: f_vaf, f_vf, f_af, f_vlf, f_vrf, f_vlrf, f_va2f, &
  f_tvaf, f_tlrf, f_trlf, f_tvamf, f_tlrmf, f_trlmf, f_va3f
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_vaf (gv, ga, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gv, ga
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gv + ga
  gr = gv - ga
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gl * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vaf
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_va2f (gva, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in), dimension(2) :: gva
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gva(1) + gva(2)
  gr = gva(1) - gva(2)
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gl * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_va2f
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_va3f (gv, ga, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gv, ga
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gv + ga
  gr = gv - ga
  vp =   v%x(3) !+ v%t
  vm = - v%x(3) !+ v%t
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gl * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_va3f
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_tvaf (gv, ga, t, psi) result (tpsi)
  type(spinor) :: tpsi
  complex(kind=default), intent(in) :: gv, ga
  type(tensor2odd), intent(in) :: t
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: e21, e21s, b12, b12s, be3, be3s
  gr   = gv + ga
  gl   = gv - ga
  e21  = t%e(2) + t%e(1)*(0,1)
  e21s = t%e(2) - t%e(1)*(0,1)
  b12  = t%b(1) + t%b(2)*(0,1)
  b12s = t%b(1) - t%b(2)*(0,1)
  be3  = t%b(3) + t%e(3)*(0,1)
  be3s = t%b(3) - t%e(3)*(0,1)
  tpsi%a(1) =   2*gl * (   psi%a(1) * be3  + psi%a(2) * ( e21 +b12s))
  tpsi%a(2) =   2*gl * ( - psi%a(2) * be3  + psi%a(1) * (-e21s+b12 ))
  tpsi%a(3) =   2*gr * (   psi%a(3) * be3s + psi%a(4) * (-e21 +b12s))
  tpsi%a(4) =   2*gr * ( - psi%a(4) * be3s + psi%a(3) * ( e21s+b12 ))
  end function f_tvaf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_tlrf (gl, gr, t, psi) result (tpsi)
  type(spinor) :: tpsi
  complex(kind=default), intent(in) :: gl, gr
  type(tensor2odd), intent(in) :: t
  type(spinor), intent(in) :: psi
  tpsi = f_tvaf (gr+gl, gr-gl, t, psi)
  end function f_tlrf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_trlf (gr, gl, t, psi) result (tpsi)
  type(spinor) :: tpsi
  complex(kind=default), intent(in) :: gl, gr
  type(tensor2odd), intent(in) :: t
  type(spinor), intent(in) :: psi
  tpsi = f_tvaf (gr+gl, gr-gl, t, psi)
  end function f_trlf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_tvamf (gv, ga, v, psi, k) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gv, ga
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  type(tensor2odd) :: t
  t = (v.wedge.k) * (0, 0.5)
  vpsi = f_tvaf(gv, ga, t, psi)
  end function f_tvamf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_tlrmf (gl, gr, v, psi, k) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gl, gr
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  vpsi = f_tvamf (gr+gl, gr-gl, v, psi, k)
  end function f_tlrmf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_trlmf (gr, gl, v, psi, k) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gl, gr
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  vpsi = f_tvamf (gr+gl, gr-gl, v, psi, k)
  end function f_trlmf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_vf (gv, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gv
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gv * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gv * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gv * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gv * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_af (ga, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: ga
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = ga * ( - vm  * psi%a(3) + v12s * psi%a(4))
  vpsi%a(2) = ga * (   v12 * psi%a(3) - vp   * psi%a(4))
  vpsi%a(3) = ga * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = ga * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_af
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_vlf (gl, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gl
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gl2
  complex(kind=default) :: vp, vm, v12, v12s
  gl2 = 2 * gl
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = 0
  vpsi%a(2) = 0
  vpsi%a(3) = gl2 * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl2 * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vlf
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_vrf (gr, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gr
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  complex(kind=default) :: gr2
  complex(kind=default) :: vp, vm, v12, v12s
  gr2 = 2 * gr
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr2 * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr2 * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = 0
  vpsi%a(4) = 0
  end function f_vrf
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_vlrf (gl, gr, v, psi) result (vpsi)
  type(spinor) :: vpsi
  complex(kind=default), intent(in) :: gl, gr
  type(vector), intent(in) :: v
  type(spinor), intent(in) :: psi
  vpsi = f_vaf (gl+gr, gl-gr, v, psi)
  end function f_vlrf
```

⟨*Declaration of spinor currents*⟩+≡

```
  public :: f_fva, f_fv, f_fa, f_fvl, f_fvr, f_fvlr, f_fva2, &
  f_ftva, f_ftlr, f_ftrl, f_ftvam, f_ftlrm, f_ftrlm, f_fva3
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fva (gv, ga, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gv + ga
  gr = gv - ga
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = gl * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = gl * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = gr * (   psibar%a(1) * vm   - psibar%a(2) * v12)
  psibarv%a(4) = gr * ( - psibar%a(1) * v12s + psibar%a(2) * vp )
  end function f_fva
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fva2 (gva, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in), dimension(2) :: gva
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gva(1) + gva(2)
  gr = gva(1) - gva(2)
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = gl * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = gl * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = gr * (   psibar%a(1) * vm   - psibar%a(2) * v12)
  psibarv%a(4) = gr * ( - psibar%a(1) * v12s + psibar%a(2) * vp )
  end function f_fva2
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fva3 (gv, ga, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gv + ga
  gr = gv - ga
  vp =   v%x(3) !+ v%t
  vm = - v%x(3) !+ v%t
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = gl * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = gl * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = gr * (   psibar%a(1) * vm   - psibar%a(2) * v12)
  psibarv%a(4) = gr * ( - psibar%a(1) * v12s + psibar%a(2) * vp )
  end function f_fva3
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftva (gv, ga, psibar, t) result (psibart)
  type(conjspinor) :: psibart
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(tensor2odd), intent(in) :: t
  complex(kind=default) :: gl, gr
  complex(kind=default) :: e21, e21s, b12, b12s, be3, be3s
  gr   = gv + ga
  gl   = gv - ga
  e21  = t%e(2) + t%e(1)*(0,1)
  e21s = t%e(2) - t%e(1)*(0,1)
  b12  = t%b(1) + t%b(2)*(0,1)
  b12s = t%b(1) - t%b(2)*(0,1)
  be3  = t%b(3) + t%e(3)*(0,1)
  be3s = t%b(3) - t%e(3)*(0,1)
  psibart%a(1) = 2*gl * (   psibar%a(1) * be3  + psibar%a(2) * (-e21s+b12 ))
  psibart%a(2) = 2*gl * ( - psibar%a(2) * be3  + psibar%a(1) * ( e21 +b12s))
  psibart%a(3) = 2*gr * (   psibar%a(3) * be3s + psibar%a(4) * ( e21s+b12 ))
  psibart%a(4) = 2*gr * ( - psibar%a(4) * be3s + psibar%a(3) * (-e21 +b12s))
  end function f_ftva
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftlr (gl, gr, psibar, t) result (psibart)
  type(conjspinor) :: psibart
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(tensor2odd), intent(in) :: t
  psibart = f_ftva (gr+gl, gr-gl, psibar, t)
  end function f_ftlr
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftrl (gr, gl, psibar, t) result (psibart)
  type(conjspinor) :: psibart
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(tensor2odd), intent(in) :: t
  psibart = f_ftva (gr+gl, gr-gl, psibar, t)
  end function f_ftrl
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftvam (gv, ga, psibar, v, k) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gv, ga
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  type(tensor2odd) :: t
  t = (v.wedge.k) * (0, 0.5)
  psibarv = f_ftva(gv, ga, psibar, t)
  end function f_ftvam
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftlrm (gl, gr, psibar, v, k) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  psibarv = f_ftvam (gr+gl, gr-gl, psibar, v, k)
  end function f_ftlrm
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_ftrlm (gr, gl, psibar, v, k) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  psibarv = f_ftvam (gr+gl, gr-gl, psibar, v, k)
  end function f_ftrlm
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fv (gv, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gv
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = gv * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = gv * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = gv * (   psibar%a(1) * vm   - psibar%a(2) * v12)
  psibarv%a(4) = gv * ( - psibar%a(1) * v12s + psibar%a(2) * vp )
  end function f_fv
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fa (ga, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: ga
  type(vector), intent(in) :: v
  type(conjspinor), intent(in) :: psibar
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = ga * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = ga * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = ga * ( - psibar%a(1) * vm   + psibar%a(2) * v12)
  psibarv%a(4) = ga * (   psibar%a(1) * v12s - psibar%a(2) * vp )
  end function f_fa
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fvl (gl, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gl
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: gl2
  complex(kind=default) :: vp, vm, v12, v12s
  gl2 = 2 * gl
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = gl2 * (   psibar%a(3) * vp   + psibar%a(4) * v12)
  psibarv%a(2) = gl2 * (   psibar%a(3) * v12s + psibar%a(4) * vm )
  psibarv%a(3) = 0
  psibarv%a(4) = 0
  end function f_fvl
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fvr (gr, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gr
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  complex(kind=default) :: gr2
  complex(kind=default) :: vp, vm, v12, v12s
  gr2 = 2 * gr
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  psibarv%a(1) = 0
  psibarv%a(2) = 0
  psibarv%a(3) = gr2 * (   psibar%a(1) * vm   - psibar%a(2) * v12)
  psibarv%a(4) = gr2 * ( - psibar%a(1) * v12s + psibar%a(2) * vp )
  end function f_fvr
```

⟨*Implementation of spinor currents*⟩+≡

```
  pure function f_fvlr (gl, gr, psibar, v) result (psibarv)
  type(conjspinor) :: psibarv
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  psibarv = f_fva (gl+gr, gl-gr, psibar, v)
  end function f_fvlr
```

AB.25.2 Fermionic Scalar and Pseudo Scalar Couplings

⟨*Declaration of spinor currents*⟩+≡
```
  public :: sp_ff, s_ff, p_ff, sl_ff, sr_ff, slr_ff
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function sp_ff (gs, gp, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gs, gp
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j =    (gs - gp) * (psibar%a(1)*psi%a(1) + psibar%a(2)*psi%a(2)) &
  + (gs + gp) * (psibar%a(3)*psi%a(3) + psibar%a(4)*psi%a(4))
  end function sp_ff
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function s_ff (gs, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gs
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j = gs * (psibar * psi)
  end function s_ff
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function p_ff (gp, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gp
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j = gp * (  psibar%a(3)*psi%a(3) + psibar%a(4)*psi%a(4) &
  - psibar%a(1)*psi%a(1) - psibar%a(2)*psi%a(2))
  end function p_ff
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function sl_ff (gl, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j =  2 * gl * (psibar%a(1)*psi%a(1) + psibar%a(2)*psi%a(2))
  end function sl_ff
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function sr_ff (gr, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j = 2 * gr * (psibar%a(3)*psi%a(3) + psibar%a(4)*psi%a(4))
  end function sr_ff
```

$$g_L(1-\gamma_5) + g_R(1+\gamma_5) = (g_R + g_L) + (g_R - g_L)\gamma_5 = g_S + g_P\gamma_5 \tag{AB.100}$$

⟨*Implementation of spinor currents*⟩+≡
```
  pure function slr_ff (gl, gr, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  type(spinor), intent(in) :: psi
  j = sp_ff (gr+gl, gr-gl, psibar, psi)
  end function slr_ff
```

⟨*Declaration of spinor currents*⟩+≡
```
  public :: f_spf, f_sf, f_pf, f_slf, f_srf, f_slrf
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_spf (gs, gp, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gs, gp
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi%a(1:2) = ((gs - gp) * phi) * psi%a(1:2)
  phipsi%a(3:4) = ((gs + gp) * phi) * psi%a(3:4)
  end function f_spf
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_sf (gs, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gs
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi%a = (gs * phi) * psi%a
  end function f_sf
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_pf (gp, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gp
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi%a(1:2) = (- gp * phi) * psi%a(1:2)
  phipsi%a(3:4) = (  gp * phi) * psi%a(3:4)
  end function f_pf
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_slf (gl, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gl
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi%a(1:2) = (2 * gl * phi) * psi%a(1:2)
  phipsi%a(3:4) = 0
  end function f_slf
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_srf (gr, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gr
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi%a(1:2) = 0
  phipsi%a(3:4) = (2 * gr * phi) * psi%a(3:4)
  end function f_srf
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_slrf (gl, gr, phi, psi) result (phipsi)
  type(spinor) :: phipsi
  complex(kind=default), intent(in) :: gl, gr
  complex(kind=default), intent(in) :: phi
  type(spinor), intent(in) :: psi
  phipsi =  f_spf (gr+gl, gr-gl, phi, psi)
  end function f_slrf
```
⟨*Declaration of spinor currents*⟩+≡
```
  public :: f_fsp, f_fs, f_fp, f_fsl, f_fsr, f_fslr
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fsp (gs, gp, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gs, gp
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi%a(1:2) = ((gs - gp) * phi) * psibar%a(1:2)
  psibarphi%a(3:4) = ((gs + gp) * phi) * psibar%a(3:4)
  end function f_fsp
```

⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fs (gs, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gs
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi%a = (gs * phi) * psibar%a
  end function f_fs
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fp (gp, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gp
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi%a(1:2) = (- gp * phi) * psibar%a(1:2)
  psibarphi%a(3:4) = (  gp * phi) * psibar%a(3:4)
  end function f_fp
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fsl (gl, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gl
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi%a(1:2) = (2 * gl * phi) * psibar%a(1:2)
  psibarphi%a(3:4) = 0
  end function f_fsl
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fsr (gr, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gr
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi%a(1:2) = 0
  psibarphi%a(3:4) = (2 * gr * phi) * psibar%a(3:4)
  end function f_fsr
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_fslr (gl, gr, psibar, phi) result (psibarphi)
  type(conjspinor) :: psibarphi
  complex(kind=default), intent(in) :: gl, gr
  type(conjspinor), intent(in) :: psibar
  complex(kind=default), intent(in) :: phi
  psibarphi = f_fsp (gr+gl, gr-gl, psibar, phi)
  end function f_fslr
```
⟨*Declaration of spinor currents*⟩+≡
```
  public :: f_gravf, f_fgrav
```
⟨*Implementation of spinor currents*⟩+≡
```
  pure function f_gravf (g, m, kb, k, t, psi) result (tpsi)
  type(spinor) :: tpsi
  complex(kind=default), intent(in) :: g
  real(kind=default), intent(in) :: m
  type(spinor), intent(in) :: psi
  type(tensor), intent(in) :: t
  type(momentum), intent(in) :: kb, k
  complex(kind=default) :: g2, g8, t_tr
  type(vector) :: kkb
  kkb = k + kb
  g2 = g / 2.0_default
  g8 = g / 8.0_default
  t_tr = t%t(0,0) - t%t(1,1) - t%t(2,2) - t%t(3,3)
  tpsi = (- f_sf (g2, cmplx (m,0.0, kind=default), psi) &
  - f_vf ((g8*m), kkb, psi)) * t_tr - &
  f_vf (g8,(t*kkb + kkb*t),psi)
  end function f_gravf
```

⟨*Implementation of spinor currents*⟩+≡
```
pure function f_fgrav (g, m, kb, k, psibar, t) result (psibart)
type(conjspinor) :: psibart
complex(kind=default), intent(in) :: g
real(kind=default), intent(in) :: m
type(conjspinor), intent(in) :: psibar
type(tensor), intent(in) :: t
type(momentum), intent(in) :: kb, k
type(vector) :: kkb
complex(kind=default) :: g2, g8, t_tr
kkb = k + kb
g2 = g / 2.0_default
g8 = g / 8.0_default
t_tr = t%t(0,0) - t%t(1,1) - t%t(2,2) - t%t(3,3)
psibart = (- f_fs (g2, psibar, cmplx (m, 0.0, kind=default)) &
- f_fv ((g8 * m), psibar, kkb)) * t_tr - &
f_fv (g8,psibar,(t*kkb + kkb*t))
end function f_fgrav
```

AB.25.3 On Shell Wave Functions

⟨*Declaration of spinor on shell wave functions*⟩≡
```
public :: u, ubar, v, vbar
private :: chi_plus, chi_minus
```

$$\chi_+(\vec{p}) = \frac{1}{\sqrt{2|\vec{p}|(|\vec{p}|+p_3)}}\begin{pmatrix}|\vec{p}|+p_3\\ p_1+\mathrm{i}p_2\end{pmatrix} \tag{AB.101a}$$

$$\chi_-(\vec{p}) = \frac{1}{\sqrt{2|\vec{p}|(|\vec{p}|+p_3)}}\begin{pmatrix}-p_1+\mathrm{i}p_2\\ |\vec{p}|+p_3\end{pmatrix} \tag{AB.101b}$$

⟨*Implementation of spinor on shell wave functions*⟩≡
```
pure function chi_plus (p) result (chi)
complex(kind=default), dimension(2) :: chi
type(momentum), intent(in) :: p
real(kind=default) :: pabs
pabs = sqrt (dot_product (p%x, p%x))
if (pabs + p%x(3) <= 1000 * epsilon (pabs) * pabs) then
chi = (/ cmplx ( 0.0, 0.0, kind=default), &
cmplx ( 1.0, 0.0, kind=default) /)
else
chi = 1 / sqrt (2*pabs*(pabs + p%x(3))) &
* (/ cmplx (pabs + p%x(3), kind=default), &
cmplx (p%x(1), p%x(2), kind=default) /)
end if
end function chi_plus
```

⟨*Implementation of spinor on shell wave functions*⟩+≡
```
pure function chi_minus (p) result (chi)
complex(kind=default), dimension(2) :: chi
type(momentum), intent(in) :: p
real(kind=default) :: pabs
pabs = sqrt (dot_product (p%x, p%x))
if (pabs + p%x(3) <= 1000 * epsilon (pabs) * pabs) then
chi = (/ cmplx (-1.0, 0.0, kind=default), &
cmplx ( 0.0, 0.0, kind=default) /)
else
chi = 1 / sqrt (2*pabs*(pabs + p%x(3))) &
* (/ cmplx (-p%x(1), p%x(2), kind=default), &
cmplx (pabs + p%x(3), kind=default) /)
end if
end function chi_minus
```

$$u_\pm(p,|m|) = \begin{pmatrix}\sqrt{p_0\mp|\vec{p}|}\cdot\chi_\pm(\vec{p})\\ \sqrt{p_0\pm|\vec{p}|}\cdot\chi_\pm(\vec{p})\end{pmatrix} \qquad u_\pm(p,-|m|) = \begin{pmatrix}-i\sqrt{p_0\mp|\vec{p}|}\cdot\chi_\pm(\vec{p})\\ +i\sqrt{p_0\pm|\vec{p}|}\cdot\chi_\pm(\vec{p})\end{pmatrix} \tag{AB.102}$$

Determining the mass from the momenta is a numerically haphazardous for light particles. Therefore, we accept some redundancy and pass the mass explicitely. Even if the mass is not used in the chiral representation, we do so for symmetry with polarization vectors and to be prepared for other representations.

⟨*Implementation of spinor on shell wave functions*⟩+≡
```
  pure function u (mass, p, s) result (psi)
  type(spinor) :: psi
  real(kind=default), intent(in) :: mass
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  complex(kind=default), dimension(2) :: chi
  real(kind=default) :: pabs, delta, m
  m = abs(mass)
  pabs = sqrt (dot_product (p%x, p%x))
  if (m < epsilon (m) * pabs) then
  delta = 0
  else
  delta = sqrt (max (p%t - pabs, 0._default))
  end if
  select case (s)
  case (1)
  chi = chi_plus (p)
  psi%a(1:2) = delta * chi
  psi%a(3:4) = sqrt (p%t + pabs) * chi
  case (-1)
  chi = chi_minus (p)
  psi%a(1:2) = sqrt (p%t + pabs) * chi
  psi%a(3:4) = delta * chi
  case default
  pabs = m ! make the compiler happy and use m
  psi%a = 0
  end select
  if (mass < 0) then
  psi%a(1:2) = - imago * psi%a(1:2)
  psi%a(3:4) = + imago * psi%a(3:4)
  end if
  end function u
```

⟨*Implementation of spinor on shell wave functions*⟩+≡
```
  pure function ubar (m, p, s) result (psibar)
  type(conjspinor) :: psibar
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  type(spinor) :: psi
  psi = u (m, p, s)
  psibar%a(1:2) = conjg (psi%a(3:4))
  psibar%a(3:4) = conjg (psi%a(1:2))
  end function ubar
```

$$v_{\pm}(p) = \begin{pmatrix} \mp\sqrt{p_0 \pm |\vec{p}|} \cdot \chi_{\mp}(\vec{p}) \\ \pm\sqrt{p_0 \mp |\vec{p}|} \cdot \chi_{\mp}(\vec{p}) \end{pmatrix} \tag{AB.103}$$

⟨*Implementation of spinor on shell wave functions*⟩+≡
```
  pure function v (mass, p, s) result (psi)
  type(spinor) :: psi
  real(kind=default), intent(in) :: mass
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  complex(kind=default), dimension(2) :: chi
  real(kind=default) :: pabs, delta, m
  m = abs(mass)
  pabs = sqrt (dot_product (p%x, p%x))
  if (m < epsilon (m) * pabs) then
  delta = 0
  else
  delta = sqrt (max (p%t - pabs, 0._default))
  end if
```

```
select case (s)
case (1)
chi = chi_minus (p)
psi%a(1:2) = - sqrt (p%t + pabs) * chi
psi%a(3:4) =   delta * chi
case (-1)
chi = chi_plus (p)
psi%a(1:2) =   delta * chi
psi%a(3:4) = - sqrt (p%t + pabs) * chi
case default
pabs = m ! make the compiler happy and use m
psi%a = 0
end select
if (mass < 0) then
psi%a(1:2) = - imago * psi%a(1:2)
psi%a(3:4) = + imago * psi%a(3:4)
end if
end function v
```

⟨*Implementation of spinor on shell wave functions*⟩+≡

```
pure function vbar (m, p, s) result (psibar)
type(conjspinor) :: psibar
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: p
integer, intent(in) :: s
type(spinor) :: psi
psi = v (m, p, s)
psibar%a(1:2) = conjg (psi%a(3:4))
psibar%a(3:4) = conjg (psi%a(1:2))
end function vbar
```

AB.25.4 Off Shell Wave Functions

I've just taken this over from Christian Schwinn's version.

⟨*Declaration of spinor off shell wave functions*⟩≡

```
public :: brs_u, brs_ubar, brs_v, brs_vbar
```

The off-shell wave functions needed for gauge checking are obtained from the LSZ-formulas:

$$\langle \mathrm{Out}|d^\dagger|\mathrm{In}\rangle = i\int d^4x \bar{v} e^{-ikx}(i\partial\!\!\!/ - m)\langle \mathrm{Out}|\psi|\mathrm{In}\rangle \tag{AB.104a}$$

$$\langle \mathrm{Out}|b|\mathrm{In}\rangle = -i\int d^4x \bar{u} e^{ikx}(i\partial\!\!\!/ - m)\langle \mathrm{Out}|\psi|\mathrm{In}\rangle \tag{AB.104b}$$

$$\langle \mathrm{Out}|d|\mathrm{In}\rangle = i\int d^4x \langle \mathrm{Out}|\bar\psi|\mathrm{In}\rangle (-i\overleftarrow{\partial\!\!\!/} - m) v e^{ikx} \tag{AB.104c}$$

$$\langle \mathrm{Out}|b^\dagger|\mathrm{In}\rangle = -i\int d^4x \langle \mathrm{Out}|\bar\psi|\mathrm{In}\rangle (-i\overleftarrow{\partial\!\!\!/} - m) u e^{-ikx} \tag{AB.104d}$$

Since the relative sign between fermions and antifermions is ignored for on-shell amplitudes we must also ignore it here, so all wavefunctions must have a $(-i)$ factor. In momentum space we have:

$$brsu(p) = (-i)(p\!\!\!/ - m)u(p) \tag{AB.105}$$

⟨*Implementation of spinor off shell wave functions*⟩≡

```
pure function brs_u (m, p, s) result (dpsi)
type(spinor) :: dpsi,psi
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: p
integer, intent(in) :: s
type (vector)::vp
complex(kind=default), parameter :: one = (1, 0)
vp=p
psi=u(m,p,s)
dpsi=cmplx(0.0,-1.0)*(f_vf(one,vp,psi)-m*psi)
end function brs_u
```

$$brsv(p) = i(\not p + m)v(p) \tag{AB.106}$$

⟨*Implementation of spinor off shell wave functions*⟩+≡
```
  pure function brs_v (m, p, s) result (dpsi)
  type(spinor) :: dpsi, psi
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) ::   s
  type (vector)::vp
  complex(kind=default), parameter :: one = (1, 0)
  vp=p
  psi=v(m,p,s)
  dpsi=cmplx(0.0,1.0)*(f_vf(one,vp,psi)+m*psi)
  end function brs_v
```

$$brs\bar u(p) = (-i)\bar u(p)(\not p - m) \tag{AB.107}$$

⟨*Implementation of spinor off shell wave functions*⟩+≡
```
  pure function brs_ubar (m, p, s)result (dpsibar)
  type(conjspinor) :: dpsibar, psibar
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  type (vector)::vp
  complex(kind=default), parameter :: one = (1, 0)
  vp=p
  psibar=ubar(m,p,s)
  dpsibar=cmplx(0.0,-1.0)*(f_fv(one,psibar,vp)-m*psibar)
  end function brs_ubar
```

$$brs\bar v(p) = (i)\bar v(p)(\not p + m) \tag{AB.108}$$

⟨*Implementation of spinor off shell wave functions*⟩+≡
```
  pure function brs_vbar (m, p, s) result (dpsibar)
  type(conjspinor) :: dpsibar,psibar
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  type(vector)::vp
  complex(kind=default), parameter :: one = (1, 0)
  vp=p
  psibar=vbar(m,p,s)
  dpsibar=cmplx(0.0,1.0)*(f_fv(one,psibar,vp)+m*psibar)
  end function brs_vbar
```

NB: The remarks on momentum flow in the propagators don't apply here since the incoming momenta are flipped for the wave functions.

AB.25.5 Propagators

NB: the common factor of i is extracted:

⟨*Declaration of spinor propagators*⟩≡
```
  public :: pr_psi, pr_psibar
  public :: pj_psi, pj_psibar
  public :: pg_psi, pg_psibar
```

$$\frac{i(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\psi \tag{AB.109}$$

NB: the sign of the momentum comes about because all momenta are treated as *outgoing* and the particle charge flow is therefore opposite to the momentum.

⟨*Implementation of spinor propagators*⟩≡
```
  pure function pr_psi (p, m, w, cms, psi) result (ppsi)
  type(spinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(spinor), intent(in) :: psi
  logical, intent(in) :: cms
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  complex(kind=default) :: num_mass
  vp = p
  if (cms) then
  num_mass = sqrt(cmplx(m**2, -m*w, kind=default))
  else
  num_mass = cmplx (m, 0, kind=default)
  end if
  ppsi = (1 / cmplx (p*p - m**2, m*w, kind=default)) &
  * (- f_vf (one, vp, psi) + num_mass * psi)
  end function pr_psi
```

$$\sqrt{\frac{\pi}{M\Gamma}}(-\not p+m)\psi \tag{AB.110}$$

⟨*Implementation of spinor propagators*⟩+≡

```
  pure function pj_psi (p, m, w, psi) result (ppsi)
  type(spinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(spinor), intent(in) :: psi
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  vp = p
  ppsi = (0, -1) * sqrt (PI / m / w) * (- f_vf (one, vp, psi) + m * psi)
  end function pj_psi
```

⟨*Implementation of spinor propagators*⟩+≡

```
  pure function pg_psi (p, m, w, psi) result (ppsi)
  type(spinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(spinor), intent(in) :: psi
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  vp = p
  ppsi = gauss(p*p, m, w) *  (- f_vf (one, vp, psi) + m * psi)
  end function pg_psi
```

$$\bar\psi\frac{i(\not p+m)}{p^2-m^2+im\Gamma} \tag{AB.111}$$

NB: the sign of the momentum comes about because all momenta are treated as *outgoing* and the antiparticle charge flow is therefore parallel to the momentum.

⟨*Implementation of spinor propagators*⟩+≡

```
  pure function pr_psibar (p, m, w, cms, psibar) result (ppsibar)
  type(conjspinor) :: ppsibar
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(conjspinor), intent(in) :: psibar
  logical, intent(in) :: cms
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  complex(kind=default) :: num_mass
  vp = p
  if (cms) then
  num_mass = sqrt(cmplx(m**2, -m*w, kind=default))
  else
  num_mass = cmplx (m, 0, kind=default)
  end if
  ppsibar = (1 / cmplx (p*p - m**2, m*w, kind=default)) &
  * (f_fv (one, psibar, vp) + num_mass * psibar)
  end function pr_psibar
```

$$\sqrt{\frac{\pi}{M\Gamma}}\bar\psi(\not p+m) \tag{AB.112}$$

NB: the sign of the momentum comes about because all momenta are treated as *outgoing* and the antiparticle charge flow is therefore parallel to the momentum.

⟨*Implementation of spinor propagators*⟩+≡

```
pure function pj_psibar (p, m, w, psibar) result (ppsibar)
type(conjspinor) :: ppsibar
type(momentum), intent(in) :: p
real(kind=default), intent(in) :: m, w
type(conjspinor), intent(in) :: psibar
type(vector) :: vp
complex(kind=default), parameter :: one = (1, 0)
vp = p
ppsibar = (0, -1) * sqrt (PI / m / w) * (f_fv (one, psibar, vp) + m * psibar)
end function pj_psibar
```

⟨*Implementation of spinor propagators*⟩+≡

```
pure function pg_psibar (p, m, w, psibar) result (ppsibar)
type(conjspinor) :: ppsibar
type(momentum), intent(in) :: p
real(kind=default), intent(in) :: m, w
type(conjspinor), intent(in) :: psibar
type(vector) :: vp
complex(kind=default), parameter :: one = (1, 0)
vp = p
ppsibar = gauss (p*p, m, w) * (f_fv (one, psibar, vp) + m * psibar)
end function pg_psibar
```

$$\frac{i(-\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\sum_n \psi_n \otimes \bar\psi_n \tag{AB.113}$$

NB: the temporary variables `psi(1:4)` are not nice, but the compilers should be able to optimize the unnecessary copies away. In any case, even if the copies are performed, they are (probably) negligible compared to the floating point multiplications anyway ...

⟨*(Not used yet) Declaration of operations for spinors*⟩≡

```
type, public :: spinordyad
! private (omegalib needs access, but DON'T TOUCH IT!)
complex(kind=default), dimension(4,4) :: a
end type spinordyad
```

⟨*(Not used yet) Implementation of spinor propagators*⟩≡

```
pure function pr_dyadleft (p, m, w, psipsibar) result (psipsibarp)
type(spinordyad) :: psipsibarp
type(momentum), intent(in) :: p
real(kind=default), intent(in) :: m, w
type(spinordyad), intent(in) :: psipsibar
integer :: i
type(vector) :: vp
type(spinor), dimension(4) :: psi
complex(kind=default) :: pole
complex(kind=default), parameter :: one = (1, 0)
vp = p
pole = 1 / cmplx (p*p - m**2, m*w, kind=default)
do i = 1, 4
psi(i)%a = psipsibar%a(:,i)
psi(i) = pole * (- f_vf (one, vp, psi(i)) + m * psi(i))
psipsibarp%a(:,i) = psi(i)%a
end do
end function pr_dyadleft
```

$$\sum_n \psi_n \otimes \bar\psi_n \frac{i(\not p + m)}{p^2 - m^2 + \mathrm{i}m\Gamma} \tag{AB.114}$$

⟨*(Not used yet) Implementation of spinor propagators*⟩+≡

```
pure function pr_dyadright (p, m, w, psipsibar) result (psipsibarp)
type(spinordyad) :: psipsibarp
type(momentum), intent(in) :: p
real(kind=default), intent(in) :: m, w
type(spinordyad), intent(in) :: psipsibar
integer :: i
type(vector) :: vp
type(conjspinor), dimension(4) :: psibar
complex(kind=default) :: pole
complex(kind=default), parameter :: one = (1, 0)
vp = p
pole = 1 / cmplx (p*p - m**2, m*w, kind=default)
do i = 1, 4
psibar(i)%a = psipsibar%a(i,:)
psibar(i) = pole * (f_fv (one, psibar(i), vp) + m * psibar(i))
psipsibarp%a(i,:) = psibar(i)%a
end do
end function pr_dyadright
```

AB.26 Spinor Couplings Revisited

⟨omega_bispinor_couplings.f90⟩≡

```
⟨Copyleft⟩
module omega_bispinor_couplings
use kinds
use constants
use omega_bispinors
use omega_vectorspinors
use omega_vectors
use omega_couplings
implicit none
private
⟨Declaration of bispinor on shell wave functions⟩
⟨Declaration of bispinor off shell wave functions⟩
⟨Declaration of bispinor currents⟩
⟨Declaration of bispinor propagators⟩
integer, parameter, public :: omega_bispinor_cpls_2010_01_A = 0
contains
⟨Implementation of bispinor on shell wave functions⟩
⟨Implementation of bispinor off shell wave functions⟩
⟨Implementation of bispinor currents⟩
⟨Implementation of bispinor propagators⟩
end module omega_bispinor_couplings
```

See table AB.1 for the names of Fortran functions. We could have used long names instead, but this would increase the chance of running past continuation line limits without adding much to the legibility.

AB.26.1 Fermionic Vector and Axial Couplings

There's more than one chiral representation. This one is compatible with HELAS [5].

$$\gamma^0 = \begin{pmatrix} 0 & \mathbf{1} \\ \mathbf{1} & 0 \end{pmatrix}, \gamma^i = \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix}, \gamma_5 = i\gamma^0\gamma^1\gamma^2\gamma^3 = \begin{pmatrix} -\mathbf{1} & 0 \\ 0 & \mathbf{1} \end{pmatrix}, \tag{AB.115a}$$

$$C = \begin{pmatrix} \epsilon & 0 \\ 0 & -\epsilon \end{pmatrix}, \qquad \epsilon = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}. \tag{AB.115b}$$

Therefore

$$g_S + g_P\gamma_5 = \begin{pmatrix} g_S - g_P & 0 & 0 & 0 \\ 0 & g_S - g_P & 0 & 0 \\ 0 & 0 & g_S + g_P & 0 \\ 0 & 0 & 0 & g_S + g_P \end{pmatrix} \tag{AB.116a}$$

$$g_V\gamma^0 - g_A\gamma^0\gamma_5 = \begin{pmatrix} 0 & 0 & g_V - g_A & 0 \\ 0 & 0 & 0 & g_V - g_A \\ g_V + g_A & 0 & 0 & 0 \\ 0 & g_V + g_A & 0 & 0 \end{pmatrix} \tag{AB.116b}$$

$$g_V\gamma^1 - g_A\gamma^1\gamma_5 = \begin{pmatrix} 0 & 0 & 0 & g_V - g_A \\ 0 & 0 & g_V - g_A & 0 \\ 0 & -g_V - g_A & 0 & 0 \\ -g_V - g_A & 0 & 0 & 0 \end{pmatrix} \tag{AB.116c}$$

$$g_V\gamma^2 - g_A\gamma^2\gamma_5 = \begin{pmatrix} 0 & 0 & 0 & -\mathrm{i}(g_V-g_A) \\ 0 & 0 & \mathrm{i}(g_V-g_A) & 0 \\ 0 & \mathrm{i}(g_V+g_A) & 0 & 0 \\ -\mathrm{i}(g_V+g_A) & 0 & 0 & 0 \end{pmatrix} \tag{AB.116d}$$

$$g_V\gamma^3 - g_A\gamma^3\gamma_5 = \begin{pmatrix} 0 & 0 & g_V-g_A & 0 \\ 0 & 0 & 0 & -g_V+g_A \\ -g_V-g_A & 0 & 0 & 0 \\ 0 & g_V+g_A & 0 & 0 \end{pmatrix} \tag{AB.116e}$$

and

$$C(g_S + g_P\gamma_5) = \begin{pmatrix} 0 & g_S-g_P & 0 & 0 \\ -g_S+g_P & 0 & 0 & 0 \\ 0 & 0 & 0 & -g_S-g_P \\ 0 & 0 & g_S+g_P & 0 \end{pmatrix} \tag{AB.117a}$$

$$C(g_V\gamma^0 - g_A\gamma^0\gamma_5) = \begin{pmatrix} 0 & 0 & 0 & g_V-g_A \\ 0 & 0 & -g_V+g_A & 0 \\ 0 & -g_V-g_A & 0 & 0 \\ g_V+g_A & 0 & 0 & 0 \end{pmatrix} \tag{AB.117b}$$

$$C(g_V\gamma^1 - g_A\gamma^1\gamma_5) = \begin{pmatrix} 0 & 0 & g_V-g_A & 0 \\ 0 & 0 & 0 & -g_V+g_A \\ g_V+g_A & 0 & 0 & 0 \\ 0 & -g_V-g_A & 0 & 0 \end{pmatrix} \tag{AB.117c}$$

$$C(g_V\gamma^2 - g_A\gamma^2\gamma_5) = \begin{pmatrix} 0 & 0 & \mathrm{i}(g_V-g_A) & 0 \\ 0 & 0 & 0 & \mathrm{i}(g_V-g_A) \\ \mathrm{i}(g_V+g_A) & 0 & 0 & 0 \\ 0 & \mathrm{i}(g_V+g_A) & 0 & 0 \end{pmatrix} \tag{AB.117d}$$

$$C(g_V\gamma^3 - g_A\gamma^3\gamma_5) = \begin{pmatrix} 0 & 0 & 0 & -g_V+g_A \\ 0 & 0 & -g_V+g_A & 0 \\ 0 & -g_V-g_A & 0 & 0 \\ -g_V-g_A & 0 & 0 & 0 \end{pmatrix} \tag{AB.117e}$$

⟨*Declaration of bispinor currents*⟩≡
```
  public :: va_ff, v_ff, a_ff, vl_ff, vr_ff, vlr_ff, va2_ff, tva_ff, tvam_ff, &
  tlr_ff, tlrm_ff
```

⟨*Implementation of bispinor currents*⟩≡
```
  pure function va_ff (gv, ga, psil, psir) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv, ga
  type(bispinor), intent(in) :: psil, psir
  complex(kind=default) :: gl, gr
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  gl = gv + ga
  gr = gv - ga
  g13 = psil%a(1)*psir%a(3)
  g14 = psil%a(1)*psir%a(4)
  g23 = psil%a(2)*psir%a(3)
  g24 = psil%a(2)*psir%a(4)
  g31 = psil%a(3)*psir%a(1)
  g32 = psil%a(3)*psir%a(2)
  g41 = psil%a(4)*psir%a(1)
  g42 = psil%a(4)*psir%a(2)
  j%t    =  gr * (   g14 - g23) + gl * ( - g32 + g41)
  j%x(1) =  gr * (   g13 - g24) + gl * (   g31 - g42)
  j%x(2) = (gr * (   g13 + g24) + gl * (   g31 + g42)) * (0, 1)
  j%x(3) =  gr * ( - g14 - g23) + gl * ( - g32 - g41)
  end function va_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function va2_ff (gva, psil, psir) result (j)
  type(vector) :: j
  complex(kind=default), intent(in), dimension(2) :: gva
  type(bispinor), intent(in) :: psil, psir
  complex(kind=default) :: gl, gr
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  gl = gva(1) + gva(2)
  gr = gva(1) - gva(2)
  g13 = psil%a(1)*psir%a(3)
  g14 = psil%a(1)*psir%a(4)
  g23 = psil%a(2)*psir%a(3)
  g24 = psil%a(2)*psir%a(4)
  g31 = psil%a(3)*psir%a(1)
  g32 = psil%a(3)*psir%a(2)
  g41 = psil%a(4)*psir%a(1)
  g42 = psil%a(4)*psir%a(2)
  j%t    =  gr * (   g14 - g23) + gl * ( - g32 + g41)
  j%x(1) =  gr * (   g13 - g24) + gl * (   g31 - g42)
  j%x(2) = (gr * (   g13 + g24) + gl * (   g31 + g42)) * (0, 1)
  j%x(3) =  gr * ( - g14 - g23) + gl * ( - g32 - g41)
  end function va2_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v_ff (gv, psil, psir) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv
  type(bispinor), intent(in) :: psil, psir
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  g13 = psil%a(1)*psir%a(3)
  g14 = psil%a(1)*psir%a(4)
  g23 = psil%a(2)*psir%a(3)
  g24 = psil%a(2)*psir%a(4)
  g31 = psil%a(3)*psir%a(1)
  g32 = psil%a(3)*psir%a(2)
  g41 = psil%a(4)*psir%a(1)
  g42 = psil%a(4)*psir%a(2)
  j%t    =   gv * (   g14 - g23 - g32 + g41)
  j%x(1) =   gv * (   g13 - g24 + g31 - g42)
  j%x(2) =   gv * (   g13 + g24 + g31 + g42) * (0, 1)
  j%x(3) =   gv * ( - g14 - g23 - g32 - g41)
  end function v_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function a_ff (ga, psil, psir) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: ga
  type(bispinor), intent(in) :: psil, psir
  complex(kind=default) :: g13, g14, g23, g24, g31, g32, g41, g42
  g13 = psil%a(1)*psir%a(3)
  g14 = psil%a(1)*psir%a(4)
  g23 = psil%a(2)*psir%a(3)
  g24 = psil%a(2)*psir%a(4)
  g31 = psil%a(3)*psir%a(1)
  g32 = psil%a(3)*psir%a(2)
  g41 = psil%a(4)*psir%a(1)
  g42 = psil%a(4)*psir%a(2)
  j%t    =  -ga * (   g14 - g23 + g32 - g41)
  j%x(1) =  -ga * (   g13 - g24 - g31 + g42)
  j%x(2) =  -ga * (   g13 + g24 - g31 - g42) * (0, 1)
  j%x(3) =  -ga * ( - g14 - g23 + g32 + g41)
  end function a_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function vl_ff (gl, psil, psir) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl
  type(bispinor), intent(in) :: psil, psir
  complex(kind=default) :: gl2
  complex(kind=default) :: g31, g32, g41, g42
  gl2 = 2 * gl
  g31 = psil%a(3)*psir%a(1)
  g32 = psil%a(3)*psir%a(2)
```

```
g41 = psil%a(4)*psir%a(1)
g42 = psil%a(4)*psir%a(2)
j%t    =   gl2 * ( - g32 + g41)
j%x(1) =   gl2 * (   g31 - g42)
j%x(2) =   gl2 * (   g31 + g42) * (0, 1)
j%x(3) =   gl2 * ( - g32 - g41)
end function vl_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function vr_ff (gr, psil, psir) result (j)
type(vector) :: j
complex(kind=default), intent(in) :: gr
type(bispinor), intent(in) :: psil, psir
complex(kind=default) :: gr2
complex(kind=default) :: g13, g14, g23, g24
gr2 = 2 * gr
g13 = psil%a(1)*psir%a(3)
g14 = psil%a(1)*psir%a(4)
g23 = psil%a(2)*psir%a(3)
g24 = psil%a(2)*psir%a(4)
j%t    = gr2 * (   g14 - g23)
j%x(1) = gr2 * (   g13 - g24)
j%x(2) = gr2 * (   g13 + g24) * (0, 1)
j%x(3) = gr2 * ( - g14 - g23)
end function vr_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function vlr_ff (gl, gr, psibar, psi) result (j)
type(vector) :: j
complex(kind=default), intent(in) :: gl, gr
type(bispinor), intent(in) :: psibar
type(bispinor), intent(in) :: psi
j = va_ff (gl+gr, gl-gr, psibar, psi)
end function vlr_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function tva_ff (gv, ga, psibar, psi) result (t)
type(tensor2odd) :: t
complex(kind=default), intent(in) :: gv, ga
type(bispinor), intent(in) :: psibar
type(bispinor), intent(in) :: psi
complex(kind=default) :: gl, gr
complex(kind=default) :: g11, g22, g33, g44, g1p2, g3p4
gr     = gv + ga
gl     = gv - ga
g11    = psibar%a(1)*psi%a(1)
g22    = psibar%a(2)*psi%a(2)
g1p2   = psibar%a(1)*psi%a(2) + psibar%a(2)*psi%a(1)
g3p4   = psibar%a(3)*psi%a(4) + psibar%a(4)*psi%a(3)
g33    = psibar%a(3)*psi%a(3)
g44    = psibar%a(4)*psi%a(4)
t%e(1) = (gl * ( - g11 + g22) + gr * ( - g33 + g44)) * (0, 1)
t%e(2) =  gl * (   g11 + g22) + gr * (   g33 + g44)
t%e(3) = (gl * (   g1p2     ) + gr * (   g3p4     )) * (0, 1)
t%b(1) =  gl * (   g11 - g22) + gr * ( - g33 + g44)
t%b(2) = (gl * (   g11 + g22) + gr * ( - g33 - g44)) * (0, 1)
t%b(3) =  gl * ( - g1p2     ) + gr * (   g3p4     )
end function tva_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function tlr_ff (gl, gr, psibar, psi) result (t)
type(tensor2odd) :: t
complex(kind=default), intent(in) :: gl, gr
type(bispinor), intent(in) :: psibar
type(bispinor), intent(in) :: psi
t = tva_ff (gr+gl, gr-gl, psibar, psi)
end function tlr_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function tvam_ff (gv, ga, psibar, psi, p) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gv, ga
  type(bispinor), intent(in) :: psibar
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: p
  j = (tva_ff(gv, ga, psibar, psi) * p) * (0,1)
  end function tvam_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function tlrm_ff (gl, gr, psibar, psi, p) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(bispinor), intent(in) :: psibar
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: p
  j = tvam_ff (gr+gl, gr-gl, psibar, psi, p)
  end function tlrm_ff
```

and

$$\not{v} - \not{a}\gamma_5 = \begin{pmatrix} 0 & 0 & v_- - a_- & -v^* + a^* \\ 0 & 0 & -v + a & v_+ - a_+ \\ v_+ + a_+ & v^* + a^* & 0 & 0 \\ v + a & v_- + a_- & 0 & 0 \end{pmatrix} \tag{AB.118}$$

with $v_\pm = v_0 \pm v_3$, $a_\pm = a_0 \pm a_3$, $v = v_1 + \mathrm{i}v_2$, $v^* = v_1 - \mathrm{i}v_2$, $a = a_1 + \mathrm{i}a_2$, and $a^* = a_1 - \mathrm{i}a_2$. But note that $\cdot^*$ is *not* complex conjugation for complex v_μ or a_μ.

