Fixpoint 2.3.2

Bertrand Jeannet

August 30, 2012

Contents

1 Overview **5**
1.1 Organization . . . 5
1.2 Overview . . . 5
1.2.1 Equation system described explicitly by an hypergraph . . . 5
1.2.2 Equation system described implicitly by a successor function . . . 6
1.2.3 Interpreting an equation system . . . 7
1.2.4 Iteration strategies . . . 7
1.3 Example . . . 7

I Main fixpoint module **9**

2 Module `Fixpoint` : Fixpoint analysis of an equation system **10**
2.1 Datatypes . . . 10
2.1.1 Manager . . . 10
2.1.2 Static equation system . . . 12
2.1.3 Dynamically explored equation system . . . 12
2.1.4 Iteration strategies . . . 12
2.1.5 Output . . . 13
2.2 Functions . . . 13
2.3 Printing Functions . . . 15

II Other fixpoint modules **16**

3 Module `FixpointType` : Fixpoint analysis of an equation system: types **17**
3.1 Public datatypes . . . 17
3.1.1 Manager . . . 17
3.1.2 Dynamically explored equation system . . . 19
3.1.3 Iteration strategies . . . 19
3.1.4 Output . . . 21
3.2 Internal datatypes . . . 21
3.3 DOT output . . . 22

4 Module `FixpointStd` : Fixpoint analysis of an equation system: standard method **23**
4.1 Utilities (internal functions) . . . 23
4.2 Initialisation of fixpoint computation . . . 23
4.3 Process a vertex (internal functions) . . . 24
4.4 Full fixpoint algorithm . . . 24

5 Module `FixpointGuided` : Guided fixpoint analysis of an equation system **26**

6 Module `FixpointDyn` : Fixpoint analysis of a dynamically explored equation system **27**

III Auxiliary modules **29**

7 Module `SHGraph` : Oriented hypergraphs **30**
- 7.1 Introduction . . . 30
- 7.2 Generic (polymorphic) interface . . . 31
  - 7.2.1 Statistics . . . 31
  - 7.2.2 Information associated to vertives and edges . . . 32
  - 7.2.3 Membership tests . . . 32
  - 7.2.4 Successors and predecessors . . . 32
  - 7.2.5 Adding and removing elements . . . 32
  - 7.2.6 Iterators . . . 33
  - 7.2.7 Copy and Transpose . . . 33
  - 7.2.8 Algorithms . . . 34
  - 7.2.9 Printing . . . 35
- 7.3 Parameter module for the functor version . . . 36
- 7.4 Signature of the functor version . . . 37
  - 7.4.1 Statistics . . . 37
  - 7.4.2 Information associated to vertives and edges . . . 38
  - 7.4.3 Membership tests . . . 38
  - 7.4.4 Successors and predecessors . . . 38
  - 7.4.5 Adding and removing elements . . . 38
  - 7.4.6 Iterators . . . 38
  - 7.4.7 Copy and Transpose . . . 39
  - 7.4.8 Algorithms . . . 39
  - 7.4.9 Printing . . . 40
- 7.5 Functor . . . 40
- 7.6 Compare interface . . . 40

8 Module `Ilist` : Imbricated lists **43**

9 Module `Sette` : Sets over ordered types (extension of standard library module and polymorphic variant)

10 Module `Hashhe` : Hash tables and hash functions (extension of standard library module) **51**

11 Module `Print` : Printing functions using module `Format` **55**
- 11.1 Printing functions for standard datatypes (lists,arrays,...) . . . 55
- 11.2 Useful functions . . . 56

12 Module `Time` : Small module to compute the duration of computations **57**

Chapter 1

Overview

1.1 Organization

The fixpoint solver library is composed of the following modules:

- `Fixpoint`: this is the only module to look at for a normal user.
- `Example`: example of use of the library;
- `FixpointType`: defines the various types and associated printing functions (including for the DOT output);
- `FixpointStd`: implements the Kleene-Bourdoncle iteration technique [Bou93] (combined with working-set algorithm) for solving equations described explicitly by an hypergraph, and offers base functions for the more sophisticated techniques;
- `FixpointGuided`: implements the Gopan-Reps guided iteration technique [GR07] for equations described explicitly by an hypergraph;
- `FixpointDyn`: exploits an implicit description (under the form if a successor function) of the equation graph, which is then explored dynamically by alternating propagation phase (to detect newly non-empty variables) and upto-convergence-iteration phase.

We also include in this documentation the modules from the `camllib` used by `fixpoint`.

- `SHGraph`: hypergraphs (used to describe equation systems);
- `Ilist`: imbricated lists (used for strongly connected sub-components);
- `Sette`: generalizes the `Set` module of the OCaml standard library;
- `Hashhe`: generalizes the `Hashtbl` module of the OCaml standard library;
- `Print`: provides printing functions for list, arrays, ... using the `Format` module;
- `Time`: for measuring times

1.2 Overview

1.2.1 Equation system described explicitly by an hypergraph

A equation system of the form

$$\left\{\begin{array}{lcl} X_0 &=& X_0^i \sqcup F_0(X_3) \\ X_1 &=& F_1(X_0) \\ X_2 &=& F_2(X_1) \\ X_3 &=& F_3(X_2) \\ X_4 &=& F_4(X_3, X_5) \\ X_5 &=& F_5(X_1) \sqcup F_6(X_4) \end{array}\right. \tag{1.1}$$