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: f_vaf, f_vf, f_af, f_vlf, f_vrf, f_vlrf, f_va2f, &
  f_tvaf, f_tlrf, f_tvamf, f_tlrmf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_vaf (gv, ga, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gv, ga
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gv + ga
  gr = gv - ga
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gl * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vaf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_va2f (gva, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in), dimension(2) :: gva
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: vp, vm, v12, v12s
  gl = gva(1) + gva(2)
  gr = gva(1) - gva(2)
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gl * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_va2f
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_vf (gv, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gv
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gv * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gv * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = gv * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gv * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_af (ga, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: ga
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: vp, vm, v12, v12s
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = ga * ( - vm  * psi%a(3) + v12s * psi%a(4))
  vpsi%a(2) = ga * (   v12 * psi%a(3) - vp   * psi%a(4))
  vpsi%a(3) = ga * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = ga * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_af
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_vlf (gl, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gl
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: gl2
  complex(kind=default) :: vp, vm, v12, v12s
  gl2 = 2 * gl
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = 0
  vpsi%a(2) = 0
  vpsi%a(3) = gl2 * (   vp  * psi%a(1) + v12s * psi%a(2))
  vpsi%a(4) = gl2 * (   v12 * psi%a(1) + vm   * psi%a(2))
  end function f_vlf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_vrf (gr, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gr
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: gr2
  complex(kind=default) :: vp, vm, v12, v12s
  gr2 = 2 * gr
  vp = v%t + v%x(3)
  vm = v%t - v%x(3)
  v12  =  v%x(1) + (0,1)*v%x(2)
  v12s =  v%x(1) - (0,1)*v%x(2)
  vpsi%a(1) = gr2 * (   vm  * psi%a(3) - v12s * psi%a(4))
  vpsi%a(2) = gr2 * ( - v12 * psi%a(3) + vp   * psi%a(4))
  vpsi%a(3) = 0
  vpsi%a(4) = 0
  end function f_vrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_vlrf (gl, gr, v, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gl, gr
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  vpsi = f_vaf (gl+gr, gl-gr, v, psi)
  end function f_vlrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_tvaf (gv, ga, t, psi) result (tpsi)
  type(bispinor) :: tpsi
  complex(kind=default), intent(in) :: gv, ga
  type(tensor2odd), intent(in) :: t
  type(bispinor), intent(in) :: psi
  complex(kind=default) :: gl, gr
  complex(kind=default) :: e21, e21s, b12, b12s, be3, be3s
  gr   = gv + ga
  gl   = gv - ga
  e21  = t%e(2) + t%e(1)*(0,1)
  e21s = t%e(2) - t%e(1)*(0,1)
  b12  = t%b(1) + t%b(2)*(0,1)
  b12s = t%b(1) - t%b(2)*(0,1)
  be3  = t%b(3) + t%e(3)*(0,1)
  be3s = t%b(3) - t%e(3)*(0,1)
  tpsi%a(1) =   2*gl * (   psi%a(1) * be3  + psi%a(2) * ( e21 +b12s))
  tpsi%a(2) =   2*gl * ( - psi%a(2) * be3  + psi%a(1) * (-e21s+b12 ))
  tpsi%a(3) =   2*gr * (   psi%a(3) * be3s + psi%a(4) * (-e21 +b12s))
  tpsi%a(4) =   2*gr * ( - psi%a(4) * be3s + psi%a(3) * ( e21s+b12 ))
  end function f_tvaf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_tlrf (gl, gr, t, psi) result (tpsi)
  type(bispinor) :: tpsi
  complex(kind=default), intent(in) :: gl, gr
  type(tensor2odd), intent(in) :: t
  type(bispinor), intent(in) :: psi
  tpsi = f_tvaf (gr+gl, gr-gl, t, psi)
  end function f_tlrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_tvamf (gv, ga, v, psi, k) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gv, ga
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  type(tensor2odd) :: t
  t = (v.wedge.k) * (0, 0.5)
  vpsi = f_tvaf(gv, ga, t, psi)
  end function f_tvamf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_tlrmf (gl, gr, v, psi, k) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: gl, gr
  type(vector), intent(in) :: v
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  vpsi = f_tvamf (gr+gl, gr-gl, v, psi, k)
  end function f_tlrmf
```

AB.26.2 Fermionic Scalar and Pseudo Scalar Couplings

⟨*Declaration of bispinor currents*⟩+≡
```
  public :: sp_ff, s_ff, p_ff, sl_ff, sr_ff, slr_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function sp_ff (gs, gp, psil, psir) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gs, gp
  type(bispinor), intent(in) :: psil, psir
  j =    (gs - gp) * (psil%a(1)*psir%a(2) - psil%a(2)*psir%a(1)) &
  + (gs + gp) * (- psil%a(3)*psir%a(4) + psil%a(4)*psir%a(3))
  end function sp_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function s_ff (gs, psil, psir) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gs
  type(bispinor), intent(in) :: psil, psir
  j = gs * (psil * psir)
  end function s_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function p_ff (gp, psil, psir) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gp
  type(bispinor), intent(in) :: psil, psir
  j = gp * (- psil%a(1)*psir%a(2) + psil%a(2)*psir%a(1) &
  - psil%a(3)*psir%a(4) + psil%a(4)*psir%a(3))
  end function p_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function sl_ff (gl, psil, psir) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl
  type(bispinor), intent(in) :: psil, psir
  j =  2 * gl * (psil%a(1)*psir%a(2) - psil%a(2)*psir%a(1))
  end function sl_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function sr_ff (gr, psil, psir) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gr
  type(bispinor), intent(in) :: psil, psir
  j = 2 * gr * (- psil%a(3)*psir%a(4) + psil%a(4)*psir%a(3))
  end function sr_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function slr_ff (gl, gr, psibar, psi) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(bispinor), intent(in) :: psibar
  type(bispinor), intent(in) :: psi
  j = sp_ff (gr+gl, gr-gl, psibar, psi)
  end function slr_ff
```

⟨*Declaration of bispinor currents*⟩+≡
```
  public :: f_spf, f_sf, f_pf, f_slf, f_srf, f_slrf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_spf (gs, gp, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gs, gp
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%a(1:2) = ((gs - gp) * phi) * psi%a(1:2)
  phipsi%a(3:4) = ((gs + gp) * phi) * psi%a(3:4)
  end function f_spf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_sf (gs, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gs
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%a = (gs * phi) * psi%a
  end function f_sf
```
⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_pf (gp, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gp
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%a(1:2) = (- gp * phi) * psi%a(1:2)
  phipsi%a(3:4) = (  gp * phi) * psi%a(3:4)
  end function f_pf
```
⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_slf (gl, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gl
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%a(1:2) = (2 * gl * phi) * psi%a(1:2)
  phipsi%a(3:4) = 0
  end function f_slf
```
⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_srf (gr, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gr
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%a(1:2) = 0
  phipsi%a(3:4) = (2 * gr * phi) * psi%a(3:4)
  end function f_srf
```
⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_slrf (gl, gr, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: gl, gr
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi =  f_spf (gr+gl, gr-gl, phi, psi)
  end function f_slrf
```

AB.26.3 Couplings for BRST Transformations

3-Couplings

The lists of needed gamma matrices can be found in the next subsection with the gravitino couplings.

⟨*Declaration of bispinor currents*⟩+≡
```
  private :: vv_ff, f_vvf
```
⟨*Declaration of bispinor currents*⟩+≡
```
  public :: vmom_ff, mom_ff, mom5_ff, moml_ff, momr_ff, lmom_ff, rmom_ff
```
⟨*Implementation of bispinor currents*⟩+≡
```
  pure function vv_ff (psibar, psi, k) result (psibarpsi)
  type(vector) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: k
  complex(kind=default) :: kp, km, k12, k12s
  type(bispinor) :: kgpsi1, kgpsi2, kgpsi3, kgpsi4
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  kgpsi1%a(1) = -k%x(3) * psi%a(1) - k12s * psi%a(2)
  kgpsi1%a(2) = -k12 * psi%a(1) + k%x(3) * psi%a(2)
  kgpsi1%a(3) = k%x(3) * psi%a(3) + k12s * psi%a(4)
  kgpsi1%a(4) = k12 * psi%a(3) - k%x(3) * psi%a(4)
  kgpsi2%a(1) = ((0,-1) * k%x(2)) * psi%a(1) - km * psi%a(2)
  kgpsi2%a(2) = - kp * psi%a(1) + ((0,1) * k%x(2)) * psi%a(2)
  kgpsi2%a(3) = ((0,-1) * k%x(2)) * psi%a(3) + kp * psi%a(4)
  kgpsi2%a(4) = km * psi%a(3) + ((0,1) * k%x(2)) * psi%a(4)
  kgpsi3%a(1) = (0,1) * (k%x(1) * psi%a(1) + km * psi%a(2))
  kgpsi3%a(2) = (0,-1) * (kp * psi%a(1) + k%x(1) * psi%a(2))
  kgpsi3%a(3) = (0,1) * (k%x(1) * psi%a(3) - kp * psi%a(4))
  kgpsi3%a(4) = (0,1) * (km * psi%a(3) - k%x(1) * psi%a(4))
  kgpsi4%a(1) = -k%t * psi%a(1) - k12s * psi%a(2)
  kgpsi4%a(2) = k12 * psi%a(1) + k%t * psi%a(2)
  kgpsi4%a(3) = k%t * psi%a(3) - k12s * psi%a(4)
  kgpsi4%a(4) = k12 * psi%a(3) - k%t * psi%a(4)
  psibarpsi%t    = 2 * (psibar * kgpsi1)
  psibarpsi%x(1) = 2 * (psibar * kgpsi2)
  psibarpsi%x(2) = 2 * (psibar * kgpsi3)
  psibarpsi%x(3) = 2 * (psibar * kgpsi4)
  end function vv_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_vvf (v, psi, k) result (kvpsi)
  type(bispinor) :: kvpsi
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: k, v
  complex(kind=default) :: kv30, kv21, kv01, kv31, kv02, kv32
  complex(kind=default) :: ap, am, bp, bm, bps, bms
  kv30 = k%x(3) * v%t - k%t * v%x(3)
  kv21 = (0,1) * (k%x(2) * v%x(1) - k%x(1) * v%x(2))
  kv01 = k%t * v%x(1) - k%x(1) * v%t
  kv31 = k%x(3) * v%x(1) - k%x(1) * v%x(3)
  kv02 = (0,1) * (k%t * v%x(2) - k%x(2) * v%t)
  kv32 = (0,1) * (k%x(3) * v%x(2) - k%x(2) * v%x(3))
  ap  = 2 * (kv30 + kv21)
  am  = 2 * (-kv30 + kv21)
  bp  = 2 * (kv01 + kv31 + kv02 + kv32)
  bm  = 2 * (kv01 - kv31 + kv02 - kv32)
  bps = 2 * (kv01 + kv31 - kv02 - kv32)
  bms = 2 * (kv01 - kv31 - kv02 + kv32)
  kvpsi%a(1) = am * psi%a(1) + bms * psi%a(2)
  kvpsi%a(2) = bp * psi%a(1) - am * psi%a(2)
  kvpsi%a(3) = ap * psi%a(3) - bps * psi%a(4)
  kvpsi%a(4) = -bm * psi%a(3) - ap * psi%a(4)
  end function f_vvf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function vmom_ff (g, psibar, psi, k) result (psibarpsi)
  type(vector) :: psibarpsi
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  psibarpsi = g * vv_ff (psibar, psi, vk)
  end function vmom_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function mom_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  type(bispinor) :: kmpsi
  complex(kind=default) :: kp, km, k12, k12s
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  kmpsi%a(1) = km * psi%a(3) - k12s * psi%a(4)
  kmpsi%a(2) = kp * psi%a(4) - k12 * psi%a(3)
  kmpsi%a(3) = kp * psi%a(1) + k12s * psi%a(2)
  kmpsi%a(4) = k12 * psi%a(1) + km * psi%a(2)
  psibarpsi = g * (psibar * kmpsi) + s_ff (m, psibar, psi)
  end function mom_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function mom5_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  type(bispinor) :: g5psi
  g5psi%a(1:2) = - psi%a(1:2)
  g5psi%a(3:4) = psi%a(3:4)
  psibarpsi = mom_ff (g, m, psibar, g5psi, k)
  end function mom5_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function moml_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  type(bispinor) :: leftpsi
  leftpsi%a(1:2) = 2 * psi%a(1:2)
  leftpsi%a(3:4) = 0
  psibarpsi = mom_ff (g, m, psibar, leftpsi, k)
  end function moml_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function momr_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  type(bispinor) :: rightpsi
  rightpsi%a(1:2) = 0
  rightpsi%a(3:4) = 2 * psi%a(3:4)
  psibarpsi = mom_ff (g, m, psibar, rightpsi, k)
  end function momr_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function lmom_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  psibarpsi = mom_ff  (g, m, psibar, psi, k) + &
  mom5_ff (g,-m, psibar, psi, k)
  end function lmom_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function rmom_ff (g, m, psibar, psi, k) result (psibarpsi)
  complex(kind=default) :: psibarpsi
  type(bispinor), intent(in) :: psibar, psi
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g, m
  psibarpsi = mom_ff  (g, m, psibar, psi, k) - &
  mom5_ff (g,-m, psibar, psi, k)
  end function rmom_ff
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: f_vmomf, f_momf, f_mom5f, f_momlf, f_momrf, f_lmomf, f_rmomf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_vmomf (g, v, psi, k) result (kvpsi)
  type(bispinor) :: kvpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: g
  type(momentum), intent(in) :: k
  type(vector), intent(in) :: v
  type(vector) :: vk
  vk = k
  kvpsi = g * f_vvf (v, psi, vk)
  end function f_vmomf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_momf (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  complex(kind=default) :: kp, km, k12, k12s
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  kmpsi%a(1) = km * psi%a(3) - k12s * psi%a(4)
  kmpsi%a(2) = -k12 * psi%a(3) + kp * psi%a(4)
  kmpsi%a(3) = kp * psi%a(1) + k12s * psi%a(2)
  kmpsi%a(4) = k12 * psi%a(1) + km * psi%a(2)
  kmpsi = g * (phi * kmpsi) + f_sf (m, phi, psi)
  end function f_momf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_mom5f (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  type(bispinor) :: g5psi
  g5psi%a(1:2) = - psi%a(1:2)
  g5psi%a(3:4) =   psi%a(3:4)
  kmpsi = f_momf (g, m, phi, g5psi, k)
  end function f_mom5f
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_momlf (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  type(bispinor) :: leftpsi
  leftpsi%a(1:2) = 2 * psi%a(1:2)
  leftpsi%a(3:4) = 0
  kmpsi = f_momf (g, m, phi, leftpsi, k)
  end function f_momlf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_momrf (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  type(bispinor) :: rightpsi
  rightpsi%a(1:2) = 0
  rightpsi%a(3:4) = 2 * psi%a(3:4)
  kmpsi = f_momf (g, m, phi, rightpsi, k)
  end function f_momrf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_lmomf (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  kmpsi = f_momf  (g, m, phi, psi, k) + &
  f_mom5f (g,-m, phi, psi, k)
  end function f_lmomf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function f_rmomf (g, m, phi, psi, k) result (kmpsi)
  type(bispinor) :: kmpsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: phi, g, m
  type(momentum), intent(in) :: k
  kmpsi = f_momf  (g, m, phi, psi, k) - &
  f_mom5f (g,-m, phi, psi, k)
  end function f_rmomf
```

4-Couplings

⟨*Declaration of bispinor currents*⟩+≡
```
  public :: v2_ff, sv1_ff, sv2_ff, pv1_ff, pv2_ff, svl1_ff, svl2_ff, &
  svr1_ff, svr2_ff, svlr1_ff, svlr2_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function v2_ff (g, psibar, v, psi) result (v2)
  type(vector) :: v2
  complex (kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  v2 = (-g) * vv_ff (psibar, psi, v)
  end function v2_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function sv1_ff (g, psibar, v, psi) result (phi)
  complex(kind=default) :: phi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g
  phi = psibar * f_vf (g, v, psi)
  end function sv1_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function sv2_ff (g, psibar, phi, psi) result (v)
  type(vector) :: v
  complex(kind=default), intent(in) :: phi, g
  type(bispinor), intent(in) :: psibar, psi
  v = phi * v_ff (g, psibar, psi)
  end function sv2_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function pv1_ff (g, psibar, v, psi) result (phi)
  complex(kind=default) :: phi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g
  phi = - (psibar * f_af (g, v, psi))
  end function pv1_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function pv2_ff (g, psibar, phi, psi) result (v)
  type(vector) :: v
  complex(kind=default), intent(in) :: phi, g
  type(bispinor), intent(in) :: psibar, psi
  v = -(phi * a_ff (g, psibar, psi))
  end function pv2_ff
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function svl1_ff (g, psibar, v, psi) result (phi)
  complex(kind=default) :: phi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g
  phi = psibar * f_vlf (g, v, psi)
  end function svl1_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function svl2_ff (g, psibar, phi, psi) result (v)
  type(vector) :: v
  complex(kind=default), intent(in) :: phi, g
  type(bispinor), intent(in) :: psibar, psi
  v = phi * vl_ff (g, psibar, psi)
  end function svl2_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function svr1_ff (g, psibar, v, psi) result (phi)
  complex(kind=default) :: phi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g
  phi = psibar * f_vrf (g, v, psi)
  end function svr1_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function svr2_ff (g, psibar, phi, psi) result (v)
  type(vector) :: v
  complex(kind=default), intent(in) :: phi, g
  type(bispinor), intent(in) :: psibar, psi
  v = phi * vr_ff (g, psibar, psi)
  end function svr2_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function svlr1_ff (gl, gr, psibar, v, psi) result (phi)
  complex(kind=default) :: phi
  type(bispinor), intent(in) :: psibar, psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gl, gr
  phi = psibar * f_vlrf (gl, gr, v, psi)
  end function svlr1_ff
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function svlr2_ff (gl, gr, psibar, phi, psi) result (v)
  type(vector) :: v
  complex(kind=default), intent(in) :: phi, gl, gr
  type(bispinor), intent(in) :: psibar, psi
  v = phi * vlr_ff (gl, gr, psibar, psi)
  end function svlr2_ff
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: f_v2f, f_svf, f_pvf, f_svlf, f_svrf, f_svlrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_v2f (g, v1, v2, psi) result (vpsi)
  type(bispinor) :: vpsi
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v1, v2
  vpsi = g * f_vvf (v2, psi, v1)
  end function f_v2f
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_svf (g, phi, v, psi) result (pvpsi)
  type(bispinor) :: pvpsi
  complex(kind=default), intent(in) :: g, phi
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  pvpsi = phi * f_vf (g, v, psi)
  end function f_svf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_pvf (g, phi, v, psi) result (pvpsi)
type(bispinor) :: pvpsi
complex(kind=default), intent(in) :: g, phi
type(bispinor), intent(in) :: psi
type(vector), intent(in) :: v
pvpsi = -(phi * f_af (g, v, psi))
end function f_pvf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_svlf (g, phi, v, psi) result (pvpsi)
type(bispinor) :: pvpsi
complex(kind=default), intent(in) :: g, phi
type(bispinor), intent(in) :: psi
type(vector), intent(in) :: v
pvpsi = phi * f_vlf (g, v, psi)
end function f_svlf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_svrf (g, phi, v, psi) result (pvpsi)
type(bispinor) :: pvpsi
complex(kind=default), intent(in) :: g, phi
type(bispinor), intent(in) :: psi
type(vector), intent(in) :: v
pvpsi = phi * f_vrf (g, v, psi)
end function f_svrf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_svlrf (gl, gr, phi, v, psi) result (pvpsi)
type(bispinor) :: pvpsi
complex(kind=default), intent(in) :: gl, gr, phi
type(bispinor), intent(in) :: psi
type(vector), intent(in) :: v
pvpsi = phi * f_vlrf (gl, gr, v, psi)
end function f_svlrf
```

AB.26.4 Gravitino Couplings

⟨*Declaration of bispinor currents*⟩+≡
```
public :: pot_grf, pot_fgr, s_grf, s_fgr, p_grf, p_fgr, &
sl_grf, sl_fgr, sr_grf, sr_fgr, slr_grf, slr_fgr
```

⟨*Declaration of bispinor currents*⟩+≡
```
private :: fgvgr, fgvg5gr, fggvvgr, grkgf, grkggf, grkkggf, &
fgkgr, fg5gkgr, grvgf, grg5vgf, grkgggf, fggkggr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function pot_grf (g, gravbar, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(vectorspinor) :: gamma_psi
gamma_psi%psi(1)%a(1) = psi%a(3)
gamma_psi%psi(1)%a(2) = psi%a(4)
gamma_psi%psi(1)%a(3) = psi%a(1)
gamma_psi%psi(1)%a(4) = psi%a(2)
gamma_psi%psi(2)%a(1) = psi%a(4)
gamma_psi%psi(2)%a(2) = psi%a(3)
gamma_psi%psi(2)%a(3) = - psi%a(2)
gamma_psi%psi(2)%a(4) = - psi%a(1)
gamma_psi%psi(3)%a(1) = (0,-1) * psi%a(4)
gamma_psi%psi(3)%a(2) = (0,1) * psi%a(3)
gamma_psi%psi(3)%a(3) = (0,1) * psi%a(2)
gamma_psi%psi(3)%a(4) = (0,-1) * psi%a(1)
gamma_psi%psi(4)%a(1) = psi%a(3)
gamma_psi%psi(4)%a(2) = - psi%a(4)
gamma_psi%psi(4)%a(3) = - psi%a(1)
```

```
  gamma_psi%psi(4)%a(4) = psi%a(2)
  j = g * (gravbar * gamma_psi)
  end function pot_grf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function pot_fgr (g, psibar, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(bispinor) :: gamma_grav
  gamma_grav%a(1) = grav%psi(1)%a(3) - grav%psi(2)%a(4) + &
  ((0,1)*grav%psi(3)%a(4)) - grav%psi(4)%a(3)
  gamma_grav%a(2) = grav%psi(1)%a(4) - grav%psi(2)%a(3) - &
  ((0,1)*grav%psi(3)%a(3)) + grav%psi(4)%a(4)
  gamma_grav%a(3) = grav%psi(1)%a(1) + grav%psi(2)%a(2) - &
  ((0,1)*grav%psi(3)%a(2)) + grav%psi(4)%a(1)
  gamma_grav%a(4) = grav%psi(1)%a(2) + grav%psi(2)%a(1) + &
  ((0,1)*grav%psi(3)%a(1)) - grav%psi(4)%a(2)
  j = g * (psibar * gamma_grav)
  end function pot_fgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function grvgf (gravbar, psi, k) result (j)
  complex(kind=default) :: j
  complex(kind=default) :: kp, km, k12, k12s
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: k
  type(vectorspinor) :: kg_psi
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  !!! Since we are taking the spinor product here, NO explicit
  !!! charge conjugation matrix is needed!
  kg_psi%psi(1)%a(1) = km * psi%a(1) - k12s * psi%a(2)
  kg_psi%psi(1)%a(2) = (-k12) * psi%a(1) + kp * psi%a(2)
  kg_psi%psi(1)%a(3) = kp * psi%a(3) + k12s * psi%a(4)
  kg_psi%psi(1)%a(4) = k12 * psi%a(3) + km * psi%a(4)
  kg_psi%psi(2)%a(1) = k12s * psi%a(1) - km * psi%a(2)
  kg_psi%psi(2)%a(2) = (-kp) * psi%a(1) + k12 * psi%a(2)
  kg_psi%psi(2)%a(3) = k12s * psi%a(3) + kp * psi%a(4)
  kg_psi%psi(2)%a(4) = km * psi%a(3) + k12 * psi%a(4)
  kg_psi%psi(3)%a(1) = (0,1) * (k12s * psi%a(1) + km * psi%a(2))
  kg_psi%psi(3)%a(2) = (0,1) * (- kp * psi%a(1) - k12 * psi%a(2))
  kg_psi%psi(3)%a(3) = (0,1) * (k12s * psi%a(3) - kp * psi%a(4))
  kg_psi%psi(3)%a(4) = (0,1) * (km * psi%a(3) - k12 * psi%a(4))
  kg_psi%psi(4)%a(1) = (-km) * psi%a(1) - k12s * psi%a(2)
  kg_psi%psi(4)%a(2) = k12 * psi%a(1) + kp * psi%a(2)
  kg_psi%psi(4)%a(3) = kp * psi%a(3) - k12s * psi%a(4)
  kg_psi%psi(4)%a(4) = k12 * psi%a(3) - km * psi%a(4)
  j = gravbar * kg_psi
  end function grvgf
```

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function grg5vgf (gravbar, psi, k) result (j)
  complex(kind=default) :: j
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: k
  type(bispinor) :: g5_psi
  g5_psi%a(1:2) = - psi%a(1:2)
  g5_psi%a(3:4) =   psi%a(3:4)
  j = grvgf (gravbar, g5_psi, k)
  end function grg5vgf
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function s_grf (g, gravbar, psi, k) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(momentum), intent(in) :: k
type(vector) :: vk
vk = k
j = g * grvgf (gravbar, psi, vk)
end function s_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function sl_grf (gl, gravbar, psi, k) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gl
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(bispinor) :: psi_l
type(momentum), intent(in) :: k
psi_l%a(1:2) = psi%a(1:2)
psi_l%a(3:4) = 0
j = s_grf (gl, gravbar, psi_l, k)
end function sl_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function sr_grf (gr, gravbar, psi, k) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gr
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(bispinor) :: psi_r
type(momentum), intent(in) :: k
psi_r%a(1:2) = 0
psi_r%a(3:4) = psi%a(3:4)
j = s_grf (gr, gravbar, psi_r, k)
end function sr_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function slr_grf (gl, gr, gravbar, psi, k) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gl, gr
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(momentum), intent(in) :: k
j = sl_grf (gl, gravbar, psi, k) + sr_grf (gr, gravbar, psi, k)
end function slr_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function fgkgr (psibar, grav, k) result (j)
complex(kind=default) :: j
complex(kind=default) :: kp, km, k12, k12s
type(bispinor), intent(in) :: psibar
type(vectorspinor), intent(in) :: grav
type(vector), intent(in) :: k
type(bispinor) :: gk_grav
kp = k%t + k%x(3)
km = k%t - k%x(3)
k12  =  k%x(1) + (0,1)*k%x(2)
k12s =  k%x(1) - (0,1)*k%x(2)
!!! Since we are taking the spinor product here, NO explicit
!!! charge conjugation matrix is needed!
gk_grav%a(1) =  kp * grav%psi(1)%a(1) + k12s * grav%psi(1)%a(2) &
- k12 * grav%psi(2)%a(1) - km * grav%psi(2)%a(2) &
+ (0,1) * k12 * grav%psi(3)%a(1)   &
+ (0,1) * km * grav%psi(3)%a(2) &
- kp * grav%psi(4)%a(1) - k12s * grav%psi(4)%a(2)
gk_grav%a(2) = k12 * grav%psi(1)%a(1) + km * grav%psi(1)%a(2) &
- kp * grav%psi(2)%a(1) - k12s * grav%psi(2)%a(2) &
```

```
  - (0,1) * kp * grav%psi(3)%a(1) &
  - (0,1) * k12s * grav%psi(3)%a(2)  &
  + k12 * grav%psi(4)%a(1) + km * grav%psi(4)%a(2)
  gk_grav%a(3) = km * grav%psi(1)%a(3) - k12s * grav%psi(1)%a(4) &
  - k12 * grav%psi(2)%a(3) + kp * grav%psi(2)%a(4) &
  + (0,1) * k12 * grav%psi(3)%a(3)   &
  - (0,1) * kp * grav%psi(3)%a(4) &
  + km * grav%psi(4)%a(3) - k12s * grav%psi(4)%a(4)
  gk_grav%a(4) = - k12 * grav%psi(1)%a(3) + kp * grav%psi(1)%a(4) &
  + km * grav%psi(2)%a(3) - k12s * grav%psi(2)%a(4) &
  + (0,1) * km * grav%psi(3)%a(3) &
  - (0,1) * k12s * grav%psi(3)%a(4)  &
  + k12 * grav%psi(4)%a(3) - kp * grav%psi(4)%a(4)
  j = psibar * gk_grav
  end function fgkgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function fg5gkgr (psibar, grav, k) result (j)
  complex(kind=default) :: j
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: k
  type(bispinor) :: psibar_g5
  psibar_g5%a(1:2) = - psibar%a(1:2)
  psibar_g5%a(3:4) =   psibar%a(3:4)
  j = fgkgr (psibar_g5, grav, k)
  end function fg5gkgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function s_fgr (g, psibar, grav, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  j = g * fgkgr (psibar, grav, vk)
  end function s_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function sl_fgr (gl, psibar, grav, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_l
  type(vectorspinor), intent(in) :: grav
  type(momentum), intent(in) :: k
  psibar_l%a(1:2) = psibar%a(1:2)
  psibar_l%a(3:4) = 0
  j = s_fgr (gl, psibar_l, grav, k)
  end function sl_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function sr_fgr (gr, psibar, grav, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gr
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_r
  type(vectorspinor), intent(in) :: grav
  type(momentum), intent(in) :: k
  psibar_r%a(1:2) = 0
  psibar_r%a(3:4) = psibar%a(3:4)
  j = s_fgr (gr, psibar_r, grav, k)
  end function sr_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function slr_fgr (gl, gr, psibar, grav, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(momentum), intent(in) :: k
  j = sl_fgr (gl, psibar, grav, k) + sr_fgr (gr, psibar, grav, k)
  end function slr_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function p_grf (g, gravbar, psi, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  j = g * grg5vgf (gravbar, psi, vk)
  end function p_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function p_fgr (g, psibar, grav, k) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  j = g * fg5gkgr (psibar, grav, vk)
  end function p_fgr
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: f_potgr, f_sgr, f_pgr, f_vgr, f_vlrgr, f_slgr, f_srgr, f_slrgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_potgr (g, phi, psi) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(vectorspinor), intent(in) :: psi
  phipsi%a(1) = (g * phi) * (psi%psi(1)%a(3) - psi%psi(2)%a(4) + &
  ((0,1)*psi%psi(3)%a(4)) - psi%psi(4)%a(3))
  phipsi%a(2) = (g * phi) * (psi%psi(1)%a(4) - psi%psi(2)%a(3) - &
  ((0,1)*psi%psi(3)%a(3)) + psi%psi(4)%a(4))
  phipsi%a(3) = (g * phi) * (psi%psi(1)%a(1) + psi%psi(2)%a(2) - &
  ((0,1)*psi%psi(3)%a(2)) + psi%psi(4)%a(1))
  phipsi%a(4) = (g * phi) * (psi%psi(1)%a(2) + psi%psi(2)%a(1) + &
  ((0,1)*psi%psi(3)%a(1)) - psi%psi(4)%a(2))
  end function f_potgr
```

The slashed notation:

$$\not{k} = \begin{pmatrix} 0 & 0 & k_- & -k^* \\ 0 & 0 & -k & k_+ \\ k_+ & k^* & 0 & 0 \\ k & k_- & 0 & 0 \end{pmatrix}, \qquad \not{k}\gamma_5 = \begin{pmatrix} 0 & 0 & k_- & -k^* \\ 0 & 0 & -k & k_+ \\ -k_+ & -k^* & 0 & 0 \\ -k & -k_- & 0 & 0 \end{pmatrix} \tag{AB.119}$$

with $k_\pm = k_0 \pm k_3$, $k = k_1 + \mathrm{i}k_2$, $k^* = k_1 - \mathrm{i}k_2$. But note that $\cdot^*$ is *not* complex conjugation for complex k_μ.

$$\gamma^0\not{k} = \begin{pmatrix} k_+ & k^* & 0 & 0 \\ k & k_- & 0 & 0 \\ 0 & 0 & k_- & -k^* \\ 0 & 0 & -k & k_+ \end{pmatrix}, \qquad \gamma^0\not{k}\gamma^5 = \begin{pmatrix} -k_+ & -k^* & 0 & 0 \\ -k & -k_- & 0 & 0 \\ 0 & 0 & k_- & -k^* \\ 0 & 0 & -k & k_+ \end{pmatrix} \tag{AB.120a}$$

$$\gamma^1\not{k} = \begin{pmatrix} k & k_- & 0 & 0 \\ k_+ & k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & -k_- & k^* \end{pmatrix}, \qquad \gamma^1\not{k}\gamma^5 = \begin{pmatrix} -k & -k_- & 0 & 0 \\ -k_+ & -k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & -k_- & k^* \end{pmatrix} \tag{AB.120b}$$

$$\gamma^2\not{k} = \begin{pmatrix} -\mathrm{i}k & -\mathrm{i}k_- & 0 & 0 \\ \mathrm{i}k_+ & \mathrm{i}k^* & 0 & 0 \\ 0 & 0 & -\mathrm{i}k & \mathrm{i}k_+ \\ 0 & 0 & -\mathrm{i}k_- & \mathrm{i}k^* \end{pmatrix}, \qquad \gamma^2\not{k}\gamma^5 = \begin{pmatrix} \mathrm{i}k & \mathrm{i}k_- & 0 & 0 \\ -\mathrm{i}k_+ & -\mathrm{i}k^* & 0 & 0 \\ 0 & 0 & -\mathrm{i}k & \mathrm{i}k_+ \\ 0 & 0 & -\mathrm{i}k_- & \mathrm{i}k^* \end{pmatrix} \tag{AB.120c}$$

$$\gamma^3\not{k} = \begin{pmatrix} k_+ & k^* & 0 & 0 \\ -k & -k_- & 0 & 0 \\ 0 & 0 & -k_- & k^* \\ 0 & 0 & -k & k_+ \end{pmatrix}, \qquad \gamma^3\not{k}\gamma^5 = \begin{pmatrix} -k_+ & -k^* & 0 & 0 \\ k & k_- & 0 & 0 \\ 0 & 0 & -k_- & k^* \\ 0 & 0 & -k & k_+ \end{pmatrix} \tag{AB.120d}$$

and

$$\not{k}\gamma^0 = \begin{pmatrix} k_- & -k^* & 0 & 0 \\ -k & k_+ & 0 & 0 \\ 0 & 0 & k_+ & k^* \\ 0 & 0 & k & k_- \end{pmatrix}, \qquad \not{k}\gamma^0\gamma^5 = \begin{pmatrix} -k_- & k^* & 0 & 0 \\ k & -k_+ & 0 & 0 \\ 0 & 0 & k_+ & k^* \\ 0 & 0 & k & k_- \end{pmatrix} \tag{AB.121a}$$

$$\not{k}\gamma^1 = \begin{pmatrix} k^* & -k_- & 0 & 0 \\ -k_+ & k & 0 & 0 \\ 0 & 0 & k^* & k_+ \\ 0 & 0 & k_- & k \end{pmatrix}, \qquad \not{k}\gamma^1\gamma^5 = \begin{pmatrix} -k^* & k_- & 0 & 0 \\ k_+ & -k & 0 & 0 \\ 0 & 0 & k^* & k_+ \\ 0 & 0 & k_- & k \end{pmatrix} \tag{AB.121b}$$

$$\not{k}\gamma^2 = \begin{pmatrix} \mathrm{i}k^* & \mathrm{i}k_- & 0 & 0 \\ -\mathrm{i}k_+ & -\mathrm{i}k & 0 & 0 \\ 0 & 0 & \mathrm{i}k^* & -\mathrm{i}k_+ \\ 0 & 0 & \mathrm{i}k_- & -\mathrm{i}k \end{pmatrix}, \qquad \not{k}\gamma^2\gamma^5 = \begin{pmatrix} -\mathrm{i}k^* & -\mathrm{i}k_- & 0 & 0 \\ \mathrm{i}k_+ & \mathrm{i}k & 0 & 0 \\ 0 & 0 & \mathrm{i}k^* & -\mathrm{i}k_+ \\ 0 & 0 & \mathrm{i}k_- & -\mathrm{i}k \end{pmatrix} \tag{AB.121c}$$

$$\not{k}\gamma^3 = \begin{pmatrix} -k_- & -k^* & 0 & 0 \\ k & k_+ & 0 & 0 \\ 0 & 0 & k_+ & -k^* \\ 0 & 0 & k & -k_- \end{pmatrix}, \qquad \not{k}\gamma^3\gamma^5 = \begin{pmatrix} k_- & k^* & 0 & 0 \\ -k & -k_+ & 0 & 0 \\ 0 & 0 & k_+ & -k^* \\ 0 & 0 & k & -k_- \end{pmatrix} \tag{AB.121d}$$

and

$$C\gamma^0\not{k} = \begin{pmatrix} k & k_- & 0 & 0 \\ -k_+ & -k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & k_- & -k^* \end{pmatrix}, \qquad C\gamma^0\not{k}\gamma^5 = \begin{pmatrix} -k & -k_- & 0 & 0 \\ k_+ & k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & k_- & -k^* \end{pmatrix} \tag{AB.122a}$$

$$C\gamma^1\not{k} = \begin{pmatrix} k_+ & k^* & 0 & 0 \\ -k & -k_- & 0 & 0 \\ 0 & 0 & k_- & -k^* \\ 0 & 0 & k & -k_+ \end{pmatrix}, \qquad C\gamma^1\not{k}\gamma^5 = \begin{pmatrix} -k_+ & -k^* & 0 & 0 \\ k & k_- & 0 & 0 \\ 0 & 0 & k_- & -k^* \\ 0 & 0 & k & -k_+ \end{pmatrix} \tag{AB.122b}$$

$$C\gamma^2\not{k} = \begin{pmatrix} \mathrm{i}k_+ & \mathrm{i}k^* & 0 & 0 \\ \mathrm{i}k & \mathrm{i}k_- & 0 & 0 \\ 0 & 0 & \mathrm{i}k_- & -\mathrm{i}k^* \\ 0 & 0 & -\mathrm{i}k & \mathrm{i}k_+ \end{pmatrix}, \qquad C\gamma^2\not{k}\gamma^5 = \begin{pmatrix} -\mathrm{i}k_+ & -\mathrm{i}k^* & 0 & 0 \\ -\mathrm{i}k & -\mathrm{i}k_- & 0 & 0 \\ 0 & 0 & \mathrm{i}k_- & -\mathrm{i}k^* \\ 0 & 0 & -\mathrm{i}k & \mathrm{i}k_+ \end{pmatrix} \tag{AB.122c}$$

$$C\gamma^3\not{k} = \begin{pmatrix} -k & -k_- & 0 & 0 \\ -k_+ & -k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & -k_- & k^* \end{pmatrix}, \qquad C\gamma^3\not{k}\gamma^5 = \begin{pmatrix} k & k_- & 0 & 0 \\ k_+ & k^* & 0 & 0 \\ 0 & 0 & k & -k_+ \\ 0 & 0 & -k_- & k^* \end{pmatrix} \tag{AB.122d}$$

and

$$C\not{k}\gamma^0 = \begin{pmatrix} -k & k^+ & 0 & 0 \\ -k_- & k^* & 0 & 0 \\ 0 & 0 & -k & -k_- \\ 0 & 0 & k_+ & k^* \end{pmatrix}, \qquad C\not{k}\gamma^0\gamma^5 = \begin{pmatrix} k & -k_+ & 0 & 0 \\ k_- & -k^* & 0 & 0 \\ 0 & 0 & -k & -k_- \\ 0 & 0 & k_+ & k^* \end{pmatrix} \tag{AB.123a}$$

$$C\not{k}\gamma^1 = \begin{pmatrix} -k_+ & k & 0 & 0 \\ -k^* & k_- & 0 & 0 \\ 0 & 0 & -k_- & -k \\ 0 & 0 & k^* & k_+ \end{pmatrix}, \qquad C\not{k}\gamma^1\gamma^5 = \begin{pmatrix} k_+ & -k & 0 & 0 \\ k^* & -k_- & 0 & 0 \\ 0 & 0 & -k_- & -k \\ 0 & 0 & k^* & k_+ \end{pmatrix} \tag{AB.123b}$$

$$C\not{k}\gamma^2 = \begin{pmatrix} -\mathrm{i}k_+ & -\mathrm{i}k & 0 & 0 \\ -\mathrm{i}k^* & -\mathrm{i}k_- & 0 & 0 \\ 0 & 0 & -\mathrm{i}k_- & \mathrm{i}k \\ 0 & 0 & \mathrm{i}k^* & -\mathrm{i}k_+ \end{pmatrix}, \qquad C\not{k}\gamma^2\gamma^5 = \begin{pmatrix} \mathrm{i}k_+ & \mathrm{i}k & 0 & 0 \\ \mathrm{i}k^* & \mathrm{i}k_- & 0 & 0 \\ 0 & 0 & -\mathrm{i}k_- & \mathrm{i}k \\ 0 & 0 & \mathrm{i}k^* & -\mathrm{i}k_+ \end{pmatrix} \tag{AB.123c}$$

$$C\not{k}\gamma^3 = \begin{pmatrix} k & k_+ & 0 & 0 \\ k_- & k^* & 0 & 0 \\ 0 & 0 & -k & k_- \\ 0 & 0 & k_+ & -k^* \end{pmatrix}, \qquad C\not{k}\gamma^3\gamma^5 = \begin{pmatrix} -k & -k_+ & 0 & 0 \\ -k_- & -k^* & 0 & 0 \\ 0 & 0 & -k & k_- \\ 0 & 0 & k_+ & -k^* \end{pmatrix} \tag{AB.123d}$$

⟨*Implementation of bispinor currents*⟩+≡
```
pure function fgvgr (psi, k) result (kpsi)
type(bispinor) :: kpsi
complex(kind=default) :: kp, km, k12, k12s
type(vector), intent(in) :: k
type(vectorspinor), intent(in) :: psi
kp = k%t + k%x(3)
km = k%t - k%x(3)
k12  =  k%x(1) + (0,1)*k%x(2)
k12s =  k%x(1) - (0,1)*k%x(2)
kpsi%a(1) = kp * psi%psi(1)%a(1) + k12s * psi%psi(1)%a(2) &
- k12 * psi%psi(2)%a(1) - km * psi%psi(2)%a(2) &
+ (0,1) * k12 * psi%psi(3)%a(1) + (0,1) * km * psi%psi(3)%a(2) &
- kp * psi%psi(4)%a(1) - k12s * psi%psi(4)%a(2)
kpsi%a(2) = k12 * psi%psi(1)%a(1) + km * psi%psi(1)%a(2) &
- kp * psi%psi(2)%a(1) - k12s * psi%psi(2)%a(2) &
- (0,1) * kp * psi%psi(3)%a(1) - (0,1) * k12s * psi%psi(3)%a(2) &
+ k12 * psi%psi(4)%a(1) + km * psi%psi(4)%a(2)
kpsi%a(3) = km * psi%psi(1)%a(3) - k12s * psi%psi(1)%a(4) &
- k12 * psi%psi(2)%a(3) + kp * psi%psi(2)%a(4) &
+ (0,1) * k12 * psi%psi(3)%a(3) - (0,1) * kp * psi%psi(3)%a(4) &
+ km * psi%psi(4)%a(3) - k12s * psi%psi(4)%a(4)
kpsi%a(4) = - k12 * psi%psi(1)%a(3) + kp * psi%psi(1)%a(4) &
+ km * psi%psi(2)%a(3) - k12s * psi%psi(2)%a(4) &
+ (0,1) * km * psi%psi(3)%a(3) - (0,1) * k12s * psi%psi(3)%a(4) &
+ k12 * psi%psi(4)%a(3) - kp * psi%psi(4)%a(4)
end function fgvgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_sgr (g, phi, psi, k) result (phipsi)
type(bispinor) :: phipsi
complex(kind=default), intent(in) :: g
complex(kind=default), intent(in) :: phi
type(momentum), intent(in) :: k
type(vectorspinor), intent(in) :: psi
type(vector) :: vk
vk = k
phipsi = (g * phi) * fgvgr (psi, vk)
end function f_sgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_slgr (gl, phi, psi, k) result (phipsi)
type(bispinor) :: phipsi
complex(kind=default), intent(in) :: gl
complex(kind=default), intent(in) :: phi
type(momentum), intent(in) :: k
type(vectorspinor), intent(in) :: psi
phipsi = f_sgr (gl, phi, psi, k)
phipsi%a(3:4) = 0
end function f_slgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function f_srgr (gr, phi, psi, k) result (phipsi)
type(bispinor) :: phipsi
complex(kind=default), intent(in) :: gr
complex(kind=default), intent(in) :: phi
type(momentum), intent(in) :: k
type(vectorspinor), intent(in) :: psi
phipsi = f_sgr (gr, phi, psi, k)
phipsi%a(1:2) = 0
end function f_srgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_slrgr (gl, gr, phi, psi, k) result (phipsi)
  type(bispinor) :: phipsi, phipsi_l, phipsi_r
  complex(kind=default), intent(in) :: gl, gr
  complex(kind=default), intent(in) :: phi
  type(momentum), intent(in) :: k
  type(vectorspinor), intent(in) :: psi
  phipsi_l = f_slgr (gl, phi, psi, k)
  phipsi_r = f_srgr (gr, phi, psi, k)
  phipsi%a(1:2) = phipsi_l%a(1:2)
  phipsi%a(3:4) = phipsi_r%a(3:4)
  end function f_slrgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function fgvg5gr (psi, k) result (kpsi)
  type(bispinor) :: kpsi
  type(vector), intent(in) :: k
  type(vectorspinor), intent(in) :: psi
  type(bispinor) :: kpsi_dum
  kpsi_dum = fgvgr (psi, k)
  kpsi%a(1:2) = - kpsi_dum%a(1:2)
  kpsi%a(3:4) =   kpsi_dum%a(3:4)
  end function fgvg5gr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_pgr (g, phi, psi, k) result (phipsi)
  type(bispinor) :: phipsi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(momentum), intent(in) :: k
  type(vectorspinor), intent(in) :: psi
  type(vector) :: vk
  vk = k
  phipsi = (g * phi) * fgvg5gr (psi, vk)
  end function f_pgr
```

The needed construction of gamma matrices involving the commutator of two gamma matrices. For the slashed terms we use as usual the abbreviations $k_\pm = k_0 \pm k_3$, $k = k_1 + \mathrm{i}k_2$, $k^* = k_1 - \mathrm{i}k_2$ and analogous expressions for the vector v^μ. We remind you that $\cdot^*$ is *not* complex conjugation for complex k_μ. Furthermore we introduce (in what follows the brackets around the vector indices have the usual meaning of antisymmetrizing with respect to the indices inside the brackets, here without a factor two in the denominator)

$$\begin{aligned}
a_+ &= k_+v_- + kv^* - k_-v_+ - k^*v = 2(k_{[3}v_{0]} + \mathrm{i}k_{[2}v_{1]}) && \text{(AB.124a)}\\
a_- &= k_-v_+ + kv^* - k_+v_- - k^*v = 2(-k_{[3}v_{0]} + \mathrm{i}k_{[2}v_{1]}) && \text{(AB.124b)}\\
b_+ &= 2(k_+v - kv_+) \qquad = 2(k_{[0}v_{1]} + k_{[3}v_{1]} + \mathrm{i}k_{[0}v_{2]} + \mathrm{i}k_{[3}v_{2]}) && \text{(AB.124c)}\\
b_- &= 2(k_-v - kv_-) \qquad = 2(k_{[0}v_{1]} - k_{[3}v_{1]} + \mathrm{i}k_{[0}v_{2]} - \mathrm{i}k_{[3}v_{2]}) && \text{(AB.124d)}\\
b_{+*} &= 2(k_+v^* - k^*v_+) \qquad = 2(k_{[0}v_{1]} + k_{[3}v_{1]} - \mathrm{i}k_{[0}v_{2]} - \mathrm{i}k_{[3}v_{2]}) && \text{(AB.124e)}\\
b_{-*} &= 2(k_-v^* - k^*v_-) \qquad = 2(k_{[0}v_{1]} - k_{[3}v_{1]} - \mathrm{i}k_{[0}v_{2]} + \mathrm{i}k_{[3}v_{2]}) && \text{(AB.124f)}
\end{aligned}$$

Of course, one could introduce a more advanced notation, but we don't want to become confused.

$$[\not{k},\gamma^0] = \begin{pmatrix} -2k_3 & -2k^* & 0 & 0 \\ -2k & 2k_3 & 0 & 0 \\ 0 & 0 & 2k_3 & 2k^* \\ 0 & 0 & 2k & -2k_3 \end{pmatrix} \tag{AB.125a}$$

$$[\not{k},\gamma^1] = \begin{pmatrix} -2\mathrm{i}k_2 & -2k_- & 0 & 0 \\ -2k_+ & 2\mathrm{i}k_2 & 0 & 0 \\ 0 & 0 & -2\mathrm{i}k_2 & 2k_+ \\ 0 & 0 & 2k_- & 2\mathrm{i}k_2 \end{pmatrix} \tag{AB.125b}$$

$$[\not{k},\gamma^2] = \begin{pmatrix} 2\mathrm{i}k_1 & 2\mathrm{i}k_- & 0 & 0 \\ -2\mathrm{i}k_+ & -2\mathrm{i}k_1 & 0 & 0 \\ 0 & 0 & 2\mathrm{i}k_1 & -2\mathrm{i}k_+ \\ 0 & 0 & 2\mathrm{i}k_- & -2\mathrm{i}k_1 \end{pmatrix} \tag{AB.125c}$$

$$[\not{k},\gamma^3]=\begin{pmatrix}-2k_0 & -2k^* & 0 & 0\\ 2k & 2k_0 & 0 & 0\\ 0 & 0 & 2k_0 & -2k^*\\ 0 & 0 & 2k & -2k_0\end{pmatrix} \tag{AB.125d}$$

$$[\not{k},\not{V}]=\begin{pmatrix}a_- & b_{-*} & 0 & 0\\ b_+ & -a_- & 0 & 0\\ 0 & 0 & a_+ & -b_{+*}\\ 0 & 0 & -b_- & -a_+\end{pmatrix} \tag{AB.125e}$$

$$\gamma^5\gamma^0[\not{k},\not{V}]=\begin{pmatrix}0 & 0 & -a_+ & b_{+*}\\ 0 & 0 & b_- & a_+\\ a_- & b_{-*} & 0 & 0\\ b_+ & -a_- & 0 & 0\end{pmatrix} \tag{AB.125f}$$

$$\gamma^5\gamma^1[\not{k},\not{V}]=\begin{pmatrix}0 & 0 & b_- & a_+\\ 0 & 0 & -a_+ & b_{+*}\\ -b_+ & a_- & 0 & 0\\ -a_- & -b_{-*} & 0 & 0\end{pmatrix} \tag{AB.125g}$$

$$\gamma^5\gamma^2[\not{k},\not{V}]=\begin{pmatrix}0 & 0 & -\mathrm{i}b_- & -\mathrm{i}a_+\\ 0 & 0 & -\mathrm{i}a_+ & \mathrm{i}b_{+*}\\ \mathrm{i}b_+ & -\mathrm{i}a_- & 0 & 0\\ -\mathrm{i}a_- & -\mathrm{i}b_{-*} & 0 & 0\end{pmatrix} \tag{AB.125h}$$

$$\gamma^5\gamma^3[\not{k},\not{V}]=\begin{pmatrix}0 & 0 & -a_+ & b_{+*}\\ 0 & 0 & -b_- & -a_+\\ -a_- & -b_{-*} & 0 & 0\\ b_+ & -a_- & 0 & 0\end{pmatrix} \tag{AB.125i}$$

and

$$[\not{k},\not{V}]\gamma^0\gamma^5=\begin{pmatrix}0 & 0 & a_- & b_{-*}\\ 0 & 0 & b_+ & -a_-\\ -a_+ & b_{+*} & 0 & 0\\ b_- & a_+ & 0 & 0\end{pmatrix} \tag{AB.126a}$$

$$[\not{k},\not{V}]\gamma^1\gamma^5=\begin{pmatrix}0 & 0 & b_{-*} & a_-\\ 0 & 0 & -a_- & b_+\\ -b_{+*} & a_+ & 0 & 0\\ -a_+ & -b_- & 0 & 0\end{pmatrix} \tag{AB.126b}$$

$$[\not{k},\not{V}]\gamma^2\gamma^5=\begin{pmatrix}0 & 0 & \mathrm{i}b_{-*} & -\mathrm{i}a_-\\ 0 & 0 & -\mathrm{i}a_- & -\mathrm{i}b_+\\ -\mathrm{i}b_{+*} & -\mathrm{i}a_+ & 0 & 0\\ -\mathrm{i}a_+ & \mathrm{i}b_- & 0 & 0\end{pmatrix} \tag{AB.126c}$$

$$[\not{k},\not{V}]\gamma^3\gamma^5=\begin{pmatrix}0 & 0 & a_- & -b_{-*}\\ 0 & 0 & b_+ & a_-\\ a_+ & b_{+*} & 0 & 0\\ -b_- & a_+ & 0 & 0\end{pmatrix} \tag{AB.126d}$$

In what follows l always means twice the value of k, e.g. $l_+ = 2k_+$. We use the abbreviation $C^{\mu\nu} \equiv C[\not{k},\gamma^\mu]\gamma^\nu\gamma^5$.

$$C^{00}=\begin{pmatrix}0 & 0 & -l & -l_3\\ 0 & 0 & l_3 & l^*\\ l & -l_3 & 0 & 0\\ -l_3 & -l^* & 0 & 0\end{pmatrix},\qquad C^{20}=\begin{pmatrix}0 & 0 & -\mathrm{i}l_+ & -\mathrm{i}l_1\\ 0 & 0 & -\mathrm{i}l_1 & -\mathrm{i}l_-\\ \mathrm{i}l_- & -\mathrm{i}l_1 & 0 & 0\\ -\mathrm{i}l_1 & \mathrm{i}l_+ & 0 & 0\end{pmatrix} \tag{AB.127a}$$

$$C^{01}=\begin{pmatrix}0 & 0 & l_3 & -l\\ 0 & 0 & l^* & l_3\\ l_3 & -l & 0 & 0\\ l^* & l_3 & 0 & 0\end{pmatrix},\qquad C^{21}=\begin{pmatrix}0 & 0 & -\mathrm{i}l_1 & -\mathrm{i}l_+\\ 0 & 0 & -\mathrm{i}l_- & -\mathrm{i}l_1\\ \mathrm{i}l_1 & -\mathrm{i}l_- & 0 & 0\\ -\mathrm{i}l_+ & \mathrm{i}l_1 & 0 & 0\end{pmatrix} \tag{AB.127b}$$

$$C^{02}=\begin{pmatrix}0 & 0 & \mathrm{i}l_3 & \mathrm{i}l\\ 0 & 0 & \mathrm{i}l^* & -\mathrm{i}l_3\\ \mathrm{i}l_3 & \mathrm{i}l & 0 & 0\\ \mathrm{i}l^* & -\mathrm{i}l_3 & 0 & 0\end{pmatrix},\qquad C^{22}=\begin{pmatrix}0 & 0 & l_1 & -l_+\\ 0 & 0 & l_- & -l_1\\ -l_1 & -l_- & 0 & 0\\ l_+ & l_1 & 0 & 0\end{pmatrix} \tag{AB.127c}$$

$$C^{03} = \begin{pmatrix} 0 & 0 & -l & -l_3 \\ 0 & 0 & l_3 & -l^* \\ -l & -l_3 & 0 & 0 \\ l_3 & -l^* & 0 & 0 \end{pmatrix}, \qquad C^{23} = \begin{pmatrix} 0 & 0 & -\mathrm{i}l_+ & \mathrm{i}l_1 \\ 0 & 0 & -\mathrm{i}l_1 & \mathrm{i}l_- \\ -\mathrm{i}l_- & -\mathrm{i}l_1 & 0 & 0 \\ \mathrm{i}l_1 & \mathrm{i}l_+ & 0 & 0 \end{pmatrix} \tag{AB.127d}$$

$$C^{10} = \begin{pmatrix} 0 & 0 & -l_+ & \mathrm{i}l_2 \\ 0 & 0 & \mathrm{i}l_2 & l_- \\ l_- & \mathrm{i}l_2 & 0 & 0 \\ \mathrm{i}l_2 & -l_+ & 0 & 0 \end{pmatrix}, \qquad C^{30} = \begin{pmatrix} 0 & 0 & l & l_0 \\ 0 & 0 & l_0 & l^* \\ l & -l_0 & 0 & 0 \\ -l_0 & l^* & 0 & 0 \end{pmatrix} \tag{AB.127e}$$

$$C^{11} = \begin{pmatrix} 0 & 0 & \mathrm{i}l_2 & -l_+ \\ 0 & 0 & l_- & \mathrm{i}l_2 \\ -\mathrm{i}l_2 & -l_- & 0 & 0 \\ l_+ & -\mathrm{i}l_2 & 0 & 0 \end{pmatrix}, \qquad C^{31} = \begin{pmatrix} 0 & 0 & l_0 & l \\ 0 & 0 & l^* & l_0 \\ l_0 & -l & 0 & 0 \\ -l^* & l_0 & 0 & 0 \end{pmatrix} \tag{AB.127f}$$

$$C^{12} = \begin{pmatrix} 0 & 0 & -l_2 & \mathrm{i}l_+ \\ 0 & 0 & \mathrm{i}l_- & l_2 \\ l_2 & \mathrm{i}l_- & 0 & 0 \\ \mathrm{i}l_+ & -l_2 & 0 & 0 \end{pmatrix}, \qquad C^{32} = \begin{pmatrix} 0 & 0 & \mathrm{i}l_0 & -\mathrm{i}l \\ 0 & 0 & \mathrm{i}l^* & -\mathrm{i}l_0 \\ \mathrm{i}l_0 & \mathrm{i}l & 0 & 0 \\ -\mathrm{i}l^* & -\mathrm{i}l_0 & 0 & 0 \end{pmatrix} \tag{AB.127g}$$

$$C^{13} = \begin{pmatrix} 0 & 0 & -l_+ & -\mathrm{i}l_2 \\ 0 & 0 & \mathrm{i}l_2 & -l_- \\ -l_- & \mathrm{i}l_2 & 0 & 0 \\ -\mathrm{i}l_2 & -l_+ & 0 & 0 \end{pmatrix}, \qquad C^{33} = \begin{pmatrix} 0 & 0 & l & -l_0 \\ 0 & 0 & l_0 & -l^* \\ -l & -l_0 & 0 & 0 \\ l_0 & l^* & 0 & 0 \end{pmatrix} \tag{AB.127h}$$

and, with the abbreviation $\tilde{C}^{\mu\nu} \equiv C\gamma^5\gamma^\nu[\not k, \gamma^\mu]$ (note the reversed order of the indices!)

$$\tilde{C}^{00} = \begin{pmatrix} 0 & 0 & -l & l_3 \\ 0 & 0 & l_3 & l^* \\ l & -l_3 & 0 & 0 \\ -l_3 & -l^* & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{20} = \begin{pmatrix} 0 & 0 & -\mathrm{i}l_- & \mathrm{i}l_1 \\ 0 & 0 & \mathrm{i}l_1 & -\mathrm{i}l_+ \\ \mathrm{i}l_+ & \mathrm{i}l_1 & 0 & 0 \\ \mathrm{i}l_1 & \mathrm{i}l_- & 0 & 0 \end{pmatrix} \tag{AB.128a}$$

$$\tilde{C}^{01} = \begin{pmatrix} 0 & 0 & -l_3 & -l^* \\ 0 & 0 & l & -l_3 \\ -l_3 & -l^* & 0 & 0 \\ l & -l_3 & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{21} = \begin{pmatrix} 0 & 0 & -\mathrm{i}l_1 & \mathrm{i}l_+ \\ 0 & 0 & \mathrm{i}l_- & -\mathrm{i}l_1 \\ \mathrm{i}l_1 & \mathrm{i}l_- & 0 & 0 \\ \mathrm{i}l_+ & \mathrm{i}l_1 & 0 & 0 \end{pmatrix} \tag{AB.128b}$$

$$\tilde{C}^{02} = \begin{pmatrix} 0 & 0 & -\mathrm{i}l_3 & -\mathrm{i}l^* \\ 0 & 0 & -\mathrm{i}l & \mathrm{i}l_3 \\ -\mathrm{i}l_3 & -\mathrm{i}l^* & 0 & 0 \\ -\mathrm{i}l & \mathrm{i}l_3 & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{22} = \begin{pmatrix} 0 & 0 & l_1 & -l_+ \\ 0 & 0 & l_- & -l_1 \\ -l_1 & -l_- & 0 & 0 \\ l_+ & l_1 & 0 & 0 \end{pmatrix} \tag{AB.128c}$$

$$\tilde{C}^{03} = \begin{pmatrix} 0 & 0 & l & -l_3 \\ 0 & 0 & l_3 & l^* \\ l & -l_3 & 0 & 0 \\ l_3 & l^* & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{23} = \begin{pmatrix} 0 & 0 & \mathrm{i}l_- & -\mathrm{i}l_1 \\ 0 & 0 & \mathrm{i}l_1 & -\mathrm{i}l_+ \\ \mathrm{i}l_+ & \mathrm{i}l_1 & 0 & 0 \\ -\mathrm{i}l_1 & -\mathrm{i}l_- & 0 & 0 \end{pmatrix} \tag{AB.128d}$$

$$\tilde{C}^{10} = \begin{pmatrix} 0 & 0 & -l_- & -\mathrm{i}l_2 \\ 0 & 0 & -\mathrm{i}l_2 & l_+ \\ l_+ & -\mathrm{i}l_2 & 0 & 0 \\ -\mathrm{i}l_2 & -l_- & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{30} = \begin{pmatrix} 0 & 0 & -l & l_0 \\ 0 & 0 & l_0 & -l^* \\ -l & -l_0 & 0 & 0 \\ -l_0 & -l^* & 0 & 0 \end{pmatrix} \tag{AB.128e}$$

$$\tilde{C}^{11} = \begin{pmatrix} 0 & 0 & \mathrm{i}l_2 & -l_+ \\ 0 & 0 & l_- & \mathrm{i}l_2 \\ -\mathrm{i}l_2 & -l_- & 0 & 0 \\ l_+ & -\mathrm{i}l_2 & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{31} = \begin{pmatrix} 0 & 0 & -l_0 & l^* \\ 0 & 0 & l & -l_0 \\ -l_0 & -l^* & 0 & 0 \\ -l & -l_0 & 0 & 0 \end{pmatrix} \tag{AB.128f}$$

$$\tilde{C}^{12} = \begin{pmatrix} 0 & 0 & -l_2 & -\mathrm{i}l_+ \\ 0 & 0 & -\mathrm{i}l_- & l_2 \\ l_2 & -\mathrm{i}l_- & 0 & 0 \\ -\mathrm{i}l_+ & -l_2 & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{32} = \begin{pmatrix} 0 & 0 & -\mathrm{i}l_0 & \mathrm{i}l^* \\ 0 & 0 & -\mathrm{i}l & \mathrm{i}l_0 \\ -\mathrm{i}l_0 & -\mathrm{i}l^* & 0 & 0 \\ \mathrm{i}l & \mathrm{i}l_0 & 0 & 0 \end{pmatrix} \tag{AB.128g}$$

$$\tilde{C}^{13} = \begin{pmatrix} 0 & 0 & l_- & \mathrm{i}l_2 \\ 0 & 0 & -\mathrm{i}l_2 & l_+ \\ l_+ & -\mathrm{i}l_2 & 0 & 0 \\ \mathrm{i}l_2 & l_- & 0 & 0 \end{pmatrix}, \qquad \tilde{C}^{33} = \begin{pmatrix} 0 & 0 & l & -l_0 \\ 0 & 0 & l_0 & -l^* \\ -l & -l_0 & 0 & 0 \\ l_0 & l^* & 0 & 0 \end{pmatrix} \tag{AB.128h}$$

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function fggvvgr (v, psi, k) result (psikv)
  type(bispinor) :: psikv
  type(vectorspinor), intent(in) :: psi
  type(vector), intent(in) :: v, k
  complex(kind=default) :: kv30, kv21, kv01, kv31, kv02, kv32
  complex(kind=default) :: ap, am, bp, bm, bps, bms
  kv30 = k%x(3) * v%t - k%t * v%x(3)
  kv21 = (0,1) * (k%x(2) * v%x(1) - k%x(1) * v%x(2))
  kv01 = k%t * v%x(1) - k%x(1) * v%t
  kv31 = k%x(3) * v%x(1) - k%x(1) * v%x(3)
  kv02 = (0,1) * (k%t * v%x(2) - k%x(2) * v%t)
  kv32 = (0,1) * (k%x(3) * v%x(2) - k%x(2) * v%x(3))
  ap  = 2 * (kv30 + kv21)
  am  = 2 * (-kv30 + kv21)
  bp  = 2 * (kv01 + kv31 + kv02 + kv32)
  bm  = 2 * (kv01 - kv31 + kv02 - kv32)
  bps = 2 * (kv01 + kv31 - kv02 - kv32)
  bms = 2 * (kv01 - kv31 - kv02 + kv32)
  psikv%a(1) = (-ap) * psi%psi(1)%a(3) + bps * psi%psi(1)%a(4) &
  + (-bm) * psi%psi(2)%a(3) + (-ap) * psi%psi(2)%a(4) &
  + (0,1) * (bm * psi%psi(3)%a(3) + ap * psi%psi(3)%a(4)) &
  + ap * psi%psi(4)%a(3) + (-bps) * psi%psi(4)%a(4)
  psikv%a(2) =  bm * psi%psi(1)%a(3) + ap * psi%psi(1)%a(4) &
  + ap * psi%psi(2)%a(3) + (-bps) * psi%psi(2)%a(4) &
  + (0,1) * (ap * psi%psi(3)%a(3) - bps * psi%psi(3)%a(4)) &
  + bm * psi%psi(4)%a(3) + ap * psi%psi(4)%a(4)
  psikv%a(3) =  am * psi%psi(1)%a(1) + bms * psi%psi(1)%a(2) &
  + bp * psi%psi(2)%a(1) + (-am) * psi%psi(2)%a(2) &
  + (0,-1) * (bp * psi%psi(3)%a(1) + (-am) * psi%psi(3)%a(2)) &
  + am * psi%psi(4)%a(1) + bms * psi%psi(4)%a(2)
  psikv%a(4) =  bp * psi%psi(1)%a(1) + (-am) * psi%psi(1)%a(2) &
  + am * psi%psi(2)%a(1) + bms * psi%psi(2)%a(2) &
  + (0,1) * (am * psi%psi(3)%a(1) + bms * psi%psi(3)%a(2)) &
  + (-bp) * psi%psi(4)%a(1) + am * psi%psi(4)%a(2)
  end function fggvvgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_vgr (g, v, psi, k) result (psikkkv)
  type(bispinor) :: psikkkv
  type(vectorspinor), intent(in) :: psi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g
  type(vector) :: vk
  vk = k
  psikkkv = g * (fggvvgr (v, psi, vk))
  end function f_vgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_vlrgr (gl, gr, v, psi, k) result (psikv)
  type(bispinor) :: psikv
  type(vectorspinor), intent(in) :: psi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: gl, gr
  type(vector) :: vk
  vk = k
  psikv = fggvvgr (v, psi, vk)
  psikv%a(1:2) = gl * psikv%a(1:2)
  psikv%a(3:4) = gr * psikv%a(3:4)
  end function f_vlrgr
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: gr_potf, gr_sf, gr_pf, gr_vf, gr_vlrf, gr_slf, gr_srf, gr_slrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_potf (g, phi, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  phipsi%psi(1)%a(1) = (g * phi) * psi%a(3)
  phipsi%psi(1)%a(2) = (g * phi) * psi%a(4)
  phipsi%psi(1)%a(3) = (g * phi) * psi%a(1)
  phipsi%psi(1)%a(4) = (g * phi) * psi%a(2)
  phipsi%psi(2)%a(1) = (g * phi) * psi%a(4)
  phipsi%psi(2)%a(2) = (g * phi) * psi%a(3)
  phipsi%psi(2)%a(3) = ((-g) * phi) * psi%a(2)
  phipsi%psi(2)%a(4) = ((-g) * phi) * psi%a(1)
  phipsi%psi(3)%a(1) = ((0,-1) * g * phi) * psi%a(4)
  phipsi%psi(3)%a(2) = ((0,1) * g * phi) * psi%a(3)
  phipsi%psi(3)%a(3) = ((0,1) * g * phi) * psi%a(2)
  phipsi%psi(3)%a(4) = ((0,-1) * g * phi) * psi%a(1)
  phipsi%psi(4)%a(1) = (g * phi) * psi%a(3)
  phipsi%psi(4)%a(2) = ((-g) * phi) * psi%a(4)
  phipsi%psi(4)%a(3) = ((-g) * phi) * psi%a(1)
  phipsi%psi(4)%a(4) = (g * phi) * psi%a(2)
  end function gr_potf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function grkgf (psi, k) result (kpsi)
  type(vectorspinor) :: kpsi
  complex(kind=default) :: kp, km, k12, k12s
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: k
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  kpsi%psi(1)%a(1) = km * psi%a(1) - k12s * psi%a(2)
  kpsi%psi(1)%a(2) = (-k12) * psi%a(1) + kp * psi%a(2)
  kpsi%psi(1)%a(3) = kp * psi%a(3) + k12s * psi%a(4)
  kpsi%psi(1)%a(4) = k12 * psi%a(3) + km * psi%a(4)
  kpsi%psi(2)%a(1) = k12s * psi%a(1) - km * psi%a(2)
  kpsi%psi(2)%a(2) = (-kp) * psi%a(1) + k12 * psi%a(2)
  kpsi%psi(2)%a(3) = k12s * psi%a(3) + kp * psi%a(4)
  kpsi%psi(2)%a(4) = km * psi%a(3) + k12 * psi%a(4)
  kpsi%psi(3)%a(1) = (0,1) * (k12s * psi%a(1) + km * psi%a(2))
  kpsi%psi(3)%a(2) = (0,-1) * (kp * psi%a(1) + k12 * psi%a(2))
  kpsi%psi(3)%a(3) = (0,1) * (k12s * psi%a(3) - kp * psi%a(4))
  kpsi%psi(3)%a(4) = (0,1) * (km * psi%a(3) - k12 * psi%a(4))
  kpsi%psi(4)%a(1) = -(km * psi%a(1) + k12s * psi%a(2))
  kpsi%psi(4)%a(2) = k12 * psi%a(1) + kp * psi%a(2)
  kpsi%psi(4)%a(3) = kp * psi%a(3) - k12s * psi%a(4)
  kpsi%psi(4)%a(4) = k12 * psi%a(3) - km * psi%a(4)
  end function grkgf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_sf (g, phi, psi, k) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  phipsi = (g * phi) * grkgf (psi, vk)
  end function gr_sf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_slf (gl, phi, psi, k) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: gl
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l
  type(momentum), intent(in) :: k
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  phipsi = gr_sf (gl, phi, psi_l, k)
  end function gr_slf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_srf (gr, phi, psi, k) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: gr
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_r
  type(momentum), intent(in) :: k
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  phipsi = gr_sf (gr, phi, psi_r, k)
  end function gr_srf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_slrf (gl, gr, phi, psi, k) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: gl, gr
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  phipsi = gr_slf (gl, phi, psi, k) + gr_srf (gr, phi, psi, k)
  end function gr_slrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function grkggf (psi, k) result (kpsi)
  type(vectorspinor) :: kpsi
  complex(kind=default) :: kp, km, k12, k12s
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: k
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  kpsi%psi(1)%a(1) = (-km) * psi%a(1) + k12s * psi%a(2)
  kpsi%psi(1)%a(2) = k12 * psi%a(1) - kp * psi%a(2)
  kpsi%psi(1)%a(3) = kp * psi%a(3) + k12s * psi%a(4)
  kpsi%psi(1)%a(4) = k12 * psi%a(3) + km * psi%a(4)
  kpsi%psi(2)%a(1) = (-k12s) * psi%a(1) + km * psi%a(2)
  kpsi%psi(2)%a(2) = kp * psi%a(1) - k12 * psi%a(2)
  kpsi%psi(2)%a(3) = k12s * psi%a(3) + kp * psi%a(4)
  kpsi%psi(2)%a(4) = km * psi%a(3) + k12 * psi%a(4)
  kpsi%psi(3)%a(1) = (0,-1) * (k12s * psi%a(1) + km * psi%a(2))
  kpsi%psi(3)%a(2) = (0,1) * (kp * psi%a(1) + k12 * psi%a(2))
  kpsi%psi(3)%a(3) = (0,1) * (k12s * psi%a(3) - kp * psi%a(4))
  kpsi%psi(3)%a(4) = (0,1) * (km * psi%a(3) - k12 * psi%a(4))
  kpsi%psi(4)%a(1) = km * psi%a(1) + k12s * psi%a(2)
  kpsi%psi(4)%a(2) = -(k12 * psi%a(1) + kp * psi%a(2))
  kpsi%psi(4)%a(3) = kp * psi%a(3) - k12s * psi%a(4)
  kpsi%psi(4)%a(4) = k12 * psi%a(3) - km * psi%a(4)
  end function grkggf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_pf (g, phi, psi, k) result (phipsi)
  type(vectorspinor) :: phipsi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi
  type(bispinor), intent(in) :: psi
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  phipsi = (g * phi) * grkggf (psi, vk)
  end function gr_pf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function grkkggf (v, psi, k) result (psikv)
  type(vectorspinor) :: psikv
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v, k
  complex(kind=default) :: kv30, kv21, kv01, kv31, kv02, kv32
  complex(kind=default) :: ap, am, bp, bm, bps, bms, imago
  imago = (0.0_default,1.0_default)
  kv30 = k%x(3) * v%t - k%t * v%x(3)
  kv21 = imago * (k%x(2) * v%x(1) - k%x(1) * v%x(2))
  kv01 = k%t * v%x(1) - k%x(1) * v%t
  kv31 = k%x(3) * v%x(1) - k%x(1) * v%x(3)
  kv02 = imago * (k%t * v%x(2) - k%x(2) * v%t)
  kv32 = imago * (k%x(3) * v%x(2) - k%x(2) * v%x(3))
  ap  = 2 * (kv30 + kv21)
  am  = 2 * ((-kv30) + kv21)
  bp  = 2 * (kv01 + kv31 + kv02 + kv32)
  bm  = 2 * (kv01 - kv31 + kv02 - kv32)
  bps = 2 * (kv01 + kv31 - kv02 - kv32)
  bms = 2 * (kv01 - kv31 - kv02 + kv32)
  psikv%psi(1)%a(1) = am * psi%a(3) + bms * psi%a(4)
  psikv%psi(1)%a(2) = bp * psi%a(3) + (-am) * psi%a(4)
  psikv%psi(1)%a(3) = (-ap) * psi%a(1) + bps * psi%a(2)
  psikv%psi(1)%a(4) = bm * psi%a(1) + ap * psi%a(2)
  psikv%psi(2)%a(1) = bms * psi%a(3) + am * psi%a(4)
  psikv%psi(2)%a(2) = (-am) * psi%a(3) + bp * psi%a(4)
  psikv%psi(2)%a(3) = (-bps) * psi%a(1) + ap * psi%a(2)
  psikv%psi(2)%a(4) = (-ap) * psi%a(1) + (-bm) * psi%a(2)
  psikv%psi(3)%a(1) = imago * (bms * psi%a(3) - am * psi%a(4))
  psikv%psi(3)%a(2) = (-imago) * (am * psi%a(3) + bp * psi%a(4))
  psikv%psi(3)%a(3) = (-imago) * (bps * psi%a(1) + ap * psi%a(2))
  psikv%psi(3)%a(4) = imago * ((-ap) * psi%a(1) + bm * psi%a(2))
  psikv%psi(4)%a(1) = am * psi%a(3) + (-bms) * psi%a(4)
  psikv%psi(4)%a(2) = bp * psi%a(3) + am * psi%a(4)
  psikv%psi(4)%a(3) = ap * psi%a(1) + bps * psi%a(2)
  psikv%psi(4)%a(4) = (-bm) * psi%a(1) + ap * psi%a(2)
  end function grkkggf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_vf (g, v, psi, k) result (psikv)
  type(vectorspinor) :: psikv
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: g
  type(vector) :: vk
  vk = k
  psikv = g * (grkkggf (v, psi, vk))
  end function gr_vf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_vlrf (gl, gr, v, psi, k) result (psikv)
  type(vectorspinor) :: psikv
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l, psi_r
  type(vector), intent(in) :: v
  type(momentum), intent(in) :: k
  complex(kind=default), intent(in) :: gl, gr
  type(vector) :: vk
  vk = k
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  psikv = gl * grkkggf (v, psi_l, vk) + gr * grkkggf (v, psi_r, vk)
  end function gr_vlrf
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: v_grf, v_fgr
```

⟨*Declaration of bispinor currents*⟩+≡
```
  public :: vlr_grf, vlr_fgr
```
$V^\mu = \psi_\rho^T C^{\mu\rho} \psi$

⟨*Implementation of bispinor currents*⟩+≡
```
  pure function grkgggf (psil, psir, k) result (j)
  type(vector) :: j
  type(vectorspinor), intent(in) :: psil
  type(bispinor), intent(in) :: psir
  type(vector), intent(in) :: k
  type(vectorspinor) :: c_psir0, c_psir1, c_psir2, c_psir3
  complex(kind=default) :: kp, km, k12, k12s, ik2
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  (k%x(1) + (0,1)*k%x(2))
  k12s =  (k%x(1) - (0,1)*k%x(2))
  ik2 = (0,1) * k%x(2)
  !!! New version:
  c_psir0%psi(1)%a(1) = (-k%x(3)) * psir%a(3) + (-k12s) * psir%a(4)
  c_psir0%psi(1)%a(2) = (-k12) * psir%a(3) + k%x(3) * psir%a(4)
  c_psir0%psi(1)%a(3) = (-k%x(3)) * psir%a(1) + (-k12s) * psir%a(2)
  c_psir0%psi(1)%a(4) = (-k12) * psir%a(1) + k%x(3) * psir%a(2)
  c_psir0%psi(2)%a(1) = (-k12s) * psir%a(3) + (-k%x(3)) * psir%a(4)
  c_psir0%psi(2)%a(2) = k%x(3) * psir%a(3) + (-k12) * psir%a(4)
  c_psir0%psi(2)%a(3) = k12s * psir%a(1) + k%x(3) * psir%a(2)
  c_psir0%psi(2)%a(4) = (-k%x(3)) * psir%a(1) + k12 * psir%a(2)
  c_psir0%psi(3)%a(1) = (0,1) * ((-k12s) * psir%a(3) + k%x(3) * psir%a(4))
  c_psir0%psi(3)%a(2) = (0,1) * (k%x(3) * psir%a(3) + k12 * psir%a(4))
  c_psir0%psi(3)%a(3) = (0,1) * (k12s * psir%a(1) + (-k%x(3)) * psir%a(2))
  c_psir0%psi(3)%a(4) = (0,1) * ((-k%x(3)) * psir%a(1) + (-k12) * psir%a(2))
  c_psir0%psi(4)%a(1) = (-k%x(3)) * psir%a(3) + k12s * psir%a(4)
  c_psir0%psi(4)%a(2) = (-k12) * psir%a(3) + (-k%x(3)) * psir%a(4)
  c_psir0%psi(4)%a(3) = k%x(3) * psir%a(1) + (-k12s) * psir%a(2)
  c_psir0%psi(4)%a(4) = k12 * psir%a(1) + k%x(3) * psir%a(2)
  !!!
  c_psir1%psi(1)%a(1) = (-ik2) * psir%a(3) + (-km) * psir%a(4)
  c_psir1%psi(1)%a(2) = (-kp) * psir%a(3) + ik2 * psir%a(4)
  c_psir1%psi(1)%a(3) = ik2 * psir%a(1) + (-kp) * psir%a(2)
  c_psir1%psi(1)%a(4) = (-km) * psir%a(1) + (-ik2) * psir%a(2)
  c_psir1%psi(2)%a(1) = (-km) * psir%a(3) + (-ik2) * psir%a(4)
  c_psir1%psi(2)%a(2) = ik2 * psir%a(3) + (-kp) * psir%a(4)
  c_psir1%psi(2)%a(3) = kp * psir%a(1) + (-ik2) * psir%a(2)
  c_psir1%psi(2)%a(4) = ik2 * psir%a(1) + km * psir%a(2)
  c_psir1%psi(3)%a(1) = ((0,-1) * km) * psir%a(3) + (-k%x(2)) * psir%a(4)
  c_psir1%psi(3)%a(2) = (-k%x(2)) * psir%a(3) + ((0,1) * kp) * psir%a(4)
  c_psir1%psi(3)%a(3) = ((0,1) * kp) * psir%a(1) + (-k%x(2)) * psir%a(2)
  c_psir1%psi(3)%a(4) = (-k%x(2)) * psir%a(1) + ((0,-1) * km) * psir%a(2)
  c_psir1%psi(4)%a(1) = (-ik2) * psir%a(3) + km * psir%a(4)
  c_psir1%psi(4)%a(2) = (-kp) * psir%a(3) + (-ik2) * psir%a(4)
  c_psir1%psi(4)%a(3) = (-ik2) *  psir%a(1) + (-kp) * psir%a(2)
  c_psir1%psi(4)%a(4) = km * psir%a(1) + (-ik2) * psir%a(2)
  !!!
  c_psir2%psi(1)%a(1) = (0,1) * (k%x(1) * psir%a(3) + km * psir%a(4))
  c_psir2%psi(1)%a(2) = (0,-1) * (kp * psir%a(3) + k%x(1) * psir%a(4))
  c_psir2%psi(1)%a(3) = (0,1) * ((-k%x(1)) * psir%a(1) + kp * psir%a(2))
  c_psir2%psi(1)%a(4) = (0,1) * ((-km) * psir%a(1) + k%x(1) * psir%a(2))
  c_psir2%psi(2)%a(1) = (0,1) * (km * psir%a(3) + k%x(1) * psir%a(4))
  c_psir2%psi(2)%a(2) = (0,-1) * (k%x(1) * psir%a(3) + kp * psir%a(4))
  c_psir2%psi(2)%a(3) = (0,-1) * (kp * psir%a(1) + (-k%x(1)) * psir%a(2))
  c_psir2%psi(2)%a(4) = (0,-1) * (k%x(1) * psir%a(1) + (-km) * psir%a(2))
  c_psir2%psi(3)%a(1) = (-km) * psir%a(3) + k%x(1) * psir%a(4)
  c_psir2%psi(3)%a(2) = k%x(1) * psir%a(3) + (-kp) * psir%a(4)
  c_psir2%psi(3)%a(3) = kp * psir%a(1) + k%x(1) * psir%a(2)
  c_psir2%psi(3)%a(4) = k%x(1) * psir%a(1) + km * psir%a(2)
  c_psir2%psi(4)%a(1) = (0,1) * (k%x(1) * psir%a(3) + (-km) * psir%a(4))
  c_psir2%psi(4)%a(2) = (0,1) * ((-kp) * psir%a(3) + k%x(1) * psir%a(4))
  c_psir2%psi(4)%a(3) = (0,1) * (k%x(1) * psir%a(1) + kp * psir%a(2))
  c_psir2%psi(4)%a(4) = (0,1) * (km * psir%a(1) + k%x(1) * psir%a(2))
  !!!
  c_psir3%psi(1)%a(1) = (-k%t) * psir%a(3) - k12s * psir%a(4)
  c_psir3%psi(1)%a(2) = k12 * psir%a(3) + k%t * psir%a(4)
  c_psir3%psi(1)%a(3) = (-k%t) * psir%a(1) + k12s * psir%a(2)
  c_psir3%psi(1)%a(4) = (-k12) * psir%a(1) + k%t * psir%a(2)
  c_psir3%psi(2)%a(1) = (-k12s) * psir%a(3) + (-k%t) * psir%a(4)
  c_psir3%psi(2)%a(2) = k%t * psir%a(3) + k12 * psir%a(4)
  c_psir3%psi(2)%a(3) = (-k12s) * psir%a(1) + k%t * psir%a(2)
  c_psir3%psi(2)%a(4) = (-k%t) * psir%a(1) + k12 * psir%a(2)
  c_psir3%psi(3)%a(1) = (0,-1) * (k12s * psir%a(3) + (-k%t) * psir%a(4))
  c_psir3%psi(3)%a(2) = (0,1) * (k%t * psir%a(3) + (-k12) * psir%a(4))
  c_psir3%psi(3)%a(3) = (0,-1) * (k12s * psir%a(1) + k%t * psir%a(2))
  c_psir3%psi(3)%a(4) = (0,-1) * (k%t * psir%a(1) + k12 * psir%a(2))
  c_psir3%psi(4)%a(1) = (-k%t) * psir%a(3) + k12s * psir%a(4)
  c_psir3%psi(4)%a(2) = k12 * psir%a(3) + (-k%t) * psir%a(4)
  c_psir3%psi(4)%a(3) = k%t * psir%a(1) + k12s * psir%a(2)
  c_psir3%psi(4)%a(4) = k12 * psir%a(1) + k%t * psir%a(2)
  j%t    =   2 * (psil * c_psir0)
  j%x(1) =   2 * (psil * c_psir1)
  j%x(2) =   2 * (psil * c_psir2)
  j%x(3) =   2 * (psil * c_psir3)
  end function grkgggf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v_grf (g, psil, psir, k) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: psil
  type(bispinor), intent(in) :: psir
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  j = g * grkgggf (psil, psir, vk)
  end function v_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function vlr_grf (gl, gr, psil, psir, k) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(vectorspinor), intent(in) :: psil
  type(bispinor), intent(in) :: psir
  type(bispinor) :: psir_l, psir_r
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  psir_l%a(1:2) = psir%a(1:2)
  psir_l%a(3:4) = 0
  psir_r%a(1:2) = 0
  psir_r%a(3:4) = psir%a(3:4)
  j = gl * grkgggf (psil, psir_l, vk) + gr * grkgggf (psil, psir_r, vk)
  end function vlr_grf
```

$V^\mu = \psi^T \tilde{C}^{\mu\rho}\psi_\rho$; remember the reversed index order in $\tilde{C}$.

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function fggkggr (psil, psir, k) result (j)
  type(vector) :: j
  type(vectorspinor), intent(in) :: psir
  type(bispinor), intent(in) :: psil
  type(vector), intent(in) :: k
  type(bispinor) :: c_psir0, c_psir1, c_psir2, c_psir3
  complex(kind=default) :: kp, km, k12, k12s, ik1, ik2
  kp = k%t + k%x(3)
  km = k%t - k%x(3)
  k12  =  k%x(1) + (0,1)*k%x(2)
  k12s =  k%x(1) - (0,1)*k%x(2)
  ik1 = (0,1) * k%x(1)
  ik2 = (0,1) * k%x(2)
```

```
c_psir0%a(1) = k%x(3) * (psir%psi(1)%a(4) + psir%psi(4)%a(4) &
+ psir%psi(2)%a(3) + (0,1) * psir%psi(3)%a(3)) &
- k12 * (psir%psi(1)%a(3) + psir%psi(4)%a(3)) &
+ k12s * (psir%psi(2)%a(4) + (0,1) * psir%psi(3)%a(4))
c_psir0%a(2) = k%x(3) * (psir%psi(1)%a(3) - psir%psi(4)%a(3) + &
psir%psi(2)%a(4) - (0,1) * psir%psi(3)%a(4)) + &
k12s * (psir%psi(1)%a(4) - psir%psi(4)%a(4)) - &
k12 * (psir%psi(2)%a(3) - (0,1) * psir%psi(3)%a(3))
c_psir0%a(3) = k%x(3) * (-psir%psi(1)%a(2) + psir%psi(4)%a(2) + &
psir%psi(2)%a(1) + (0,1) * psir%psi(3)%a(1)) + &
k12 * (psir%psi(1)%a(1) - psir%psi(4)%a(1)) + &
k12s * (psir%psi(2)%a(2) + (0,1) * psir%psi(3)%a(2))
c_psir0%a(4) = k%x(3) * (-psir%psi(1)%a(1) - psir%psi(4)%a(1) + &
psir%psi(2)%a(2) - (0,1) * psir%psi(3)%a(2)) -  &
k12s * (psir%psi(1)%a(2) + psir%psi(4)%a(2)) - &
k12 * (psir%psi(2)%a(1) - (0,1) * psir%psi(3)%a(1))
!!!
c_psir1%a(1) = ik2 * (-psir%psi(1)%a(4) - psir%psi(4)%a(4) - &
psir%psi(2)%a(3) - (0,1) * psir%psi(3)%a(3)) - &
km * (psir%psi(1)%a(3) + psir%psi(4)%a(3)) + &
kp * (psir%psi(2)%a(4) + (0,1) * psir%psi(3)%a(4))
c_psir1%a(2) = ik2 * (-psir%psi(1)%a(3) - psir%psi(2)%a(4) + &
psir%psi(4)%a(3) + (0,1) * psir%psi(3)%a(4)) + &
kp * (psir%psi(1)%a(4) - psir%psi(4)%a(4)) - &
km * (psir%psi(2)%a(3) - (0,1) * psir%psi(3)%a(3))
c_psir1%a(3) = ik2 * (-psir%psi(1)%a(2) + psir%psi(2)%a(1) + &
psir%psi(4)%a(2) + (0,1) * psir%psi(3)%a(1)) + &
kp * (psir%psi(1)%a(1) - psir%psi(4)%a(1)) + &
km * (psir%psi(2)%a(2) + (0,1) * psir%psi(3)%a(2))
c_psir1%a(4) = ik2 * (-psir%psi(1)%a(1) + psir%psi(2)%a(2) - &
psir%psi(4)%a(1) - (0,1) * psir%psi(3)%a(2)) - &
km * (psir%psi(1)%a(2) + psir%psi(4)%a(2)) - &
kp * (psir%psi(2)%a(1) - (0,1) * psir%psi(3)%a(1))
!!!
c_psir2%a(1) = ik1 * (psir%psi(2)%a(3) + psir%psi(1)%a(4) &
+ psir%psi(4)%a(4) + (0,1) * psir%psi(3)%a(3)) - &
((0,1)*km) * (psir%psi(1)%a(3) + psir%psi(4)%a(3)) &
+ kp * (psir%psi(3)%a(4) - (0,1) * psir%psi(2)%a(4))
c_psir2%a(2) = ik1 * (psir%psi(1)%a(3) + psir%psi(2)%a(4) - &
psir%psi(4)%a(3) - (0,1) * psir%psi(3)%a(4)) - &
((0,1)*kp) * (psir%psi(1)%a(4) - psir%psi(4)%a(4)) &
- km * (psir%psi(3)%a(3) + (0,1) * psir%psi(2)%a(3))
c_psir2%a(3) = ik1 * (psir%psi(1)%a(2) - psir%psi(2)%a(1) - &
psir%psi(4)%a(2) - (0,1) * psir%psi(3)%a(1)) + &
((0,1)*kp) * (psir%psi(1)%a(1) - psir%psi(4)%a(1)) &
+ km * (psir%psi(3)%a(2) - (0,1) * psir%psi(2)%a(2))
c_psir2%a(4) = ik1 * (psir%psi(1)%a(1) - psir%psi(2)%a(2) + &
psir%psi(4)%a(1) + (0,1) * psir%psi(3)%a(2)) + &
((0,1)*km) * (psir%psi(1)%a(2) + psir%psi(4)%a(2)) - &
kp * (psir%psi(3)%a(1) + (0,1) * psir%psi(2)%a(1))
!!!
c_psir3%a(1) = k%t * (psir%psi(1)%a(4) + psir%psi(4)%a(4) + &
psir%psi(2)%a(3) + (0,1) * psir%psi(3)%a(3)) - &
k12 * (psir%psi(1)%a(3) + psir%psi(4)%a(3)) - &
k12s * (psir%psi(2)%a(4) + (0,1) * psir%psi(3)%a(4))
c_psir3%a(2) = k%t * (psir%psi(1)%a(3) - psir%psi(4)%a(3) + &
psir%psi(2)%a(4) - (0,1) * psir%psi(3)%a(4)) - &
k12s * (psir%psi(1)%a(4) - psir%psi(4)%a(4)) - &
k12 * (psir%psi(2)%a(3) - (0,1) * psir%psi(3)%a(3))
c_psir3%a(3) = k%t * (-psir%psi(1)%a(2) + psir%psi(2)%a(1) + &
psir%psi(4)%a(2) + (0,1) * psir%psi(3)%a(1)) - &
k12 * (psir%psi(1)%a(1) - psir%psi(4)%a(1)) + &
k12s * (psir%psi(2)%a(2) + (0,1) * psir%psi(3)%a(2))
c_psir3%a(4) = k%t * (-psir%psi(1)%a(1) + psir%psi(2)%a(2) - &
psir%psi(4)%a(1) - (0,1) * psir%psi(3)%a(2)) - &
k12s * (psir%psi(1)%a(2) + psir%psi(4)%a(2)) + &
```

```
  k12 * (psir%psi(2)%a(1) - (0,1) * psir%psi(3)%a(1))
  !!! Because we explicitly multiplied the charge conjugation matrix
  !!! we have to omit it from the spinor product and take the
  !!! ordinary product!
  j%t    =   2 * dot_product (conjg (psil%a), c_psir0%a)
  j%x(1) =   2 * dot_product (conjg (psil%a), c_psir1%a)
  j%x(2) =   2 * dot_product (conjg (psil%a), c_psir2%a)
  j%x(3) =   2 * dot_product (conjg (psil%a), c_psir3%a)
  end function fggkggr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v_fgr (g, psil, psir, k) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: psir
  type(bispinor), intent(in) :: psil
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  j = g * fggkggr (psil, psir, vk)
  end function v_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function vlr_fgr (gl, gr, psil, psir, k) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(vectorspinor), intent(in) :: psir
  type(bispinor), intent(in) :: psil
  type(bispinor) :: psil_l
  type(bispinor) :: psil_r
  type(momentum), intent(in) :: k
  type(vector) :: vk
  vk = k
  psil_l%a(1:2) = psil%a(1:2)
  psil_l%a(3:4) = 0
  psil_r%a(1:2) = 0
  psil_r%a(3:4) = psil%a(3:4)
  j = gl * fggkggr (psil_l, psir, vk) + gr * fggkggr (psil_r, psir, vk)
  end function vlr_fgr
```