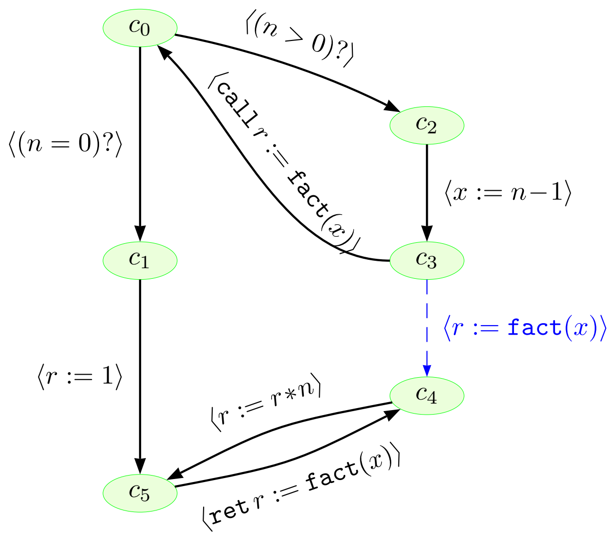

Figure 1.1: Control-Flow-Graph of the Factorial program

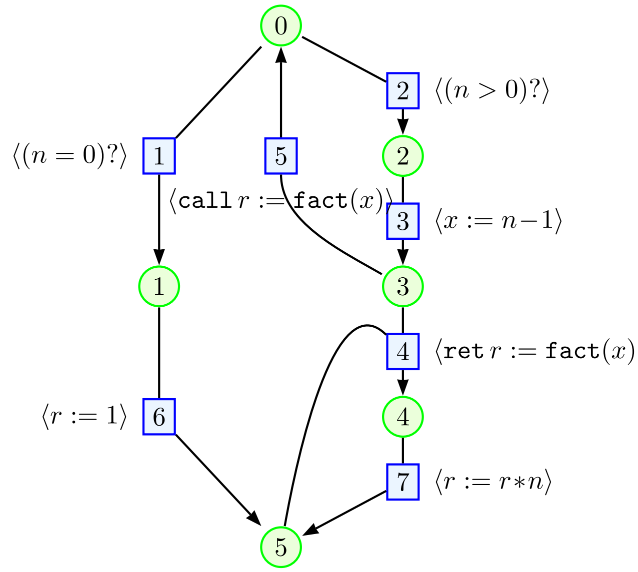

Figure 1.2: Hypergraph representing the equation system of Eqn (1.1)

as generated for instance by the relational interprocedural analysis of the factorial program depicted on Fig. 1.1 will be represented by an hypergraph, depicted on Fig. 1.2, the vertices of which denotes variables, and the hyperedges functions. Hyperedges are needed for functions taking multiple arguments, such as the "combine" function F_4 which models procedure return by combining the value at the call-site (X_3) and the value at the return-site (X_5).

The library use the module `SHGraph` and the type `('vertex, 'hedge, 'a,'b,'c) SHGraph.t` to represent such hypergraphs. This provided datatype distinguishes vertex and hyperedge identifiers from the information associated to them.

1.2.2 Equation system described implicitly by a successor function

Alternatively, the equation system (1.1) can be described by the following function:

$$\begin{array}{lcl}
0 & \longmapsto & \left\{ \begin{array}{lcl} \boxed{1} & \mapsto & ([0],\ 1) \\ \boxed{2} & \mapsto & ([0],\ 2) \end{array} \right\} \\
1 & \longmapsto & \left\{ \boxed{5} \mapsto ([1],\ 5) \right\} \\
2 & \longmapsto & \left\{ \boxed{3} \mapsto ([2],\ 3) \right\} \\
3 & \longmapsto & \left\{ \boxed{4} \mapsto ([3;5],\ 4) \right\} \\
4 & \longmapsto & \left\{ \boxed{6} \mapsto ([4],\ 5) \right\} \\
5 & \longmapsto & \left\{ \boxed{4} \mapsto ([3;5],\ 4) \right\}
\end{array} \tag{1.2}$$

This function associates to each vertex v a map associating to each successor hyperedge h a pair composed of (i) the array of predecessor vertex of h (including v), and (ii) the successor vertex of h.

The library use the type `('vertex,'hedge) Fixpoint.equation` to describe such a function.

1.2.3 Interpreting an equation system

`Fixpoint` is parameterized by an object of type [('vertex, 'hedge, 'abstract, 'arc) Fixpoint.manager], which defines the type of vertex and hyperedge identifiers, the type of abstract values attached to vertices (which is the value of the variable), and the type of values attached to hyperedges (optional, maybe `unit`). The manager defines the needed functions on abstract values, the function that computes the effect of an hyperedge, and a set of options and debugging functions.

1.2.4 Iteration strategies

Fixpoint solving iterations are parametrized by an object of type `('vertex,'hedge) Fixpoint.strategy`. We refer to [Bou93] for complete explanations, and to the documentation of module `Fixpoint`. In short, a strategy is an imbricated list like $[1;[[2;3];4];[5];6]$, which means, during the iterative solving:

1. update vertex 1;
2. update 2 and 3, and loop until stabilization;
3. update 4, and come back to 2. until stabilization;
4. update 5 and loop until stabilization;
5. update 6 and ends the analysis.

[Bou93] represents such an iteration strategy by the regular expression $(1\ ((2\ 3)^*\ 4)^*\ (5^*)\ 6)$. Alternatively, one could use a flatter iteration strategy like $(1\ (2\ 3\ 4)^*\ (5^*)\ 6)$.
Standard users (as myself) is advised to use the function `Fixpoint.make_strategy` to generate correct strategies from an hypergraph.

1.3 Example

I suggest to look at the example file `example.ml` and to play a bit with it.

Bibliography

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, 1993.

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, August 2007.