AB.26.5 Gravitino 4-Couplings

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: f_s2gr, f_svgr, f_slvgr, f_srvgr, f_slrvgr, f_pvgr, f_v2gr, f_v2lrgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_s2gr (g, phi1, phi2, psi) result (phipsi)
  type(bispinor) :: phipsi
  type(vectorspinor), intent(in) :: psi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi1, phi2
  phipsi = phi2 * f_potgr (g, phi1, psi)
  end function f_s2gr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_svgr (g, phi, v, grav) result (phigrav)
  type(bispinor) :: phigrav
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g, phi
  phigrav = (g * phi) * fgvg5gr (grav, v)
  end function f_svgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_slvgr (gl, phi, v, grav) result (phigrav)
  type(bispinor) :: phigrav, phidum
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gl, phi
  phidum = (gl * phi) * fgvg5gr (grav, v)
  phigrav%a(1:2) = phidum%a(1:2)
  phigrav%a(3:4) = 0
  end function f_slvgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_srvgr (gr, phi, v, grav) result (phigrav)
  type(bispinor) :: phigrav, phidum
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gr, phi
  phidum = (gr * phi) * fgvg5gr (grav, v)
  phigrav%a(1:2) = 0
  phigrav%a(3:4) = phidum%a(3:4)
  end function f_srvgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_slrvgr (gl, gr, phi, v, grav) result (phigrav)
  type(bispinor) :: phigrav
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gl, gr, phi
  phigrav = f_slvgr (gl, phi, v, grav) + f_srvgr (gr, phi, v, grav)
  end function f_slrvgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_pvgr (g, phi, v, grav) result (phigrav)
  type(bispinor) :: phigrav
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g, phi
  phigrav = (g * phi) * fgvgr (grav, v)
  end function f_pvgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_v2gr (g, v1, v2, grav) result (psi)
  type(bispinor) :: psi
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v1, v2
  psi = g * fggvvgr (v2, grav, v1)
  end function f_v2gr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function f_v2lrgr (gl, gr, v1, v2, grav) result (psi)
  type(bispinor) :: psi
  complex(kind=default), intent(in) :: gl, gr
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v1, v2
  psi = fggvvgr (v2, grav, v1)
  psi%a(1:2) = gl * psi%a(1:2)
  psi%a(3:4) = gr * psi%a(3:4)
  end function f_v2lrgr
```

⟨*Declaration of bispinor currents*⟩+≡

```
  public :: gr_s2f, gr_svf, gr_pvf, gr_slvf, gr_srvf, gr_slrvf, gr_v2f, gr_v2lrf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_s2f (g, phi1, phi2, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  complex(kind=default), intent(in) :: g
  complex(kind=default), intent(in) :: phi1, phi2
  phipsi = phi2 * gr_potf (g, phi1, psi)
  end function gr_s2f
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_svf (g, phi, v, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g, phi
  phipsi = (g * phi) * grkggf (psi, v)
  end function gr_svf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_slvf (gl, phi, v, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gl, phi
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  phipsi = (gl * phi) * grkggf (psi_l, v)
  end function gr_slvf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_srvf (gr, phi, v, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_r
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gr, phi
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  phipsi = (gr * phi) * grkggf (psi_r, v)
  end function gr_srvf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_slrvf (gl, gr, phi, v, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: gl, gr, phi
  phipsi = gr_slvf (gl, phi, v, psi) + gr_srvf (gr, phi, v, psi)
  end function gr_slrvf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_pvf (g, phi, v, psi) result (phipsi)
  type(vectorspinor) :: phipsi
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  complex(kind=default), intent(in) :: g, phi
  phipsi = (g * phi) * grkgf (psi, v)
  end function gr_pvf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_v2f (g, v1, v2, psi) result (vvpsi)
  type(vectorspinor) :: vvpsi
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v1, v2
  vvpsi = g * grkkggf (v2, psi, v1)
  end function gr_v2f
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function gr_v2lrf (gl, gr, v1, v2, psi) result (vvpsi)
  type(vectorspinor) :: vvpsi
  complex(kind=default), intent(in) :: gl, gr
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l, psi_r
  type(vector), intent(in) :: v1, v2
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  vvpsi = gl * grkkggf (v2, psi_l, v1) + gr * grkkggf (v2, psi_r, v1)
  end function gr_v2lrf
```

⟨*Declaration of bispinor currents*⟩+≡
```
public :: s2_grf, s2_fgr, sv1_grf, sv2_grf, sv1_fgr, sv2_fgr, &
slv1_grf, slv2_grf, slv1_fgr, slv2_fgr, &
srv1_grf, srv2_grf, srv1_fgr, srv2_fgr, &
slrv1_grf, slrv2_grf, slrv1_fgr, slrv2_fgr, &
pv1_grf, pv2_grf, pv1_fgr, pv2_fgr, v2_grf, v2_fgr, &
v2lr_grf, v2lr_fgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function s2_grf (g, gravbar, phi, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g, phi
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
j = phi * pot_grf (g, gravbar, psi)
end function s2_grf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function s2_fgr (g, psibar, phi, grav) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g, phi
type(bispinor), intent(in) :: psibar
type(vectorspinor), intent(in) :: grav
j = phi * pot_fgr (g, psibar, grav)
end function s2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function sv1_grf (g, gravbar, v, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(vector), intent(in) :: v
j = g * grg5vgf (gravbar, psi, v)
end function sv1_grf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function slv1_grf (gl, gravbar, v, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gl
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(bispinor) :: psi_l
type(vector), intent(in) :: v
psi_l%a(1:2) = psi%a(1:2)
psi_l%a(3:4) = 0
j = gl * grg5vgf (gravbar, psi_l, v)
end function slv1_grf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function srv1_grf (gr, gravbar, v, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gr
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(bispinor) :: psi_r
type(vector), intent(in) :: v
psi_r%a(1:2) = 0
psi_r%a(3:4) = psi%a(3:4)
j = gr * grg5vgf (gravbar, psi_r, v)
end function srv1_grf
```

⟨*Implementation of bispinor currents*⟩+≡
```
pure function slrv1_grf (gl, gr, gravbar, v, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: gl, gr
type(vectorspinor), intent(in) :: gravbar
type(bispinor), intent(in) :: psi
type(bispinor) :: psi_l, psi_r
```

```
  type(vector), intent(in) :: v
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  j = gl * grg5vgf (gravbar, psi_l, v) + gr * grg5vgf (gravbar, psi_r, v)
  end function slrv1_grf
```

$$C\gamma^0\gamma^0 = -C\gamma^1\gamma^1 = -C\gamma^2\gamma^2 = C\gamma^3\gamma^3 = C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{AB.129a}$$

$$C\gamma^0\gamma^1 = -C\gamma^1\gamma^0 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{AB.129b}$$

$$C\gamma^0\gamma^2 = -C\gamma^2\gamma^0 = \begin{pmatrix} -\mathrm{i} & 0 & 0 & 0 \\ 0 & -\mathrm{i} & 0 & 0 \\ 0 & 0 & -\mathrm{i} & 0 \\ 0 & 0 & 0 & -\mathrm{i} \end{pmatrix} \tag{AB.129c}$$

$$C\gamma^0\gamma^3 = -C\gamma^3\gamma^0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{AB.129d}$$

$$C\gamma^1\gamma^2 = -C\gamma^2\gamma^1 = \begin{pmatrix} 0 & \mathrm{i} & 0 & 0 \\ \mathrm{i} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\mathrm{i} \\ 0 & 0 & -\mathrm{i} & 0 \end{pmatrix} \tag{AB.129e}$$

$$C\gamma^1\gamma^3 = -C\gamma^3\gamma^1 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{AB.129f}$$

$$C\gamma^2\gamma^3 = -C\gamma^3\gamma^2 = \begin{pmatrix} -\mathrm{i} & 0 & 0 & 0 \\ 0 & \mathrm{i} & 0 & 0 \\ 0 & 0 & \mathrm{i} & 0 \\ 0 & 0 & 0 & -\mathrm{i} \end{pmatrix} \tag{AB.129g}$$

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function sv2_grf (g, gravbar, phi, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g, phi
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(vectorspinor) :: g0_psi, g1_psi, g2_psi, g3_psi
  g0_psi%psi(1)%a(1:2) = - psi%a(1:2)
  g0_psi%psi(1)%a(3:4) = psi%a(3:4)
  g0_psi%psi(2)%a(1) = psi%a(2)
  g0_psi%psi(2)%a(2) = psi%a(1)
  g0_psi%psi(2)%a(3) = psi%a(4)
  g0_psi%psi(2)%a(4) = psi%a(3)
  g0_psi%psi(3)%a(1) = (0,-1) * psi%a(2)
  g0_psi%psi(3)%a(2) = (0,1) * psi%a(1)
  g0_psi%psi(3)%a(3) = (0,-1) * psi%a(4)
  g0_psi%psi(3)%a(4) = (0,1) * psi%a(3)
  g0_psi%psi(4)%a(1) = psi%a(1)
  g0_psi%psi(4)%a(2) = - psi%a(2)
  g0_psi%psi(4)%a(3) = psi%a(3)
  g0_psi%psi(4)%a(4) = - psi%a(4)
  g1_psi%psi(1)%a(1:4) = - g0_psi%psi(2)%a(1:4)
  g1_psi%psi(2)%a(1:4) = - g0_psi%psi(1)%a(1:4)
  g1_psi%psi(3)%a(1) = (0,1) * psi%a(1)
  g1_psi%psi(3)%a(2) = (0,-1) * psi%a(2)
  g1_psi%psi(3)%a(3) = (0,-1) * psi%a(3)
  g1_psi%psi(3)%a(4) = (0,1) * psi%a(4)
  g1_psi%psi(4)%a(1) = - psi%a(2)
  g1_psi%psi(4)%a(2) = psi%a(1)
  g1_psi%psi(4)%a(3) = psi%a(4)
  g1_psi%psi(4)%a(4) = - psi%a(3)
  g2_psi%psi(1)%a(1:4) = - g0_psi%psi(3)%a(1:4)
  g2_psi%psi(2)%a(1:4) = - g1_psi%psi(3)%a(1:4)
  g2_psi%psi(3)%a(1:4) = - g0_psi%psi(1)%a(1:4)
  g2_psi%psi(4)%a(1) = (0,1) * psi%a(2)
  g2_psi%psi(4)%a(2) = (0,1) * psi%a(1)
  g2_psi%psi(4)%a(3) = (0,-1) * psi%a(4)
  g2_psi%psi(4)%a(4) = (0,-1) * psi%a(3)
  g3_psi%psi(1)%a(1:4) = - g0_psi%psi(4)%a(1:4)
  g3_psi%psi(2)%a(1:4) = - g1_psi%psi(4)%a(1:4)
  g3_psi%psi(3)%a(1:4) = - g2_psi%psi(4)%a(1:4)
  g3_psi%psi(4)%a(1:4) = - g0_psi%psi(1)%a(1:4)
  j%t    =   (g * phi) * (gravbar * g0_psi)
  j%x(1) =   (g * phi) * (gravbar * g1_psi)
  j%x(2) =   (g * phi) * (gravbar * g2_psi)
  j%x(3) =   (g * phi) * (gravbar * g3_psi)
  end function sv2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function slv2_grf (gl, gravbar, phi, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, phi
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  j = sv2_grf (gl, gravbar, phi, psi_l)
  end function slv2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function srv2_grf (gr, gravbar, phi, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gr, phi
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_r
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  j = sv2_grf (gr, gravbar, phi, psi_r)
  end function srv2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function slrv2_grf (gl, gr, gravbar, phi, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr, phi
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l, psi_r
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  j = sv2_grf (gl, gravbar, phi, psi_l) + sv2_grf (gr, gravbar, phi, psi_r)
  end function slrv2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function sv1_fgr (g, psibar, v, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  j = g * fg5gkgr (psibar, grav, v)
  end function sv1_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function slv1_fgr (gl, psibar, v, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_l
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  psibar_l%a(1:2) = psibar%a(1:2)
  psibar_l%a(3:4) = 0
  j = gl * fg5gkgr (psibar_l, grav, v)
  end function slv1_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function srv1_fgr (gr, psibar, v, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gr
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_r
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  psibar_r%a(1:2) = 0
  psibar_r%a(3:4) = psibar%a(3:4)
  j = gr * fg5gkgr (psibar_r, grav, v)
  end function srv1_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function slrv1_fgr (gl, gr, psibar, v, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_l, psibar_r
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  psibar_l%a(1:2) = psibar%a(1:2)
  psibar_l%a(3:4) = 0
  psibar_r%a(1:2) = 0
  psibar_r%a(3:4) = psibar%a(3:4)
  j = gl * fg5gkgr (psibar_l, grav, v)  + gr * fg5gkgr (psibar_r, grav, v)
  end function slrv1_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function sv2_fgr (g, psibar, phi, grav) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g, phi
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(bispinor) :: g0_grav, g1_grav, g2_grav, g3_grav
  g0_grav%a(1) = -grav%psi(1)%a(1) +  grav%psi(2)%a(2) - &
  (0,1) * grav%psi(3)%a(2) + grav%psi(4)%a(1)
  g0_grav%a(2) = -grav%psi(1)%a(2) + grav%psi(2)%a(1) + &
  (0,1) * grav%psi(3)%a(1) - grav%psi(4)%a(2)
  g0_grav%a(3) = grav%psi(1)%a(3) + grav%psi(2)%a(4) - &
  (0,1) * grav%psi(3)%a(4) + grav%psi(4)%a(3)
  g0_grav%a(4) = grav%psi(1)%a(4) + grav%psi(2)%a(3) + &
  (0,1) * grav%psi(3)%a(3) - grav%psi(4)%a(4)
  !!!
  g1_grav%a(1) = grav%psi(1)%a(2) - grav%psi(2)%a(1) + &
  (0,1) * grav%psi(3)%a(1) - grav%psi(4)%a(2)
  g1_grav%a(2) = grav%psi(1)%a(1) - grav%psi(2)%a(2) - &
  (0,1) * grav%psi(3)%a(2) + grav%psi(4)%a(1)
  g1_grav%a(3) = grav%psi(1)%a(4) + grav%psi(2)%a(3) - &
  (0,1) * grav%psi(3)%a(3) + grav%psi(4)%a(4)
  g1_grav%a(4) = grav%psi(1)%a(3) + grav%psi(2)%a(4) + &
```

```
(0,1) * grav%psi(3)%a(4) - grav%psi(4)%a(3)
!!!
g2_grav%a(1) = (0,1) * (-grav%psi(1)%a(2) - grav%psi(2)%a(1) + &
grav%psi(4)%a(2)) - grav%psi(3)%a(1)
g2_grav%a(2) = (0,1) * (grav%psi(1)%a(1) + grav%psi(2)%a(2) + &
grav%psi(4)%a(1)) - grav%psi(3)%a(2)
g2_grav%a(3) = (0,1) * (-grav%psi(1)%a(4) + grav%psi(2)%a(3) - &
grav%psi(4)%a(4)) + grav%psi(3)%a(3)
g2_grav%a(4) = (0,1) * (grav%psi(1)%a(3) - grav%psi(2)%a(4) - &
grav%psi(4)%a(3)) + grav%psi(3)%a(4)
!!!
g3_grav%a(1) = -grav%psi(1)%a(2) + grav%psi(2)%a(2) - &
(0,1) * grav%psi(3)%a(2) - grav%psi(4)%a(1)
g3_grav%a(2) = grav%psi(1)%a(1) - grav%psi(2)%a(1) - &
(0,1) * grav%psi(3)%a(1) - grav%psi(4)%a(2)
g3_grav%a(3) = -grav%psi(1)%a(2) - grav%psi(2)%a(4) + &
(0,1) * grav%psi(3)%a(4) + grav%psi(4)%a(3)
g3_grav%a(4) = -grav%psi(1)%a(4) + grav%psi(2)%a(3) + &
(0,1) * grav%psi(3)%a(3) + grav%psi(4)%a(4)
j%t    =   (g * phi) * (psibar * g0_grav)
j%x(1) =   (g * phi) * (psibar * g1_grav)
j%x(2) =   (g * phi) * (psibar * g2_grav)
j%x(3) =   (g * phi) * (psibar * g3_grav)
end function sv2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function slv2_fgr (gl, psibar, phi, grav) result (j)
type(vector) :: j
complex(kind=default), intent(in) :: gl, phi
type(bispinor), intent(in) :: psibar
type(bispinor) :: psibar_l
type(vectorspinor), intent(in) :: grav
psibar_l%a(1:2) = psibar%a(1:2)
psibar_l%a(3:4) = 0
j = sv2_fgr (gl, psibar_l, phi, grav)
end function slv2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function srv2_fgr (gr, psibar, phi, grav) result (j)
type(vector) :: j
complex(kind=default), intent(in) :: gr, phi
type(bispinor), intent(in) :: psibar
type(bispinor) :: psibar_r
type(vectorspinor), intent(in) :: grav
psibar_r%a(1:2) = 0
psibar_r%a(3:4) = psibar%a(3:4)
j = sv2_fgr (gr, psibar_r, phi, grav)
end function srv2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function slrv2_fgr (gl, gr, psibar, phi, grav) result (j)
type(vector) :: j
complex(kind=default), intent(in) :: gl, gr, phi
type(bispinor), intent(in) :: psibar
type(bispinor) :: psibar_l, psibar_r
type(vectorspinor), intent(in) :: grav
psibar_l%a(1:2) = psibar%a(1:2)
psibar_l%a(3:4) = 0
psibar_r%a(1:2) = 0
psibar_r%a(3:4) = psibar%a(3:4)
j = sv2_fgr (gl, psibar_l, phi, grav) + sv2_fgr (gr, psibar_r, phi, grav)
end function slrv2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
pure function pv1_grf (g, gravbar, v, psi) result (j)
complex(kind=default) :: j
complex(kind=default), intent(in) :: g
type(vectorspinor), intent(in) :: gravbar
```

```
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  j = g * grvgf (gravbar, psi, v)
  end function pv1_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function pv2_grf (g, gravbar, phi, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g, phi
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(bispinor) :: g5_psi
  g5_psi%a(1:2) = - psi%a(1:2)
  g5_psi%a(3:4) = psi%a(3:4)
  j = sv2_grf (g, gravbar, phi, g5_psi)
  end function pv2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function pv1_fgr (g, psibar, v, grav) result (j)
  complex(kind=default) :: j
  complex(kind=default), intent(in) :: g
  type(bispinor), intent(in) :: psibar
  type(vectorspinor), intent(in) :: grav
  type(vector), intent(in) :: v
  j = g * fgkgr (psibar, grav, v)
  end function pv1_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function pv2_fgr (g, psibar, phi, grav) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g, phi
  type(vectorspinor), intent(in) :: grav
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_g5
  psibar_g5%a(1:2) = - psibar%a(1:2)
  psibar_g5%a(3:4) = psibar%a(3:4)
  j = sv2_fgr (g, psibar_g5, phi, grav)
  end function pv2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v2_grf (g, gravbar, v, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(vector), intent(in) :: v
  j = -g * grkgggf (gravbar, psi, v)
  end function v2_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v2lr_grf (gl, gr, gravbar, v, psi) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(vectorspinor), intent(in) :: gravbar
  type(bispinor), intent(in) :: psi
  type(bispinor) :: psi_l, psi_r
  type(vector), intent(in) :: v
  psi_l%a(1:2) = psi%a(1:2)
  psi_l%a(3:4) = 0
  psi_r%a(1:2) = 0
  psi_r%a(3:4) = psi%a(3:4)
  j = -(gl * grkgggf (gravbar, psi_l, v) + gr * grkgggf (gravbar, psi_r, v))
  end function v2lr_grf
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v2_fgr (g, psibar, v, grav) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: g
  type(vectorspinor), intent(in) :: grav
  type(bispinor), intent(in) :: psibar
  type(vector), intent(in) :: v
  j = -g * fggkggr (psibar, grav, v)
  end function v2_fgr
```

⟨*Implementation of bispinor currents*⟩+≡

```
  pure function v2lr_fgr (gl, gr, psibar, v, grav) result (j)
  type(vector) :: j
  complex(kind=default), intent(in) :: gl, gr
  type(vectorspinor), intent(in) :: grav
  type(bispinor), intent(in) :: psibar
  type(bispinor) :: psibar_l, psibar_r
  type(vector), intent(in) :: v
  psibar_l%a(1:2) = psibar%a(1:2)
  psibar_l%a(3:4) = 0
  psibar_r%a(1:2) = 0
  psibar_r%a(3:4) = psibar%a(3:4)
  j = -(gl * fggkggr (psibar_l, grav, v) + gr * fggkggr (psibar_r, grav, v))
  end function v2lr_fgr
```

AB.26.6 On Shell Wave Functions

⟨*Declaration of bispinor on shell wave functions*⟩≡

```
  public :: u, v, ghost
```

$$\chi_+(\vec{p}) = \frac{1}{\sqrt{2|\vec{p}|(|\vec{p}|+p_3)}} \begin{pmatrix} |\vec{p}|+p_3 \\ p_1 + \mathrm{i}p_2 \end{pmatrix} \tag{AB.130a}$$

$$\chi_-(\vec{p}) = \frac{1}{\sqrt{2|\vec{p}|(|\vec{p}|+p_3)}} \begin{pmatrix} -p_1 + \mathrm{i}p_2 \\ |\vec{p}|+p_3 \end{pmatrix} \tag{AB.130b}$$

$$u_\pm(p) = \begin{pmatrix} \sqrt{p_0 \mp |\vec{p}|} \cdot \chi_\pm(\vec{p}) \\ \sqrt{p_0 \pm |\vec{p}|} \cdot \chi_\pm(\vec{p}) \end{pmatrix} \tag{AB.131}$$

⟨*Implementation of bispinor on shell wave functions*⟩≡

```
  pure function u (mass, p, s) result (psi)
  type(bispinor) :: psi
  real(kind=default), intent(in) :: mass
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  complex(kind=default), dimension(2) :: chip, chim
  real(kind=default) :: pabs, norm, delta, m
  m = abs(mass)
  pabs = sqrt (dot_product (p%x, p%x))
  if (m < epsilon (m) * pabs) then
  delta = 0
  else
  delta = sqrt (max (p%t - pabs, 0._default))
  end if
  if (pabs + p%x(3) <= 1000 * epsilon (pabs) * pabs) then
  chip = (/ cmplx ( 0.0, 0.0, kind=default), &
  cmplx ( 1.0, 0.0, kind=default) /)
  chim = (/ cmplx (-1.0, 0.0, kind=default), &
  cmplx ( 0.0, 0.0, kind=default) /)
  else
  norm = 1 / sqrt (2*pabs*(pabs + p%x(3)))
  chip = norm * (/ cmplx (pabs + p%x(3), kind=default), &
  cmplx (p%x(1), p%x(2), kind=default) /)
  chim = norm * (/ cmplx (-p%x(1), p%x(2), kind=default), &
  cmplx (pabs + p%x(3), kind=default) /)
  end if
  if (s > 0) then
  psi%a(1:2) = delta * chip
  psi%a(3:4) = sqrt (p%t + pabs) * chip
  else
```

```
psi%a(1:2) = sqrt (p%t + pabs) * chim
psi%a(3:4) = delta * chim
end if
pabs = m ! make the compiler happy and use m
if (mass < 0) then
psi%a(1:2) = - imago * psi%a(1:2)
psi%a(3:4) = + imago * psi%a(3:4)
end if
end function u
```

$$v_\pm(p) = \begin{pmatrix} \mp\sqrt{p_0 \pm |\vec{p}|} \cdot \chi_\mp(\vec{p}) \\ \pm\sqrt{p_0 \mp |\vec{p}|} \cdot \chi_\mp(\vec{p}) \end{pmatrix} \tag{AB.132}$$

⟨*Implementation of bispinor on shell wave functions*⟩+≡

```
pure function v (mass, p, s) result (psi)
type(bispinor) :: psi
real(kind=default), intent(in) :: mass
type(momentum), intent(in) :: p
integer, intent(in) :: s
complex(kind=default), dimension(2) :: chip, chim
real(kind=default) :: pabs, norm, delta, m
pabs = sqrt (dot_product (p%x, p%x))
m = abs(mass)
if (m < epsilon (m) * pabs) then
delta = 0
else
delta = sqrt (max (p%t - pabs, 0._default))
end if
if (pabs + p%x(3) <= 1000 * epsilon (pabs) * pabs) then
chip = (/ cmplx ( 0.0, 0.0, kind=default), &
cmplx ( 1.0, 0.0, kind=default) /)
chim = (/ cmplx (-1.0, 0.0, kind=default), &
cmplx ( 0.0, 0.0, kind=default) /)
else
norm = 1 / sqrt (2*pabs*(pabs + p%x(3)))
chip = norm * (/ cmplx (pabs + p%x(3), kind=default), &
cmplx (p%x(1), p%x(2), kind=default) /)
chim = norm * (/ cmplx (-p%x(1), p%x(2), kind=default), &
cmplx (pabs + p%x(3), kind=default) /)
end if
if (s > 0) then
psi%a(1:2) = - sqrt (p%t + pabs) * chim
psi%a(3:4) = delta * chim
else
psi%a(1:2) = delta * chip
psi%a(3:4) = - sqrt (p%t + pabs) * chip
end if
pabs = m ! make the compiler happy and use m
if (mass < 0) then
psi%a(1:2) = - imago * psi%a(1:2)
psi%a(3:4) = + imago * psi%a(3:4)
end if
end function v
```

⟨*Implementation of bispinor on shell wave functions*⟩+≡

```
pure function ghost (m, p, s) result (psi)
type(bispinor) :: psi
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: p
integer, intent(in) :: s
psi%a(:) = 0
select case (s)
case (1)
psi%a(1)   = 1
psi%a(2:4) = 0
case (2)
psi%a(1)   = 0
psi%a(2)   = 1
```

```
  psi%a(3:4) = 0
  case (3)
  psi%a(1:2) = 0
  psi%a(3)   = 1
  psi%a(4)   = 0
  case (4)
  psi%a(1:3) = 0
  psi%a(4)   = 1
  case (5)
  psi%a(1) =    1.4
  psi%a(2) = -  2.3
  psi%a(3) = - 71.5
  psi%a(4) =    0.1
  end select
  end function ghost
```

AB.26.7 Off Shell Wave Functions

This is the same as for the Dirac fermions except that the expressions for [ubar] and [vbar] are missing.

⟨*Declaration of bispinor off shell wave functions*⟩≡

```
  public :: brs_u, brs_v
```

In momentum space we have:

$$brsu(p) = (-i)(\not{p} - m)u(p) \tag{AB.133}$$

⟨*Implementation of bispinor off shell wave functions*⟩≡

```
  pure function brs_u (m, p, s) result (dpsi)
  type(bispinor) :: dpsi, psi
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) :: s
  type (vector)::vp
  complex(kind=default), parameter :: one = (1, 0)
  vp=p
  psi=u(m,p,s)
  dpsi=cmplx(0.0,-1.0)*(f_vf(one,vp,psi)-m*psi)
  end function brs_u
```

$$brsv(p) = i(\not{p} + m)v(p) \tag{AB.134}$$

⟨*Implementation of bispinor off shell wave functions*⟩+≡

```
  pure function brs_v (m, p, s) result (dpsi)
  type(bispinor) :: dpsi, psi
  real(kind=default), intent(in) :: m
  type(momentum), intent(in) :: p
  integer, intent(in) ::   s
  type (vector)::vp
  complex(kind=default), parameter :: one = (1, 0)
  vp=p
  psi=v(m,p,s)
  dpsi=cmplx(0.0,1.0)*(f_vf(one,vp,psi)+m*psi)
  end function brs_v
```

AB.26.8 Propagators

⟨*Declaration of bispinor propagators*⟩≡

```
  public :: pr_psi, pr_grav
  public :: pj_psi, pg_psi
```

$$\frac{\mathrm{i}(-\not{p} + m)}{p^2 - m^2 + \mathrm{i}m\Gamma}\psi \tag{AB.135}$$

NB: the sign of the momentum comes about because all momenta are treated as *outgoing* and the particle charge flow is therefore opposite to the momentum.

⟨*Implementation of bispinor propagators*⟩≡

```
  pure function pr_psi (p, m, w, cms, psi) result (ppsi)
  type(bispinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(bispinor), intent(in) :: psi
  logical, intent(in) :: cms
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  complex(kind=default) :: num_mass
  vp = p
  if (cms) then
  num_mass = sqrt(cmplx(m**2, -m*w, kind=default))
  else
  num_mass = cmplx (m, 0, kind=default)
  end if
  ppsi = (1 / cmplx (p*p - m**2, m*w, kind=default)) &
  * (- f_vf (one, vp, psi) + num_mass * psi)
  end function pr_psi
```

$$\sqrt{\frac{\pi}{M\Gamma}}(-\not p+m)\psi \tag{AB.136}$$

⟨*Implementation of bispinor propagators*⟩+≡

```
  pure function pj_psi (p, m, w, psi) result (ppsi)
  type(bispinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(bispinor), intent(in) :: psi
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  vp = p
  ppsi = (0, -1) * sqrt (PI / m / w) * (- f_vf (one, vp, psi) + m * psi)
  end function pj_psi
```

⟨*Implementation of bispinor propagators*⟩+≡

```
  pure function pg_psi (p, m, w, psi) result (ppsi)
  type(bispinor) :: ppsi
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(bispinor), intent(in) :: psi
  type(vector) :: vp
  complex(kind=default), parameter :: one = (1, 0)
  vp = p
  ppsi = gauss (p*p, m, w) * (- f_vf (one, vp, psi) + m * psi)
  end function pg_psi
```

$$\frac{\mathrm{i}\Big\{(-\not p+m)\left(-\eta_{\mu\nu}+\frac{p_\mu p_\nu}{m^2}\right)+\frac{1}{3}\left(\gamma_\mu-\frac{p_\mu}{m}\right)(\not p+m)\left(\gamma_\nu-\frac{p_\nu}{m}\right)\Big\}}{p^2-m^2+\mathrm{i}m\Gamma}\,\psi^\nu \tag{AB.137}$$

⟨*Implementation of bispinor propagators*⟩+≡

```
  pure function pr_grav (p, m, w, grav) result (propgrav)
  type(vectorspinor) :: propgrav
  type(momentum), intent(in) :: p
  real(kind=default), intent(in) :: m, w
  type(vectorspinor), intent(in) :: grav
  type(vector) :: vp
  type(bispinor) :: pgrav, ggrav, ggrav1, ggrav2, ppgrav
  type(vectorspinor) :: etagrav_dum, etagrav, pppgrav, &
  gg_grav_dum, gg_grav
  complex(kind=default), parameter :: one = (1, 0)
  real(kind=default) :: minv
  integer :: i
  vp = p
  minv = 1/m
  pgrav = p%t    * grav%psi(1) - p%x(1) * grav%psi(2) - &
  p%x(2) * grav%psi(3) - p%x(3) * grav%psi(4)
  ggrav%a(1) = grav%psi(1)%a(3) - grav%psi(2)%a(4) + (0,1) * &
  grav%psi(3)%a(4) - grav%psi(4)%a(3)
  ggrav%a(2) = grav%psi(1)%a(4) - grav%psi(2)%a(3) - (0,1) * &
  grav%psi(3)%a(3) + grav%psi(4)%a(4)
  ggrav%a(3) = grav%psi(1)%a(1) + grav%psi(2)%a(2) - (0,1) * &
```

```
grav%psi(3)%a(2) + grav%psi(4)%a(1)
gg rav%a(4) = grav%psi(1)%a(2) + grav%psi(2)%a(1) + (0,1) * &
grav%psi(3)%a(1) - grav%psi(4)%a(2)
ggrav1 = ggrav - minv * pgrav
ggrav2 = f_vf (one, vp, ggrav1) + m * ggrav - pgrav
ppgrav = (-minv**2) * f_vf (one, vp, pgrav) + minv * pgrav
do i = 1, 4
etagrav_dum%psi(i) = f_vf (one, vp, grav%psi(i))
end do
etagrav = etagrav_dum - m * grav
pppgrav%psi(1) = p%t    * ppgrav
pppgrav%psi(2) = p%x(1) * ppgrav
pppgrav%psi(3) = p%x(2) * ppgrav
pppgrav%psi(4) = p%x(3) * ppgrav
gg_grav_dum%psi(1) = p%t    * ggrav2
gg_grav_dum%psi(2) = p%x(1) * ggrav2
gg_grav_dum%psi(3) = p%x(2) * ggrav2
gg_grav_dum%psi(4) = p%x(3) * ggrav2
gg_grav = gr_potf (one, one, ggrav2) - minv * gg_grav_dum
propgrav = (1 / cmplx (p*p - m**2, m*w, kind=default)) * &
(etagrav + pppgrav + (1/3.0_default) * gg_grav)
end function pr_grav
```

AB.27 Polarization vectorspinors

Here we construct the wavefunctions for (massive) gravitinos out of the wavefunctions of (massive) vectorbosons and (massive) Majorana fermions.

$$\psi^\mu_{(u;3/2)}(k) = \epsilon^\mu_+(k)\cdot u(k,+) \tag{AB.138a}$$

$$\psi^\mu_{(u;1/2)}(k) = \sqrt{\frac{1}{3}}\,\epsilon^\mu_+(k)\cdot u(k,-) + \sqrt{\frac{2}{3}}\,\epsilon^\mu_0(k)\cdot u(k,+) \tag{AB.138b}$$

$$\psi^\mu_{(u;-1/2)}(k) = \sqrt{\frac{2}{3}}\,\epsilon^\mu_0(k)\cdot u(k,-) + \sqrt{\frac{1}{3}}\,\epsilon^\mu_-(k)\cdot u(k,+) \tag{AB.138c}$$

$$\psi^\mu_{(u;-3/2)}(k) = \epsilon^\mu_-(k)\cdot u(k,-) \tag{AB.138d}$$

and in the same manner for $\psi^\mu_{(v;s)}$ with u replaced by v and with the conjugated polarization vectors. These gravitino wavefunctions obey the Dirac equation, they are transverse and they fulfill the irreducibility condition

$$\gamma_\mu \psi^\mu_{(u/v;s)} = 0. \tag{AB.139}$$

⟨`omega_vspinor_polarizations.f90`⟩≡

```
  ⟨Copyleft⟩
  module omega_vspinor_polarizations
  use kinds
  use constants
  use omega_vectors
  use omega_bispinors
  use omega_bispinor_couplings
  use omega_vectorspinors
  implicit none
  ⟨Declaration of polarization vectorspinors⟩
  integer, parameter, public :: omega_vspinor_pols_2010_01_A = 0
  contains
  ⟨Implementation of polarization vectorspinors⟩
  end module omega_vspinor_polarizations
```

⟨*Declaration of polarization vectorspinors*⟩≡

```
  public :: ueps, veps
  private :: eps
  private :: outer_product
```

Here we implement the polarization vectors for vectorbosons with trigonometric functions, without the rotating of components done in HELAS [5]. These are only used for generating the polarization vectorspinors.

$$\epsilon^\mu_+(k) = \frac{-e^{+\mathrm{i}\phi}}{\sqrt{2}}\,(0;\cos\theta\cos\phi - \mathrm{i}\sin\phi, \cos\theta\sin\phi + \mathrm{i}\cos\phi, -\sin\theta) \tag{AB.140a}$$

$$\epsilon^\mu_-(k) = \frac{e^{-\mathrm{i}\phi}}{\sqrt{2}} (0; \cos\theta\cos\phi + \mathrm{i}\sin\phi, \cos\theta\sin\phi - \mathrm{i}\cos\phi, -\sin\theta) \tag{AB.140b}$$

$$\epsilon^\mu_0(k) = \frac{1}{m}\left(|\vec{k}|; k^0\sin\theta\cos\phi, k^0\sin\theta\sin\phi, k^0\cos\theta\right) \tag{AB.140c}$$

Determining the mass from the momenta is a numerically haphazardous for light particles. Therefore, we accept some redundancy and pass the mass explicitely. For the case that the momentum lies totally in the z-direction we take the convention $\cos\phi = 1$ and $\sin\phi = 0$.

⟨*Implementation of polarization vectorspinors*⟩≡
```
pure function eps (mass, k, s) result (e)
type(vector) :: e
real(kind=default), intent(in) :: mass
type(momentum), intent(in) :: k
integer, intent(in) :: s
real(kind=default) :: kabs, kabs2, sqrt2, m
real(kind=default) :: cos_phi, sin_phi, cos_th, sin_th
complex(kind=default) :: epiphi, emiphi
sqrt2 = sqrt (2.0_default)
kabs2 = dot_product (k%x, k%x)
m = abs(mass)
if (kabs2 > 0) then
kabs = sqrt (kabs2)
if ((k%x(1) == 0) .and. (k%x(2) == 0)) then
cos_phi = 1
sin_phi = 0
else
cos_phi = k%x(1) / sqrt(k%x(1)**2 + k%x(2)**2)
sin_phi = k%x(2) / sqrt(k%x(1)**2 + k%x(2)**2)
end if
cos_th = k%x(3) / kabs
sin_th = sqrt(1 - cos_th**2)
epiphi = cos_phi + (0,1) * sin_phi
emiphi = cos_phi - (0,1) * sin_phi
e%t = 0
e%x = 0
select case (s)
case (1)
e%x(1) = epiphi * (-cos_th * cos_phi + (0,1) * sin_phi) / sqrt2
e%x(2) = epiphi * (-cos_th * sin_phi - (0,1) * cos_phi) / sqrt2
e%x(3) = epiphi * ( sin_th / sqrt2)
case (-1)
e%x(1) = emiphi * ( cos_th * cos_phi + (0,1) * sin_phi) / sqrt2
e%x(2) = emiphi * ( cos_th * sin_phi - (0,1) * cos_phi) / sqrt2
e%x(3) = emiphi * (-sin_th / sqrt2)
case (0)
if (m > 0) then
e%t = kabs / m
e%x = k%t / (m*kabs) * k%x
end if
case (4)
if (m > 0) then
e = (1 / m) * k
else
e = (1 / k%t) * k
end if
end select
else   !!! for particles in their rest frame defined to be
!!! polarized along the 3-direction
e%t = 0
e%x = 0
select case (s)
case (1)
e%x(1) = cmplx ( - 1,   0, kind=default) / sqrt2
e%x(2) = cmplx (   0,   1, kind=default) / sqrt2
case (-1)
e%x(1) = cmplx (   1,   0, kind=default) / sqrt2
```

```
e%x(2) = cmplx (   0,   1, kind=default) / sqrt2
case (0)
if (m > 0) then
e%x(3) = 1
end if
case (4)
if (m > 0) then
e = (1 / m) * k
else
e = (1 / k%t) * k
end if
end select
end if
end function eps
```

⟨*Implementation of polarization vectorspinors*⟩+≡

```
pure function ueps (m, k, s) result (t)
type(vectorspinor) :: t
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
integer, intent(in) :: s
integer :: i
type(vector) :: ep, e0, em
type(bispinor) :: up, um
do i = 1, 4
t%psi(i)%a = 0
end do
select case (s)
case (2)
ep = eps (m, k, 1)
up = u (m, k, 1)
t = outer_product (ep, up)
case (1)
ep = eps (m, k, 1)
e0 = eps (m, k, 0)
up = u (m, k, 1)
um = u (m, k, -1)
t = (1 / sqrt (3.0_default)) * (outer_product (ep, um) &
+ sqrt (2.0_default) * outer_product (e0, up))
case (-1)
e0 = eps (m, k, 0)
em = eps (m, k, -1)
up = u (m, k, 1)
um = u (m, k, -1)
t = (1 / sqrt (3.0_default)) * (sqrt (2.0_default) * &
outer_product (e0, um) + outer_product (em, up))
case (-2)
em = eps (m, k, -1)
um = u (m, k, -1)
t = outer_product (em, um)
end select
end function ueps
```

⟨*Implementation of polarization vectorspinors*⟩+≡

```
pure function veps (m, k, s) result (t)
type(vectorspinor) :: t
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
integer, intent(in) :: s
integer :: i
type(vector) :: ep, e0, em
type(bispinor) :: vp, vm
do i = 1, 4
t%psi(i)%a = 0
end do
select case (s)
case (2)
```

```
ep = conjg(eps (m, k, 1))
vp = v (m, k, 1)
t = outer_product (ep, vp)
case (1)
ep = conjg(eps (m, k, 1))
e0 = conjg(eps (m, k, 0))
vp = v (m, k, 1)
vm = v (m, k, -1)
t = (1 / sqrt (3.0_default)) * (outer_product (ep, vm) &
+ sqrt (2.0_default) * outer_product (e0, vp))
case (-1)
e0 = conjg(eps (m, k,  0))
em = conjg(eps (m, k, -1))
vp = v (m, k, 1)
vm = v (m, k, -1)
t = (1 / sqrt (3.0_default)) * (sqrt (2.0_default) &
* outer_product (e0, vm) + outer_product (em, vp))
case (-2)
em = conjg(eps (m, k, -1))
vm = v (m, k, -1)
t = outer_product (em, vm)
end select
end function veps
```

⟨*Implementation of polarization vectorspinors*⟩+≡

```
pure function outer_product (ve, sp) result (vs)
type(vectorspinor) :: vs
type(vector), intent(in) :: ve
type(bispinor), intent(in) :: sp
integer :: i
vs%psi(1)%a(1:4) = ve%t * sp%a(1:4)
do i = 1, 3
vs%psi((i+1))%a(1:4) = ve%x(i) * sp%a(1:4)
end do
end function outer_product
```

AB.28 Color

⟨`omega_color.f90`⟩≡

```
⟨Copyleft⟩
module omega_color
use kinds
implicit none
private
⟨Declaration of color types⟩
⟨Declaration of color functions⟩
integer, parameter, public :: omega_color_2010_01_A = 0
contains
⟨Implementation of color functions⟩
end module omega_color
```

AB.28.1 Color Sum

⟨*Declaration of color types*⟩≡

```
public :: omega_color_factor
type omega_color_factor
integer :: i1, i2
real(kind=default) :: factor
end type omega_color_factor
```

⟨*Declaration of color functions*⟩≡

```
public :: omega_color_sum
```

The `!$omp` instruction will result in parallel code if compiled with support for OpenMP otherwise it is ignored.

⟨*Implementation of color functions*⟩≡

```
⟨pure unless OpenMP⟩
```

```
function omega_color_sum (flv, hel, amp, cf) result (amp2)
complex(kind=default) :: amp2
integer, intent(in) :: flv, hel
complex(kind=default), dimension(:,:,:), intent(in) :: amp
type(omega_color_factor), dimension(:), intent(in) :: cf
integer :: n
amp2 = 0
!$omp parallel do reduction(+:amp2)
do n = 1, size (cf)
amp2 = amp2 + cf(n)%factor * &
amp(flv,cf(n)%i1,hel) * conjg (amp(flv,cf(n)%i2,hel))
end do
!$omp end parallel do
end function omega_color_sum
```

In the bytecode for the OVM, we only save the symmetric part of the color factor table. This almost halves the size of n gluon amplitudes for $n > 6$. For $2 \to (5,6)\,g$ the reduced color factor table still amounts for $\sim (75,93)\%$ of the bytecode, making it desirable to omit it completely by computing it dynamically to reduce memory requirements. Note that $2\text{Re}(A_{i_1}A^*_{i_2}) = A_{i_1}A^*_{i_2} + A_{i_2}A^*_{i_1}$.

⟨*Declaration of color functions*⟩+≡
```
public :: ovm_color_sum
```

⟨*Implementation of color functions*⟩+≡
```
⟨pure unless OpenMP⟩
function ovm_color_sum (flv, hel, amp, cf) result (amp2)
real(kind=default) :: amp2
integer, intent(in) :: flv, hel
complex(kind=default), dimension(:,:,:), intent(in) :: amp
type(omega_color_factor), dimension(:), intent(in) :: cf
integer :: n
amp2 = 0
!$omp parallel do reduction(+:amp2)
do n = 1, size (cf)
if (cf(n)%i1 == cf(n)%i2) then
amp2 = amp2 + cf(n)%factor * &
real(amp(flv,cf(n)%i1,hel) * conjg(amp(flv,cf(n)%i2,hel)))
else
amp2 = amp2 + cf(n)%factor * 2 * &
real(amp(flv,cf(n)%i1,hel) * conjg(amp(flv,cf(n)%i2,hel)))
end if
end do
!$omp end parallel do
end function ovm_color_sum
```

AB.29 Utilities

⟨omega_utils.f90⟩≡
```
⟨Copyleft⟩
module omega_utils
use kinds
use omega_vectors
use omega_polarizations
implicit none
private
⟨Declaration of utility functions⟩
⟨Numerical tolerances⟩
integer, parameter, public :: omega_utils_2010_01_A = 0
contains
⟨Implementation of utility functions⟩
end module omega_utils
```

AB.29.1 Helicity Selection Rule Heuristics

⟨*Declaration of utility functions*⟩≡
```
public :: omega_update_helicity_selection
```

⟨*Implementation of utility functions*⟩≡
```
pure subroutine omega_update_helicity_selection &
(count, amp, max_abs, sum_abs, mask, threshold, cutoff, mask_dirty)
integer, intent(inout) :: count
complex(kind=default), dimension(:,:,:), intent(in) :: amp
real(kind=default), dimension(:), intent(inout) :: max_abs
real(kind=default), intent(inout) :: sum_abs
logical, dimension(:), intent(inout) :: mask
real(kind=default), intent(in) :: threshold
integer, intent(in) :: cutoff
logical, intent(out) :: mask_dirty
integer :: h
real(kind=default) :: avg
mask_dirty = .false.
if (threshold > 0) then
count = count + 1
if (count <= cutoff) then
forall (h = lbound (amp, 3) : ubound (amp, 3))
max_abs(h) = max (max_abs(h), maxval (abs (amp(:,:,h))))
end forall
sum_abs = sum_abs + sum (abs (amp))
if (count == cutoff) then
avg = sum_abs / size (amp) / cutoff
mask = max_abs >= threshold * epsilon (avg) * avg
mask_dirty = .true.
end if
end if
end if
end subroutine omega_update_helicity_selection
```

AB.29.2 Diagnostics

⟨*Declaration of utility functions*⟩+≡
```
public :: omega_report_helicity_selection
```
We shoul try to use `msg_message` from WHIZARD's `diagnostics` module, but this would spoil independent builds.

⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_report_helicity_selection (mask, spin_states, threshold, unit)
logical, dimension(:), intent(in) :: mask
integer, dimension(:,:), intent(in) :: spin_states
real(kind=default), intent(in) :: threshold
integer, intent(in), optional :: unit
integer :: u
integer :: h, i
if (present(unit)) then
u = unit
else
u = 6
end if
if (u >= 0) then
write (unit = u, &
fmt = "('| ','Contributing Helicity Combinations: ', I5, ' of ', I5)") &
count (mask), size (mask)
write (unit = u, &
fmt = "('| ','Threshold: amp / avg > ', E9.2, ' = ', E9.2, ' * epsilon()')") &
threshold * epsilon (threshold), threshold
i = 0
do h = 1, size (mask)
if (mask(h)) then
i = i + 1
write (unit = u, fmt = "('| ',I4,': ',20I4)") i, spin_states (:, h)
end if
end do
end if
end subroutine omega_report_helicity_selection
```

⟨*Declaration of utility functions*⟩+≡
```
public :: omega_ward_warn, omega_ward_panic
```
The O'Mega amplitudes have only one particle off shell and are the sum of *all* possible diagrams with the other particles on-shell.

The problem with these gauge checks is that are numerically very small amplitudes that vanish analytically and that violate transversality. The hard part is to determine the thresholds that make threse tests usable.

⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_ward_warn (name, m, k, e)
character(len=*), intent(in) :: name
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
type(vector), intent(in) :: e
type(vector) :: ek
real(kind=default) :: abs_eke, abs_ek_abs_e
ek = eps (m, k, 4)
abs_eke = abs (ek * e)
abs_ek_abs_e = abs (ek) * abs (e)
print *, name, ":", abs_eke / abs_ek_abs_e, abs (ek), abs (e)
if (abs_eke > 1000 * epsilon (abs_ek_abs_e)) then
print *, "O'Mega: warning: non-transverse vector field: ", &
name, ":", abs_eke / abs_ek_abs_e, abs (e)
end if
end subroutine omega_ward_warn
```
⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_ward_panic (name, m, k, e)
character(len=*), intent(in) :: name
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
type(vector), intent(in) :: e
type(vector) :: ek
real(kind=default) :: abs_eke, abs_ek_abs_e
ek = eps (m, k, 4)
abs_eke = abs (ek * e)
abs_ek_abs_e = abs (ek) * abs (e)
if (abs_eke > 1000 * epsilon (abs_ek_abs_e)) then
print *, "O'Mega: panic: non-transverse vector field: ", &
name, ":", abs_eke / abs_ek_abs_e, abs (e)
stop
end if
end subroutine omega_ward_panic
```
⟨*Declaration of utility functions*⟩+≡
```
public :: omega_slavnov_warn, omega_slavnov_panic
```
⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_slavnov_warn (name, m, k, e, phi)
character(len=*), intent(in) :: name
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
type(vector), intent(in) :: e
complex(kind=default), intent(in) :: phi
type(vector) :: ek
real(kind=default) :: abs_eke, abs_ek_abs_e
ek = eps (m, k, 4)
abs_eke = abs (ek * e - phi)
abs_ek_abs_e = abs (ek) * abs (e)
print *, name, ":", abs_eke / abs_ek_abs_e, abs (ek), abs (e)
if (abs_eke > 1000 * epsilon (abs_ek_abs_e)) then
print *, "O'Mega: warning: non-transverse vector field: ", &
name, ":", abs_eke / abs_ek_abs_e, abs (e)
end if
end subroutine omega_slavnov_warn
```
⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_slavnov_panic (name, m, k, e, phi)
```

```
character(len=*), intent(in) :: name
real(kind=default), intent(in) :: m
type(momentum), intent(in) :: k
type(vector), intent(in) :: e
complex(kind=default), intent(in) :: phi
type(vector) :: ek
real(kind=default) :: abs_eke, abs_ek_abs_e
ek = eps (m, k, 4)
abs_eke = abs (ek * e - phi)
abs_ek_abs_e = abs (ek) * abs (e)
if (abs_eke > 1000 * epsilon (abs_ek_abs_e)) then
print *, "O'Mega: panic: non-transverse vector field: ", &
name, ":", abs_eke / abs_ek_abs_e, abs (e)
stop
end if
end subroutine omega_slavnov_panic
```

⟨*Declaration of utility functions*⟩+≡

```
public :: omega_check_arguments_warn, omega_check_arguments_panic
```

⟨*Implementation of utility functions*⟩+≡

```
subroutine omega_check_arguments_warn (n, k)
integer, intent(in) :: n
real(kind=default), dimension(0:,:), intent(in) :: k
integer :: i
i = size(k,dim=1)
if (i /= 4) then
print *, "O'Mega: warning: wrong # of dimensions:", i
end if
i = size(k,dim=2)
if (i /= n) then
print *, "O'Mega: warning: wrong # of momenta:", i, &
", expected", n
end if
end subroutine omega_check_arguments_warn
```

⟨*Implementation of utility functions*⟩+≡

```
subroutine omega_check_arguments_panic (n, k)
integer, intent(in) :: n
real(kind=default), dimension(0:,:), intent(in) :: k
logical :: error
integer :: i
error = .false.
i = size(k,dim=1)
if (i /= n) then
print *, "O'Mega: warning: wrong # of dimensions:", i
error = .true.
end if
i = size(k,dim=2)
if (i /= n) then
print *, "O'Mega: warning: wrong # of momenta:", i, &
", expected", n
error = .true.
end if
if (error) then
stop
end if
end subroutine omega_check_arguments_panic
```

⟨*Declaration of utility functions*⟩+≡

```
public :: omega_check_helicities_warn, omega_check_helicities_panic
private :: omega_check_helicity
```

⟨*Implementation of utility functions*⟩+≡

```
function omega_check_helicity (m, smax, s) result (error)
real(kind=default), intent(in) :: m
integer, intent(in) :: smax, s
logical :: error
select case (smax)
```

```
case (0)
error = (s /= 0)
case (1)
error = (abs (s) /= 1)
case (2)
if (m == 0.0_default) then
error = .not. (abs (s) == 1 .or. abs (s) == 4)
else
error = .not. (abs (s) <= 1 .or. abs (s) == 4)
end if
case (4)
error = .true.
case default
error = .true.
end select
end function omega_check_helicity
```

⟨*Implementation of utility functions*⟩+≡

```
subroutine omega_check_helicities_warn (m, smax, s)
real(kind=default), dimension(:), intent(in) :: m
integer, dimension(:), intent(in) :: smax, s
integer :: i
do i = 1, size (m)
if (omega_check_helicity (m(i), smax(i), s(i))) then
print *, "O'Mega: warning: invalid helicity", s(i)
end if
end do
end subroutine omega_check_helicities_warn
```

⟨*Implementation of utility functions*⟩+≡

```
subroutine omega_check_helicities_panic (m, smax, s)
real(kind=default), dimension(:), intent(in) :: m
integer, dimension(:), intent(in) :: smax, s
logical :: error
logical :: error1
integer :: i
error = .false.
do i = 1, size (m)
error1 = omega_check_helicity (m(i), smax(i), s(i))
if (error1) then
print *, "O'Mega: panic: invalid helicity", s(i)
error = .true.
end if
end do
if (error) then
stop
end if
end subroutine omega_check_helicities_panic
```

⟨*Declaration of utility functions*⟩+≡

```
public :: omega_check_momenta_warn, omega_check_momenta_panic
private :: check_momentum_conservation, check_mass_shell
```

⟨*Numerical tolerances*⟩≡

```
integer, parameter, private :: MOMENTUM_TOLERANCE = 10000
```

⟨*Implementation of utility functions*⟩+≡

```
function check_momentum_conservation (k) result (error)
real(kind=default), dimension(0:,:), intent(in) :: k
logical :: error
error = any (abs (sum (k(:,3:), dim = 2) - k(:,1) - k(:,2)) > &
MOMENTUM_TOLERANCE * epsilon (maxval (abs (k), dim = 2)))
if (error) then
print *, sum (k(:,3:), dim = 2) - k(:,1) - k(:,2)
print *, MOMENTUM_TOLERANCE * epsilon (maxval (abs (k), dim = 2)), &
maxval (abs (k), dim = 2)
end if
end function check_momentum_conservation
```

⟨*Numerical tolerances*⟩+≡
```
integer, parameter, private :: ON_SHELL_TOLERANCE = 1000000
```

⟨*Implementation of utility functions*⟩+≡
```
function check_mass_shell (m, k) result (error)
real(kind=default), intent(in) :: m
real(kind=default), dimension(0:), intent(in) :: k
real(kind=default) :: e2
logical :: error
e2 = k(1)**2 + k(2)**2 + k(3)**2 + m**2
error = abs (k(0)**2 - e2) > ON_SHELL_TOLERANCE * epsilon (max (k(0)**2, e2))
if (error) then
print *, k(0)**2 - e2
print *, ON_SHELL_TOLERANCE * epsilon (max (k(0)**2, e2)), max (k(0)**2, e2)
end if
end function check_mass_shell
```

⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_check_momenta_warn (m, k)
real(kind=default), dimension(:), intent(in) :: m
real(kind=default), dimension(0:,:), intent(in) :: k
integer :: i
if (check_momentum_conservation (k)) then
print *, "O'Mega: warning: momentum not conserved"
end if
do i = 1, size(m)
if (check_mass_shell (m(i), k(:,i))) then
print *, "O'Mega: warning: particle #", i, "not on-shell"
end if
end do
end subroutine omega_check_momenta_warn
```

⟨*Implementation of utility functions*⟩+≡
```
subroutine omega_check_momenta_panic (m, k)
real(kind=default), dimension(:), intent(in) :: m
real(kind=default), dimension(0:,:), intent(in) :: k
logical :: error
logical :: error1
integer :: i
error = check_momentum_conservation (k)
if (error) then
print *, "O'Mega: panic: momentum not conserved"
end if
do i = 1, size(m)
error1 = check_mass_shell (m(i), k(0:,i))
if (error1) then
print *, "O'Mega: panic: particle #", i, "not on-shell"
error = .true.
end if
end do
if (error) then
stop
end if
end subroutine omega_check_momenta_panic
```

AB.29.3 Obsolete Summation

Spin/Helicity Summation

⟨*Declaration of obsolete utility functions*⟩≡
```
public :: omega_sum, omega_sum_nonzero, omega_nonzero
private :: state_index
```

⟨*Implementation of obsolete utility functions*⟩≡
```
pure function omega_sum (omega, p, states, fixed) result (sigma)
real(kind=default) :: sigma
real(kind=default), dimension(0:,:), intent(in) :: p
integer, dimension(:), intent(in), optional :: states, fixed
```

```
⟨interface for O'Mega Amplitude⟩
integer, dimension(size(p,dim=2)) :: s, nstates
integer :: j
complex(kind=default) :: a
if (present (states)) then
nstates = states
else
nstates = 2
end if
sigma = 0
s = -1
sum_spins: do
if (present (fixed)) then
!!! print *, 's = ', s, ', fixed = ', fixed, ', nstates = ', nstates, &
!!!    ', fixed|s = ', merge (fixed, s, mask = nstates == 0)
a = omega (p, merge (fixed, s, mask = nstates == 0))
else
a = omega (p, s)
end if
sigma = sigma + a * conjg(a)
⟨Step s like a n-ary number and terminate when all (s == -1)⟩
end do sum_spins
sigma = sigma / num_states (2, nstates(1:2))
end function omega_sum
```

We're looping over all spins like a n-ary numbers $(-1,\ldots,-1,-1)$, $(-1,\ldots,-1,0)$, $(-1,\ldots,-1,1)$, $(-1,\ldots,0,-1)$, $\ldots$, $(1,\ldots,1,0)$, $(1,\ldots,1,1)$:

⟨*Step* `s` *like a n-ary number and terminate when* `all (s == -1)`⟩≡

```
do j = size (p, dim = 2), 1, -1
select case (nstates (j))
case (3) ! massive vectors
s(j) = modulo (s(j) + 2, 3) - 1
case (2) ! spinors, massless vectors
s(j) = - s(j)
case (1) ! scalars
s(j) = -1
case (0) ! fized spin
s(j) = -1
case default ! ???
s(j) = -1
end select
if (s(j) /= -1) then
cycle sum_spins
end if
end do
exit sum_spins
```

The dual operation evaluates an n-number:

⟨*Implementation of obsolete utility functions*⟩+≡

```
pure function state_index (s, states) result (n)
integer, dimension(:), intent(in) :: s
integer, dimension(:), intent(in), optional :: states
integer :: n
integer :: j, p
n = 1
p = 1
if (present (states)) then
do j = size (s), 1, -1
select case (states(j))
case (3)
n = n + p * (s(j) + 1)
case (2)
n = n + p * (s(j) + 1) / 2
end select
p = p * states(j)
end do
else
```

```
do j = size (s), 1, -1
n = n + p * (s(j) + 1) / 2
p = p * 2
end do
end if
end function state_index
```

⟨interface *for O'Mega Amplitude*⟩≡

```
interface
pure function omega (p, s) result (me)
use kinds
implicit none
complex(kind=default) :: me
real(kind=default), dimension(0:,:), intent(in) :: p
integer, dimension(:), intent(in) :: s
end function omega
end interface
```

⟨*Declaration of obsolete utility functions*⟩+≡

```
public :: num_states
```

⟨*Implementation of obsolete utility functions*⟩+≡

```
pure function num_states (n, states) result (ns)
integer, intent(in) :: n
integer, dimension(:), intent(in), optional :: states
integer :: ns
if (present (states)) then
ns = product (states, mask = states == 2 .or. states == 3)
else
ns = 2**n
end if
end function num_states
```

AB.30 omega95

⟨omega95.f90⟩≡

```
⟨Copyleft⟩
module omega95
use constants
use omega_spinors
use omega_vectors
use omega_polarizations
use omega_tensors
use omega_tensor_polarizations
use omega_couplings
use omega_spinor_couplings
use omega_color
use omega_utils
public
end module omega95
```

AB.31 omega95 *Revisited*

⟨omega95_bispinors.f90⟩≡

```
⟨Copyleft⟩
module omega95_bispinors
use constants
use omega_bispinors
use omega_vectors
use omega_vectorspinors
use omega_polarizations
use omega_vspinor_polarizations
use omega_couplings
use omega_bispinor_couplings
use omega_color
use omega_utils
public
end module omega95_bispinors
```

AB.32 Testing

⟨omega_testtools.f90⟩≡
  ⟨*Copyleft*⟩
```
  module omega_testtools
  use kinds
  implicit none
  private
  real(kind=default), parameter, private :: ABS_THRESHOLD_DEFAULT = 1E-17
  real(kind=default), parameter, private :: THRESHOLD_DEFAULT = 0.6
  real(kind=default), parameter, private :: THRESHOLD_WARN = 0.8
```
  ⟨*Declaration of test support functions*⟩
```
  contains
```
  ⟨*Implementation of test support functions*⟩
```
  end module omega_testtools
```

Quantify the agreement of two real or complex numbers

$$\text{agreement}(x,y) = \frac{\ln \Delta(x,y)}{\ln \epsilon} \in [0,1] \tag{AB.141}$$

with

$$\Delta(x,y) = \frac{|x-y|}{\max(|x|,|y|)} \tag{AB.142}$$

and values outside $[0,1]$ replaced the closed value in the interval. In other words

- 1 for $x - y = \max(|x|,|y|) \cdot \mathcal{O}(\epsilon)$ and
- 0 for $x - y = \max(|x|,|y|) \cdot \mathcal{O}(1)$

with logarithmic interpolation. The cases $x = 0$ and $y = 0$ must be treated separately.

⟨*Declaration of test support functions*⟩≡
```
  public :: agreement
  interface agreement
  module procedure agreement_real, agreement_complex, &
  agreement_real_complex, agreement_complex_real, &
  agreement_integer_complex, agreement_complex_integer, &
  agreement_integer_real, agreement_real_integer
  end interface
  private :: agreement_real, agreement_complex, &
  agreement_real_complex, agreement_complex_real, &
  agreement_integer_complex, agreement_complex_integer, &
  agreement_integer_real, agreement_real_integer
```

⟨*Implementation of test support functions*⟩≡
```
  elemental function agreement_real (x, y, base) result (a)
  real(kind=default) :: a
  real(kind=default), intent(in) :: x, y
  real(kind=default), intent(in), optional :: base
  real(kind=default) :: scale, dxy
  if (present (base)) then
  scale = max (abs (x), abs (y), abs (base))
  else
  scale = max (abs (x), abs (y))
  end if
  if (ieee_is_nan (x) .or. ieee_is_nan (y)) then
  a = 0
  else if (scale <= 0) then
  a = -1
  else
  dxy = abs (x - y) / scale
  if (dxy <= 0.0_default) then
  a = 1
  else
  a = log (dxy) / log (epsilon (scale))
  a = max (0.0_default, min (1.0_default, a))
  if (ieee_is_nan (a)) then
  a = 0
  end if
  end if
  end if
  if (ieee_is_nan (a)) then
  a = 0
  end if
  end function agreement_real
```

Poor man's replacement

⟨*Implementation of test support functions*⟩+≡

```
  elemental function ieee_is_nan (x) result (yorn)
  logical :: yorn
  real (kind=default), intent(in) :: x
  yorn = (x /= x)
  end function ieee_is_nan
```

⟨*Implementation of test support functions*⟩+≡

```
  elemental function agreement_complex (x, y, base) result (a)
  real(kind=default) :: a
  complex(kind=default), intent(in) :: x, y
  real(kind=default), intent(in), optional :: base
  real(kind=default) :: scale, dxy
  if (present (base)) then
  scale = max (abs (x), abs (y), abs (base))
  else
  scale = max (abs (x), abs (y))
  end if
  if (      ieee_is_nan (real (x, kind=default)) .or. ieee_is_nan (aimag (x)) &
  .or. ieee_is_nan (real (y, kind=default)) .or. ieee_is_nan (aimag (y))) then
  a = 0
  else if (scale <= 0) then
  a = -1
  else
  dxy = abs (x - y) / scale
  if (dxy <= 0.0_default) then
  a = 1
  else
  a = log (dxy) / log (epsilon (scale))
  a = max (0.0_default, min (1.0_default, a))
  if (ieee_is_nan (a)) then
  a = 0
  end if
  end if
  end if
  if (ieee_is_nan (a)) then
  a = 0
  end if
  end function agreement_complex
```

⟨*Implementation of test support functions*⟩+≡

```
  elemental function agreement_real_complex (x, y, base) result (a)
  real(kind=default) :: a
  real(kind=default), intent(in) :: x
  complex(kind=default), intent(in) :: y
  real(kind=default), intent(in), optional :: base
  a = agreement_complex (cmplx (x, kind=default), y, base)
  end function agreement_real_complex
```

⟨*Implementation of test support functions*⟩+≡

```
  elemental function agreement_complex_real (x, y, base) result (a)
  real(kind=default) :: a
  complex(kind=default), intent(in) :: x
```

```
real(kind=default), intent(in) :: y
real(kind=default), intent(in), optional :: base
a = agreement_complex (x, cmplx (y, kind=default), base)
end function agreement_complex_real
```

⟨*Implementation of test support functions*⟩+≡
```
elemental function agreement_integer_complex (x, y, base) result (a)
real(kind=default) :: a
integer, intent(in) :: x
complex(kind=default), intent(in) :: y
real(kind=default), intent(in), optional :: base
a = agreement_complex (cmplx (x, kind=default), y, base)
end function agreement_integer_complex
```

⟨*Implementation of test support functions*⟩+≡
```
elemental function agreement_complex_integer (x, y, base) result (a)
real(kind=default) :: a
complex(kind=default), intent(in) :: x
integer, intent(in) :: y
real(kind=default), intent(in), optional :: base
a = agreement_complex (x, cmplx (y, kind=default), base)
end function agreement_complex_integer
```

⟨*Implementation of test support functions*⟩+≡
```
elemental function agreement_integer_real (x, y, base) result (a)
real(kind=default) :: a
integer, intent(in) :: x
real(kind=default), intent(in) :: y
real(kind=default), intent(in), optional :: base
a = agreement_real (real(x, kind=default), y, base)
end function agreement_integer_real
```

⟨*Implementation of test support functions*⟩+≡
```
elemental function agreement_real_integer (x, y, base) result (a)
real(kind=default) :: a
real(kind=default), intent(in) :: x
integer, intent(in) :: y
real(kind=default), intent(in), optional :: base
a = agreement_real (x, real (y, kind=default), base)
end function agreement_real_integer
```

⟨*Declaration of test support functions*⟩+≡
```
public:: vanishes
interface vanishes
module procedure vanishes_real, vanishes_complex
end interface
private :: vanishes_real, vanishes_complex
```

⟨*Implementation of test support functions*⟩+≡
```
elemental function vanishes_real (x, scale) result (a)
real(kind=default) :: a
real(kind=default), intent(in) :: x
real(kind=default), intent(in), optional :: scale
real(kind=default) :: scaled_x
if (x == 0.0_default) then
a = 1
return
else if (ieee_is_nan (x)) then
a = 0
return
end if
scaled_x = x
if (present (scale)) then
if (scale /= 0) then
scaled_x = x / abs (scale)
else
a = 0
return
end if
```

```
else
end if
a = log (abs (scaled_x)) / log (epsilon (scaled_x))
a = max (0.0_default, min (1.0_default, a))
if (ieee_is_nan (a)) then
a = 0
end if
end function vanishes_real
```

⟨*Implementation of test support functions*⟩+≡

```
elemental function vanishes_complex (x, scale) result (a)
real(kind=default) :: a
complex(kind=default), intent(in) :: x
real(kind=default), intent(in), optional :: scale
a = vanishes_real (abs (x), scale)
end function vanishes_complex
```

⟨*Declaration of test support functions*⟩+≡

```
public :: expect
interface expect
module procedure expect_integer, expect_real, expect_complex, &
expect_real_integer, expect_integer_real, &
expect_complex_integer, expect_integer_complex, &
expect_complex_real, expect_real_complex
end interface
private :: expect_integer, expect_real, expect_complex, &
expect_real_integer, expect_integer_real, &
expect_complex_integer, expect_integer_complex, &
expect_complex_real, expect_real_complex
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_integer (x, x0, msg, passed, quiet, buffer, unit)
integer, intent(in) :: x, x0
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
logical, intent(in), optional :: quiet
character(len=*), intent(inout), optional :: buffer
integer, intent(in), optional :: unit
logical :: failed, verbose
character(len=*), parameter :: fmt = "(1X,A,': ',A)"
character(len=*), parameter :: &
fmt_verbose = "(1X,A,': ',A,' [expected ',I6,', got ',I6,']')"
failed = .false.
verbose = .true.
if (present (quiet)) then
verbose = .not.quiet
end if
if (x == x0) then
if (verbose) then
if (.not. (present (buffer) .or. present (unit))) then
write (unit = *, fmt = fmt) msg, "passed"
end if
if (present (unit)) then
write (unit = unit, fmt = fmt) msg, "passed"
end if
if (present (buffer)) then
write (unit = buffer, fmt = fmt) msg, "passed"
end if
end if
else
if (.not. (present (buffer) .or. present (unit))) then
write (unit = *, fmt = fmt_verbose) msg, "failed", x0, x
end if
if (present (unit)) then
write (unit = unit, fmt = fmt_verbose) msg, "failed", x0, x
end if
if (present (buffer)) then
write (unit = buffer, fmt = fmt_verbose) msg, "failed", x0, x
```

```
end if
failed = .true.
end if
if (present (passed)) then
passed = passed .and. .not.failed
end if
end subroutine expect_integer
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_real (x, x0, msg, passed, threshold, quiet, abs_threshold)
real(kind=default), intent(in) :: x, x0
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
real(kind=default), intent(in), optional :: threshold
real(kind=default), intent(in), optional :: abs_threshold
logical, intent(in), optional :: quiet
logical :: failed, verbose
real(kind=default) :: agreement_threshold, abs_agreement_threshold
character(len=*), parameter :: fmt = "(1X,A,': ',A,' at ',I4,'%')"
character(len=*), parameter :: fmt_verbose = "(1X,A,': ',A,' at ',I4,'%'," // &
"' [expected ',E10.3,', got ',E10.3,']')"
real(kind=default) :: a
failed = .false.
verbose = .true.
if (present (quiet)) then
verbose = .not.quiet
end if
if (x == x0) then
if (verbose) then
write (unit = *, fmt = fmt) msg, "passed", 100
end if
else
if (x0 == 0) then
a = vanishes (x)
else
a = agreement (x, x0)
end if
if (present (threshold)) then
agreement_threshold = threshold
else
agreement_threshold = THRESHOLD_DEFAULT
end if
if (present (abs_threshold)) then
abs_agreement_threshold = abs_threshold
else
abs_agreement_threshold = ABS_THRESHOLD_DEFAULT
end if
if (a >= agreement_threshold .or. &
max(abs(x), abs(x0)) <= abs_agreement_threshold) then
if (verbose) then
if (a >= THRESHOLD_WARN) then
write (unit = *, fmt = fmt) msg, "passed", int (a * 100)
else
write (unit = *, fmt = fmt_verbose) msg, "passed", int (a * 100), x0, x
end if
end if
else
failed = .true.
write (unit = *, fmt = fmt_verbose) msg, "failed", int (a * 100), x0, x
end if
end if
if (present (passed)) then
passed = passed .and. .not. failed
end if
end subroutine expect_real
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_complex (x, x0, msg, passed, threshold, quiet, abs_threshold)
complex(kind=default), intent(in) :: x, x0
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
real(kind=default), intent(in), optional :: threshold
real(kind=default), intent(in), optional :: abs_threshold
logical, intent(in), optional :: quiet
logical :: failed, verbose
real(kind=default) :: agreement_threshold, abs_agreement_threshold
character(len=*), parameter :: fmt = "(1X,A,': ',A,' at ',I4,'%')"
character(len=*), parameter :: fmt_verbose = "(1X,A,': ',A,' at ',I4,'%'," // &
"' [expected (',E10.3,',',E10.3,'), got (',E10.3,',',E10.3,')]')"
character(len=*), parameter :: fmt_phase = "(1X,A,': ',A,' at ',I4,'%'," // &
"' [modulus passed at ',I4,'%',', phases ',F5.3,' vs. ',F5.3,']')"
real(kind=default) :: a, a_modulus
failed = .false.
verbose = .true.
if (present (quiet)) then
verbose = .not.quiet
end if
if (x == x0) then
if (verbose) then
write (unit = *, fmt = fmt) msg, "passed", 100
end if
else
if (x0 == 0) then
a = vanishes (x)
else
a = agreement (x, x0)
end if
if (present (threshold)) then
agreement_threshold = threshold
else
agreement_threshold = THRESHOLD_DEFAULT
end if
if (present (abs_threshold)) then
abs_agreement_threshold = abs_threshold
else
abs_agreement_threshold = ABS_THRESHOLD_DEFAULT
end if
if (a >= agreement_threshold .or. &
max(abs(x), abs(x0)) <= abs_agreement_threshold) then
if (verbose) then
if (a >= THRESHOLD_WARN) then
write (unit = *, fmt = fmt) msg, "passed", int (a * 100)
else
write (unit = *, fmt = fmt_verbose) msg, "passed", int (a * 100), x0, x
end if
end if
else
a_modulus = agreement (abs (x), abs (x0))
if (a_modulus >= agreement_threshold) then
write (unit = *, fmt = fmt_phase) msg, "failed", int (a * 100), &
int (a_modulus * 100), &
atan2 (real (x, kind=default), aimag (x)), &
atan2 (real (x0, kind=default), aimag (x0))
else
write (unit = *, fmt = fmt_verbose) msg, "failed", int (a * 100), x0, x
end if
failed = .true.
end if
end if
if (present (passed)) then
passed = passed .and. .not.failed
end if
end subroutine expect_complex
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_real_integer (x, x0, msg, passed, threshold, quiet)
  real(kind=default), intent(in) :: x
  integer, intent(in) :: x0
  character(len=*), intent(in) :: msg
  real(kind=default), intent(in), optional :: threshold
  logical, intent(inout), optional :: passed
  logical, intent(in), optional :: quiet
  call expect_real (x, real (x0, kind=default), msg, passed, threshold, quiet)
  end subroutine expect_real_integer
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_integer_real (x, x0, msg, passed, threshold, quiet)
  integer, intent(in) :: x
  real(kind=default), intent(in) :: x0
  character(len=*), intent(in) :: msg
  real(kind=default), intent(in), optional :: threshold
  logical, intent(inout), optional :: passed
  logical, intent(in), optional :: quiet
  call expect_real (real (x, kind=default), x0, msg, passed, threshold, quiet)
  end subroutine expect_integer_real
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_complex_integer (x, x0, msg, passed, threshold, quiet)
  complex(kind=default), intent(in) :: x
  integer, intent(in) :: x0
  character(len=*), intent(in) :: msg
  logical, intent(inout), optional :: passed
  real(kind=default), intent(in), optional :: threshold
  logical, intent(in), optional :: quiet
  call expect_complex (x, cmplx (x0, kind=default), msg, passed, threshold, quiet)
  end subroutine expect_complex_integer
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_integer_complex (x, x0, msg, passed, threshold, quiet)
  integer, intent(in) :: x
  complex(kind=default), intent(in) :: x0
  character(len=*), intent(in) :: msg
  logical, intent(inout), optional :: passed
  real(kind=default), intent(in), optional :: threshold
  logical, intent(in), optional :: quiet
  call expect_complex (cmplx (x, kind=default), x0, msg, passed, threshold, quiet)
  end subroutine expect_integer_complex
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_complex_real (x, x0, msg, passed, threshold, quiet)
  complex(kind=default), intent(in) :: x
  real(kind=default), intent(in) :: x0
  character(len=*), intent(in) :: msg
  logical, intent(inout), optional :: passed
  real(kind=default), intent(in), optional :: threshold
  logical, intent(in), optional :: quiet
  call expect_complex (x, cmplx (x0, kind=default), msg, passed, threshold, quiet)
  end subroutine expect_complex_real
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine expect_real_complex (x, x0, msg, passed, threshold, quiet)
  real(kind=default), intent(in) :: x
  complex(kind=default), intent(in) :: x0
  character(len=*), intent(in) :: msg
  logical, intent(inout), optional :: passed
  real(kind=default), intent(in), optional :: threshold
  logical, intent(in), optional :: quiet
  call expect_complex (cmplx (x, kind=default), x0, msg, passed, threshold, quiet)
  end subroutine expect_real_complex
```

⟨*Declaration of test support functions*⟩+≡
```
  public :: expect_zero
  interface expect_zero
```

```
module procedure expect_zero_integer, expect_zero_real, expect_zero_complex
end interface
private :: expect_zero_integer, expect_zero_real, expect_zero_complex
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_zero_integer (x, msg, passed)
integer, intent(in) :: x
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
call expect_integer (x, 0, msg, passed)
end subroutine expect_zero_integer
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_zero_real (x, scale, msg, passed, threshold, quiet)
real(kind=default), intent(in) :: x, scale
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
real(kind=default), intent(in), optional :: threshold
logical, intent(in), optional :: quiet
logical :: failed, verbose
real(kind=default) :: agreement_threshold
character(len=*), parameter :: fmt = "(1X,A,': ',A,' at ',I4,'%')"
character(len=*), parameter :: fmt_verbose = "(1X,A,': ',A,' at ',I4,'%'," // &
"' [expected 0 (relative to ',E10.3,') got ',E10.3,']')"
real(kind=default) :: a
failed = .false.
verbose = .true.
if (present (quiet)) then
verbose = .not.quiet
end if
if (x == 0) then
if (verbose) then
write (unit = *, fmt = fmt) msg, "passed", 100
end if
else
a = vanishes (x, scale = scale)
if (present (threshold)) then
agreement_threshold = threshold
else
agreement_threshold = THRESHOLD_DEFAULT
end if
if (a >= agreement_threshold) then
if (verbose) then
if (a >= THRESHOLD_WARN) then
write (unit = *, fmt = fmt) msg, "passed", int (a * 100)
else
write (unit = *, fmt = fmt_verbose) msg, "passed", int (a * 100), scale, x
end if
end if
else
failed = .true.
write (unit = *, fmt = fmt_verbose) msg, "failed", int (a * 100), scale, x
end if
end if
if (present (passed)) then
passed = passed .and. .not.failed
end if
end subroutine expect_zero_real
```

⟨*Implementation of test support functions*⟩+≡

```
subroutine expect_zero_complex (x, scale, msg, passed, threshold, quiet)
complex(kind=default), intent(in) :: x
real(kind=default), intent(in) :: scale
character(len=*), intent(in) :: msg
logical, intent(inout), optional :: passed
real(kind=default), intent(in), optional :: threshold
logical, intent(in), optional :: quiet
call expect_zero_real (abs (x), scale, msg, passed, threshold, quiet)
```

```
  end subroutine expect_zero_complex
```

⟨*Implementation of test support functions*⟩+≡
```
  subroutine print_matrix (a)
  complex(kind=default), dimension(:,:), intent(in) :: a
  integer :: row
  do row = 1, size (a, dim=1)
  write (unit = *, fmt = "(10(tr2, f5.2, '+', f5.2, 'I'))") a(row,:)
  end do
  end subroutine print_matrix
```

⟨*Declaration of test support functions*⟩+≡
```
  public :: print_matrix
```

⟨`test_omega95.f90`⟩≡
```
  ⟨Copyleft⟩
  program test_omega95
  use kinds
  use omega95
  use omega_testtools
  implicit none
  real(kind=default) :: m, pabs, qabs, w
  real(kind=default), dimension(0:3) :: r
  complex(kind=default) :: c_one, c_nil
  type(momentum) :: p, q, p0
  type(vector) :: vp, vq, vtest, v0
  type(tensor) :: ttest
  type(spinor) :: test_psi, test_spinor1, test_spinor2
  type(conjspinor) :: test_psibar, test_conjspinor1, test_conjspinor2
  integer, dimension(8) :: date_time
  integer :: rsize, i
  logical :: passed
  call date_and_time (values = date_time)
  call random_seed (size = rsize)
  call random_seed (put = spread (product (date_time), dim = 1, ncopies = rsize))
  w = 1.4142
  c_one = 1.0_default
  c_nil = 0.0_default
  m = 13
  pabs = 42
  qabs = 137
  call random_number (r)
  vtest%t = cmplx (10.0_default * r(0), kind=default)
  vtest%x(1:3) = cmplx (10.0_default * r(1:3), kind=default)
  ttest = vtest.tprod.vtest
  call random_momentum (p, pabs, m)
  call random_momentum (q, qabs, m)
  call random_momentum (p0, 0.0_default, m)
  vp = p
  vq = q
  v0 = p0
  passed = .true.
  ⟨Test omega95⟩
  if (.not. passed) then
  stop 1
  end if
  end program test_omega95
```

⟨*Test* omega95⟩≡
```
  print *, "*** Checking the equations of motion ***:"
  call expect (abs(f_vf(c_one,vp,u(m,p,+1))-m*u(m,p,+1)), 0, "|[p-m]u(+)|=0", passed)
  call expect (abs(f_vf(c_one,vp,u(m,p,-1))-m*u(m,p,-1)), 0, "|[p-m]u(-)|=0", passed)
  call expect (abs(f_vf(c_one,vp,v(m,p,+1))+m*v(m,p,+1)), 0, "|[p+m]v(+)|=0", passed)
  call expect (abs(f_vf(c_one,vp,v(m,p,-1))+m*v(m,p,-1)), 0, "|[p+m]v(-)|=0", passed)
  call expect (abs(f_fv(c_one,ubar(m,p,+1),vp)-m*ubar(m,p,+1)), 0, "|ubar(+)[p-m]|=0", passed)
  call expect (abs(f_fv(c_one,ubar(m,p,-1),vp)-m*ubar(m,p,-1)), 0, "|ubar(-)[p-m]|=0", passed)
  call expect (abs(f_fv(c_one,vbar(m,p,+1),vp)+m*vbar(m,p,+1)), 0, "|vbar(+)[p+m]|=0", passed)
  call expect (abs(f_fv(c_one,vbar(m,p,-1),vp)+m*vbar(m,p,-1)), 0, "|vbar(-)[p+m]|=0", passed)
```

```
print *, "*** Checking the equations of motion for negative mass***:"
call expect (abs(f_vf(c_one,vp,u(-m,p,+1))+m*u(-m,p,+1)), 0, "|[p+m]u(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,u(-m,p,-1))+m*u(-m,p,-1)), 0, "|[p+m]u(-)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(-m,p,+1))-m*v(-m,p,+1)), 0, "|[p-m]v(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(-m,p,-1))-m*v(-m,p,-1)), 0, "|[p-m]v(-)|=0", passed)
call expect (abs(f_fv(c_one,ubar(-m,p,+1),vp)+m*ubar(-m,p,+1)), 0, "|ubar(+)[p+m]|=0", passed)
call expect (abs(f_fv(c_one,ubar(-m,p,-1),vp)+m*ubar(-m,p,-1)), 0, "|ubar(-)[p+m]|=0", passed)
call expect (abs(f_fv(c_one,vbar(-m,p,+1),vp)-m*vbar(-m,p,+1)), 0, "|vbar(+)[p-m]|=0", passed)
call expect (abs(f_fv(c_one,vbar(-m,p,-1),vp)-m*vbar(-m,p,-1)), 0, "|vbar(-)[p-m]|=0", passed)
```

⟨*Test* omega95⟩+≡

```
print *, "*** Spin Sums"
test_psi%a = [one, two, three, four]
test_spinor1 = f_vf (c_one, vp, test_psi) + m * test_psi
test_spinor2 = u (m, p, +1) * (ubar (m, p, +1) * test_psi) + &
u (m, p, -1) * (ubar (m, p, -1) * test_psi)
do i = 1, 4
call expect (test_spinor1%a(i), test_spinor2%a(i), "(p+m)1=(sum u ubar)1", passed)
end do
test_spinor1 = f_vf (c_one, vp, test_psi) - m * test_psi
test_spinor2 = v (m, p, +1) * (vbar (m, p, +1) * test_psi) + &
v (m, p, -1) * (vbar (m, p, -1) * test_psi)
do i = 1, 4
call expect (test_spinor1%a(i), test_spinor2%a(i), "(p-m)1=(sum v vbar)1", passed)
end do
test_psibar%a = [one, two, three, four]
test_conjspinor1 = f_fv (c_one, test_psibar, vp) - m * test_psibar
test_conjspinor2 = (test_psibar * v (m, p, +1)) * vbar (m, p, +1) + &
(test_psibar * v (m, p, -1)) * vbar (m, p, -1)
do i = 1, 4
call expect (test_conjspinor1%a(i), test_conjspinor2%a(i), "(p-m)1=(sum v vbar)1", passed)
end do
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking the normalization ***:"
call expect (ubar(m,p,+1)*u(m,p,+1), +2*m, "ubar(+)*u(+)=+2m", passed)
call expect (ubar(m,p,-1)*u(m,p,-1), +2*m, "ubar(-)*u(-)=+2m", passed)
call expect (vbar(m,p,+1)*v(m,p,+1), -2*m, "vbar(+)*v(+)=-2m", passed)
call expect (vbar(m,p,-1)*v(m,p,-1), -2*m, "vbar(-)*v(-)=-2m", passed)
call expect (ubar(m,p,+1)*v(m,p,+1),    0, "ubar(+)*v(+)=0  ", passed)
call expect (ubar(m,p,-1)*v(m,p,-1),    0, "ubar(-)*v(-)=0  ", passed)
call expect (vbar(m,p,+1)*u(m,p,+1),    0, "vbar(+)*u(+)=0  ", passed)
call expect (vbar(m,p,-1)*u(m,p,-1),    0, "vbar(-)*u(-)=0  ", passed)
print *, "*** Checking the normalization for negative masses***:"
call expect (ubar(-m,p,+1)*u(-m,p,+1), -2*m, "ubar(+)*u(+)=-2m", passed)
call expect (ubar(-m,p,-1)*u(-m,p,-1), -2*m, "ubar(-)*u(-)=-2m", passed)
call expect (vbar(-m,p,+1)*v(-m,p,+1), +2*m, "vbar(+)*v(+)=+2m", passed)
call expect (vbar(-m,p,-1)*v(-m,p,-1), +2*m, "vbar(-)*v(-)=+2m", passed)
call expect (ubar(-m,p,+1)*v(-m,p,+1),    0, "ubar(+)*v(+)=0  ", passed)
call expect (ubar(-m,p,-1)*v(-m,p,-1),    0, "ubar(-)*v(-)=0  ", passed)
call expect (vbar(-m,p,+1)*u(-m,p,+1),    0, "vbar(+)*u(+)=0  ", passed)
call expect (vbar(-m,p,-1)*u(-m,p,-1),    0, "vbar(-)*u(-)=0  ", passed)
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking the currents ***:"
call expect (abs(v_ff(c_one,ubar(m,p,+1),u(m,p,+1))-2*vp), 0, "ubar(+).V.u(+)=2p", passed)
call expect (abs(v_ff(c_one,ubar(m,p,-1),u(m,p,-1))-2*vp), 0, "ubar(-).V.u(-)=2p", passed)
call expect (abs(v_ff(c_one,vbar(m,p,+1),v(m,p,+1))-2*vp), 0, "vbar(+).V.v(+)=2p", passed)
call expect (abs(v_ff(c_one,vbar(m,p,-1),v(m,p,-1))-2*vp), 0, "vbar(-).V.v(-)=2p", passed)
print *, "*** Checking the currents for negative masses***:"
call expect (abs(v_ff(c_one,ubar(-m,p,+1),u(-m,p,+1))-2*vp), 0, "ubar(+).V.u(+)=2p", passed)
call expect (abs(v_ff(c_one,ubar(-m,p,-1),u(-m,p,-1))-2*vp), 0, "ubar(-).V.u(-)=2p", passed)
call expect (abs(v_ff(c_one,vbar(-m,p,+1),v(-m,p,+1))-2*vp), 0, "vbar(+).V.v(+)=2p", passed)
call expect (abs(v_ff(c_one,vbar(-m,p,-1),v(-m,p,-1))-2*vp), 0, "vbar(-).V.v(-)=2p", passed)
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking current conservation ***:"
call expect ((vp-vq)*v_ff(c_one,ubar(m,p,+1),u(m,q,+1)), 0, "d(ubar(+).V.u(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,ubar(m,p,-1),u(m,q,-1)), 0, "d(ubar(-).V.u(-))=0", passed)
call expect ((vp-vq)*v_ff(c_one,vbar(m,p,+1),v(m,q,+1)), 0, "d(vbar(+).V.v(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,vbar(m,p,-1),v(m,q,-1)), 0, "d(vbar(-).V.v(-))=0", passed)
print *, "*** Checking current conservation for negative masses***:"
call expect ((vp-vq)*v_ff(c_one,ubar(-m,p,+1),u(-m,q,+1)), 0, "d(ubar(+).V.u(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,ubar(-m,p,-1),u(-m,q,-1)), 0, "d(ubar(-).V.u(-))=0", passed)
call expect ((vp-vq)*v_ff(c_one,vbar(-m,p,+1),v(-m,q,+1)), 0, "d(vbar(+).V.v(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,vbar(-m,p,-1),v(-m,q,-1)), 0, "d(vbar(-).V.v(-))=0", passed)
```

⟨*Test* omega95⟩+≡

```
if (m == 0) then
print *, "*** Checking axial current conservation ***:"
call expect ((vp-vq)*a_ff(c_one,ubar(m,p,+1),u(m,q,+1)), 0, "d(ubar(+).A.u(+))=0", passed)
call expect ((vp-vq)*a_ff(c_one,ubar(m,p,-1),u(m,q,-1)), 0, "d(ubar(-).A.u(-))=0", passed)
call expect ((vp-vq)*a_ff(c_one,vbar(m,p,+1),v(m,q,+1)), 0, "d(vbar(+).A.v(+))=0", passed)
call expect ((vp-vq)*a_ff(c_one,vbar(m,p,-1),v(m,q,-1)), 0, "d(vbar(-).A.v(-))=0", passed)
end if
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking implementation of the sigma vertex funktions ***:"
call expect ((vp*tvam_ff(c_one,c_nil,ubar(m,p,+1),u(m,q,+1),q) - (p*q-m**2)*(ubar(m,p,+1)*u(m,q,+1))),
0, &
"p*[ubar(p,+).(Isigma*q).u(q,+)] - (p*q-m^2)*ubar(p,+).u(q,+) = 0", passed)
call expect ((vp*tvam_ff(c_one,c_nil,ubar(m,p,-1),u(m,q,-1),q) - (p*q-m**2)*(ubar(m,p,-1)*u(m,q,-1))),
0, &
"p*[ubar(p,-).(Isigma*q).u(q,-)] - (p*q-m^2)*ubar(p,-).u(q,-) = 0", passed)
call expect ((vp*tvam_ff(c_one,c_nil,vbar(m,p,+1),v(m,q,+1),q) - (p*q-m**2)*(vbar(m,p,+1)*v(m,q,+1))),
0, &
"p*[vbar(p,+).(Isigma*q).v(q,+)] - (p*q-m^2)*vbar(p,+).v(q,+) = 0", passed)
call expect ((vp*tvam_ff(c_one,c_nil,vbar(m,p,-1),v(m,q,-1),q) - (p*q-m**2)*(vbar(m,p,-1)*v(m,q,-1))),
0, &
"p*[vbar(p,-).(Isigma*q).v(q,-)] - (p*q-m^2)*vbar(p,-).v(q,-) = 0", passed)
call expect ((ubar(m,p,+1)*f_tvamf(c_one,c_nil,vp,u(m,q,+1),q) - (p*q-m**2)*(ubar(m,p,+1)*u(m,q,+1))),
0, &
"ubar(p,+).[p*(Isigma*q).u(q,+)] - (p*q-m^2)*ubar(p,+).u(q,+) = 0", passed)
call expect ((ubar(m,p,-1)*f_tvamf(c_one,c_nil,vp,u(m,q,-1),q) - (p*q-m**2)*(ubar(m,p,-1)*u(m,q,-1))),
0, &
"ubar(p,-).[p*(Isigma*q).u(q,-)] - (p*q-m^2)*ubar(p,-).u(q,-) = 0", passed)
call expect ((vbar(m,p,+1)*f_tvamf(c_one,c_nil,vp,v(m,q,+1),q) - (p*q-m**2)*(vbar(m,p,+1)*v(m,q,+1))),
0, &
"vbar(p,+).[p*(Isigma*q).v(q,+)] - (p*q-m^2)*vbar(p,+).v(q,+) = 0", passed)
call expect ((vbar(m,p,-1)*f_tvamf(c_one,c_nil,vp,v(m,q,-1),q) - (p*q-m**2)*(vbar(m,p,-1)*v(m,q,-1))),
0, &
"vbar(p,-).[p*(Isigma*q).v(q,-)] - (p*q-m^2)*vbar(p,-).v(q,-) = 0", passed)
call expect ((f_ftvam(c_one,c_nil,ubar(m,p,+1),vp,q)*u(m,q,+1) - (p*q-m**2)*(ubar(m,p,+1)*u(m,q,+1))),
0, &
"[ubar(p,+).p*(Isigma*q)].u(q,+) - (p*q-m^2)*ubar(p,+).u(q,+) = 0", passed)
call expect ((f_ftvam(c_one,c_nil,ubar(m,p,-1),vp,q)*u(m,q,-1) - (p*q-m**2)*(ubar(m,p,-1)*u(m,q,-1))),
0, &
"[ubar(p,-).p*(Isigma*q)].u(q,-) - (p*q-m^2)*ubar(p,-).u(q,-) = 0", passed)
call expect ((f_ftvam(c_one,c_nil,vbar(m,p,+1),vp,q)*v(m,q,+1) - (p*q-m**2)*(vbar(m,p,+1)*v(m,q,+1))),
0, &
"[vbar(p,+).p*(Isigma*q)].v(q,+) - (p*q-m^2)*vbar(p,+).v(q,+) = 0", passed)
call expect ((f_ftvam(c_one,c_nil,vbar(m,p,-1),vp,q)*v(m,q,-1) - (p*q-m**2)*(vbar(m,p,-1)*v(m,q,-1))),
0, &
"[vbar(p,-).p*(Isigma*q)].v(q,-) - (p*q-m^2)*vbar(p,-).v(q,-) = 0", passed)

call expect ((vp*tvam_ff(c_nil,c_one,ubar(m,p,+1),u(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,ubar(m,p,+1),u(m,q,+1))),
0, &
"p*[ubar(p,+).(Isigma*q).g5.u(q,+)] - (p*q+m^2)*ubar(p,+).g5.u(q,+) = 0", passed)
call expect ((vp*tvam_ff(c_nil,c_one,ubar(m,p,-1),u(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,ubar(m,p,-1),u(m,q,-1))),
0, &
"p*[ubar(p,-).(Isigma*q).g5.u(q,-)] - (p*q+m^2)*ubar(p,-).g5.u(q,-) = 0", passed)
call expect ((vp*tvam_ff(c_nil,c_one,vbar(m,p,+1),v(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,vbar(m,p,+1),v(m,q,+1))),
0, &
"p*[vbar(p,+).(Isigma*q).g5.v(q,+)] - (p*q+m^2)*vbar(p,+).g5.v(q,+) = 0", passed)
```

```
call expect ((vp*tvam_ff(c_nil,c_one,vbar(m,p,-1),v(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,vbar(m,p,-1),v(m,q,-1))),
0, &
"p*[vbar(p,-).(Isigma*q).g5.v(q,-)] - (p*q+m^2)*vbar(p,-).g5.v(q,-) = 0", passed)
call expect ((ubar(m,p,+1)*f_tvamf(c_nil,c_one,vp,u(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,ubar(m,p,+1),u(m,q,+1))),
0, &
"p*[ubar(p,+).(Isigma*q).g5.u(q,+)] - (p*q+m^2)*ubar(p,+).g5.u(q,+) = 0", passed)
call expect ((ubar(m,p,-1)*f_tvamf(c_nil,c_one,vp,u(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,ubar(m,p,-1),u(m,q,-1))),
0, &
"p*[ubar(p,-).(Isigma*q).g5.u(q,-)] - (p*q+m^2)*ubar(p,-).g5.u(q,-) = 0", passed)
call expect ((vbar(m,p,+1)*f_tvamf(c_nil,c_one,vp,v(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,vbar(m,p,+1),v(m,q,+1))),
0, &
"p*[vbar(p,+).(Isigma*q).g5.v(q,+)] - (p*q+m^2)*vbar(p,+).g5.v(q,+) = 0", passed)
call expect ((vbar(m,p,-1)*f_tvamf(c_nil,c_one,vp,v(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,vbar(m,p,-1),v(m,q,-1))),
0, &
"p*[vbar(p,-).(Isigma*q).g5.v(q,-)] - (p*q+m^2)*vbar(p,-).g5.v(q,-) = 0", passed)
call expect ((f_ftvam(c_nil,c_one,ubar(m,p,+1),vp,q)*u(m,q,+1) - (p*q+m**2)*p_ff(c_one,ubar(m,p,+1),u(m,q,+1))),
0, &
"p*[ubar(p,+).(Isigma*q).g5.u(q,+)] - (p*q+m^2)*ubar(p,+).g5.u(q,+) = 0", passed)
call expect ((f_ftvam(c_nil,c_one,ubar(m,p,-1),vp,q)*u(m,q,-1) - (p*q+m**2)*p_ff(c_one,ubar(m,p,-1),u(m,q,-1))),
0, &
"p*[ubar(p,-).(Isigma*q).g5.u(q,-)] - (p*q+m^2)*ubar(p,-).g5.u(q,-) = 0", passed)
call expect ((f_ftvam(c_nil,c_one,vbar(m,p,+1),vp,q)*v(m,q,+1) - (p*q+m**2)*p_ff(c_one,vbar(m,p,+1),v(m,q,+1))),
0, &
"p*[vbar(p,+).(Isigma*q).g5.v(q,+)] - (p*q+m^2)*vbar(p,+).g5.v(q,+) = 0", passed)
call expect ((f_ftvam(c_nil,c_one,vbar(m,p,-1),vp,q)*v(m,q,-1) - (p*q+m**2)*p_ff(c_one,vbar(m,p,-1),v(m,q,-1))),
0, &
"p*[vbar(p,-).(Isigma*q).g5.v(q,-)] - (p*q+m^2)*vbar(p,-).g5.v(q,-) = 0", passed)
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking polarisation vectors: ***"
call expect (conjg(eps(m,p, 1))*eps(m,p, 1), -1, "e( 1).e( 1)=-1", passed)
call expect (conjg(eps(m,p, 1))*eps(m,p,-1),  0, "e( 1).e(-1)= 0", passed)
call expect (conjg(eps(m,p,-1))*eps(m,p, 1),  0, "e(-1).e( 1)= 0", passed)
call expect (conjg(eps(m,p,-1))*eps(m,p,-1), -1, "e(-1).e(-1)=-1", passed)
call expect (                p*eps(m,p, 1),  0, "    p.e( 1)= 0", passed)
call expect (                p*eps(m,p,-1),  0, "    p.e(-1)= 0", passed)
if (m > 0) then
call expect (conjg(eps(m,p, 1))*eps(m,p, 0),  0, "e( 1).e( 0)= 0", passed)
call expect (conjg(eps(m,p, 0))*eps(m,p, 1),  0, "e( 0).e( 1)= 0", passed)
call expect (conjg(eps(m,p, 0))*eps(m,p, 0), -1, "e( 0).e( 0)=-1", passed)
call expect (conjg(eps(m,p, 0))*eps(m,p,-1),  0, "e( 0).e(-1)= 0", passed)
call expect (conjg(eps(m,p,-1))*eps(m,p, 0),  0, "e(-1).e( 0)= 0", passed)
call expect (               p*eps(m,p, 0),  0, "    p.e( 0)= 0", passed)
end if
```