Part I

Main fixpoint module

Chapter 2

Module Fixpoint : Fixpoint analysis of an equation system

2.1 Datatypes

2.1.1 Manager

The manager parameterizes the fixpoint solver with the following types:

- 'vertex: type of vertex/variable (identifier) in the hypergraph describing the equation system;
- 'hedge: type of hyperedge/function (identifier) in the hypergraph describing the equation system;
- 'abstract: type of abstract values associated to vertices/variables;
- 'arc: type of information associated to hyperedges/functions;

and values:

- Lattice operations on abstract values; the argument of type 'vertex indicates to which vertex the result of the function is associated (useful when abstract values are typed by an environment, for instance)
- Functions to initialize variables and to interpret hyperedges/functions;
- Printing functions for type parameters;
- Options, mainly about widening;
- Debugging options, for text (and possibly DOT) output on the formatter print_fmt;
- Printing functions for DOT output on the optional formatter dot_fmt.

```
type ('vertex, 'hedge, 'abstract, 'arc) manager = ('vertex, 'hedge, 'abstract, 'arc) Fixpoint-
Type.manager = {
  mutable bottom : 'vertex -> 'abstract ;
```

Create a bottom value

```
  mutable canonical : 'vertex -> 'abstract -> unit ;
```

Make an abstract value canonical

```
  mutable is_bottom : 'vertex -> 'abstract -> bool ;
```

Emptiness test

```
mutable is_leq : 'vertex -> 'abstract -> 'abstract -> bool ;
```

Inclusion test

```
mutable join : 'vertex -> 'abstract -> 'abstract -> 'abstract ;
mutable join_list : 'vertex -> 'abstract list -> 'abstract ;
```

Binary and n-ary join operation

```
mutable widening : 'vertex -> 'abstract -> 'abstract -> 'abstract ;
```

Apply widening at the given point, with the two arguments. Widening will always be applied with first argument being included in the second one.

```
mutable odiff : ('vertex -> 'abstract -> 'abstract -> 'abstract) option ;
```

Sound approximation of set difference (optional)

```
mutable abstract_init : 'vertex -> 'abstract ;
```

Return the non-bottom initial value associated to the given vertex

```
mutable arc_init : 'hedge -> 'arc ;
```

Initial value for arcs

```
mutable apply : 'hedge -> 'abstract array -> 'arc * 'abstract ;
```

Apply the function indexed by `hedge` to the array of arguments.

It returns the new abstract value, but also a user-defined information that will be associated to the hyperedge in the result.

```
mutable print_vertex : Format.formatter -> 'vertex -> unit ;
mutable print_hedge : Format.formatter -> 'hedge -> unit ;
mutable print_abstract : Format.formatter -> 'abstract -> unit ;
mutable print_arc : Format.formatter -> 'arc -> unit ;
```

Printing functions

```
mutable accumulate : bool ;
```

If true, during ascending phase, compute the union of old reachable value with growing incoming hyperedges. If false, recompute all incoming hyperedges.

```
mutable print_fmt : Format.formatter ;
```

Typically equal to `Format.std_formatter`

```
mutable print_analysis : bool ;
mutable print_component : bool ;
mutable print_step : bool ;
mutable print_state : bool ;
mutable print_postpre : bool ;
mutable print_workingsets : bool ;
```

Printing Options

```
mutable dot_fmt : Format.formatter option ;
```

`Some fmt` enables DOT output. You can set dummy values to the fields below if you always set `None` and you do not want DOT output.

```
mutable dot_vertex : Format.formatter -> 'vertex -> unit ;
```

Print vertex identifiers in DOT format

```
mutable dot_hedge : Format.formatter -> 'hedge -> unit ;
```

Print hyperedge identifiers in DOT format (vertices and hyperedges identifiers should be different, as they are represented by DOT vertices

```
mutable dot_attrvertex : Format.formatter -> 'vertex -> unit ;
```

Print the displayed information in boxes

```
  mutable dot_attrhedge : Format.formatter -> 'hedge -> unit ;
```

Print the displayed information for hyperedges

```
}
```

2.1.2 Static equation system

A static equation system is defined by an hypergraph, of type `PSHGraph.t`, see `Fixpoint.analysis_std`[2.2] and `Fixpoint.analysis_guided`[2.2]

2.1.3 Dynamically explored equation system

```
type ('vertex, 'hedge) equation = 'vertex -> ('hedge, 'vertex array * 'vertex) PMappe.t
```

Function that explores dynamically an equation system. `equation vertex` returns a map hat associates to each successor hyperedge a pair of composed of the set of predecessor vertices, and the successor vertex.

2.1.4 Iteration strategies

```
type strategy_iteration = FixpointType.strategy_iteration = {
  mutable widening_start : int ;
```

Nb of initial steps without widening in the current strategy

```
  mutable widening_descend : int ;
```

Maximum nb. of descending steps in the current strategy

```
  mutable ascending_nb : int ;
```

For stats

```
  mutable descending_nb : int ;
```

For stats

```
  mutable descending_stable : bool ;
```

For stats

```
}
```

Widening and Descending Options

```
type ('vertex, 'hedge) strategy_vertex = ('vertex, 'hedge) FixpointType.strategy_vertex = {
  mutable vertex : 'vertex ;
  mutable hedges : 'hedge list ;
```

Order in which the incoming hyperedges will be applied

```
  mutable widen : bool ;
```

Should this vertex be a widening point ?

```
}
```

Strategy to be applied for the vertex `vertex`.

- `hedges` is a list of incoming hyperedges. The effect of hyperedges are applied ”in parallel” and the destination vertex is updated. Be cautious: if an incoming hyperedge is forgotten in this list, it won't be taken into account in the analysis.
- `widen` specifies whether the vertex is a widening point or not.

```
type ('vertex, 'hedge) strategy = (strategy_iteration, ('vertex, 'hedge) strategy_vertex)
  Ilist.t
```

Type for defining iteration strategies. For instance, `[1; [2;3]; 4; [5]; 6]` means:

- update `1`;
- update `2` then `3`, and loop until stabilization;
- update `4`;
- update `5` and loop until stabilization;
- update `6` and ends the analysis.

Moreover, to each (imbricated) list is associated a record of type `strategy_iteration`, which indicates when to start the widening, and the maximum number of descending iterations.

Some observations on this example:

- The user should specify correctly the strategy. Two vertices belonging to the same connex component should always belong to a loop. Here, if there is an edge from `6` to `2`, the loop will not be iterated.
- A vertex may appear more than once in the strategy, if it is useful.
- Definition of the set of widening point is independent from the order
- of application, here. it is alos the user-responsability to ensure that
- the computation will end.

So-called stabilization loops can be recursive, like that: `[1; [2; [3;4]; [5]]; 6]`, where the loop `[3;4]` needs to be (temporarily stable) before going on with `5`.

2.1.5 Output

```
type stat_iteration = FixpointType.stat_iteration = {
  mutable nb : int ;
  mutable stable : bool ;
}
```

```
type stat = FixpointType.stat = {
  mutable time : float ;
  mutable ascending : (stat_iteration, unit) Ilist.t ;
  mutable descending : (stat_iteration, unit) Ilist.t ;
}
```

statistics at the end of the analysis

```
type ('vertex, 'hedge, 'abstract, 'arc) output = ('vertex, 'hedge, 'abstract, 'arc, stat) PSHGraph.t
```

result of the analysis

2.2 Functions

```
val make_strategy_default :
  ?depth:int ->
  ?widening_start:int ->
  ?widening_descend:int ->
  ?priority:'hedge PSHGraph.priority ->
  vertex_dummy:'vertex ->
  hedge_dummy:'hedge ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t -> ('vertex, 'hedge) strategy
```

Build a "default" strategy, with the following options:

- depth: to apply the recursive strategy of Bourdoncle's paper. Default value is 2, which means that Strongly Connected Components are stabilized independently: the iteration order looks like `(1 2 (3 4 5) 6 (7 8) 9)` where `1, 2, 6, 9` are SCC by themselves. A higher value defines a more recursive behavior, like `(1 2 (3 (4 5)) 6 (7 (8)) 9)`.

- iteration: for each strategy, nb if initial steps without widening and max nb. of descending steps (the latter being used only for depth 2). Default is `(0,1)`.

- priority: specify which hedges should be taken into account in the computation of the iteration order and the widening points (the one such that `priority h >= 0` and the widening points, and also indicates which hyperedge should be explored first at a point of choice.

One known usage for filtering: guided analysis, where one analyse a subgraph of the equation graph.

```
val analysis_std :
  ('vertex, 'hedge, 'abstract, 'arc) manager ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t ->
  ('vertex, 'hedge) strategy ->
  ('vertex, 'hedge, 'abstract, 'arc) output
```

Performs initialization, fixpoint analysis and descending, and measures the global analysis time.

`analysis_std manager graph sinit strategy` takes a graph giving the structure of the equation system, a manager indicating how to interpret the equation system, a (super)set `sinit` of the variables to be initialized to a non-empty value, and an iteration strategy `strategy`.

```
val analysis_guided :
  ('vertex, 'hedge, 'attr, 'arc) manager ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t ->
  (('hedge -> bool) -> ('vertex, 'hedge) strategy) ->
  ('vertex, 'hedge, 'attr, 'arc) output
```

Same as `Fixpoint.analysis_std`[2.2], but with the technique of Gopan and Reps published in Static Anlaysis Symposium, SAS'2007.

`analysis_guided manager graph sinit make_strategy`: compared to `Fixpoint.analysis_std`[2.2], instead of providing a strategy, one provides a function `make_strategy` generating strategies, which takes as input a function filtering the edges to be considered. A typical value for the argument `make_strategy` is `(fun p -> make_strategy_default ~priority:(PSHGraph.Filter p) vdummy hdummy graph sinit)`.

```
val equation_of_graph :
  ?filter:('hedge -> bool) ->
  ('vertex, 'hedge, 'attr, 'arc, 'e) PSHGraph.t ->
  ('vertex, 'hedge) equation
```

Generate from a graph a function of type `('vertex, 'hedge) equation` or dynamically exploring the graph. The `filter` function allows to select a part of the graph.

```
val graph_of_equation :
  ('vertex, 'hedge) PSHGraph.compare ->
  ?filter:('hedge -> bool) ->
  make_attrvertex:('vertex -> 'attr) ->
  make_attrhedge:('hedge -> 'arc) ->
```

```
  info:'e ->
  ('vertex, 'hedge) equation ->
  'vertex PSette.t -> ('vertex, 'hedge, 'attr, 'arc, 'e) PSHGraph.t
```

Generate from an equation a graph, using `make_attrvertex`, `make_attrhedge` and `info`.

```
val analysis_dyn :
  ('a, 'b) SHGraph.compare ->
  guided:bool ->
  ('a, 'b, 'c, 'd) manager ->
  ('a, 'b) equation ->
  'a PSette.t ->
  (('a, 'b, 'c, 'd) FixpointType.graph -> ('a, 'b) strategy) ->
  ('a, 'b, 'c, 'd) output
```

Dynamic analysis.

2.3 Printing Functions

```
val print_strategy_vertex :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b) strategy_vertex -> unit
```

`print_strategy_vertex man fmt sv` prints an object of type `strategy_vertex`, using the manager `man` for printing vertices and hyperedges. The output has the form `(boolean,vertex,[list of list of hedges])`.

```
val print_strategy :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b) strategy -> unit
```

`print_strategy_vertex man fmt sv` prints an object of type `strategy`, using the manager `man` for printing vertices and hyperedges.

```
val print_stat : Format.formatter -> stat -> unit
```

Prints statistics

```
val print_output :
  ('vertex, 'hedge, 'attr, 'arc) manager ->
  Format.formatter -> ('vertex, 'hedge, 'attr, 'arc) output -> unit
```

Prints the result of an analysis.