⟨*Test* omega95⟩+≡

```
print *, "*** Checking epsilon tensor: ***"
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,q,1),eps(m,p,1),eps(m,p,0),eps(m,q,0)), "eps(1<->2)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,p,0),eps(m,q,1),eps(m,p,1),eps(m,q,0)), "eps(1<->3)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,q,0),eps(m,q,1),eps(m,p,0),eps(m,p,1)), "eps(1<->4)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,p,1),eps(m,p,0),eps(m,q,1),eps(m,q,0)), "eps(2<->3)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,p,1),eps(m,q,0),eps(m,p,0),eps(m,q,1)), "eps(2<->4)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
- pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,q,0),eps(m,p,0)), "eps(3<->4)", passed)
call expect (  pseudo_scalar(eps(m,p,1),eps(m,q,1),eps(m,p,0),eps(m,q,0)), &
eps(m,p,1)*pseudo_vector(eps(m,q,1),eps(m,p,0),eps(m,q,0)), "eps'", passed)
```

$$\frac{1}{2}[x \wedge y]^*_{\mu\nu}[x \wedge y]^{\mu\nu} = \frac{1}{2}(x^*_\mu y^*_\nu - x^*_\nu y^*_\mu)(x^\mu y^\nu - x^\nu y^\mu) = (x^* x)(y^* y) - (x^* y)(y^* x) \tag{AB.143}$$

⟨*Test* omega95⟩+≡

```
print *, "*** Checking tensors: ***"
call expect (conjg(p.wedge.q)*(p.wedge.q), (p*p)*(q*q)-(p*q)**2, &
"[p,q].[q,p]=p.p*q.q-p.q^2", passed)
```

```
  call expect (conjg(p.wedge.q)*(q.wedge.p), (p*q)**2-(p*p)*(q*q), &
  "[p,q].[q,p]=p.q^2-p.p*q.q", passed)
```

i. e.

$$\frac{1}{2}[p \wedge \epsilon(p,i)]^*_{\mu\nu}[p \wedge \epsilon(p,j)]^{\mu\nu} = -p^2\delta_{ij} \tag{AB.144}$$

⟨*Test* omega95⟩+≡

```
  call expect (conjg(p.wedge.eps(m,p, 1))*(p.wedge.eps(m,p, 1)), -p*p, &
  "[p,e( 1)].[p,e( 1)]=-p.p", passed)
  call expect (conjg(p.wedge.eps(m,p, 1))*(p.wedge.eps(m,p,-1)),    0, &
  "[p,e( 1)].[p,e(-1)]=0", passed)
  call expect (conjg(p.wedge.eps(m,p,-1))*(p.wedge.eps(m,p, 1)),    0, &
  "[p,e(-1)].[p,e( 1)]=0", passed)
  call expect (conjg(p.wedge.eps(m,p,-1))*(p.wedge.eps(m,p,-1)), -p*p, &
  "[p,e(-1)].[p,e(-1)]=-p.p", passed)
  if (m > 0) then
  call expect (conjg(p.wedge.eps(m,p, 1))*(p.wedge.eps(m,p, 0)),    0, &
  "[p,e( 1)].[p,e( 0)]=0", passed)
  call expect (conjg(p.wedge.eps(m,p, 0))*(p.wedge.eps(m,p, 1)),    0, &
  "[p,e( 0)].[p,e( 1)]=0", passed)
  call expect (conjg(p.wedge.eps(m,p, 0))*(p.wedge.eps(m,p, 0)), -p*p, &
  "[p,e( 0)].[p,e( 0)]=-p.p", passed)
  call expect (conjg(p.wedge.eps(m,p, 0))*(p.wedge.eps(m,p,-1)),    0, &
  "[p,e( 1)].[p,e(-1)]=0", passed)
  call expect (conjg(p.wedge.eps(m,p,-1))*(p.wedge.eps(m,p, 0)),    0, &
  "[p,e(-1)].[p,e( 0)]=0", passed)
  end if
```

also

$$[x \wedge y]_{\mu\nu} z^\nu = x_\mu(yz) - y_\mu(xz) \tag{AB.145}$$
$$z_\mu [x \wedge y]^{\mu\nu} = (zx)y^\nu - (zy)x^\nu \tag{AB.146}$$

⟨*Test* omega95⟩+≡

```
  call expect (abs ((p.wedge.eps(m,p, 1))*p + (p*p)*eps(m,p, 1)), 0, &
  "[p,e( 1)].p=-p.p*e( 1)]", passed)
  call expect (abs ((p.wedge.eps(m,p, 0))*p + (p*p)*eps(m,p, 0)), 0, &
  "[p,e( 0)].p=-p.p*e( 0)]", passed)
  call expect (abs ((p.wedge.eps(m,p,-1))*p + (p*p)*eps(m,p,-1)), 0, &
  "[p,e(-1)].p=-p.p*e(-1)]", passed)
  call expect (abs (p*(p.wedge.eps(m,p, 1)) - (p*p)*eps(m,p, 1)), 0, &
  "p.[p,e( 1)]=p.p*e( 1)]", passed)
  call expect (abs (p*(p.wedge.eps(m,p, 0)) - (p*p)*eps(m,p, 0)), 0, &
  "p.[p,e( 0)]=p.p*e( 0)]", passed)
  call expect (abs (p*(p.wedge.eps(m,p,-1)) - (p*p)*eps(m,p,-1)), 0, &
  "p.[p,e(-1)]=p.p*e(-1)]", passed)
```

⟨*Test* omega95⟩+≡

```
  print *, "*** Checking polarisation tensors: ***"
  call expect (conjg(eps2(m,p, 2))*eps2(m,p, 2), 1, "e2( 2).e2( 2)=1", passed)
  call expect (conjg(eps2(m,p, 2))*eps2(m,p,-2), 0, "e2( 2).e2(-2)=0", passed)
  call expect (conjg(eps2(m,p,-2))*eps2(m,p, 2), 0, "e2(-2).e2( 2)=0", passed)
  call expect (conjg(eps2(m,p,-2))*eps2(m,p,-2), 1, "e2(-2).e2(-2)=1", passed)
  if (m > 0) then
  call expect (conjg(eps2(m,p, 2))*eps2(m,p, 1), 0, "e2( 2).e2( 1)=0", passed)
  call expect (conjg(eps2(m,p, 2))*eps2(m,p, 0), 0, "e2( 2).e2( 0)=0", passed)
  call expect (conjg(eps2(m,p, 2))*eps2(m,p,-1), 0, "e2( 2).e2(-1)=0", passed)
  call expect (conjg(eps2(m,p, 1))*eps2(m,p, 2), 0, "e2( 1).e2( 2)=0", passed)
  call expect (conjg(eps2(m,p, 1))*eps2(m,p, 1), 1, "e2( 1).e2( 1)=1", passed)
  call expect (conjg(eps2(m,p, 1))*eps2(m,p, 0), 0, "e2( 1).e2( 0)=0", passed)
  call expect (conjg(eps2(m,p, 1))*eps2(m,p,-1), 0, "e2( 1).e2(-1)=0", passed)
  call expect (conjg(eps2(m,p, 1))*eps2(m,p,-2), 0, "e2( 1).e2(-2)=0", passed)
  call expect (conjg(eps2(m,p, 0))*eps2(m,p, 2), 0, "e2( 0).e2( 2)=0", passed)
  call expect (conjg(eps2(m,p, 0))*eps2(m,p, 1), 0, "e2( 0).e2( 1)=0", passed)
  call expect (conjg(eps2(m,p, 0))*eps2(m,p, 0), 1, "e2( 0).e2( 0)=1", passed)
  call expect (conjg(eps2(m,p, 0))*eps2(m,p,-1), 0, "e2( 0).e2(-1)=0", passed)
  call expect (conjg(eps2(m,p, 0))*eps2(m,p,-2), 0, "e2( 0).e2(-2)=0", passed)
  call expect (conjg(eps2(m,p,-1))*eps2(m,p, 2), 0, "e2(-1).e2( 2)=0", passed)
  call expect (conjg(eps2(m,p,-1))*eps2(m,p, 1), 0, "e2(-1).e2( 1)=0", passed)
  call expect (conjg(eps2(m,p,-1))*eps2(m,p, 0), 0, "e2(-1).e2( 0)=0", passed)
  call expect (conjg(eps2(m,p,-1))*eps2(m,p,-1), 1, "e2(-1).e2(-1)=1", passed)
  call expect (conjg(eps2(m,p,-1))*eps2(m,p,-2), 0, "e2(-1).e2(-2)=0", passed)
  call expect (conjg(eps2(m,p,-2))*eps2(m,p, 1), 0, "e2(-2).e2( 1)=0", passed)
  call expect (conjg(eps2(m,p,-2))*eps2(m,p, 0), 0, "e2(-2).e2( 0)=0", passed)
  call expect (conjg(eps2(m,p,-2))*eps2(m,p,-1), 0, "e2(-2).e2(-1)=0", passed)
  end if
```

⟨*Test* omega95⟩+≡

```
  call expect (           abs(p*eps2(m,p, 2)  ), 0, " |p.e2( 2)|  =0", passed)
  call expect (             abs(eps2(m,p, 2)*p), 0, "   |e2( 2).p|=0", passed)
  call expect (           abs(p*eps2(m,p,-2)  ), 0, " |p.e2(-2)|  =0", passed)
  call expect (             abs(eps2(m,p,-2)*p), 0, "   |e2(-2).p|=0", passed)
  if (m > 0) then
  call expect (           abs(p*eps2(m,p, 1)  ), 0, " |p.e2( 1)|  =0", passed)
  call expect (             abs(eps2(m,p, 1)*p), 0, "   |e2( 1).p|=0", passed)
  call expect (           abs(p*eps2(m,p, 0)  ), 0, " |p.e2( 0)|  =0", passed)
  call expect (             abs(eps2(m,p, 0)*p), 0, "   |e2( 0).p|=0", passed)
  call expect (           abs(p*eps2(m,p,-1)  ), 0, " |p.e2(-1)|  =0", passed)
  call expect (             abs(eps2(m,p,-1)*p), 0, "   |e2(-1).p|=0", passed)
  end if
```

⟨*XXX Test* omega95⟩≡

```
  print *, " *** Checking the polarization tensors for massive gravitons:"
  call expect (abs(p * eps2(m,p,2)),  0, "p.e(+2)=0", passed)
  call expect (abs(p * eps2(m,p,1)),  0, "p.e(+1)=0", passed)
  call expect (abs(p * eps2(m,p,0)),  0, "p.e( 0)=0", passed)
  call expect (abs(p * eps2(m,p,-1)), 0, "p.e(-1)=0", passed)
  call expect (abs(p * eps2(m,p,-2)), 0, "p.e(-2)=0", passed)
  call expect (abs(trace(eps2 (m,p,2))),  0, "Tr[e(+2)]=0", passed)
  call expect (abs(trace(eps2 (m,p,1))),  0, "Tr[e(+1)]=0", passed)
  call expect (abs(trace(eps2 (m,p,0))),  0, "Tr[e( 0)]=0", passed)
  call expect (abs(trace(eps2 (m,p,-1))), 0, "Tr[e(-1)]=0", passed)
  call expect (abs(trace(eps2 (m,p,-2))), 0, "Tr[e(-2)]=0", passed)
  call expect (abs(eps2(m,p,2) * eps2(m,p,2)),   1, &
  "e(2).e(2)   = 1", passed)
  call expect (abs(eps2(m,p,2) * eps2(m,p,1)),   0, &
  "e(2).e(1)   = 0", passed)
  call expect (abs(eps2(m,p,2) * eps2(m,p,0)),   0, &
  "e(2).e(0)   = 0", passed)
  call expect (abs(eps2(m,p,2) * eps2(m,p,-1)),  0, &
  "e(2).e(-1)  = 0", passed)
  call expect (abs(eps2(m,p,2) * eps2(m,p,-2)),  0, &
  "e(2).e(-2)  = 0", passed)
  call expect (abs(eps2(m,p,1) * eps2(m,p,1)),   1, &
  "e(1).e(1)   = 1", passed)
  call expect (abs(eps2(m,p,1) * eps2(m,p,0)),   0, &
  "e(1).e(0)   = 0", passed)
  call expect (abs(eps2(m,p,1) * eps2(m,p,-1)),  0, &
  "e(1).e(-1)  = 0", passed)
  call expect (abs(eps2(m,p,1) * eps2(m,p,-2)),  0, &
  "e(1).e(-2)  = 0", passed)
  call expect (abs(eps2(m,p,0) * eps2(m,p,0)),   1, &
  "e(0).e(0)   = 1", passed)
  call expect (abs(eps2(m,p,0) * eps2(m,p,-1)),  0, &
  "e(0).e(-1)  = 0", passed)
  call expect (abs(eps2(m,p,0) * eps2(m,p,-2)),  0, &
  "e(0).e(-2)  = 0", passed)
  call expect (abs(eps2(m,p,-1) * eps2(m,p,-1)), 1, &
  "e(-1).e(-1) = 1", passed)
  call expect (abs(eps2(m,p,-1) * eps2(m,p,-2)), 0, &
  "e(-1).e(-2) = 0", passed)
  call expect (abs(eps2(m,p,-2) * eps2(m,p,-2)), 1, &
  "e(-2).e(-2) = 1", passed)
```

⟨*Test* omega95⟩+≡

```
  print *, " *** Checking the graviton propagator:"
```

```
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,eps2(m,p,-2)))), 0, "p.pr.e(-2)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,eps2(m,p,-1)))), 0, "p.pr.e(-1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,eps2(m,p,0)))), 0, "p.pr.e(0)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,eps2(m,p,1)))), 0, "p.pr.e(1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,eps2(m,p,2)))), 0, "p.pr.e(2)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_tensor(p,m,w,ttest))), 0, "p.pr.ttest", passed)
```

⟨test_omega95_bispinors.f90⟩≡

```
⟨Copyleft⟩
program test_omega95_bispinors
use kinds
use omega95_bispinors
use omega_vspinor_polarizations
use omega_testtools
implicit none
integer :: i, j
real(kind=default) :: m, pabs, qabs, tabs, zabs, w
real(kind=default), dimension(4) :: r
complex(kind=default) :: c_nil, c_one, c_two
type(momentum) :: p, q, t, z, p_0
type(vector) :: vp, vq, vt, vz
type(vectorspinor) :: testv
type(bispinor) :: vv
logical :: passed
call random_seed ()
c_nil = 0.0_default
c_one = 1.0_default
c_two = 2.0_default
w = 1.4142
m = 13
pabs = 42
qabs = 137
tabs = 84
zabs = 3.1415
p_0%t = m
p_0%x = 0
call random_momentum (p, pabs, m)
call random_momentum (q, qabs, m)
call random_momentum (t, tabs, m)
call random_momentum (z, zabs, m)
call random_number (r)
do i = 1, 4
testv%psi(1)%a(i) = (0.0_default, 0.0_default)
end do
do i = 2, 3
do j = 1, 4
testv%psi(i)%a(j) = cmplx (10.0_default * r(j), kind=default)
end do
end do
testv%psi(4)%a(1) = (1.0_default, 0.0_default)
testv%psi(4)%a(2) = (0.0_default, 2.0_default)
testv%psi(4)%a(3) = (1.0_default, 0.0_default)
testv%psi(4)%a(4) = (3.0_default, 0.0_default)
vp = p
vq = q
vt = t
vz = z
passed = .true.
vv%a(1) = (1.0_default, 0.0_default)
vv%a(2) = (0.0_default, 2.0_default)
vv%a(3) = (1.0_default, 0.0_default)
```

```
vv%a(4) = (3.0_default, 0.0_default)
vv = pr_psi(p, m, w, .false., vv)
⟨Test omega95_bispinors⟩
if (.not. passed) then
stop 1
end if
end program test_omega95_bispinors
```

⟨*Test* omega95_bispinors⟩≡

```
print *, "*** Checking the equations of motion ***:"
call expect (abs(f_vf(c_one,vp,u(m,p,+1))-m*u(m,p,+1)), 0, "|[p-m]u(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,u(m,p,-1))-m*u(m,p,-1)), 0, "|[p-m]u(-)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(m,p,+1))+m*v(m,p,+1)), 0, "|[p+m]v(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(m,p,-1))+m*v(m,p,-1)), 0, "|[p+m]v(-)|=0", passed)
print *, "*** Checking the equations of motion for negative masses***:"
call expect (abs(f_vf(c_one,vp,u(-m,p,+1))+m*u(-m,p,+1)), 0, "|[p+m]u(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,u(-m,p,-1))+m*u(-m,p,-1)), 0, "|[p+m]u(-)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(-m,p,+1))-m*v(-m,p,+1)), 0, "|[p-m]v(+)|=0", passed)
call expect (abs(f_vf(c_one,vp,v(-m,p,-1))-m*v(-m,p,-1)), 0, "|[p-m]v(-)|=0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Checking the normalization ***:"
call expect (s_ff(c_one,v(m,p,+1),u(m,p,+1)), +2*m, "ubar(+)*u(+)=+2m", passed)
call expect (s_ff(c_one,v(m,p,-1),u(m,p,-1)), +2*m, "ubar(-)*u(-)=+2m", passed)
call expect (s_ff(c_one,u(m,p,+1),v(m,p,+1)), -2*m, "vbar(+)*v(+)=-2m", passed)
call expect (s_ff(c_one,u(m,p,-1),v(m,p,-1)), -2*m, "vbar(-)*v(-)=-2m", passed)
call expect (s_ff(c_one,v(m,p,+1),v(m,p,+1)),    0, "ubar(+)*v(+)=0  ", passed)
call expect (s_ff(c_one,v(m,p,-1),v(m,p,-1)),    0, "ubar(-)*v(-)=0  ", passed)
call expect (s_ff(c_one,u(m,p,+1),u(m,p,+1)),    0, "vbar(+)*u(+)=0  ", passed)
call expect (s_ff(c_one,u(m,p,-1),u(m,p,-1)),    0, "vbar(-)*u(-)=0  ", passed)
print *, "*** Checking the normalization for negative masses***:"
call expect (s_ff(c_one,v(-m,p,+1),u(-m,p,+1)), -2*m, "ubar(+)*u(+)=-2m", passed)
call expect (s_ff(c_one,v(-m,p,-1),u(-m,p,-1)), -2*m, "ubar(-)*u(-)=-2m", passed)
call expect (s_ff(c_one,u(-m,p,+1),v(-m,p,+1)), +2*m, "vbar(+)*v(+)=+2m", passed)
call expect (s_ff(c_one,u(-m,p,-1),v(-m,p,-1)), +2*m, "vbar(-)*v(-)=+2m", passed)
call expect (s_ff(c_one,v(-m,p,+1),v(-m,p,+1)),    0, "ubar(+)*v(+)=0  ", passed)
call expect (s_ff(c_one,v(-m,p,-1),v(-m,p,-1)),    0, "ubar(-)*v(-)=0  ", passed)
call expect (s_ff(c_one,u(-m,p,+1),u(-m,p,+1)),    0, "vbar(+)*u(+)=0  ", passed)
call expect (s_ff(c_one,u(-m,p,-1),u(-m,p,-1)),    0, "vbar(-)*u(-)=0  ", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Checking the currents ***:"
call expect (abs(v_ff(c_one,v(m,p,+1),u(m,p,+1))-2*vp), 0, "ubar(+).V.u(+)=2p", passed)
call expect (abs(v_ff(c_one,v(m,p,-1),u(m,p,-1))-2*vp), 0, "ubar(-).V.u(-)=2p", passed)
call expect (abs(v_ff(c_one,u(m,p,+1),v(m,p,+1))-2*vp), 0, "vbar(+).V.v(+)=2p", passed)
call expect (abs(v_ff(c_one,u(m,p,-1),v(m,p,-1))-2*vp), 0, "vbar(-).V.v(-)=2p", passed)
print *, "*** Checking the currents for negative masses***:"
call expect (abs(v_ff(c_one,v(-m,p,+1),u(-m,p,+1))-2*vp), 0, "ubar(+).V.u(+)=2p", passed)
call expect (abs(v_ff(c_one,v(-m,p,-1),u(-m,p,-1))-2*vp), 0, "ubar(-).V.u(-)=2p", passed)
call expect (abs(v_ff(c_one,u(-m,p,+1),v(-m,p,+1))-2*vp), 0, "vbar(+).V.v(+)=2p", passed)
call expect (abs(v_ff(c_one,u(-m,p,-1),v(-m,p,-1))-2*vp), 0, "vbar(-).V.v(-)=2p", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Checking current conservation ***:"
call expect ((vp-vq)*v_ff(c_one,v(m,p,+1),u(m,q,+1)), 0, "d(ubar(+).V.u(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,v(m,p,-1),u(m,q,-1)), 0, "d(ubar(-).V.u(-))=0", passed)
call expect ((vp-vq)*v_ff(c_one,u(m,p,+1),v(m,q,+1)), 0, "d(vbar(+).V.v(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,u(m,p,-1),v(m,q,-1)), 0, "d(vbar(-).V.v(-))=0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Checking current conservation for negative masses***:"
call expect ((vp-vq)*v_ff(c_one,v(-m,p,+1),u(-m,q,+1)), 0, "d(ubar(+).V.u(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,v(-m,p,-1),u(-m,q,-1)), 0, "d(ubar(-).V.u(-))=0", passed)
call expect ((vp-vq)*v_ff(c_one,u(-m,p,+1),v(-m,q,+1)), 0, "d(vbar(+).V.v(+))=0", passed)
call expect ((vp-vq)*v_ff(c_one,u(-m,p,-1),v(-m,q,-1)), 0, "d(vbar(-).V.v(-))=0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
if (m == 0) then
print *, "*** Checking axial current conservation ***:"
```

```
  call expect ((vp-vq)*a_ff(c_one,v(m,p,+1),u(m,q,+1)), 0, "d(ubar(+).A.u(+))=0", passed)
  call expect ((vp-vq)*a_ff(c_one,v(m,p,-1),u(m,q,-1)), 0, "d(ubar(-).A.u(-))=0", passed)
  call expect ((vp-vq)*a_ff(c_one,u(m,p,+1),v(m,q,+1)), 0, "d(vbar(+).A.v(+))=0", passed)
  call expect ((vp-vq)*a_ff(c_one,u(m,p,-1),v(m,q,-1)), 0, "d(vbar(-).A.v(-))=0", passed)
  end if
```

⟨*Test* omega95_bispinors⟩+≡

```
  print *, "*** Checking implementation of the sigma vertex funktions ***:"
  call expect ((vp*tvam_ff(c_one,c_nil,v(m,p,+1),u(m,q,+1),q) - (p*q-m**2)*(v(m,p,+1)*u(m,q,+1))), 0, &
  "p*[ubar(p,+).(Isigma*q).u(q,+)] - (p*q-m^2)*ubar(p,+).u(q,+) = 0", passed)
  call expect ((vp*tvam_ff(c_one,c_nil,v(m,p,-1),u(m,q,-1),q) - (p*q-m**2)*(v(m,p,-1)*u(m,q,-1))), 0, &
  "p*[ubar(p,-).(Isigma*q).u(q,-)] - (p*q-m^2)*ubar(p,-).u(q,-) = 0", passed)
  call expect ((vp*tvam_ff(c_one,c_nil,u(m,p,+1),v(m,q,+1),q) - (p*q-m**2)*(u(m,p,+1)*v(m,q,+1))), 0, &
  "p*[vbar(p,+).(Isigma*q).v(q,+)] - (p*q-m^2)*vbar(p,+).v(q,+) = 0", passed)
  call expect ((vp*tvam_ff(c_one,c_nil,u(m,p,-1),v(m,q,-1),q) - (p*q-m**2)*(u(m,p,-1)*v(m,q,-1))), 0, &
  "p*[vbar(p,-).(Isigma*q).v(q,-)] - (p*q-m^2)*vbar(p,-).v(q,-) = 0", passed)
  call expect ((v(m,p,+1)*f_tvamf(c_one,c_nil,vp,u(m,q,+1),q) - (p*q-m**2)*(v(m,p,+1)*u(m,q,+1))), 0, &
  "ubar(p,+).[p*(Isigma*q).u(q,+)] - (p*q-m^2)*ubar(p,+).u(q,+) = 0", passed)
  call expect ((v(m,p,-1)*f_tvamf(c_one,c_nil,vp,u(m,q,-1),q) - (p*q-m**2)*(v(m,p,-1)*u(m,q,-1))), 0, &
  "ubar(p,-).[p*(Isigma*q).u(q,-)] - (p*q-m^2)*ubar(p,-).u(q,-) = 0", passed)
  call expect ((u(m,p,+1)*f_tvamf(c_one,c_nil,vp,v(m,q,+1),q) - (p*q-m**2)*(u(m,p,+1)*v(m,q,+1))), 0, &
  "vbar(p,+).[p*(Isigma*q).v(q,+)] - (p*q-m^2)*vbar(p,+).v(q,+) = 0", passed)
  call expect ((u(m,p,-1)*f_tvamf(c_one,c_nil,vp,v(m,q,-1),q) - (p*q-m**2)*(u(m,p,-1)*v(m,q,-1))), 0, &
  "vbar(p,-).[p*(Isigma*q).v(q,-)] - (p*q-m^2)*vbar(p,-).v(q,-) = 0", passed)

  call expect ((vp*tvam_ff(c_nil,c_one,v(m,p,+1),u(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,v(m,p,+1),u(m,q,+1))),
  0, &
  "p*[ubar(p,+).(Isigma*q).g5.u(q,+)] - (p*q+m^2)*ubar(p,+).g5.u(q,+) = 0", passed)
  call expect ((vp*tvam_ff(c_nil,c_one,v(m,p,-1),u(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,v(m,p,-1),u(m,q,-1))),
  0, &
  "p*[ubar(p,-).(Isigma*q).g5.u(q,-)] - (p*q+m^2)*ubar(p,-).g5.u(q,-) = 0", passed)
  call expect ((vp*tvam_ff(c_nil,c_one,u(m,p,+1),v(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,u(m,p,+1),v(m,q,+1))),
  0, &
  "p*[vbar(p,+).(Isigma*q).g5.v(q,+)] - (p*q+m^2)*vbar(p,+).g5.v(q,+) = 0", passed)
  call expect ((vp*tvam_ff(c_nil,c_one,u(m,p,-1),v(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,u(m,p,-1),v(m,q,-1))),
  0, &
  "p*[vbar(p,-).(Isigma*q).g5.v(q,-)] - (p*q+m^2)*vbar(p,-).g5.v(q,-) = 0", passed)
  call expect ((v(m,p,+1)*f_tvamf(c_nil,c_one,vp,u(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,v(m,p,+1),u(m,q,+1))),
  0, &
  "p*[ubar(p,+).(Isigma*q).g5.u(q,+)] - (p*q+m^2)*ubar(p,+).g5.u(q,+) = 0", passed)
  call expect ((v(m,p,-1)*f_tvamf(c_nil,c_one,vp,u(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,v(m,p,-1),u(m,q,-1))),
  0, &
  "p*[ubar(p,-).(Isigma*q).g5.u(q,-)] - (p*q+m^2)*ubar(p,-).g5.u(q,-) = 0", passed)
  call expect ((u(m,p,+1)*f_tvamf(c_nil,c_one,vp,v(m,q,+1),q) - (p*q+m**2)*p_ff(c_one,u(m,p,+1),v(m,q,+1))),
  0, &
  "p*[vbar(p,+).(Isigma*q).g5.v(q,+)] - (p*q+m^2)*vbar(p,+).g5.v(q,+) = 0", passed)
  call expect ((u(m,p,-1)*f_tvamf(c_nil,c_one,vp,v(m,q,-1),q) - (p*q+m**2)*p_ff(c_one,u(m,p,-1),v(m,q,-1))),
  0, &
  "p*[vbar(p,-).(Isigma*q).g5.v(q,-)] - (p*q+m^2)*vbar(p,-).g5.v(q,-) = 0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
  print *, "*** Checking polarization vectors: ***"
  call expect (conjg(eps(m,p, 1))*eps(m,p, 1), -1, "e( 1).e( 1)=-1", passed)
  call expect (conjg(eps(m,p, 1))*eps(m,p,-1),  0, "e( 1).e(-1)= 0", passed)
  call expect (conjg(eps(m,p,-1))*eps(m,p, 1),  0, "e(-1).e( 1)= 0", passed)
  call expect (conjg(eps(m,p,-1))*eps(m,p,-1), -1, "e(-1).e(-1)=-1", passed)
  call expect (                 p*eps(m,p, 1),  0, "    p.e( 1)= 0", passed)
  call expect (                 p*eps(m,p,-1),  0, "    p.e(-1)= 0", passed)
  if (m > 0) then
  call expect (conjg(eps(m,p, 1))*eps(m,p, 0),  0, "e( 1).e( 0)= 0", passed)
  call expect (conjg(eps(m,p, 0))*eps(m,p, 1),  0, "e( 0).e( 1)= 0", passed)
  call expect (conjg(eps(m,p, 0))*eps(m,p, 0), -1, "e( 0).e( 0)=-1", passed)
  call expect (conjg(eps(m,p, 0))*eps(m,p,-1),  0, "e( 0).e(-1)= 0", passed)
  call expect (conjg(eps(m,p,-1))*eps(m,p, 0),  0, "e(-1).e( 0)= 0", passed)
  call expect (                 p*eps(m,p, 0),  0, "    p.e( 0)= 0", passed)
  end if
```

⟨*Test* omega95_bispinors⟩+≡

```
  print *, "*** Checking polarization vectorspinors: ***"
  call expect (abs(p * ueps(m, p,  2)),  0, "p.ueps ( 2)= 0", passed)
  call expect (abs(p * ueps(m, p,  1)),  0, "p.ueps ( 1)= 0", passed)
  call expect (abs(p * ueps(m, p, -1)),  0, "p.ueps (-1)= 0", passed)
  call expect (abs(p * ueps(m, p, -2)),  0, "p.ueps (-2)= 0", passed)
  call expect (abs(p * veps(m, p,  2)),  0, "p.veps ( 2)= 0", passed)
  call expect (abs(p * veps(m, p,  1)),  0, "p.veps ( 1)= 0", passed)
  call expect (abs(p * veps(m, p, -1)),  0, "p.veps (-1)= 0", passed)
  call expect (abs(p * veps(m, p, -2)),  0, "p.veps (-2)= 0", passed)
  print *, "*** Checking polarization vectorspinors (neg. masses): ***"
  call expect (abs(p * ueps(-m, p,  2)),  0, "p.ueps ( 2)= 0", passed)
  call expect (abs(p * ueps(-m, p,  1)),  0, "p.ueps ( 1)= 0", passed)
  call expect (abs(p * ueps(-m, p, -1)),  0, "p.ueps (-1)= 0", passed)
  call expect (abs(p * ueps(-m, p, -2)),  0, "p.ueps (-2)= 0", passed)
  call expect (abs(p * veps(-m, p,  2)),  0, "p.veps ( 2)= 0", passed)
  call expect (abs(p * veps(-m, p,  1)),  0, "p.veps ( 1)= 0", passed)
  call expect (abs(p * veps(-m, p, -1)),  0, "p.veps (-1)= 0", passed)
  call expect (abs(p * veps(-m, p, -2)),  0, "p.veps (-2)= 0", passed)
  print *, "*** in the rest frame ***"
  call expect (abs(p_0 * ueps(m, p_0,  2)),  0, "p0.ueps ( 2)= 0", passed)
  call expect (abs(p_0 * ueps(m, p_0,  1)),  0, "p0.ueps ( 1)= 0", passed)
  call expect (abs(p_0 * ueps(m, p_0, -1)),  0, "p0.ueps (-1)= 0", passed)
  call expect (abs(p_0 * ueps(m, p_0, -2)),  0, "p0.ueps (-2)= 0", passed)
  call expect (abs(p_0 * veps(m, p_0,  2)),  0, "p0.veps ( 2)= 0", passed)
  call expect (abs(p_0 * veps(m, p_0,  1)),  0, "p0.veps ( 1)= 0", passed)
  call expect (abs(p_0 * veps(m, p_0, -1)),  0, "p0.veps (-1)= 0", passed)
  call expect (abs(p_0 * veps(m, p_0, -2)),  0, "p0.veps (-2)= 0", passed)
  print *, "*** in the rest frame (neg. masses) ***"
  call expect (abs(p_0 * ueps(-m, p_0,  2)),  0, "p0.ueps ( 2)= 0", passed)
  call expect (abs(p_0 * ueps(-m, p_0,  1)),  0, "p0.ueps ( 1)= 0", passed)
  call expect (abs(p_0 * ueps(-m, p_0, -1)),  0, "p0.ueps (-1)= 0", passed)
  call expect (abs(p_0 * ueps(-m, p_0, -2)),  0, "p0.ueps (-2)= 0", passed)
  call expect (abs(p_0 * veps(-m, p_0,  2)),  0, "p0.veps ( 2)= 0", passed)
  call expect (abs(p_0 * veps(-m, p_0,  1)),  0, "p0.veps ( 1)= 0", passed)
  call expect (abs(p_0 * veps(-m, p_0, -1)),  0, "p0.veps (-1)= 0", passed)
  call expect (abs(p_0 * veps(-m, p_0, -2)),  0, "p0.veps (-2)= 0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
  print *, "*** Checking the irreducibility condition: ***"
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p,  2))),  0, "g.ueps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p,  1))),  0, "g.ueps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p, -1))),  0, "g.ueps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p, -2))),  0, "g.ueps (-2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p,  2))),  0, "g.veps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p,  1))),  0, "g.veps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p, -1))),  0, "g.veps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p, -2))),  0, "g.veps (-2)", passed)
  print *, "*** Checking the irreducibility condition (neg. masses): ***"
  call expect (abs(f_potgr (c_one, c_one, ueps(-m, p,  2))),  0, "g.ueps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(-m, p,  1))),  0, "g.ueps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(-m, p, -1))),  0, "g.ueps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(-m, p, -2))),  0, "g.ueps (-2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(-m, p,  2))),  0, "g.veps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(-m, p,  1))),  0, "g.veps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(-m, p, -1))),  0, "g.veps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(-m, p, -2))),  0, "g.veps (-2)", passed)
  print *, "*** in the rest frame ***"
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0,  2))),  0, "g.ueps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0,  1))),  0, "g.ueps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0, -1))),  0, "g.ueps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0, -2))),  0, "g.ueps (-2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0,  2))),  0, "g.veps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0,  1))),  0, "g.veps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0, -1))),  0, "g.veps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0, -2))),  0, "g.veps (-2)", passed)
  print *, "*** in the rest frame (neg. masses) ***"
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0,  2))),  0, "g.ueps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0,  1))),  0, "g.ueps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0, -1))),  0, "g.ueps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, ueps(m, p_0, -2))),  0, "g.ueps (-2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0,  2))),  0, "g.veps ( 2)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0,  1))),  0, "g.veps ( 1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0, -1))),  0, "g.veps (-1)", passed)
  call expect (abs(f_potgr (c_one, c_one, veps(m, p_0, -2))),  0, "g.veps (-2)", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
  print *, "*** Testing vectorspinor normalization ***"
  call expect (veps(m,p, 2)*ueps(m,p, 2), -2*m, "ueps( 2).ueps( 2)= -2m", passed)
  call expect (veps(m,p, 1)*ueps(m,p, 1), -2*m, "ueps( 1).ueps( 1)= -2m", passed)
  call expect (veps(m,p,-1)*ueps(m,p,-1), -2*m, "ueps(-1).ueps(-1)= -2m", passed)
  call expect (veps(m,p,-2)*ueps(m,p,-2), -2*m, "ueps(-2).ueps(-2)= -2m", passed)
  call expect (ueps(m,p, 2)*veps(m,p, 2),  2*m, "veps( 2).veps( 2)= +2m", passed)
  call expect (ueps(m,p, 1)*veps(m,p, 1),  2*m, "veps( 1).veps( 1)= +2m", passed)
  call expect (ueps(m,p,-1)*veps(m,p,-1),  2*m, "veps(-1).veps(-1)= +2m", passed)
  call expect (ueps(m,p,-2)*veps(m,p,-2),  2*m, "veps(-2).veps(-2)= +2m", passed)
  call expect (ueps(m,p, 2)*ueps(m,p, 2),    0, "ueps( 2).veps( 2)=   0", passed)
  call expect (ueps(m,p, 1)*ueps(m,p, 1),    0, "ueps( 1).veps( 1)=   0", passed)
  call expect (ueps(m,p,-1)*ueps(m,p,-1),    0, "ueps(-1).veps(-1)=   0", passed)
  call expect (ueps(m,p,-2)*ueps(m,p,-2),    0, "ueps(-2).veps(-2)=   0", passed)
  call expect (veps(m,p, 2)*veps(m,p, 2),    0, "veps( 2).ueps( 2)=   0", passed)
  call expect (veps(m,p, 1)*veps(m,p, 1),    0, "veps( 1).ueps( 1)=   0", passed)
  call expect (veps(m,p,-1)*veps(m,p,-1),    0, "veps(-1).ueps(-1)=   0", passed)
  call expect (veps(m,p,-2)*veps(m,p,-2),    0, "veps(-2).ueps(-2)=   0", passed)
  print *, "*** Testing vectorspinor normalization (neg. masses) ***"
  call expect (veps(-m,p, 2)*ueps(-m,p, 2), +2*m, "ueps( 2).ueps( 2)= +2m", passed)
  call expect (veps(-m,p, 1)*ueps(-m,p, 1), +2*m, "ueps( 1).ueps( 1)= +2m", passed)
  call expect (veps(-m,p,-1)*ueps(-m,p,-1), +2*m, "ueps(-1).ueps(-1)= +2m", passed)
  call expect (veps(-m,p,-2)*ueps(-m,p,-2), +2*m, "ueps(-2).ueps(-2)= +2m", passed)
  call expect (ueps(-m,p, 2)*veps(-m,p, 2), -2*m, "veps( 2).veps( 2)= -2m", passed)
  call expect (ueps(-m,p, 1)*veps(-m,p, 1), -2*m, "veps( 1).veps( 1)= -2m", passed)
  call expect (ueps(-m,p,-1)*veps(-m,p,-1), -2*m, "veps(-1).veps(-1)= -2m", passed)
  call expect (ueps(-m,p,-2)*veps(-m,p,-2), -2*m, "veps(-2).veps(-2)= -2m", passed)
  call expect (ueps(-m,p, 2)*ueps(-m,p, 2),    0, "ueps( 2).veps( 2)=   0", passed)
  call expect (ueps(-m,p, 1)*ueps(-m,p, 1),    0, "ueps( 1).veps( 1)=   0", passed)
  call expect (ueps(-m,p,-1)*ueps(-m,p,-1),    0, "ueps(-1).veps(-1)=   0", passed)
  call expect (ueps(-m,p,-2)*ueps(-m,p,-2),    0, "ueps(-2).veps(-2)=   0", passed)
  call expect (veps(-m,p, 2)*veps(-m,p, 2),    0, "veps( 2).ueps( 2)=   0", passed)
  call expect (veps(-m,p, 1)*veps(-m,p, 1),    0, "veps( 1).ueps( 1)=   0", passed)
  call expect (veps(-m,p,-1)*veps(-m,p,-1),    0, "veps(-1).ueps(-1)=   0", passed)
  call expect (veps(-m,p,-2)*veps(-m,p,-2),    0, "veps(-2).ueps(-2)=   0", passed)
  print *, "*** in the rest frame ***"
  call expect (veps(m,p_0, 2)*ueps(m,p_0, 2), -2*m, "ueps( 2).ueps( 2)= -2m", passed)
  call expect (veps(m,p_0, 1)*ueps(m,p_0, 1), -2*m, "ueps( 1).ueps( 1)= -2m", passed)
  call expect (veps(m,p_0,-1)*ueps(m,p_0,-1), -2*m, "ueps(-1).ueps(-1)= -2m", passed)
  call expect (veps(m,p_0,-2)*ueps(m,p_0,-2), -2*m, "ueps(-2).ueps(-2)= -2m", passed)
  call expect (ueps(m,p_0, 2)*veps(m,p_0, 2),  2*m, "veps( 2).veps( 2)= +2m", passed)
  call expect (ueps(m,p_0, 1)*veps(m,p_0, 1),  2*m, "veps( 1).veps( 1)= +2m", passed)
  call expect (ueps(m,p_0,-1)*veps(m,p_0,-1),  2*m, "veps(-1).veps(-1)= +2m", passed)
  call expect (ueps(m,p_0,-2)*veps(m,p_0,-2),  2*m, "veps(-2).veps(-2)= +2m", passed)
  call expect (ueps(m,p_0, 2)*ueps(m,p_0, 2),    0, "ueps( 2).veps( 2)=   0", passed)
  call expect (ueps(m,p_0, 1)*ueps(m,p_0, 1),    0, "ueps( 1).veps( 1)=   0", passed)
  call expect (ueps(m,p_0,-1)*ueps(m,p_0,-1),    0, "ueps(-1).veps(-1)=   0", passed)
  call expect (ueps(m,p_0,-2)*ueps(m,p_0,-2),    0, "ueps(-2).veps(-2)=   0", passed)
  call expect (veps(m,p_0, 2)*veps(m,p_0, 2),    0, "veps( 2).ueps( 2)=   0", passed)
  call expect (veps(m,p_0, 1)*veps(m,p_0, 1),    0, "veps( 1).ueps( 1)=   0", passed)
  call expect (veps(m,p_0,-1)*veps(m,p_0,-1),    0, "veps(-1).ueps(-1)=   0", passed)
  call expect (veps(m,p_0,-2)*veps(m,p_0,-2),    0, "veps(-2).ueps(-2)=   0", passed)
  print *, "*** in the rest frame (neg. masses) ***"
  call expect (veps(-m,p_0, 2)*ueps(-m,p_0, 2), +2*m, "ueps( 2).ueps( 2)= +2m", passed)
  call expect (veps(-m,p_0, 1)*ueps(-m,p_0, 1), +2*m, "ueps( 1).ueps( 1)= +2m", passed)
  call expect (veps(-m,p_0,-1)*ueps(-m,p_0,-1), +2*m, "ueps(-1).ueps(-1)= +2m", passed)
  call expect (veps(-m,p_0,-2)*ueps(-m,p_0,-2), +2*m, "ueps(-2).ueps(-2)= +2m", passed)
  call expect (ueps(-m,p_0, 2)*veps(-m,p_0, 2), -2*m, "veps( 2).veps( 2)= -2m", passed)
  call expect (ueps(-m,p_0, 1)*veps(-m,p_0, 1), -2*m, "veps( 1).veps( 1)= -2m", passed)
```

```
call expect (ueps(-m,p_0,-1)*veps(-m,p_0,-1), -2*m, "veps(-1).veps(-1)= -2m", passed)
call expect (ueps(-m,p_0,-2)*veps(-m,p_0,-2), -2*m, "veps(-2).veps(-2)= -2m", passed)
call expect (ueps(-m,p_0, 2)*ueps(-m,p_0, 2),    0, "ueps( 2).veps( 2)=   0", passed)
call expect (ueps(-m,p_0, 1)*ueps(-m,p_0, 1),    0, "ueps( 1).veps( 1)=   0", passed)
call expect (ueps(-m,p_0,-1)*ueps(-m,p_0,-1),    0, "ueps(-1).veps(-1)=   0", passed)
call expect (ueps(-m,p_0,-2)*ueps(-m,p_0,-2),    0, "ueps(-2).veps(-2)=   0", passed)
call expect (veps(-m,p_0, 2)*veps(-m,p_0, 2),    0, "veps( 2).ueps( 2)=   0", passed)
call expect (veps(-m,p_0, 1)*veps(-m,p_0, 1),    0, "veps( 1).ueps( 1)=   0", passed)
call expect (veps(-m,p_0,-1)*veps(-m,p_0,-1),    0, "veps(-1).ueps(-1)=   0", passed)
call expect (veps(-m,p_0,-2)*veps(-m,p_0,-2),    0, "veps(-2).ueps(-2)=   0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Majorana properties of gravitino vertices: ***"
call expect (abs(u (m,q,1) * f_sgr (c_one, c_one, ueps(m,p,2), t) + &
ueps(m,p,2) * gr_sf(c_one,c_one,u(m,q,1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_sgr (c_one, c_one, ueps(m,p,2), t) + &
!!!    ueps(m,p,2) * gr_sf(c_one,c_one,u(m,q,-1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,1) * f_sgr (c_one, c_one, ueps(m,p,1), t) + &
!!!    ueps(m,p,1) * gr_sf(c_one,c_one,u(m,q,1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_sgr (c_one, c_one, ueps(m,p,1), t) + &
!!!    ueps(m,p,1) * gr_sf(c_one,c_one,u(m,q,-1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,1) * f_sgr (c_one, c_one, ueps(m,p,-1), t) + &
!!!    ueps(m,p,-1) * gr_sf(c_one,c_one,u(m,q,1),t)),  0, "f_sgr   + gr_sf       = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_sgr (c_one, c_one, ueps(m,p,-1), t) + &
!!!    ueps(m,p,-1) * gr_sf(c_one,c_one,u(m,q,-1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,1) * f_sgr (c_one, c_one, ueps(m,p,-2), t) + &
!!!    ueps(m,p,-2) * gr_sf(c_one,c_one,u(m,q,1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_sgr (c_one, c_one, ueps(m,p,-2), t) + &
!!!    ueps(m,p,-2) * gr_sf(c_one,c_one,u(m,q,-1),t)),  0, "f_sgr     + gr_sf     = 0", passed)
call expect (abs(u (m,q,1) * f_slgr (c_one, c_one, ueps(m,p,2), t) + &
ueps(m,p,2) * gr_slf(c_one,c_one,u(m,q,1),t)),  0, "f_slgr    + gr_slf    = 0", passed, threshold = 0.5_default)
call expect (abs(u (m,q,1) * f_srgr (c_one, c_one, ueps(m,p,2), t) + &
ueps(m,p,2) * gr_srf(c_one,c_one,u(m,q,1),t)),  0, "f_srgr    + gr_srf    = 0", passed, threshold = 0.5_default)
call expect (abs(u (m,q,1) * f_slrgr (c_one, c_two, c_one, ueps(m,p,2), t) + &
ueps(m,p,2) * gr_slrf(c_one,c_two,c_one,u(m,q,1),t)),  0, "f_slrgr   + gr_slrf   = 0", passed, threshold
= 0.5_default)
call expect (abs(u (m,q,1) * f_pgr (c_one, c_one, ueps(m,p,2), t) + &
ueps(m,p,2) * gr_pf(c_one,c_one,u(m,q,1),t)),  0, "f_pgr     + gr_pf     = 0", passed, threshold = 0.5_default)
call expect (abs(u (m,q,1) * f_vgr (c_one, vt, ueps(m,p,2), p+q) + &
ueps(m,p,2) * gr_vf(c_one,vt,u(m,q,1),p+q)),  0, "f_vgr     + gr_vf = 0", passed, threshold = 0.5_default)
call expect (abs(u (m,q,1) * f_vlrgr (c_one, c_two, vt, ueps(m,p,2), p+q) + &
ueps(m,p,2) * gr_vlrf(c_one,c_two,vt,u(m,q,1),p+q)),  0, "f_vlrgr   + gr_vlrf   = 0", &
passed, threshold = 0.5_default)
!!! call expect (abs(u (m,q,-1) * f_vgr (c_one, vt, ueps(m,p,2), p+q) + &
!!!    ueps(m,p,2) * gr_vf(c_one,vt,u(m,q,-1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(u (m,q,1) * f_vgr (c_one, vt, ueps(m,p,1), p+q) + &
!!!    ueps(m,p,1) * gr_vf(c_one,vt,u(m,q,1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_vgr (c_one, vt, ueps(m,p,1), p+q) + &
!!!    ueps(m,p,1) * gr_vf(c_one,vt,u(m,q,-1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(u (m,q,1) * f_vgr (c_one, vt, ueps(m,p,-1), p+q) + &
!!!    ueps(m,p,-1) * gr_vf(c_one,vt,u(m,q,1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_vgr (c_one, vt, veps(m,p,-1), p+q) + &
!!!    veps(m,p,-1) * gr_vf(c_one,vt,u(m,q,-1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(v (m,q,1) * f_vgr (c_one, vt, ueps(m,p,-2), p+q) + &
!!!    ueps(m,p,-2) * gr_vf(c_one,vt,v(m,q,1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
!!! call expect (abs(u (m,q,-1) * f_vgr (c_one, vt, ueps(m,p,-2), p+q) + &
!!!    ueps(m,p,-2) * gr_vf(c_one,vt,u(m,q,-1),p+q)),  0, "f_vgr     + gr_vf     = 0", passed)
call expect (abs(s_grf (c_one, ueps(m,p,2), u(m,q,1),t) + &
s_fgr(c_one,u(m,q,1),ueps(m,p,2),t)),  0, "s_grf     + s_fgr     = 0", passed)
call expect (abs(sl_grf (c_one, ueps(m,p,2), u(m,q,1),t) + &
sl_fgr(c_one,u(m,q,1),ueps(m,p,2),t)),  0, "sl_grf    + sl_fgr    = 0", passed)
call expect (abs(sr_grf (c_one, ueps(m,p,2), u(m,q,1),t) + &
sr_fgr(c_one,u(m,q,1),ueps(m,p,2),t)),  0, "sr_grf    + sr_fgr    = 0", passed)
call expect (abs(slr_grf (c_one, c_two, ueps(m,p,2), u(m,q,1),t) + &
slr_fgr(c_one,c_two,u(m,q,1),ueps(m,p,2),t)),  0, "slr_grf   + slr_fgr   = 0", passed)
call expect (abs(p_grf (c_one, ueps(m,p,2), u(m,q,1),t) + &
p_fgr(c_one,u(m,q,1),ueps(m,p,2),t)),  0, "p_grf     + p_fgr     = 0", passed)
```

```
call expect (abs(v_grf (c_one, ueps(m,p,2), u(m,q,1),t) + &
v_fgr(c_one,u(m,q,1),ueps(m,p,2),t)),  0, "v_grf     + v_fgr     = 0", passed)
call expect (abs(vlr_grf (c_one, c_two, ueps(m,p,2), u(m,q,1),t) + &
vlr_fgr(c_one,c_two,u(m,q,1),ueps(m,p,2),t)),  0, "vlr_grf   + vlr_fgr   = 0", passed)
call expect (abs(u(m,p,1) * f_potgr (c_one,c_one,testv) - testv * gr_potf &
(c_one,c_one,u (m,p,1))), 0, "f_potgr   - gr_potf   = 0", passed)
call expect (abs (pot_fgr (c_one,u(m,p,1),testv) - pot_grf(c_one, &
testv,u(m,p,1))), 0, "pot_fgr   - pot_grf   = 0", passed)
call expect (abs(u(m,p,1) * f_s2gr (c_one,c_one,c_one,testv) - testv * gr_s2f &
(c_one,c_one,c_one,u (m,p,1))), 0, "f_s2gr    - gr_s2f    = 0", passed)
call expect (abs (s2_fgr (c_one,u(m,p,1),c_one,testv) - s2_grf(c_one, &
testv,c_one,u(m,p,1))), 0, "s2_fgr    - s2_grf    = 0", passed)
call expect (abs(u (m,q,1) * f_svgr (c_one, c_one, vt, ueps(m,p,2)) + &
ueps(m,p,2) * gr_svf(c_one,c_one,vt,u(m,q,1))),  0, "f_svgr    + gr_svf    = 0", passed)
call expect (abs(u (m,q,1) * f_slvgr (c_one, c_one, vt, ueps(m,p,2)) + &
ueps(m,p,2) * gr_slvf(c_one,c_one,vt,u(m,q,1))),  0, "f_slvgr   + gr_slvf   = 0", passed)
call expect (abs(u (m,q,1) * f_srvgr (c_one, c_one, vt, ueps(m,p,2)) + &
ueps(m,p,2) * gr_srvf(c_one,c_one,vt,u(m,q,1))),  0, "f_srvgr   + gr_srvf   = 0", passed)
call expect (abs(u (m,q,1) * f_slrvgr (c_one, c_two, c_one, vt, ueps(m,p,2)) + &
ueps(m,p,2) * gr_slrvf(c_one,c_two,c_one,vt,u(m,q,1))),  0, "f_slrvgr  + gr_slrvf  = 0", passed)
call expect (abs (sv1_fgr (c_one,u(m,p,1),vt,ueps(m,q,2)) + sv1_grf(c_one, &
ueps(m,q,2),vt,u(m,p,1))), 0, "sv1_fgr   + sv1_grf   = 0", passed)
call expect (abs (sv2_fgr (c_one,u(m,p,1),c_one,ueps(m,q,2)) + sv2_grf(c_one, &
ueps(m,q,2),c_one,u(m,p,1))), 0, "sv2_fgr   + sv2_grf   = 0", passed)
call expect (abs (slv1_fgr (c_one,u(m,p,1),vt,ueps(m,q,2)) + slv1_grf(c_one, &
ueps(m,q,2),vt,u(m,p,1))), 0, "slv1_fgr  + slv1_grf  = 0", passed)
call expect (abs (srv2_fgr (c_one,u(m,p,1),c_one,ueps(m,q,2)) + srv2_grf(c_one, &
ueps(m,q,2),c_one,u(m,p,1))), 0, "srv2_fgr  + srv2_grf  = 0", passed)
call expect (abs (slrv1_fgr (c_one,c_two,u(m,p,1),vt,ueps(m,q,2)) + slrv1_grf(c_one,c_two, &
ueps(m,q,2),vt,u(m,p,1))), 0, "slrv1_fgr + slrv1_grf = 0", passed)
call expect (abs (slrv2_fgr (c_one,c_two,u(m,p,1),c_one,ueps(m,q,2)) + slrv2_grf(c_one, &
c_two,ueps(m,q,2),c_one,u(m,p,1))), 0, "slrv2_fgr + slrv2_grf = 0", passed)
call expect (abs(u (m,q,1) * f_pvgr (c_one, c_one, vt, ueps(m,p,2)) + &
ueps(m,p,2) * gr_pvf(c_one,c_one,vt,u(m,q,1))),  0, "f_pvgr    + gr_pvf    = 0", passed)
call expect (abs (pv1_fgr (c_one,u(m,p,1),vt,ueps(m,q,2)) + pv1_grf(c_one, &
ueps(m,q,2),vt,u(m,p,1))), 0, "pv1_fgr   + pv1_grf   = 0", passed)
call expect (abs (pv2_fgr (c_one,u(m,p,1),c_one,ueps(m,q,2)) + pv2_grf(c_one, &
ueps(m,q,2),c_one,u(m,p,1))), 0, "pv2_fgr   + pv2_grf   = 0", passed)
call expect (abs(u (m,q,1) * f_v2gr (c_one, vt, vz, ueps(m,p,2)) + &
ueps(m,p,2) * gr_v2f(c_one,vt,vz,u(m,q,1))),  0, "f_v2gr    + gr_v2f    = 0", passed)
call expect (abs(u (m,q,1) * f_v2lrgr (c_one, c_two, vt, vz, ueps(m,p,2)) + &
ueps(m,p,2) * gr_v2lrf(c_one,c_two,vt,vz,u(m,q,1))),  0, "f_v2lrgr  + gr_v2lrf  = 0", passed)
call expect (abs (v2_fgr (c_one,u(m,p,1),vt,ueps(m,q,2)) + v2_grf(c_one, &
ueps(m,q,2),vt,u(m,p,1))), 0, "v2_fgr    + v2_grf    = 0", passed)
call expect (abs (v2lr_fgr (c_one,c_two,u(m,p,1),vt,ueps(m,q,2)) + v2lr_grf(c_one, c_two, &
ueps(m,q,2),vt,u(m,p,1))), 0, "v2lr_fgr  + v2lr_grf  = 0", passed)
```