Part II

Other fixpoint modules

Chapter 3

Module FixpointType : Fixpoint analysis of an equation system: types

3.1 Public datatypes

3.1.1 Manager

The manager parameterizes the fixpoint solver with the following types:

- 'vertex: type of vertex/variable (identifier) in the hypergraph describing the equation system;
- 'hedge: type of hyperedge/function (identifier) in the hypergraph describing the equation system;
- 'abstract: type of abstract values associated to vertices/variables;
- 'arc: type of information associated to hyperedges/functions;

and values:

- Lattice operations on abstract values; the argument of type 'vertex indicates to which vertex the result of the function is associated (useful when abstract values are typed by an environment, for instance)
- Functions to initialize variables and to interpret hyperedges/functions;
- Printing functions for type parameters;
- Options, mainly about widening;
- Debugging options, for text (and possibly DOT) output on the formatter `print_fmt`;
- Printing functions for DOT output on the optional formatter `dot_fmt`.

```
type ('vertex, 'hedge, 'abstract, 'arc) manager = {
  mutable bottom : 'vertex -> 'abstract ;
```

Create a bottom value

```
  mutable canonical : 'vertex -> 'abstract -> unit ;
```

Make an abstract value canonical

```
  mutable is_bottom : 'vertex -> 'abstract -> bool ;
```

Emptiness test

```
  mutable is_leq : 'vertex -> 'abstract -> 'abstract -> bool ;
```

Inclusion test

```
mutable join : 'vertex -> 'abstract -> 'abstract -> 'abstract ;
mutable join_list : 'vertex -> 'abstract list -> 'abstract ;
```

Binary and n-ary join operation

```
mutable widening : 'vertex -> 'abstract -> 'abstract -> 'abstract ;
```

Apply widening at the given point, with the two arguments. Widening will always be applied with first argument being included in the second one.

```
mutable odiff : ('vertex -> 'abstract -> 'abstract -> 'abstract) option ;
```

Sound approximation of set difference (optional)

```
mutable abstract_init : 'vertex -> 'abstract ;
```

Return the non-bottom initial value associated to the given vertex

```
mutable arc_init : 'hedge -> 'arc ;
```

Initial value for arcs

```
mutable apply : 'hedge -> 'abstract array -> 'arc * 'abstract ;
```

Apply the function indexed by `hedge` to the array of arguments.

It returns the new abstract value, but also a user-defined information that will be associated to the hyperedge in the result.

```
mutable print_vertex : Format.formatter -> 'vertex -> unit ;
mutable print_hedge : Format.formatter -> 'hedge -> unit ;
mutable print_abstract : Format.formatter -> 'abstract -> unit ;
mutable print_arc : Format.formatter -> 'arc -> unit ;
```

Printing functions

```
mutable accumulate : bool ;
```

If true, during ascending phase, compute the union of old reachable value with growing incoming hyperedges. If false, recompute all incoming hyperedges.

```
mutable print_fmt : Format.formatter ;
```

Typically equal to `Format.std_formatter`

```
mutable print_analysis : bool ;
mutable print_component : bool ;
mutable print_step : bool ;
mutable print_state : bool ;
mutable print_postpre : bool ;
mutable print_workingsets : bool ;
```

Printing Options

```
mutable dot_fmt : Format.formatter option ;
```

`Some fmt` enables DOT output. You can set dummy values to the fields below if you always set `None` and you do not want DOT output.

```
mutable dot_vertex : Format.formatter -> 'vertex -> unit ;
```

Print vertex identifiers in DOT format

```
mutable dot_hedge : Format.formatter -> 'hedge -> unit ;
```

Print hyperedge identifiers in DOT format (vertices and hyperedges identifiers should be different, as they are represented by DOT vertices

```
mutable dot_attrvertex : Format.formatter -> 'vertex -> unit ;
```

Print the displayed information in boxes

```
mutable dot_attrhedge : Format.formatter -> 'hedge -> unit ;
```

Print the displayed information for hyperedges

```
}
```

3.1.2 Dynamically explored equation system

```
type ('vertex, 'hedge) equation = 'vertex -> ('hedge, 'vertex array * 'vertex) PMappe.t
```

Function that explores dynamically an equation system. `equation vertex` returns a map hat associates to each successor hyperedge a pair of composed of the set of predecessor vertices, and the successor vertex.

3.1.3 Iteration strategies

```
type strategy_iteration = {
  mutable widening_start : int ;
```

Nb of initial steps without widening in the current strategy

```
  mutable widening_descend : int ;
```

Maximum nb. of descending steps in the current strategy

```
  mutable ascending_nb : int ;
```

For stats

```
  mutable descending_nb : int ;
```

For stats

```
  mutable descending_stable : bool ;
```

For stats

```
}
```

Widening and Descending Options

```
type ('vertex, 'hedge) strategy_vertex = {
  mutable vertex : 'vertex ;
  mutable hedges : 'hedge list ;
```

Order in which the incoming hyperedges will be applied

```
  mutable widen : bool ;
```

Should this vertex be a widening point ?

```
}
```

Strategy to be applied for the vertex `vertex`.