⟨*Test* omega95_bispinors⟩+≡

```
print *, "*** Testing the gravitino propagator: ***"
print *, "Transversality:"
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,testv))), 0, "p.pr.test", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,ueps(m,p,2)))),  0, "p.pr.ueps ( 2)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,ueps(m,p,1)))),  0, "p.pr.ueps ( 1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,ueps(m,p,-1)))), 0, "p.pr.ueps (-1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,ueps(m,p,-2)))), 0, "p.pr.ueps (-2)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,veps(m,p,2)))),  0, "p.pr.veps ( 2)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,veps(m,p,1)))),  0, "p.pr.veps ( 1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,veps(m,p,-1)))), 0, "p.pr.veps (-1)", passed)
call expect (abs(p * (cmplx (p*p - m**2, m*w, kind=default) * &
pr_grav(p,m,w,veps(m,p,-2)))), 0, "p.pr.veps (-2)", passed)
print *, "Irreducibility:"
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,testv)))), 0, "g.pr.test", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,ueps(m,p,2))))), 0, &
"g.pr.ueps ( 2)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,ueps(m,p,1))))), 0, &
"g.pr.ueps ( 1)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,ueps(m,p,-1))))), 0, &
"g.pr.ueps (-1)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,ueps(m,p,-2))))), 0, &
"g.pr.ueps (-2)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,veps(m,p,2))))), 0, &
"g.pr.veps ( 2)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,veps(m,p,1))))), 0, &
"g.pr.veps ( 1)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,veps(m,p,-1))))), 0, &
"g.pr.veps (-1)", passed)
call expect (abs(f_potgr (c_one, c_one, (cmplx (p*p - m**2, m*w, &
kind=default) * pr_grav(p,m,w,veps(m,p,-2))))), 0, &
"g.pr.veps (-2)", passed)
```

```
⟨omega_bundle.f90⟩≡
  ⟨omega_vectors.f90⟩
  ⟨omega_spinors.f90⟩
  ⟨omega_bispinors.f90⟩
  ⟨omega_vectorspinors.f90⟩
  ⟨omega_polarizations.f90⟩
  ⟨omega_tensors.f90⟩
  ⟨omega_tensor_polarizations.f90⟩
  ⟨omega_couplings.f90⟩
  ⟨omega_spinor_couplings.f90⟩
  ⟨omega_bispinor_couplings.f90⟩
  ⟨omega_vspinor_polarizations.f90⟩
  ⟨omega_utils.f90⟩
  ⟨omega95.f90⟩
  ⟨omega95_bispinors.f90⟩
  ⟨omega_parameters.f90⟩
  ⟨omega_parameters_madgraph.f90⟩
```

```
⟨omega_bundle_whizard.f90⟩≡
  ⟨omega_bundle.f90⟩
  ⟨omega_parameters_whizard.f90⟩
```

AB.33 O'Mega Virtual Machine

This module defines the O'Mega Virtual Machine (OVM) completely, whereby all environmental dependencies like masses, widths and couplings have to be given to the constructor `vm%init` at runtime.

Support for Majorana particles and vectorspinors is only partially, especially all fusions are missing. Maybe it would be easier to make an additional `omegavm95_bispinors` to avoid namespace issues. Non-type specific chunks could be reused

```
⟨omegavm95.f90⟩≡
  ⟨Copyleft⟩
  module omegavm95
  use kinds, only: default
  use constants
  use iso_varying_string, string_t => varying_string
  use, intrinsic :: iso_fortran_env, only : input_unit, output_unit, error_unit
```

```
use omega95
use omega95_bispinors, only: bispinor, vectorspinor, veps, pr_grav
use omega95_bispinors, only: bi_u => u
use omega95_bispinors, only: bi_v => v
use omega95_bispinors, only: bi_pr_psi => pr_psi
use omega_bispinors, only: operator (*), operator (+)
use omega_color, only: ovm_color_sum, OCF => omega_color_factor
implicit none
private
⟨Utilities Declarations⟩
⟨OVM Data Declarations⟩
⟨OVM Instructions⟩
contains
⟨OVM Procedure Implementations⟩
⟨Utilities Procedure Implementations⟩
end module omegavm95
```

This might not be the proper place but I don't know where to put it

⟨Utilities Declarations⟩≡

```
integer, parameter, public :: stdin = input_unit
integer, parameter, public :: stdout = output_unit
integer, parameter, public :: stderr = error_unit
integer, parameter :: MIN_UNIT = 11, MAX_UNIT = 99
```

⟨OVM Procedure Implementations⟩≡

```
subroutine find_free_unit (u, iostat)
integer, intent(out) :: u
integer, intent(out), optional :: iostat
logical :: exists, is_open
integer :: i, status
do i = MIN_UNIT, MAX_UNIT
inquire (unit = i, exist = exists, opened = is_open, &
iostat = status)
if (status == 0) then
if (exists .and. .not. is_open) then
u = i
if (present (iostat)) then
iostat = 0
end if
return
end if
end if
end do
if (present (iostat)) then
iostat = -1
end if
u = -1
end subroutine find_free_unit
```

These abstract data types would ideally be the interface to communicate quantum numbers between O'Mega and Whizard. This gives full flexibility to change the representation at any time

⟨Utilities Declarations⟩+≡

```
public :: color_t
type color_t
contains
procedure :: write => color_write
end type color_t

public :: col_discrete
type, extends(color_t) :: col_discrete
integer :: i
end type col_discrete

public :: flavor_t
type flavor_t
contains
procedure :: write => flavor_write
end type flavor_t
```

```
public :: flv_discrete
type, extends(flavor_t) :: flv_discrete
integer :: i
end type flv_discrete

public :: helicity_t
type :: helicity_t
contains
procedure :: write => helicity_write
end type helicity_t

public :: hel_discrete
type, extends(helicity_t) :: hel_discrete
integer :: i
end type hel_discrete

public :: hel_trigonometric
type, extends(helicity_t) :: hel_trigonometric
real :: theta
end type hel_trigonometric

public :: hel_exponential
type, extends(helicity_t) :: hel_exponential
real :: phi
end type hel_exponential

public :: hel_spherical
type, extends(helicity_t) :: hel_spherical
real :: theta, phi
end type hel_spherical
```

⟨*Utilities Procedure Implementations*⟩≡

```
subroutine color_write (color, fh)
class(color_t), intent(in) :: color
integer, intent(in) :: fh
select type(color)
type is (col_discrete)
write(fh, *) 'color_discrete%i           = ', color%i
end select
end subroutine color_write

subroutine helicity_write (helicity, fh)
class(helicity_t), intent(in) :: helicity
integer, intent(in) :: fh
select type(helicity)
type is (hel_discrete)
write(fh, *) 'helicity_discrete%i           = ', helicity%i
type is (hel_trigonometric)
write(fh, *) 'helicity_trigonometric%theta  = ', helicity%theta
type is (hel_exponential)
write(fh, *) 'helicity_exponential%phi      = ', helicity%phi
type is (hel_spherical)
write(fh, *) 'helicity_spherical%phi        = ', helicity%phi
write(fh, *) 'helicity_spherical%theta      = ', helicity%theta
end select
end subroutine helicity_write

subroutine flavor_write (flavor, fh)
class(flavor_t), intent(in) :: flavor
integer, intent(in) :: fh
select type(flavor)
type is (flv_discrete)
write(fh, *) 'flavor_discrete%i           = ', flavor%i
end select
end subroutine flavor_write
```

AB.33.1 Memory Layout

Some internal parameters

⟨*OVM Data Declarations*⟩≡
```
integer, parameter :: len_instructions = 8
integer, parameter :: N_version_lines = 2
! Comment lines including the first header description line
integer, parameter :: N_comments = 6
! Actual data lines plus intermediate description lines
! 'description \n 1 2 3 \n description \n 3 2 1' would count as 3
integer, parameter :: N_header_lines = 5
real(default), parameter, public :: N_ = three
```

This is the basic type of a VM

⟨*OVM Data Declarations*⟩+≡
```
type :: basic_vm_t
private
logical :: verbose
type(string_t) :: bytecode_file
integer :: bytecode_fh, out_fh
integer :: N_instructions, N_levels
integer :: N_table_lines
integer, dimension(:, :), allocatable :: instructions
integer, dimension(:), allocatable :: levels
end type
```

To allow for a lazy evaluation of amplitudes, we have to keep track whether a wave function has already been computed, to avoid multiple-computing that would arise when the bytecode has redundant fusions, which is necessary for flavor and color MC (and helicity MC when we use Weyl-van-der-Waerden-spinors)

⟨*OVM Data Declarations*⟩+≡
```
type :: vm_scalar
logical :: c
complex(kind=default) :: v
end type

type :: vm_spinor
logical :: c
type(spinor) :: v
end type

type :: vm_conjspinor
logical :: c
type(conjspinor) :: v
end type

type :: vm_bispinor
logical :: c
type(bispinor) :: v
end type

type :: vm_vector
logical :: c
type(vector) :: v
end type

type :: vm_tensor_2
logical :: c
type(tensor) :: v
end type

type :: vm_tensor_1
logical :: c
```

```
type(tensor2odd) :: v
end type

type :: vm_vectorspinor
logical :: c
type(vectorspinor) :: v
end type
```

We need a memory pool for all the intermediate results

⟨*OVM Data Declarations*⟩+≡

```
type, public, extends (basic_vm_t) :: vm_t
private
type(string_t) :: version
type(string_t) :: model
integer :: N_momenta, N_particles, N_prt_in, N_prt_out, N_amplitudes
! helicities = helicity combinations
integer :: N_helicities, N_col_flows, N_col_indices, N_flavors, N_col_factors

integer :: N_scalars, N_spinors, N_conjspinors, N_bispinors
integer :: N_vectors, N_tensors_2, N_tensors_1, N_vectorspinors

integer :: N_coupl_real, N_coupl_real2, N_coupl_cmplx, N_coupl_cmplx2

integer, dimension(:, :), allocatable :: table_flavor
integer, dimension(:, :, :), allocatable :: table_color_flows
integer, dimension(:, :), allocatable :: table_spin
logical, dimension(:, :), allocatable :: table_ghost_flags
type(OCF), dimension(:), allocatable :: table_color_factors
logical, dimension(:, :), allocatable :: table_flv_col_is_allowed

real(default), dimension(:), allocatable :: coupl_real
real(default), dimension(:, :), allocatable :: coupl_real2
complex(default), dimension(:), allocatable :: coupl_cmplx
complex(default), dimension(:, :), allocatable :: coupl_cmplx2
real(default), dimension(:), allocatable :: mass
real(default), dimension(:), allocatable :: width

type(momentum), dimension(:), allocatable :: momenta
complex(default), dimension(:), allocatable :: amplitudes
complex(default), dimension(:, :, :), allocatable :: table_amplitudes
class(flavor_t), dimension(:), allocatable :: flavor
class(color_t), dimension(:), allocatable :: color
! gfortran 4.7
!class(helicity_t), dimension(:), pointer :: helicity => null()
integer, dimension(:), allocatable :: helicity

type(vm_scalar), dimension(:), allocatable :: scalars
type(vm_spinor), dimension(:), allocatable :: spinors
type(vm_conjspinor), dimension(:), allocatable :: conjspinors
type(vm_bispinor), dimension(:), allocatable :: bispinors
type(vm_vector), dimension(:), allocatable :: vectors
type(vm_tensor_2), dimension(:), allocatable :: tensors_2
type(vm_tensor_1), dimension(:), allocatable :: tensors_1
type(vm_vectorspinor), dimension(:), allocatable :: vectorspinors

logical, dimension(:), allocatable :: hel_is_allowed
real(default), dimension(:), allocatable :: hel_max_abs
real(default) :: hel_sum_abs = 0, hel_threshold = 1E10
integer :: hel_count = 0, hel_cutoff = 100
integer, dimension(:), allocatable :: hel_map
integer :: hel_finite
logical :: cms

logical :: openmp

contains
```

```
⟨VM: TBP⟩
end type
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine alloc_arrays (vm)
type(vm_t), intent(inout) :: vm
integer :: i
allocate (vm%table_flavor(vm%N_particles, vm%N_flavors))
allocate (vm%table_color_flows(vm%N_col_indices, vm%N_particles, &
vm%N_col_flows))
allocate (vm%table_spin(vm%N_particles, vm%N_helicities))
allocate (vm%table_ghost_flags(vm%N_particles, vm%N_col_flows))
allocate (vm%table_color_factors(vm%N_col_factors))
allocate (vm%table_flv_col_is_allowed(vm%N_flavors, vm%N_col_flows))
allocate (vm%momenta(vm%N_momenta))
allocate (vm%amplitudes(vm%N_amplitudes))
allocate (vm%table_amplitudes(vm%N_flavors, vm%N_col_flows, &
vm%N_helicities))
vm%table_amplitudes = zero
allocate (vm%scalars(vm%N_scalars))
allocate (vm%spinors(vm%N_spinors))
allocate (vm%conjspinors(vm%N_conjspinors))
allocate (vm%bispinors(vm%N_bispinors))
allocate (vm%vectors(vm%N_vectors))
allocate (vm%tensors_2(vm%N_tensors_2))
allocate (vm%tensors_1(vm%N_tensors_1))
allocate (vm%vectorspinors(vm%N_vectorspinors))
allocate (vm%hel_is_allowed(vm%N_helicities))
vm%hel_is_allowed = .True.
allocate (vm%hel_max_abs(vm%N_helicities))
vm%hel_max_abs = 0
allocate (vm%hel_map(vm%N_helicities))
vm%hel_map = (/(i, i = 1, vm%N_helicities)/)
vm%hel_finite = vm%N_helicities
end subroutine alloc_arrays
```

AB.33.2 Controlling the VM

These type-bound procedures steer the VM

⟨*VM: TBP*⟩≡

```
procedure :: init => vm_init
procedure :: write => vm_write
procedure :: reset => vm_reset
procedure :: run => vm_run
procedure :: final => vm_final
```

The `init` completely sets the environment for the OVM. Parameters can be changed with `reset` without reloading the bytecode.

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_init (vm, bytecode_file, version, model, &
coupl_real, coupl_real2, coupl_cmplx, coupl_cmplx2, &
mass, width, verbose, out_fh, openmp)
class(vm_t), intent(out) :: vm
type(string_t), intent(in) :: bytecode_file
type(string_t), intent(in) :: version
type(string_t), intent(in) :: model
real(default), dimension(:), optional, intent(in) :: coupl_real
real(default), dimension(:, :), optional, intent(in) :: coupl_real2
complex(default), dimension(:), optional, intent(in) :: coupl_cmplx
complex(default), dimension(:, :), optional, intent(in) :: coupl_cmplx2
real(default), dimension(:), optional, intent(in) :: mass
real(default), dimension(:), optional, intent(in) :: width
logical, optional, intent(in) :: verbose
integer, optional, intent(in) :: out_fh
```

```
logical, optional, intent(in) :: openmp
vm%bytecode_file = bytecode_file
vm%version = version
vm%model = model
if (present (coupl_real)) then
allocate (vm%coupl_real (size (coupl_real)), source=coupl_real)
end if
if (present (coupl_real2)) then
allocate (vm%coupl_real2 (2, size (coupl_real2, 2)), source=coupl_real2)
end if
if (present (coupl_cmplx)) then
allocate (vm%coupl_cmplx (size (coupl_cmplx)), source=coupl_cmplx)
end if
if (present (coupl_cmplx2)) then
allocate (vm%coupl_cmplx2 (2, size (coupl_cmplx2, 2)), &
source=coupl_cmplx2)
end if
if (present (mass)) then
allocate (vm%mass(size(mass)), source=mass)
end if
if (present (width)) then
allocate (vm%width(size (width)), source=width)
end if
if (present (openmp)) then
vm%openmp = openmp
else
vm%openmp = .false.
end if
vm%cms = .false.

call basic_init (vm, verbose, out_fh)
end subroutine vm_init
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_reset (vm, &
coupl_real, coupl_real2, coupl_cmplx, coupl_cmplx2, &
mass, width, verbose, out_fh)
class(vm_t), intent(inout) :: vm
real(default), dimension(:), optional, intent(in) :: coupl_real
real(default), dimension(:, :), optional, intent(in) :: coupl_real2
complex(default), dimension(:), optional, intent(in) :: coupl_cmplx
complex(default), dimension(:, :), optional, intent(in) :: coupl_cmplx2
real(default), dimension(:), optional, intent(in) :: mass
real(default), dimension(:), optional, intent(in) :: width
logical, optional, intent(in) :: verbose
integer, optional, intent(in) :: out_fh
if (present (coupl_real)) then
vm%coupl_real = coupl_real
end if
if (present (coupl_real2)) then
vm%coupl_real2 = coupl_real2
end if
if (present (coupl_cmplx)) then
vm%coupl_cmplx = coupl_cmplx
end if
if (present (coupl_cmplx2)) then
vm%coupl_cmplx2 = coupl_cmplx2
end if
if (present (mass)) then
vm%mass = mass
end if
if (present (width)) then
vm%width = width
end if
if (present (verbose)) then
vm%verbose = verbose
```

```
end if
if (present (out_fh)) then
vm%out_fh = out_fh
end if
end subroutine vm_reset
```

Mainly for debugging

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_write (vm)
class(vm_t), intent(in) :: vm
integer :: i, j, k
call basic_write (vm)
write(vm%out_fh, *) 'table_flavor              = ', vm%table_flavor
write(vm%out_fh, *) 'table_color_flows         = ', vm%table_color_flows
write(vm%out_fh, *) 'table_spin                = ', vm%table_spin
write(vm%out_fh, *) 'table_ghost_flags         = ', vm%table_ghost_flags
write(vm%out_fh, *) 'table_color_factors       = '
do i = 1, size(vm%table_color_factors)
write(vm%out_fh, *)  vm%table_color_factors(i)%i1, &
vm%table_color_factors(i)%i2, &
vm%table_color_factors(i)%factor
end do

write(vm%out_fh, *) 'table_flv_col_is_allowed  = ', &
vm%table_flv_col_is_allowed
do i = 1, vm%N_flavors
do j = 1, vm%N_col_flows
do k = 1, vm%N_helicities
write(vm%out_fh, *) 'table_amplitudes(f,c,h), f, c, h = ', vm%table_amplitudes(i,j,k), i, j, k
end do
end do
end do
if (allocated(vm%coupl_real)) then
write(vm%out_fh, *) 'coupl_real          = ', vm%coupl_real
end if
if (allocated(vm%coupl_real2)) then
write(vm%out_fh, *) 'coupl_real2         = ', vm%coupl_real2
end if
if (allocated(vm%coupl_cmplx)) then
write(vm%out_fh, *) 'coupl_cmplx         = ', vm%coupl_cmplx
end if
if (allocated(vm%coupl_cmplx2)) then
write(vm%out_fh, *) 'coupl_cmplx2        = ', vm%coupl_cmplx2
end if
write(vm%out_fh, *) 'mass                = ', vm%mass
write(vm%out_fh, *) 'width               = ', vm%width
write(vm%out_fh, *) 'momenta             = ', vm%momenta
! gfortran 4.7
!do i = 1, size(vm%flavor)
!call vm%flavor(i)%write (vm%out_fh)
!end do
!do i = 1, size(vm%color)
!call vm%color(i)%write (vm%out_fh)
!end do
!do i = 1, size(vm%helicity)
!call vm%helicity(i)%write (vm%out_fh)
!end do
write(vm%out_fh, *) 'helicity            = ', vm%helicity
write(vm%out_fh, *) 'amplitudes        = ', vm%amplitudes
write(vm%out_fh, *) 'scalars       = ', vm%scalars
write(vm%out_fh, *) 'spinors       = ', vm%spinors
write(vm%out_fh, *) 'conjspinors   = ', vm%conjspinors
write(vm%out_fh, *) 'bispinors     = ', vm%bispinors
write(vm%out_fh, *) 'vectors       = ', vm%vectors
write(vm%out_fh, *) 'tensors_2     = ', vm%tensors_2
write(vm%out_fh, *) 'tensors_1     = ', vm%tensors_1
```

```
!!! !!! !!! Regression with ifort 16.0.0
!!! write(vm%out_fh, *) 'vectorspinors = ', vm%vectorspinors
write(vm%out_fh, *) 'N_momenta       = ', vm%N_momenta
write(vm%out_fh, *) 'N_particles     = ', vm%N_particles
write(vm%out_fh, *) 'N_prt_in        = ', vm%N_prt_in
write(vm%out_fh, *) 'N_prt_out       = ', vm%N_prt_out
write(vm%out_fh, *) 'N_amplitudes    = ', vm%N_amplitudes
write(vm%out_fh, *) 'N_helicities    = ', vm%N_helicities
write(vm%out_fh, *) 'N_col_flows     = ', vm%N_col_flows
write(vm%out_fh, *) 'N_col_indices   = ', vm%N_col_indices
write(vm%out_fh, *) 'N_flavors       = ', vm%N_flavors
write(vm%out_fh, *) 'N_col_factors   = ', vm%N_col_factors
write(vm%out_fh, *) 'N_scalars       = ', vm%N_scalars
write(vm%out_fh, *) 'N_spinors       = ', vm%N_spinors
write(vm%out_fh, *) 'N_conjspinors   = ', vm%N_conjspinors
write(vm%out_fh, *) 'N_bispinors     = ', vm%N_bispinors
write(vm%out_fh, *) 'N_vectors       = ', vm%N_vectors
write(vm%out_fh, *) 'N_tensors_2     = ', vm%N_tensors_2
write(vm%out_fh, *) 'N_tensors_1     = ', vm%N_tensors_1
write(vm%out_fh, *) 'N_vectorspinors = ', vm%N_vectorspinors
write(vm%out_fh, *) 'Overall size of VM: '
! GNU extension
! write(vm%out_fh, *) 'sizeof(wavefunctions) = ', &
!   sizeof(vm%scalars) + sizeof(vm%spinors) + sizeof(vm%conjspinors) + &
!   sizeof(vm%bispinors) + sizeof(vm%vectors) + sizeof(vm%tensors_2) + &
!   sizeof(vm%tensors_1) +  sizeof(vm%vectorspinors)
! write(vm%out_fh, *) 'sizeof(mometa) = ', sizeof(vm%momenta)
! write(vm%out_fh, *) 'sizeof(amplitudes) = ', sizeof(vm%amplitudes)
! write(vm%out_fh, *) 'sizeof(tables) = ', &
!   sizeof(vm%table_amplitudes) + sizeof(vm%table_spin) + &
!   sizeof(vm%table_flavor) + sizeof(vm%table_flv_col_is_allowed) + &
!   sizeof(vm%table_color_flows) + sizeof(vm%table_color_factors) + &
!   sizeof(vm%table_ghost_flags)
end subroutine vm_write
```

Most of this is redundant (Fortran will deallocate when we leave the scope) but when we change from `allocatables` to `pointers`, it is necessary to avoid leaks

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_final (vm)
class(vm_t), intent(inout) :: vm
deallocate (vm%table_flavor)
deallocate (vm%table_color_flows)
deallocate (vm%table_spin)
deallocate (vm%table_ghost_flags)
deallocate (vm%table_color_factors)
deallocate (vm%table_flv_col_is_allowed)
if (allocated (vm%coupl_real)) then
deallocate (vm%coupl_real)
end if
if (allocated (vm%coupl_real2)) then
deallocate (vm%coupl_real2)
end if
if (allocated (vm%coupl_cmplx)) then
deallocate (vm%coupl_cmplx)
end if
if (allocated (vm%coupl_cmplx2)) then
deallocate (vm%coupl_cmplx2)
end if
if (allocated (vm%mass)) then
deallocate (vm%mass)
end if
if (allocated (vm%width)) then
deallocate (vm%width)
end if
deallocate (vm%momenta)
deallocate (vm%flavor)
deallocate (vm%color)
deallocate (vm%helicity)
deallocate (vm%amplitudes)
deallocate (vm%table_amplitudes)
deallocate (vm%scalars)
deallocate (vm%spinors)
deallocate (vm%conjspinors)
deallocate (vm%bispinors)
deallocate (vm%vectors)
deallocate (vm%tensors_2)
deallocate (vm%tensors_1)
deallocate (vm%vectorspinors)
end subroutine vm_final
```

Handing over the polymorph object helicity didn't work out as planned. A work-around is the use of `pointer`s. `flavor` and `color` are not yet used but would have to be changed to `pointer`s as well. At least this potentially avoids copying. Actually, neither the allocatable nor the pointer version works in `gfortran 4.7` due to the broken `select type`. Back to Stone Age, i.e. integers.

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_run (vm, mom, flavor, color, helicity)
class(vm_t), intent(inout) :: vm
real(default), dimension(0:3, *), intent(in) :: mom
class(flavor_t), dimension(:), optional, intent(in) :: flavor
class(color_t), dimension(:), optional, intent(in) :: color
! gfortran 4.7
!class(helicity_t), dimension(:), optional, target, intent(in) :: helicity
integer, dimension(:), optional, intent(in) :: helicity
integer :: i, h, hi
do i = 1, vm%N_particles
if (i <= vm%N_prt_in) then
vm%momenta(i) = - mom(:, i)          ! incoming, crossing symmetry
else
vm%momenta(i) = mom(:, i)            ! outgoing
end if
end do
if (present (flavor)) then
allocate(vm%flavor(size(flavor)), source=flavor)
else
if (.not. (allocated (vm%flavor))) then
allocate(flv_discrete::vm%flavor(vm%N_particles))
end if
end if
if (present (color)) then
allocate(vm%color(size(color)), source=color)
else
if (.not. (allocated (vm%color))) then
allocate(col_discrete::vm%color(vm%N_col_flows))
end if
end if
! gfortran 4.7
if (present (helicity)) then
!vm%helicity => helicity
vm%helicity = helicity
call vm_run_one_helicity (vm, 1)
else
!if (.not. (associated (vm%helicity))) then
!allocate(hel_discrete::vm%helicity(vm%N_particles))
!end if
if (.not. (allocated (vm%helicity))) then
allocate(vm%helicity(vm%N_particles))
end if
if (vm%hel_finite == 0) return
do hi = 1, vm%hel_finite
h = vm%hel_map(hi)
!<Work around [[gfortran 4.7 Bug 56731]] Implementation»
```

```
vm%helicity = vm%table_spin(:,h)
call vm_run_one_helicity (vm, h)
end do
end if
end subroutine vm_run
```

This only removes the `ICE` but still leads to a segmentation fault in `gfortran 4.7`. I am running out of ideas how to make this compiler work with arrays of polymorph datatypes.

⟨*Work around* `gfortran 4.7 Bug 56731` *Declarations*⟩≡

```
integer :: hj
```

⟨*Work around* `gfortran 4.7 Bug 56731` *Implementation*⟩≡

```
do hj = 1, size(vm%helicity)
select type (hel => vm%helicity(hj))
type is (hel_discrete)
hel%i = vm%table_spin(hj,h)
end select
end do
```

⟨*Original version*⟩≡

```
select type (hel => vm%helicity)
type is (hel_discrete)
hel(:)%i = vm%table_spin(:,h)
end select
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine vm_run_one_helicity (vm, h)
class(vm_t), intent(inout) :: vm
integer, intent(in) :: h
integer :: f, c, i
vm%amplitudes = zero
if (vm%N_levels > 0) then
call null_all_wfs (vm)
call iterate_instructions (vm)
end if
i = 1
do c = 1, vm%N_col_flows
do f = 1, vm%N_flavors
if (vm%table_flv_col_is_allowed(f,c)) then
vm%table_amplitudes(f,c,h) = vm%amplitudes(i)
i = i + 1
end if
end do
end do
end subroutine
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine null_all_wfs (vm)
type(vm_t), intent(inout) :: vm
integer :: i, j
vm%scalars%c = .False.
vm%scalars%v = zero
vm%spinors%c = .False.
vm%conjspinors%c = .False.
vm%bispinors%c = .False.
vm%vectorspinors%c = .False.
do i = 1, 4
vm%spinors%v%a(i) = zero
vm%conjspinors%v%a(i) = zero
vm%bispinors%v%a(i) = zero
do j = 1, 4
vm%vectorspinors%v%psi(i)%a(j) = zero
end do
end do
vm%vectors%c = .False.
vm%vectors%v%t = zero
vm%tensors_1%c = .False.
```

```
vm%tensors_2%c = .False.
do i = 1, 3
vm%vectors%v%x(i) = zero
vm%tensors_1%v%e(i) = zero
vm%tensors_1%v%b(i) = zero
do j = 1, 3
vm%tensors_2%v%t(i,j) = zero
end do
end do
end subroutine
```

AB.33.3 Reading the bytecode

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine load_header (vm, IO)
type(vm_t), intent(inout) :: vm
integer, intent(inout) :: IO
integer, dimension(len_instructions) :: line
read(vm%bytecode_fh, fmt = *, iostat = IO) line
vm%N_momenta = line(1)
vm%N_particles = line(2)
vm%N_prt_in = line(3)
vm%N_prt_out = line(4)
vm%N_amplitudes = line(5)
vm%N_helicities = line(6)
vm%N_col_flows = line(7)
if (vm%N_momenta == 0) then
vm%N_col_indices = 2
else
vm%N_col_indices = line(8)
end if
read(vm%bytecode_fh, fmt = *, iostat = IO)
read(vm%bytecode_fh, fmt = *, iostat = IO) line
vm%N_flavors = line(1)
vm%N_col_factors = line(2)
vm%N_scalars = line(3)
vm%N_spinors = line(4)
vm%N_conjspinors = line(5)
vm%N_bispinors = line(6)
vm%N_vectors = line(7)
vm%N_tensors_2 = line(8)
read(vm%bytecode_fh, fmt = *, iostat = IO)
read(vm%bytecode_fh, fmt = *, iostat = IO) line
vm%N_tensors_1 = line(1)
vm%N_vectorspinors = line(2)
! Add 1 for seperating label lines like 'Another table'
vm%N_table_lines = vm%N_helicities + 1 + vm%N_flavors + 1 + vm%N_col_flows &
+ 1 + vm%N_col_flows + 1 + vm%N_col_factors + 1 + vm%N_col_flows
end subroutine load_header
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine read_tables (vm, IO)
type(vm_t), intent(inout) :: vm
integer, intent(inout) :: IO
integer :: i
integer, dimension(2) :: tmpcf
integer, dimension(3) :: tmpfactor
integer, dimension(vm%N_flavors) :: tmpF
integer, dimension(vm%N_particles) :: tmpP
real(default) :: factor
do i = 1, vm%N_helicities
read(vm%bytecode_fh, fmt = *, iostat = IO) vm%table_spin(:, i)
end do
```

```
read(vm%bytecode_fh, fmt = *, iostat = IO)
do i = 1, vm%N_flavors
read(vm%bytecode_fh, fmt = *, iostat = IO) vm%table_flavor(:, i)
end do

read(vm%bytecode_fh, fmt = *, iostat = IO)
do i = 1, vm%N_col_flows
read(vm%bytecode_fh, fmt = *, iostat = IO) vm%table_color_flows(:, :, i)
end do

read(vm%bytecode_fh, fmt = *, iostat = IO)
do i = 1, vm%N_col_flows
read(vm%bytecode_fh, fmt = *, iostat = IO) tmpP
vm%table_ghost_flags(:, i) = int_to_log(tmpP)
end do

read(vm%bytecode_fh, fmt = *, iostat = IO)
do i = 1, vm%N_col_factors
read(vm%bytecode_fh, fmt = '(2I9)', iostat = IO, advance='no') tmpcf
factor = zero
do
read(vm%bytecode_fh, fmt = '(3I9)', iostat = IO, advance='no', EOR=10) tmpfactor
factor = factor + color_factor(tmpfactor(1), tmpfactor(2), tmpfactor(3))
end do
10 vm%table_color_factors(i) = OCF(tmpcf(1), tmpcf(2), factor)
end do

read(vm%bytecode_fh, fmt = *, iostat = IO)
do i = 1, vm%N_col_flows
read(vm%bytecode_fh, fmt = *, iostat = IO) tmpF
vm%table_flv_col_is_allowed(:, i) = int_to_log(tmpF)
end do
end subroutine read_tables
```

This checking has proven useful more than once

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine extended_version_check (vm, IO)
type(vm_t), intent(in) :: vm
integer, intent(inout) :: IO
character(256) :: buffer
read(vm%bytecode_fh, fmt = "(A)", iostat = IO) buffer
if (vm%version /= buffer) then
print *, "Warning: Bytecode has been generated with an older O'Mega version."
else
if (vm%verbose) then
write (vm%out_fh, fmt = *) "Bytecode version fits."
end if
end if
end subroutine extended_version_check
```

This chunk is copied verbatim from the `basic_vm`

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine basic_init (vm, verbose, out_fh)
type(vm_t), intent(inout) :: vm
logical, optional, intent(in) :: verbose
integer, optional, intent(in) :: out_fh
if (present (verbose)) then
vm%verbose = verbose
else
vm%verbose = .true.
end if
if (present (out_fh)) then
vm%out_fh = out_fh
else
vm%out_fh = stdout
```

```
end if
call set_stream (vm)
call alloc_and_count (vm)
if (vm%N_levels > 0) then
call read_bytecode (vm)
call sanity_check (vm)
end if
close (vm%bytecode_fh)
end subroutine basic_init

subroutine basic_write (vm)
type(vm_t), intent(in) :: vm
integer :: i
write (vm%out_fh, *) '=====> VM ', char(vm%version), ' <====='
write (vm%out_fh, *) 'verbose          =    ', vm%verbose
write (vm%out_fh, *) 'bytecode_file    =    ', char (vm%bytecode_file)
write (vm%out_fh, *) 'N_instructions   =    ', vm%N_instructions
write (vm%out_fh, *) 'N_levels         =    ', vm%N_levels
write (vm%out_fh, *) 'instructions     =    '
do i = 1, vm%N_instructions
write (vm%out_fh, *) vm%instructions(:, i)
end do
write (vm%out_fh, *) 'levels           =    ', vm%levels
end subroutine basic_write

subroutine alloc_and_count (vm)
type(vm_t), intent(inout) :: vm
integer, dimension(len_instructions) :: line
character(256) :: buffer
integer :: i, IO
read(vm%bytecode_fh, fmt = "(A)", iostat = IO) buffer
if (vm%model /= buffer) then
print *, "Warning: Bytecode has been generated with an older O'Mega version."
else
if (vm%verbose) then
write (vm%out_fh, fmt = *) "Using the model: "
write (vm%out_fh, fmt = *) char(vm%model)
end if
end if
call extended_version_check (vm, IO)
if (vm%verbose) then
write (vm%out_fh, fmt = *) "Trying to allocate."
end if
do i = 1, N_comments
read(vm%bytecode_fh, fmt = *, iostat = IO)
end do
call load_header (vm, IO)
call alloc_arrays (vm)
if (vm%N_momenta /= 0) then
do i = 1, vm%N_table_lines + 1
read(vm%bytecode_fh, fmt = *, iostat = IO)
end do
vm%N_instructions = 0
vm%N_levels = 0
do
read(vm%bytecode_fh, fmt = *, end = 42) line
if (line(1) /= 0) then
vm%N_instructions = vm%N_instructions + 1
else
vm%N_levels = vm%N_levels + 1
end if
end do
42 rewind(vm%bytecode_fh, iostat = IO)
allocate (vm%instructions(len_instructions, vm%N_instructions))
allocate (vm%levels(vm%N_levels))
if (IO /= 0) then
```

```
print *, "Error: vm.alloc : Couldn't load bytecode!"
stop 1
end if
end if
end subroutine alloc_and_count

subroutine read_bytecode (vm)
type(vm_t), intent(inout) :: vm
integer, dimension(len_instructions) :: line
integer :: i, j, IO
! Jump over version number, comments, header and first table description
do i = 1, N_version_lines + N_comments + N_header_lines + 1
read (vm%bytecode_fh, fmt = *, iostat = IO)
end do
call read_tables (vm, IO)
read (vm%bytecode_fh, fmt = *, iostat = IO)
i = 0; j = 0
do
read (vm%bytecode_fh, fmt = *, iostat = IO) line
if (IO /= 0) exit
if (line(1) == 0) then
if (j <= vm%N_levels) then
j = j + 1
vm%levels(j) = i                ! last index of a level is saved
else
print *, 'Error: vm.read_bytecode: File has more levels than anticipated!'
stop 1
end if
else
if (i <= vm%N_instructions) then
i = i + 1                       ! A valid instruction line
vm%instructions(:, i) = line
else
print *, 'Error: vm.read_bytecode: File is larger than anticipated!'
stop 1
end if
end if
end do
end subroutine read_bytecode

subroutine iterate_instructions (vm)
type(vm_t), intent(inout) :: vm
integer :: i, j
if (vm%openmp) then
!$omp parallel
do j = 1, vm%N_levels - 1
!$omp do schedule (static)
do i = vm%levels (j) + 1, vm%levels (j + 1)
call decode (vm, i)
end do
!$omp end do
end do
!$omp end parallel
else
do j = 1, vm%N_levels - 1
do i = vm%levels (j) + 1, vm%levels (j + 1)
call decode (vm, i)
end do
end do
end if
end subroutine iterate_instructions

subroutine set_stream (vm)
type(vm_t), intent(inout) :: vm
integer :: IO
call find_free_unit (vm%bytecode_fh, IO)
```

```
open (vm%bytecode_fh, file = char (vm%bytecode_file), form = 'formatted', &
access = 'sequential', status = 'old', position = 'rewind', iostat = IO, &
action = 'read')
if (IO /= 0) then
print *, "Error: vm.set_stream: Bytecode file '", char(vm%bytecode_file), &
"' not found!"
stop 1
end if
end subroutine set_stream

subroutine sanity_check (vm)
type(vm_t), intent(in) :: vm
if (vm%levels(1) /= 0) then
print *, "Error: vm.vm_init: levels(1) != 0"
stop 1
end if
if (vm%levels(vm%N_levels) /= vm%N_instructions) then
print *, "Error: vm.vm_init: levels(N_levels) != N_instructions"
stop 1
end if
if (vm%verbose) then
write(vm%out_fh, *) "vm passed sanity check. Starting calculation."
end if
end subroutine sanity_check
```

AB.33.4 Main Decode Function

This is the heart of the OVM

⟨*OVM Procedure Implementations*⟩+≡

```
! pure & ! if no warnings
subroutine decode (vm, instruction_index)
type(vm_t), intent(inout) :: vm
integer, intent(in) :: instruction_index
integer, dimension(len_instructions) :: i, curr
complex(default) :: braket
integer :: tmp
real(default) :: w
i = vm%instructions (:, instruction_index)
select case (i(1))
case ( : -1)       ! Jump over subinstructions

⟨cases of decode⟩
case (0)
print *, 'Error: Levelbreak put in decode! Line:', &
instruction_index
stop 1
case default
print *, "Error: Decode has case not catched! Line: ", &
instruction_index
stop 1
end select
end subroutine decode
```

Momenta

The most trivial instruction

⟨*OVM Instructions*⟩≡

```
integer, parameter :: ovm_ADD_MOMENTA = 1
```

⟨cases *of* decode⟩≡

```
case (ovm_ADD_MOMENTA)
vm%momenta(i(4)) = vm%momenta(i(5)) + vm%momenta(i(6))
```

```
if (i(7) > 0) then
vm%momenta(i(4)) = vm%momenta(i(4)) + vm%momenta(i(7))
end if
```

Loading External states

⟨*OVM Instructions*⟩+≡
```
integer, parameter :: ovm_LOAD_SCALAR = 10
integer, parameter :: ovm_LOAD_SPINOR_INC = 11
integer, parameter :: ovm_LOAD_SPINOR_OUT = 12
integer, parameter :: ovm_LOAD_CONJSPINOR_INC = 13
integer, parameter :: ovm_LOAD_CONJSPINOR_OUT = 14
integer, parameter :: ovm_LOAD_MAJORANA_INC = 15
integer, parameter :: ovm_LOAD_MAJORANA_OUT = 16
integer, parameter :: ovm_LOAD_VECTOR_INC = 17
integer, parameter :: ovm_LOAD_VECTOR_OUT = 18
integer, parameter :: ovm_LOAD_VECTORSPINOR_INC = 19
integer, parameter :: ovm_LOAD_VECTORSPINOR_OUT = 20
integer, parameter :: ovm_LOAD_TENSOR2_INC = 21
integer, parameter :: ovm_LOAD_TENSOR2_OUT = 22
integer, parameter :: ovm_LOAD_BRS_SCALAR = 30
integer, parameter :: ovm_LOAD_BRS_SPINOR_INC = 31
integer, parameter :: ovm_LOAD_BRS_SPINOR_OUT = 32
integer, parameter :: ovm_LOAD_BRS_CONJSPINOR_INC = 33
integer, parameter :: ovm_LOAD_BRS_CONJSPINOR_OUT = 34
integer, parameter :: ovm_LOAD_BRS_VECTOR_INC = 37
integer, parameter :: ovm_LOAD_BRS_VECTOR_OUT = 38
integer, parameter :: ovm_LOAD_MAJORANA_GHOST_INC = 23
integer, parameter :: ovm_LOAD_MAJORANA_GHOST_OUT = 24
integer, parameter :: ovm_LOAD_BRS_MAJORANA_INC = 35
integer, parameter :: ovm_LOAD_BRS_MAJORANA_OUT = 36
```

⟨cases *of* decode⟩+≡
```
case (ovm_LOAD_SCALAR)
vm%scalars(i(4))%v = one
vm%scalars(i(4))%c = .True.

case (ovm_LOAD_SPINOR_INC)
call load_spinor(vm%spinors(i(4)), - ⟨p⟩, ⟨m⟩, &
vm%helicity(i(5)), ovm_LOAD_SPINOR_INC)

case (ovm_LOAD_SPINOR_OUT)
call load_spinor(vm%spinors(i(4)), ⟨p⟩, ⟨m⟩, &
vm%helicity(i(5)), ovm_LOAD_SPINOR_OUT)

case (ovm_LOAD_CONJSPINOR_INC)
call load_conjspinor(vm%conjspinors(i(4)), - ⟨p⟩, &
⟨m⟩, vm%helicity(i(5)), ovm_LOAD_CONJSPINOR_INC)

case (ovm_LOAD_CONJSPINOR_OUT)
call load_conjspinor(vm%conjspinors(i(4)), ⟨p⟩, &
⟨m⟩, vm%helicity(i(5)), ovm_LOAD_CONJSPINOR_OUT)

case (ovm_LOAD_MAJORANA_INC)
call load_bispinor(vm%bispinors(i(4)), - ⟨p⟩, &
⟨m⟩, vm%helicity(i(5)), ovm_LOAD_MAJORANA_INC)

case (ovm_LOAD_MAJORANA_OUT)
call load_bispinor(vm%bispinors(i(4)), ⟨p⟩, ⟨m⟩, &
vm%helicity(i(5)), ovm_LOAD_MAJORANA_OUT)

case (ovm_LOAD_VECTOR_INC)
call load_vector(vm%vectors(i(4)), - ⟨p⟩, ⟨m⟩, &
vm%helicity(i(5)), ovm_LOAD_VECTOR_INC)
```

```
case (ovm_LOAD_VECTOR_OUT)
call load_vector(vm%vectors(i(4)), ⟨p⟩, ⟨m⟩, &
vm%helicity(i(5)), ovm_LOAD_VECTOR_OUT)

case (ovm_LOAD_VECTORSPINOR_INC)
!select type (h => vm%helicity(i(5)))
!type is (hel_discrete)
!vm%vectorspinors(i(4))%v = veps(⟨m⟩, - ⟨p⟩, &
!h%i)
!end select
vm%vectorspinors(i(4))%v = veps(⟨m⟩, - ⟨p⟩, &
vm%helicity(i(5)))
vm%vectorspinors(i(4))%c = .True.

case (ovm_LOAD_VECTORSPINOR_OUT)
!select type (h => vm%helicity(i(5)))
!type is (hel_discrete)
!vm%vectorspinors(i(4))%v = veps(⟨m⟩, ⟨p⟩, &
!h%i)
!end select
vm%vectorspinors(i(4))%v = veps(⟨m⟩, ⟨p⟩, &
vm%helicity(i(5)))
vm%vectorspinors(i(4))%c = .True.

case (ovm_LOAD_TENSOR2_INC)
!select type (h => vm%helicity(i(5)))
!type is (hel_discrete)
!vm%tensors_2(i(4))%v = eps2(⟨m⟩, - ⟨p⟩, &
!h%i)
!end select
vm%tensors_2(i(4))%c = .True.

case (ovm_LOAD_TENSOR2_OUT)
!select type (h => vm%helicity(i(5)))
!type is (hel_discrete)
!vm%tensors_2(i(4))%v = eps2(⟨m⟩, ⟨p⟩, h%i)
!end select
vm%tensors_2(i(4))%c = .True.

case (ovm_LOAD_BRS_SCALAR)
vm%scalars(i(4))%v = (0, -1) * (⟨p⟩ * ⟨p⟩ - &
⟨m⟩**2)
vm%scalars(i(4))%c = .True.

case (ovm_LOAD_BRS_SPINOR_INC)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_SPINOR_OUT)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_CONJSPINOR_INC)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_CONJSPINOR_OUT)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_VECTOR_INC)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_VECTOR_OUT)
print *, 'not implemented'
stop 1
case (ovm_LOAD_MAJORANA_GHOST_INC)
print *, 'not implemented'
stop 1
```

```
case (ovm_LOAD_MAJORANA_GHOST_OUT)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_MAJORANA_INC)
print *, 'not implemented'
stop 1
case (ovm_LOAD_BRS_MAJORANA_OUT)
print *, 'not implemented'
stop 1
```

Brakets and Fusions

NB: during, execution, the type of the coupling constant is implicit in the instruction

⟨*OVM Instructions*⟩+≡
```
integer, parameter :: ovm_CALC_BRAKET = 2

integer, parameter :: ovm_FUSE_V_FF = -1
integer, parameter :: ovm_FUSE_F_VF = -2
integer, parameter :: ovm_FUSE_F_FV = -3
integer, parameter :: ovm_FUSE_VA_FF = -4
integer, parameter :: ovm_FUSE_F_VAF = -5
integer, parameter :: ovm_FUSE_F_FVA = -6
integer, parameter :: ovm_FUSE_VA2_FF = -7
integer, parameter :: ovm_FUSE_F_VA2F = -8
integer, parameter :: ovm_FUSE_F_FVA2 = -9
integer, parameter :: ovm_FUSE_A_FF = -10
integer, parameter :: ovm_FUSE_F_AF = -11
integer, parameter :: ovm_FUSE_F_FA = -12
integer, parameter :: ovm_FUSE_VL_FF = -13
integer, parameter :: ovm_FUSE_F_VLF = -14
integer, parameter :: ovm_FUSE_F_FVL = -15
integer, parameter :: ovm_FUSE_VR_FF = -16
integer, parameter :: ovm_FUSE_F_VRF = -17
integer, parameter :: ovm_FUSE_F_FVR = -18
integer, parameter :: ovm_FUSE_VLR_FF = -19
integer, parameter :: ovm_FUSE_F_VLRF = -20
integer, parameter :: ovm_FUSE_F_FVLR = -21
integer, parameter :: ovm_FUSE_SP_FF = -22
integer, parameter :: ovm_FUSE_F_SPF = -23
integer, parameter :: ovm_FUSE_F_FSP = -24
integer, parameter :: ovm_FUSE_S_FF = -25
integer, parameter :: ovm_FUSE_F_SF = -26
integer, parameter :: ovm_FUSE_F_FS = -27
integer, parameter :: ovm_FUSE_P_FF = -28
integer, parameter :: ovm_FUSE_F_PF = -29
integer, parameter :: ovm_FUSE_F_FP = -30
integer, parameter :: ovm_FUSE_SL_FF = -31
integer, parameter :: ovm_FUSE_F_SLF = -32
integer, parameter :: ovm_FUSE_F_FSL = -33
integer, parameter :: ovm_FUSE_SR_FF = -34
integer, parameter :: ovm_FUSE_F_SRF = -35
integer, parameter :: ovm_FUSE_F_FSR = -36
integer, parameter :: ovm_FUSE_SLR_FF = -37
integer, parameter :: ovm_FUSE_F_SLRF = -38
integer, parameter :: ovm_FUSE_F_FSLR = -39

integer, parameter :: ovm_FUSE_G_GG = -40
integer, parameter :: ovm_FUSE_V_SS = -41
integer, parameter :: ovm_FUSE_S_VV = -42
integer, parameter :: ovm_FUSE_S_VS = -43
integer, parameter :: ovm_FUSE_V_SV = -44
integer, parameter :: ovm_FUSE_S_SS = -45
integer, parameter :: ovm_FUSE_S_SVV = -46
integer, parameter :: ovm_FUSE_V_SSV = -47
integer, parameter :: ovm_FUSE_S_SSS = -48
```

```
  integer, parameter :: ovm_FUSE_V_VVV = -49

  integer, parameter :: ovm_FUSE_S_G2 = -50
  integer, parameter :: ovm_FUSE_G_SG = -51
  integer, parameter :: ovm_FUSE_G_GS = -52
  integer, parameter :: ovm_FUSE_S_G2_SKEW = -53
  integer, parameter :: ovm_FUSE_G_SG_SKEW = -54
  integer, parameter :: ovm_FUSE_G_GS_SKEW = -55
```

Shorthands

⟨*p*⟩≡
```
  vm%momenta(i(5))
```

⟨*m*⟩≡
```
  vm%mass(i(2))
```

⟨*p1*⟩≡
```
  vm%momenta(curr(6))
```

⟨*p2*⟩≡
```
  vm%momenta(curr(8))
```

⟨*v1*⟩≡
```
  vm%vectors(curr(5))%v
```

⟨*v2*⟩≡
```
  vm%vectors(curr(7))%v
```

⟨*s1*⟩≡
```
  vm%scalars(curr(5))%v
```

⟨*s2*⟩≡
```
  vm%scalars(curr(7))%v
```

⟨*c*⟩≡
```
  sgn_coupl_cmplx(vm, curr(2))
```

⟨*c1*⟩≡
```
  sgn_coupl_cmplx2(vm, curr(2), 1)
```

⟨*c2*⟩≡
```
  sgn_coupl_cmplx2(vm, curr(2), 2)
```

⟨*check for matching color and flavor amplitude of braket (old)*⟩≡
```
  if ((i(4) == o%cols(1)) .or. (i(4) == o%cols(2)) .or. &
  ((mode%col_MC .eq. FULL_SUM) .or. (mode%col_MC .eq. DIAG_COL))) then
```

Just a stub for now. Will be reimplemented with the polymorph type `color` similar to the `select type(helicity)` when we need it.

⟨*check for matching color and flavor amplitude*⟩≡

⟨`case`*s of* `decode`⟩+≡
```
  case (ovm_CALC_BRAKET)
  ⟨check for matching color and flavor amplitude⟩
  tmp = instruction_index + 1
  do
  if (tmp > vm%N_instructions) exit
  curr = vm%instructions(:, tmp)
  if (curr(1) >= 0) exit                        ! End of fusions
  select case (curr(1))
  case (ovm_FUSE_V_FF, ovm_FUSE_VL_FF, ovm_FUSE_VR_FF)
  braket = vm%vectors(curr(4))%v * vec_ff(vm, curr)

  case (ovm_FUSE_F_VF, ovm_FUSE_F_VLF, ovm_FUSE_F_VRF)
  braket = vm%conjspinors(curr(4))%v * ferm_vf(vm, curr)

  case (ovm_FUSE_F_FV, ovm_FUSE_F_FVL, ovm_FUSE_F_FVR)
  braket = ferm_fv(vm, curr) * vm%spinors(curr(4))%v

  case (ovm_FUSE_VA_FF)
  braket = vm%vectors(curr(4))%v * vec_ff2(vm, curr)

  case (ovm_FUSE_F_VAF)
```

```
braket = vm%conjspinors(curr(4))%v * ferm_vf2(vm, curr)

case (ovm_FUSE_F_FVA)
braket = ferm_fv2(vm, curr) * vm%spinors(curr(4))%v

case (ovm_FUSE_S_FF, ovm_FUSE_SP_FF)
braket = vm%scalars(curr(4))%v * scal_ff(vm, curr)

case (ovm_FUSE_F_SF, ovm_FUSE_F_SPF)
braket = vm%conjspinors(curr(4))%v * ferm_sf(vm, curr)

case (ovm_FUSE_F_FS, ovm_FUSE_F_FSP)
braket = ferm_fs(vm, curr) * vm%spinors(curr(4))%v

case (ovm_FUSE_G_GG)
braket = vm%vectors(curr(4))%v * &
g_gg(⟨c⟩, &
⟨v1⟩, ⟨p1⟩, &
⟨v2⟩, ⟨p2⟩)

case (ovm_FUSE_S_VV)
braket = vm%scalars(curr(4))%v * ⟨c⟩ * &
(⟨v1⟩ * vm%vectors(curr(6))%v)

case (ovm_FUSE_V_SS)
braket = vm%vectors(curr(4))%v * &
v_ss(⟨c⟩, ⟨s1⟩, ⟨p1⟩, &
⟨s2⟩, ⟨p2⟩)

case (ovm_FUSE_S_G2, ovm_FUSE_S_G2_SKEW)
braket = vm%scalars(curr(4))%v * scal_g2(vm, curr)

case (ovm_FUSE_G_SG, ovm_FUSE_G_GS, ovm_FUSE_G_SG_SKEW, ovm_FUSE_G_GS_SKEW)
braket = vm%vectors(curr(4))%v * gauge_sg(vm, curr)

case (ovm_FUSE_S_VS)
braket = vm%scalars(curr(4))%v * &
s_vs(⟨c⟩, &
⟨v1⟩, ⟨p1⟩, &
⟨s2⟩, ⟨p2⟩)

case (ovm_FUSE_V_SV)
braket = (vm%vectors(curr(4))%v * vm%vectors(curr(6))%v) * &
(⟨c⟩ * ⟨s1⟩)

case (ovm_FUSE_S_SS)
braket = vm%scalars(curr(4))%v * &
⟨c⟩ * &
(⟨s1⟩ * vm%scalars(curr(6))%v)

case (ovm_FUSE_S_SSS)
braket = vm%scalars(curr(4))%v * &
⟨c⟩ * &
(⟨s1⟩ * vm%scalars(curr(6))%v * &
⟨s2⟩)

case (ovm_FUSE_S_SVV)
braket = vm%scalars(curr(4))%v * &
⟨c⟩ * &
⟨s1⟩ * (vm%vectors(curr(6))%v * &
⟨v2⟩)

case (ovm_FUSE_V_SSV)
braket = vm%vectors(curr(4))%v * &
(⟨c⟩ * ⟨s1⟩ * &
vm%scalars(curr(6))%v) * ⟨v2⟩
```

```
case (ovm_FUSE_V_VVV)
braket = ⟨c⟩ * &
(⟨v1⟩ * vm%vectors(curr(6))%v) * &
(vm%vectors(curr(4))%v * ⟨v2⟩)

case default
print *, 'Braket', curr(1), 'not implemented'
stop 1

end select
vm%amplitudes(i(4)) = vm%amplitudes(i(4)) + curr(3) * braket
tmp = tmp + 1
end do

vm%amplitudes(i(4)) = vm%amplitudes(i(4)) * i(2)
if (i(5) > 1) then
vm%amplitudes(i(4)) = vm%amplitudes(i(4)) * &         ! Symmetry factor
(one / sqrt(real(i(5), kind=default)))
end if
```

Propagators

⟨*OVM Instructions*⟩+≡
```
integer, parameter :: ovm_PROPAGATE_SCALAR = 51
integer, parameter :: ovm_PROPAGATE_COL_SCALAR = 52
integer, parameter :: ovm_PROPAGATE_GHOST = 53
integer, parameter :: ovm_PROPAGATE_SPINOR = 54
integer, parameter :: ovm_PROPAGATE_CONJSPINOR = 55
integer, parameter :: ovm_PROPAGATE_MAJORANA = 56
integer, parameter :: ovm_PROPAGATE_COL_MAJORANA = 57
integer, parameter :: ovm_PROPAGATE_UNITARITY = 58
integer, parameter :: ovm_PROPAGATE_COL_UNITARITY = 59
integer, parameter :: ovm_PROPAGATE_FEYNMAN = 60
integer, parameter :: ovm_PROPAGATE_COL_FEYNMAN = 61
integer, parameter :: ovm_PROPAGATE_VECTORSPINOR = 62
integer, parameter :: ovm_PROPAGATE_TENSOR2 = 63
integer, parameter :: ovm_PROPAGATE_NONE = 64
```

⟨*check for matching color and flavor amplitude of propagator (old)*⟩≡
```
if ((mode%col_MC .eq. FULL_SUM) .or. (mode%col_MC .eq. DIAG_COL)) then
select case(i(1))
case (ovm_PROPAGATE_PSI)
go = .not. vm%spinors%c(i(4))
case (ovm_PROPAGATE_PSIBAR)
go = .not. vm%conjspinors%c(i(4))
case (ovm_PROPAGATE_UNITARITY, ovm_PROPAGATE_FEYNMAN, &
ovm_PROPAGATE_COL_FEYNMAN)
go = .not. vm%vectors%c(i(4))
end select
else
go = (i(8) == o%cols(1)) .or. (i(8) == o%cols(2))
end if
if (go) then
```

⟨`case`*s of* `decode`⟩+≡
```
⟨check for matching color and flavor amplitude⟩
case (ovm_PROPAGATE_SCALAR : ovm_PROPAGATE_NONE)
tmp = instruction_index + 1
do
curr = vm%instructions(:,tmp)
if (curr(1) >= 0) exit                        ! End of fusions
select case (curr(1))
case (ovm_FUSE_V_FF, ovm_FUSE_VL_FF, ovm_FUSE_VR_FF)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + curr(3) * &
vec_ff(vm, curr)
```

```
case (ovm_FUSE_F_VF, ovm_FUSE_F_VLF, ovm_FUSE_F_VRF)
vm%spinors(curr(4))%v = vm%spinors(curr(4))%v + curr(3) * &
ferm_vf(vm, curr)

case (ovm_FUSE_F_FV, ovm_FUSE_F_FVL, ovm_FUSE_F_FVR)
vm%conjspinors(curr(4))%v = vm%conjspinors(curr(4))%v + curr(3) * &
ferm_fv(vm, curr)

case (ovm_FUSE_VA_FF)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + curr(3) * &
vec_ff2(vm, curr)

case (ovm_FUSE_F_VAF)
vm%spinors(curr(4))%v = vm%spinors(curr(4))%v + curr(3) * &
ferm_vf2(vm, curr)

case (ovm_FUSE_F_FVA)
vm%conjspinors(curr(4))%v = vm%conjspinors(curr(4))%v + curr(3) * &
ferm_fv2(vm, curr)

case (ovm_FUSE_S_FF, ovm_FUSE_SP_FF)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + curr(3) * &
scal_ff(vm, curr)

case (ovm_FUSE_F_SF, ovm_FUSE_F_SPF)
vm%spinors(curr(4))%v = vm%spinors(curr(4))%v + curr(3) * &
ferm_sf(vm, curr)

case (ovm_FUSE_F_FS, ovm_FUSE_F_FSP)
vm%conjspinors(curr(4))%v = vm%conjspinors(curr(4))%v + curr(3) * &
ferm_fs(vm, curr)

case (ovm_FUSE_G_GG)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + curr(3) * &
g_gg(⟨c⟩, ⟨v1⟩, &
⟨p1⟩, ⟨v2⟩, &
⟨p2⟩)

case (ovm_FUSE_S_VV)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + curr(3) * &
⟨c⟩ * &
(⟨v1⟩ * vm%vectors(curr(6))%v)

case (ovm_FUSE_V_SS)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + curr(3) * &
v_ss(⟨c⟩, ⟨s1⟩, ⟨p1⟩, &
⟨s2⟩, ⟨p2⟩)


case (ovm_FUSE_S_G2, ovm_FUSE_S_G2_SKEW)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + &
scal_g2(vm, curr) * curr(3)

case (ovm_FUSE_G_SG, ovm_FUSE_G_GS, ovm_FUSE_G_SG_SKEW, ovm_FUSE_G_GS_SKEW)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + &
gauge_sg(vm, curr) * curr(3)

case (ovm_FUSE_S_VS)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + &
s_vs(⟨c⟩, &
⟨v1⟩, ⟨p1⟩, &
⟨s2⟩, ⟨p2⟩) * curr(3)

case (ovm_FUSE_V_SV)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + &
```

```
vm%vectors(curr(6))%v * &
(⟨c⟩ * ⟨s1⟩ * curr(3))

case (ovm_FUSE_S_SS)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + &
⟨c⟩ * &
(⟨s1⟩ * vm%scalars(curr(6))%v) * curr(3)

case (ovm_FUSE_S_SSS)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + &
⟨c⟩ * &
(⟨s1⟩ * vm%scalars(curr(6))%v * &
⟨s2⟩) * curr(3)

case (ovm_FUSE_S_SVV)
vm%scalars(curr(4))%v = vm%scalars(curr(4))%v + &
⟨c⟩ * &
⟨s1⟩ * (vm%vectors(curr(6))%v * &
⟨v2⟩) * curr(3)

case (ovm_FUSE_V_SSV)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + &
(⟨c⟩ * ⟨s1⟩ * &
vm%scalars(curr(6))%v) * ⟨v2⟩ * curr(3)

case (ovm_FUSE_V_VVV)
vm%vectors(curr(4))%v = vm%vectors(curr(4))%v + &
(⟨c⟩ * (⟨v1⟩ * &
vm%vectors(curr(6))%v)) * curr(3) * ⟨v2⟩

case default
print *, 'Fusion', curr(1), 'not implemented'
stop 1

end select
tmp = tmp + 1
end do

select case (i(3))
case (0)
w = zero

case (1)
w = vm%width(i(2))
vm%cms = .false.

case (2)
w = wd_tl(⟨p⟩, vm%width(i(2)))

case (3)
w = vm%width(i(2))
vm%cms = .true.

case (4)
w = wd_run(⟨p⟩, ⟨m⟩, vm%width(i(2)))

case default
print *, 'not implemented'
stop 1

end select

select case (i(1))
⟨propagator cases in decode⟩
end select
```

⟨*propagator* cases *in* decode⟩≡
```
  case (ovm_PROPAGATE_SCALAR)
  vm%scalars(i(4))%v = pr_phi(⟨p⟩, ⟨m⟩, &
  w, vm%scalars(i(4))%v)
  vm%scalars(i(4))%c = .True.

  case (ovm_PROPAGATE_COL_SCALAR)
  vm%scalars(i(4))%v = - one / N_ * pr_phi(⟨p⟩, &
  ⟨m⟩, w, vm%scalars(i(4))%v)
  vm%scalars(i(4))%c = .True.

  case (ovm_PROPAGATE_GHOST)
  vm%scalars(i(4))%v = imago * pr_phi(⟨p⟩, ⟨m⟩, &
  w, vm%scalars(i(4))%v)
  vm%scalars(i(4))%c = .True.

  case (ovm_PROPAGATE_SPINOR)
  vm%spinors(i(4))%v = pr_psi(⟨p⟩, ⟨m⟩, &
  w, vm%cms, vm%spinors(i(4))%v)
  vm%spinors(i(4))%c = .True.

  case (ovm_PROPAGATE_CONJSPINOR)
  vm%conjspinors(i(4))%v = pr_psibar(⟨p⟩, ⟨m⟩, &
  w, vm%cms, vm%conjspinors(i(4))%v)
  vm%conjspinors(i(4))%c = .True.

  case (ovm_PROPAGATE_MAJORANA)
  vm%bispinors(i(4))%v = bi_pr_psi(⟨p⟩, ⟨m⟩, &
  w, vm%cms, vm%bispinors(i(4))%v)
  vm%bispinors(i(4))%c = .True.

  case (ovm_PROPAGATE_COL_MAJORANA)
  vm%bispinors(i(4))%v = (- one / N_) * &
  bi_pr_psi(⟨p⟩, ⟨m⟩, &
  w, vm%cms, vm%bispinors(i(4))%v)
  vm%bispinors(i(4))%c = .True.

  case (ovm_PROPAGATE_UNITARITY)
  vm%vectors(i(4))%v = pr_unitarity(⟨p⟩, ⟨m⟩, &
  w, vm%cms, vm%vectors(i(4))%v)
  vm%vectors(i(4))%c = .True.

  case (ovm_PROPAGATE_COL_UNITARITY)
  vm%vectors(i(4))%v = - one / N_ * pr_unitarity(⟨p⟩, &
  ⟨m⟩, w, vm%cms, vm%vectors(i(4))%v)
  vm%vectors(i(4))%c = .True.

  case (ovm_PROPAGATE_FEYNMAN)
  vm%vectors(i(4))%v = pr_feynman(⟨p⟩, vm%vectors(i(4))%v)
  vm%vectors(i(4))%c = .True.

  case (ovm_PROPAGATE_COL_FEYNMAN)
  vm%vectors(i(4))%v = - one / N_ * &
  pr_feynman(⟨p⟩, vm%vectors(i(4))%v)
  vm%vectors(i(4))%c = .True.

  case (ovm_PROPAGATE_VECTORSPINOR)
  vm%vectorspinors(i(4))%v = pr_grav(⟨p⟩, ⟨m⟩, &
  w, vm%vectorspinors(i(4))%v)
  vm%vectorspinors(i(4))%c = .True.

  case (ovm_PROPAGATE_TENSOR2)
  vm%tensors_2(i(4))%v = pr_tensor(⟨p⟩, ⟨m⟩, &
  w, vm%tensors_2(i(4))%v)
  vm%tensors_2(i(4))%c = .True.
```

```
case (ovm_PROPAGATE_NONE)
! This will not work with color MC. Appropriate type%c has to be set to
! .True.
```

AB.33.5 Helper functions

Factoring out these parts helps a lot to keep sane but might hurt the performance of the VM noticably. In that case, we have to copy & paste to avoid the additional function calls. Note that with preprocessor macros, we could maintain this factorized form (and factor out even more since types don't have to match), in case we would decide to allow this

⟨*load outer wave function*⟩≡

```
!select type (h)
!type is (hel_trigonometric)
!wf%v = (cos (h%theta) * load_wf (m, p, + 1) + &
!sin (h%theta) * load_wf (m, p, - 1)) * sqrt2
!type is (hel_exponential)
!wf%v = exp (+ imago * h%phi) * load_wf (m, p, + 1) + &
!exp (- imago * h%phi) * load_wf (m, p, - 1)
!type is (hel_spherical)
!wf%v = (exp (+ imago * h%phi) * cos (h%theta) * load_wf (m, p, + 1) + &
!exp (- imago * h%phi) * sin (h%theta) * load_wf (m, p, - 1)) * &
!sqrt2
!type is(hel_discrete)
!wf%v = load_wf (m, p, h%i)
!end select
wf%v = load_wf (m, p, h)
wf%c = .True.
```

Caveat: Helicity MC not tested with Majorana particles but should be fine

⟨*check for matching color and flavor amplitude of wf (old)*⟩≡

```
if ((mode%col_MC .eq. FULL_SUM) .or. (mode%col_MC .eq. DIAG_COL)) then
go = .not. vm%spinors%c(i(4))
else
go = (i(8) == o%cols(1)) .or. (i(8) == o%cols(2))
end if
if (go) ..
```

⟨*OVM Procedure Implementations*⟩+≡

```
subroutine load_bispinor(wf, p, m, h, opcode)
type(vm_bispinor), intent(out) :: wf
type(momentum), intent(in) :: p
real(default), intent(in) :: m
!class(helicity_t), intent(in) :: h
integer, intent(in) :: h
integer, intent(in) :: opcode
procedure(bi_u), pointer :: load_wf
⟨check for matching color and flavor amplitude⟩
select case (opcode)
case (ovm_LOAD_MAJORANA_INC)
load_wf => bi_u
case (ovm_LOAD_MAJORANA_OUT)
load_wf => bi_v
case default
load_wf => null()
end select
⟨load outer wave function⟩
end subroutine load_bispinor

subroutine load_spinor(wf, p, m, h, opcode)
type(vm_spinor), intent(out) :: wf
type(momentum), intent(in) :: p
real(default), intent(in) :: m
!class(helicity_t), intent(in) :: h
integer, intent(in) :: h
integer, intent(in) :: opcode
```

```
procedure(u), pointer :: load_wf
⟨check for matching color and flavor amplitude⟩
select case (opcode)
case (ovm_LOAD_SPINOR_INC)
load_wf => u
case (ovm_LOAD_SPINOR_OUT)
load_wf => v
case default
load_wf => null()
end select
⟨load outer wave function⟩
end subroutine load_spinor

subroutine load_conjspinor(wf, p, m, h, opcode)
type(vm_conjspinor), intent(out) :: wf
type(momentum), intent(in) :: p
real(default), intent(in) :: m
!class(helicity_t), intent(in) :: h
integer, intent(in) :: h
integer, intent(in) :: opcode
procedure(ubar), pointer :: load_wf
⟨check for matching color and flavor amplitude⟩
select case (opcode)
case (ovm_LOAD_CONJSPINOR_INC)
load_wf => vbar
case (ovm_LOAD_CONJSPINOR_OUT)
load_wf => ubar
case default
load_wf => null()
end select
⟨load outer wave function⟩
end subroutine load_conjspinor

subroutine load_vector(wf, p, m, h, opcode)
type(vm_vector), intent(out) :: wf
type(momentum), intent(in) :: p
real(default), intent(in) :: m
!class(helicity_t), intent(in) :: h
integer, intent(in) :: h
integer, intent(in) :: opcode
procedure(eps), pointer :: load_wf
⟨check for matching color and flavor amplitude⟩
load_wf => eps
⟨load outer wave function⟩
if (opcode == ovm_LOAD_VECTOR_OUT) then
wf%v = conjg(wf%v)
end if
end subroutine load_vector
```

⟨*OVM Procedure Implementations*⟩+≡

```
function ferm_vf(vm, curr) result (x)
type(spinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(f_vf), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_F_VF)
load_wf => f_vf
case (ovm_FUSE_F_VLF)
load_wf => f_vlf
case (ovm_FUSE_F_VRF)
load_wf => f_vrf
case default
load_wf => null()
end select
x = load_wf(⟨c⟩, ⟨v1⟩, vm%spinors(curr(6))%v)
```

```
end function ferm_vf

function ferm_vf2(vm, curr) result (x)
type(spinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(f_vaf), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_F_VAF)
load_wf => f_vaf
case default
load_wf => null()
end select
x = f_vaf(⟨c1⟩, ⟨c2⟩, ⟨v1⟩, vm%spinors(curr(6))%v)
end function ferm_vf2

function ferm_sf(vm, curr) result (x)
type(spinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
select case (curr(1))
case (ovm_FUSE_F_SF)
x = f_sf(⟨c⟩, ⟨s1⟩, vm%spinors(curr(6))%v)
case (ovm_FUSE_F_SPF)
x = f_spf(⟨c1⟩, ⟨c2⟩, ⟨s1⟩, vm%spinors(curr(6))%v)
case default
end select
end function ferm_sf

function ferm_fv(vm, curr) result (x)
type(conjspinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(f_fv), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_F_FV)
load_wf => f_fv
case (ovm_FUSE_F_FVL)
load_wf => f_fvl
case (ovm_FUSE_F_FVR)
load_wf => f_fvr
case default
load_wf => null()
end select
x = load_wf(⟨c⟩, vm%conjspinors(curr(5))%v, vm%vectors(curr(6))%v)
end function ferm_fv

function ferm_fv2(vm, curr) result (x)
type(conjspinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(f_fva), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_F_FVA)
load_wf => f_fva
case default
load_wf => null()
end select
x = f_fva(⟨c1⟩, ⟨c2⟩, &
vm%conjspinors(curr(5))%v, vm%vectors(curr(6))%v)
end function ferm_fv2

function ferm_fs(vm, curr) result (x)
type(conjspinor) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
```

```
procedure(f_fs), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_F_FS)
x = f_fs(⟨c⟩, vm%conjspinors(curr(5))%v, vm%scalars(curr(6))%v)
case (ovm_FUSE_F_FSP)
x = f_fsp(⟨c1⟩, ⟨c2⟩, &
vm%conjspinors(curr(5))%v, vm%scalars(curr(6))%v)
case default
x%a = zero
end select
end function ferm_fs

function vec_ff(vm, curr) result (x)
type(vector) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(v_ff), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_V_FF)
load_wf => v_ff
case (ovm_FUSE_VL_FF)
load_wf => vl_ff
case (ovm_FUSE_VR_FF)
load_wf => vr_ff
case default
load_wf => null()
end select
x = load_wf(⟨c⟩, vm%conjspinors(curr(5))%v, vm%spinors(curr(6))%v)
end function vec_ff

function vec_ff2(vm, curr) result (x)
type(vector) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
procedure(va_ff), pointer :: load_wf
select case (curr(1))
case (ovm_FUSE_VA_FF)
load_wf => va_ff
case default
load_wf => null()
end select
x = load_wf(⟨c1⟩, ⟨c2⟩, &
vm%conjspinors(curr(5))%v, vm%spinors(curr(6))%v)
end function vec_ff2

function scal_ff(vm, curr) result (x)
complex(default) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
select case (curr(1))
case (ovm_FUSE_S_FF)
x = s_ff(⟨c⟩, &
vm%conjspinors(curr(5))%v, vm%spinors(curr(6))%v)
case (ovm_FUSE_SP_FF)
x = sp_ff(⟨c1⟩, ⟨c2⟩, &
vm%conjspinors(curr(5))%v, vm%spinors(curr(6))%v)
case default
x = zero
end select
end function scal_ff

function scal_g2(vm, curr) result (x)
complex(default) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
select case (curr(1))
```

```
case (ovm_FUSE_S_G2)
x = ⟨c⟩ * ((⟨p1⟩ * ⟨v2⟩) * &
(⟨p2⟩ * ⟨v1⟩) - &
(⟨p1⟩ * ⟨p2⟩) * &
(⟨v2⟩ * ⟨v1⟩))
case (ovm_FUSE_S_G2_SKEW)
x = - phi_vv(⟨c⟩, ⟨p1⟩, ⟨p2⟩, &
⟨v1⟩, ⟨v2⟩)
case default
x = zero
end select
end function scal_g2

pure function gauge_sg(vm, curr) result (x)
type(vector) :: x
class(vm_t), intent(in) :: vm
integer, dimension(:), intent(in) :: curr
select case (curr(1))
case (ovm_FUSE_G_SG)
x = ⟨c⟩ * ⟨s1⟩ * ( &
-((⟨p1⟩ + ⟨p2⟩) * &
⟨v2⟩) * ⟨p2⟩ - &
(-(⟨p1⟩ + ⟨p2⟩) * &
⟨p2⟩) * ⟨v2⟩)
case (ovm_FUSE_G_GS)
x = ⟨c⟩ * ⟨s1⟩ * ( &
-((⟨p1⟩ + ⟨p2⟩) * &
⟨v2⟩) * ⟨p2⟩ - &
(-(⟨p1⟩ + ⟨p2⟩) * &
⟨p2⟩) * ⟨v2⟩)
case (ovm_FUSE_G_SG_SKEW)
x = - v_phiv(⟨c⟩, ⟨s1⟩, ⟨p1⟩, &
⟨p2⟩, ⟨v2⟩)
case (ovm_FUSE_G_GS_SKEW)
x = - v_phiv(⟨c⟩, ⟨s2⟩, ⟨p1⟩, &
⟨p2⟩, ⟨v1⟩)
case default
x = [zero, zero, zero, zero]
end select
end function gauge_sg
```

Some really tiny ones that hopefully get inlined by the compiler

⟨OVM Procedure Implementations⟩+≡

```
elemental function sgn_coupl_cmplx(vm, j) result (s)
class(vm_t), intent(in) :: vm
integer, intent(in) :: j
complex(default) :: s
s = isign(1, j) * vm%coupl_cmplx(abs(j))
end function sgn_coupl_cmplx

elemental function sgn_coupl_cmplx2(vm, j, i) result (s)
class(vm_t), intent(in) :: vm
integer, intent(in) :: j, i
complex(default) :: s
if (i == 1) then
s = isign(1, j) * vm%coupl_cmplx2(i, abs(j))
else
s = isign(1, j) * vm%coupl_cmplx2(i, abs(j))
end if
end function sgn_coupl_cmplx2

elemental function int_to_log(i) result(yorn)
integer, intent(in) :: i
logical :: yorn
if (i /= 0) then
yorn = .true.
```

```
else
yorn = .false.
end if
end function

elemental function color_factor(num, den, pwr) result (cf)
integer, intent(in) :: num, den, pwr
real(kind=default) :: cf
if (pwr == 0) then
cf = (one * num) / den
else
cf = (one * num) / den * (N_**pwr)
end if
end function color_factor
```

AB.33.6 O'Mega Interface

We want to keep the interface close to the native Fortran code but of course one has to hand over the `vm` additionally

⟨*VM: TBP*⟩+≡

```
procedure :: number_particles_in => vm_number_particles_in
procedure :: number_particles_out => vm_number_particles_out
procedure :: number_color_indices => vm_number_color_indices
procedure :: reset_helicity_selection => vm_reset_helicity_selection
procedure :: new_event => vm_new_event
procedure :: color_sum => vm_color_sum
procedure :: spin_states => vm_spin_states
procedure :: number_spin_states => vm_number_spin_states
procedure :: number_color_flows => vm_number_color_flows
procedure :: flavor_states => vm_flavor_states
procedure :: number_flavor_states => vm_number_flavor_states
procedure :: color_flows => vm_color_flows
procedure :: color_factors => vm_color_factors
procedure :: number_color_factors => vm_number_color_factors
procedure :: is_allowed => vm_is_allowed
procedure :: get_amplitude => vm_get_amplitude
```

⟨*OVM Procedure Implementations*⟩+≡

```
elemental function vm_number_particles_in (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_prt_in
end function vm_number_particles_in

elemental function vm_number_particles_out (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_prt_out
end function vm_number_particles_out

elemental function vm_number_spin_states (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_helicities
end function vm_number_spin_states

pure subroutine vm_spin_states (vm, a)
class(vm_t), intent(in) :: vm
integer, dimension(:,:), intent(out) :: a
a = vm%table_spin
end subroutine vm_spin_states

elemental function vm_number_flavor_states (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
```

```
n = vm%N_flavors
end function vm_number_flavor_states

pure subroutine vm_flavor_states (vm, a)
class(vm_t), intent(in) :: vm
integer, dimension(:,:), intent(out) :: a
a = vm%table_flavor
end subroutine vm_flavor_states

elemental function vm_number_color_indices (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_col_indices
end function vm_number_color_indices

elemental function vm_number_color_flows (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_col_flows
end function vm_number_color_flows

pure subroutine vm_color_flows (vm, a, g)
class(vm_t), intent(in) :: vm
integer, dimension(:,:,:), intent(out) :: a
logical, dimension(:,:), intent(out) :: g
a = vm%table_color_flows
g = vm%table_ghost_flags
end subroutine vm_color_flows

elemental function vm_number_color_factors (vm) result (n)
class(vm_t), intent(in) :: vm
integer :: n
n = vm%N_col_factors
end function vm_number_color_factors

pure subroutine vm_color_factors (vm, cf)
class(vm_t), intent(in) :: vm
type(OCF), dimension(:), intent(out) :: cf
cf = vm%table_color_factors
end subroutine vm_color_factors

! pure & ! pure unless OpenMp
function vm_color_sum (vm, flv, hel) result (amp2)
class(vm_t), intent(in) :: vm
integer, intent(in) :: flv, hel
real(default) :: amp2
amp2 = ovm_color_sum (flv, hel, vm%table_amplitudes, vm%table_color_factors)
end function vm_color_sum

subroutine vm_new_event (vm, p)
class(vm_t), intent(inout) :: vm
real(default), dimension(0:3,*), intent(in) :: p
logical :: mask_dirty
integer :: hel
call vm%run (p)
if ((vm%hel_threshold .gt. 0) .and. (vm%hel_count .le. vm%hel_cutoff)) then
call omega_update_helicity_selection (vm%hel_count, vm%table_amplitudes, &
vm%hel_max_abs, vm%hel_sum_abs, vm%hel_is_allowed, vm%hel_threshold, &
vm%hel_cutoff, mask_dirty)
if (mask_dirty) then
vm%hel_finite = 0
do hel = 1, vm%N_helicities
if (vm%hel_is_allowed(hel)) then
vm%hel_finite = vm%hel_finite + 1
vm%hel_map(vm%hel_finite) = hel
end if
end do
end if
end if
end subroutine vm_new_event

pure subroutine vm_reset_helicity_selection (vm, threshold, cutoff)
class(vm_t), intent(inout) :: vm
real(kind=default), intent(in) :: threshold
integer, intent(in) :: cutoff
integer :: i
vm%hel_is_allowed = .True.
vm%hel_max_abs = 0
vm%hel_sum_abs = 0
vm%hel_count = 0
vm%hel_threshold = threshold
vm%hel_cutoff = cutoff
vm%hel_map = (/(i, i = 1, vm%N_helicities)/)
vm%hel_finite = vm%N_helicities
end subroutine vm_reset_helicity_selection

pure function vm_is_allowed (vm, flv, hel, col) result (yorn)
class(vm_t), intent(in) :: vm
logical :: yorn
integer, intent(in) :: flv, hel, col
yorn = vm%table_flv_col_is_allowed(flv,col) .and. vm%hel_is_allowed(hel)
end function vm_is_allowed

pure function vm_get_amplitude (vm, flv, hel, col) result (amp_result)
class(vm_t), intent(in) :: vm
complex(kind=default) :: amp_result
integer, intent(in) :: flv, hel, col
amp_result = vm%table_amplitudes(flv, col, hel)
end function vm_get_amplitude
```

⟨*Copyleft*⟩≡

—AC—
Index

This index has been generated automatically and might not be 100%ly accurate. In particular, hyperlinks have been observed to be off by one page.