- `hedges` is a list of incoming hyperedges. The effect of hyperedges are applied ”in parallel” and the destination vertex is updated. Be cautious: if an incoming hyperedge is forgotten in this list, it won’t be taken into account in the analysis.
- `widen` specifies whether the vertex is a widening point or not.

```
type ('vertex, 'hedge) strategy = (strategy_iteration,
   ('vertex, 'hedge) strategy_vertex)
  Ilist.t
```

Type for defining iteration strategies. For instance, `[1; [2;3]; 4; [5]; 6]` means:

- update `1`;
- update `2` then `3`, and loop until stabilization;
- update `4`;
- update `5` and loop until stabilization;
- update `6` and ends the analysis.

Moreover, to each (imbricated) list is associated a record of type `strategy_iteration`, which indicates when to start the widening, and the maximum number of descending iterations.

Some observations on this example:

- The user should specify correctly the strategy. Two vertices belonging to the same connex component should always belong to a loop. Here, if there is an edge from `6` to `2`, the loop will not be iterated.
- A vertex may appear more than once in the strategy, if it is useful.
- Definition of the set of widening point is independent from the order
- of application, here. it is alos the user-responsability to ensure that
- the computation will end.

So-called stabilization loops can be recursive, like that: `[1; [2; [3;4]; [5]]; 6]`, where the loop `[3;4]` needs to be (temporarily stable) before going on with `5`.

```
val print_strategy_iteration : Format.formatter -> strategy_iteration -> unit
```

```
val print_strategy_vertex :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b) strategy_vertex -> unit
```

`print_strategy_vertex man fmt sv` prints an object of type `strategy_vertex`, using the manager `man` for printing vertices and hyperedges. The output has the form `(boolean,vertex,[list of list of hedges])`.

```
val print_strategy :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b) strategy -> unit
```

`print_strategy_vertex man fmt sv` prints an object of type `strategy`, using the manager `man` for printing vertices and hyperedges.

```
val make_strategy_iteration :
  ?widening_start:int ->
  ?widening_descend:int -> unit -> strategy_iteration
```

```
val make_strategy_default :
  ?depth:int ->
  ?widening_start:int ->
  ?widening_descend:int ->
  ?priority:'hedge PSHGraph.priority ->
  vertex_dummy:'vertex ->
  hedge_dummy:'hedge ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t -> ('vertex, 'hedge) strategy
```

Build a "default" strategy, with the following options:

- depth: to apply the recursive strategy of Bourdoncle's paper. Default value is 2, which means that Strongly Connected Components are stabilized independently: the iteration order looks like `(1 2 (3 4 5) 6 (7 8) 9)` where `1, 2, 6, 9` are SCC by themselves. A higher value defines a more recursive behavior, like `(1 2 (3 (4 5)) 6 (7 (8)) 9)`.

- iteration: for each strategy, nb if initial steps without widening and max nb. of descending steps (the latter being used only for depth 2). Default is `(0,1)`.

- priority: specify which hedges should be taken into account in the computation of the iteration order and the widening points (the one such that `priority h >= 0` and the widening points, and also indicates which hyperedge should be explored first at a point of choice.

One known usage for filtering: guided analysis, where one analyse a subgraph of the equation graph.

3.1.4 Output

```
type stat_iteration = {
  mutable nb : int ;
  mutable stable : bool ;
}
```

```
type stat = {
  mutable time : float ;
  mutable ascending : (stat_iteration, unit) Ilist.t ;
  mutable descending : (stat_iteration, unit) Ilist.t ;
}
```

```
type ('vertex, 'hedge, 'abstract, 'arc) output = ('vertex, 'hedge, 'abstract, 'arc, stat) PSHGraph.t
```

result of the analysis

```
val ilist_map_condense : ('a -> 'c) -> ('a, 'b) Ilist.t -> ('c, 'd) Ilist.t
```

```
val stat_iteration_merge : (stat_iteration, unit) Ilist.t -> stat_iteration
```

```
val print_stat_iteration : Format.formatter -> stat_iteration -> unit
```

```
val print_stat_iteration_ilist :
  Format.formatter -> (stat_iteration, 'a) Ilist.t -> unit
```

```
val print_stat : Format.formatter -> stat -> unit
```

Prints statistics

```
val print_output :
  ('vertex, 'hedge, 'attr, 'arc) manager ->
  Format.formatter ->
  ('vertex, 'hedge, 'attr, 'arc) output -> unit
```

Prints the result of an analysis.

3.2 Internal datatypes

```
type 'abstract attr = {
  mutable reach : 'abstract ;
  mutable diff : 'abstract ;
  mutable empty : bool ;
}
```

```
type 'arc arc = {
  mutable arc : 'arc ;
  mutable aempty : bool ;
}
```

```
type ('vertex, 'hedge) infodyn = {
  mutable iaddhedge : ('hedge, 'vertex array * 'vertex) PHashhe.t ;
  iequation : ('vertex, 'hedge) equation ;
}
```

```
type ('vertex, 'hedge) info = {
  iinit : 'vertex PSette.t ;
  itime : float Pervasives.ref ;
  mutable iascending : (stat_iteration, unit) Ilist.t ;
```

```
  mutable idescending : (stat_iteration, unit) Ilist.t ;
  mutable iworkvertex : ('vertex, unit) PHashhe.t ;
  mutable iworkhedge : ('hedge, unit) PHashhe.t ;
  iinfodyn : ('vertex, 'hedge) infodyn option ;
}
```

```
type ('vertex, 'hedge, 'abstract, 'arc) graph = ('vertex, 'hedge, 'abstract attr, 'arc arc,
   ('vertex, 'hedge) info)
  PSHGraph.t
```

```
val print_attr :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> 'c attr -> unit
```

```
val print_arc :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> 'd arc -> unit
```

```
val print_info :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b) info -> unit
```

```
val print_workingsets :
  ('a, 'b, 'c, 'd) manager ->
  Format.formatter -> ('a, 'b, 'c, 'd) graph -> unit
```

```
val print_graph :
  ('vertex, 'hedge, 'attr, 'arc) manager ->
  Format.formatter -> ('vertex, 'hedge, 'attr, 'arc) graph -> unit
```

Prints internal graph.

3.3 DOT output

```
val dot_graph :
  ?style:string ->
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle:string ->
  ?strategy:('a, 'b) strategy ->
  ?vertex:'a ->
  ('a, 'b, 'c, 'd) manager ->
  ('a, 'b, 'c, 'd) graph -> title:string -> unit
```

Prints internal graph on the (optional) formatter `man.dot_fmt`, see type `FixpointType.manager`[3.1.1].

Chapter 4

Module `FixpointStd` : Fixpoint analysis of an equation system: standard method

4.1 Utilities (internal functions)

```
val is_tvertex :
  ('vertex, 'hedge, 'abstract, 'arc) FixpointType.graph ->
  'vertex array -> bool
```

Does the array of vertices belong to the graph, all with non bottom values ?

```
val treach_of_tvertex :
  descend:bool ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  'vertex array -> 'attr array
```

Return the array of abstract values associated to the vertices

```
val update_workingsets :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  hedge:bool -> 'vertex -> unit
```

Update working sets assuming that the abstract value associated to the vertex has been modified. If `hedge=true`, then also consider the working set associated to hyperhedges.

4.2 Initialisation of fixpoint computation

```
val init :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t -> ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph
```

`init manager input sinit` creates the internal graph structure (from the equation graph `input`) and initialize the working sets (from the set of initial points `sinit`) (stored in the info field of the internal graph).

4.3 Process a vertex (internal functions)

```
val accumulate_vertex :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  ('vertex, 'hedge) FixpointType.strategy_vertex ->
  'attr FixpointType.attr -> bool
```

Compute the least upper bound of the current value of the vertex/variable with the values propagated by the incoming hyperedges belonging to the working set. Returns `true` if the value is strictly increasing.

```
val propagate_vertex :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  descend:bool ->
  ('vertex, 'hedge) FixpointType.strategy_vertex ->
  'attr FixpointType.attr -> bool
val process_vertex :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  widening:bool -> ('vertex, 'hedge) FixpointType.strategy_vertex -> bool
```

4.4 Full fixpoint algorithm

```
val process_strategy :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  depth:int -> ('vertex, 'hedge) FixpointType.strategy -> bool
val descend_strategy :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  ('vertex, 'hedge) FixpointType.strategy -> bool
```

Internal functions

```
val descend :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  ('vertex, 'hedge) FixpointType.strategy -> bool
```

(Rather internal)

`descend manager graph strategy` performs descending iterations on the part of the graph represented by the strategy

```
val process_toplevel_strategy :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  ('vertex, 'hedge) FixpointType.strategy -> bool * bool
```

(Rather internal function)

`process_toplevel_strategy manager graph strategy`: assuming that `graph` has been created with the function `FixpointStd.init`[4.2], this function solves iteratively the fixpoint equation, using `manager` for interpreting the equation system `graph`, and the strategy `strategy` for the iteration order and the application of widening.

Descending iterations are applied separately to each upper-level component of the strategy, before processing the next such component in the strategy.

The first returned Boolean indicates if some growth has been observed (before descending iteration), and the second one indicates if some reduction has been observed after descending iteration.

```
val output_of_graph :
  ('vertex, 'hedge, 'abstract, 'arc) FixpointType.graph ->
  ('vertex, 'hedge, 'abstract, 'arc) FixpointType.output
```

(Rather internal function)

Getting the result of the analysis from the internal representation.

```
val analysis :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t ->
  ('vertex, 'hedge) FixpointType.strategy ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.output
```

Main user function: `analysis manager equation_graph sinit strategy` takes a graph giving the structure of the equation system, a manager indicating how to interpret the equation system, a (super)set of the variables to be initialized to a non-empty value, and an iteration strategy. It returns the result of the full analysis with an object of type `Fixpoint.output`[2.1.5].

Chapter 5

Module `FixpointGuided` : Guided fixpoint analysis of an equation system

Technique of `Gopand and Reps, SAS'07`

```
val analysis :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'e, 'f, 'g) PSHGraph.t ->
  'vertex PSette.t ->
  (('hedge -> bool) -> ('vertex, 'hedge) FixpointType.strategy) ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.output
```

Same as `FixpointStd.analysis`[4.4], but with the technique of Gopan and Reps published in Static Anlaysis Symposium, SAS'2007.

```
val add_active_hedges :
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'attr, 'arc) FixpointType.graph ->
  ('hedge, unit) PHashhe.t -> bool
```

Chapter 6

Module FixpointDyn : Fixpoint analysis of a dynamically explored equation system

```
val init :
  ('vertex, 'hedge) SHGraph.compare ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.manager ->
  ('vertex, 'hedge) FixpointType.equation ->
  'vertex PSette.t -> ('vertex, 'hedge, 'abs, 'arc) FixpointType.graph
```

```
val propagate :
  guided:bool ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.graph -> bool
```

```
val fixpoint :
  guided:bool ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.manager ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.graph ->
  (('vertex, 'hedge, 'abs, 'arc) FixpointType.graph ->
   ('vertex, 'hedge) FixpointType.strategy) ->
  bool
```

```
val analysis :
  ('vertex, 'hedge) SHGraph.compare ->
  guided:bool ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.manager ->
  ('vertex, 'hedge) FixpointType.equation ->
  'vertex PSette.t ->
  (('vertex, 'hedge, 'abs, 'arc) FixpointType.graph ->
   ('vertex, 'hedge) FixpointType.strategy) ->
  ('vertex, 'hedge, 'abs, 'arc) FixpointType.output
```

```
val equation_of_graph :
  ?filter:('hedge -> bool) ->
  ('vertex, 'hedge, 'attr, 'arc, 'e) PSHGraph.t ->
  ('vertex, 'hedge) FixpointType.equation
```

```
val graph_of_equation :
  ('vertex, 'hedge) SHGraph.compare ->
  ?filter:('hedge -> bool) ->
  make_attrvertex:('vertex -> 'attr) ->
```

```
make_attrhedge:('hedge -> 'arc) ->
info:'e ->
('vertex, 'hedge) FixpointType.equation ->
'vertex PSette.t -> ('vertex, 'hedge, 'attr, 'arc, 'e) PSHGraph.t
```

Part III

Auxiliary modules

Chapter 7

Module SHGraph : Oriented hypergraphs

7.1 Introduction

This module provides an abstract datatypes and functions for manipulating hypergraphs, that is, graphs where edges relates potentially more than 2 vertices. The considered hypergraphs are *oriented*: one distinguishes for a vertex incoming and outgoing hyperedges, and for an hyperedge incoming (or origin) and outgoing (or destination) vertices.

Origin and destination vertices of an hyperedge are ordered (by using arrays), in contrast with incoming and outgoing hyperedges of a vertex.

A possible use of such hypergraphs is the representation of a (fixpoint) equation system, where the unknown are the vertices and the functions the hyperedges, taking a vector of unknowns as arguments and delivering a vector of results.

A last note about the notion of connectivity, which is relevant for operations like depth-first-search, reachability and connex components notions. A destination vertex of an hyperedge is considered as reachable from an origin vertex through this hyperedge only if *all* origin vertices are reachable.

```
type 'a priority =
  | Filter of ('a -> bool)
  | Priority of ('a -> int)
```

Filtering or priority function (used by `SHGraph.topological_sort`[7.2.8], `SHGraph.cfc`[7.2.8], `SHGraph.scfc`[7.2.8], `SHGraph.topological_sort_multi`[7.2.8], `SHGraph.cfc_multi`[7.2.8], `SHGraph.scfc_multi`[7.2.8]).

`Filter p` specifies `p` as a filtering function for hyperedges: only those satisfying `p` are taken into account.

`Priority p` specifies `p` as a priority function. Hyperedges `h` with `p h < 0` are not taken into account. Otherwise, hyperedges with highest priority are explored first.

```
type ('a, 'b) compare = {
  hashv : 'a Hashhe.compare ;
  hashh : 'b Hashhe.compare ;
  comparev : 'a -> 'a -> int ;
  compareh : 'b -> 'b -> int ;
}
```

```
type ('b, 'c) vertex_n = {
  attrvertex : 'c ;
  mutable predhedge : 'b Sette.set ;
  mutable succhedge : 'b Sette.set ;
```

```
}
type ('a, 'd) hedge_n = {
  attrhedge : 'd ;
  predvertex : 'a array ;
  succvertex : 'a array ;
}
type ('a, 'b, 'c, 'd, 'e) graph = {
  vertex : ('a, ('b, 'c) vertex_n) Hashhe.hashtbl ;
  hedge : ('b, ('a, 'd) hedge_n) Hashhe.hashtbl ;
  info : 'e ;
}
```

7.2 Generic (polymorphic) interface

`val stdcompare : ('a, 'b) compare`

`type ('a, 'b, 'c, 'd, 'e) t = ('a, 'b, 'c, 'd, 'e) graph`

The type of hypergraphs where

- 'a : type of vertices
- 'b : type of hedges
- 'c : information associated to vertices
- 'd : information associated to hedges
- 'e : user-information associated to an hypergraph

`val create : int -> 'e -> ('a, 'b, 'c, 'd, 'e) t`

`create n data` creates an hypergraph, using `n` for the initial size of internal hashtables, and `data` for the user information

`val clear : ('a, 'b, 'c, 'd, 'e) t -> unit`

Remove all vertices and hyperedges of the graph.

`val is_empty : ('a, 'b, 'c, 'd, 'e) t -> bool`

Is the graph empty ?

7.2.1 Statistics

`val size_vertex : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of vertices in the hypergraph

`val size_hedge : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of hyperedges in the hypergraph

`val size_edgevh : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of edges (vertex,hyperedge) in the hypergraph

`val size_edgehv : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of edges (hyperedge,vertex) in the hypergraph

`val size : ('a, 'b, 'c, 'd, 'e) t -> int * int * int * int`

`size graph` returns `(nbvertex,nbhedge,nbedgevh,nbedgehv)`

7.2.2 Information associated to vertives and edges

`val attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c`

`attrvertex graph vertex` returns the information associated to the vertex `vertex`

`val attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd`

`attrhedge graph hedge` returns the information associated to the hyperedge `hedge`

`val info : ('a, 'b, 'c, 'd, 'e) t -> 'e`

`info g` returns the user-information attached to the graph `g`

7.2.3 Membership tests

```
val is_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> bool
val is_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> bool
```

7.2.4 Successors and predecessors

`val succhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b Sette.t`

Successor hyperedges of a vertex

`val predhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b Sette.t`

Predecessor hyperedges of a vertex

`val succvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array`

Successor vertices of an hyperedge

`val predvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array`

Predecessor vertices of an hyperedge

`val succ_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a Sette.t`

Successor vertices of a vertex by any hyperedge

`val pred_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a Sette.t`

Predecessor vertices of a vertex by any hyperedge

7.2.5 Adding and removing elements

`val add_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit`

Add a vertex

```
val add_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  'b -> 'd -> pred:'a array -> succ:'a array -> unit
```

Add an hyperedge. The predecessor and successor vertices should already exist in the graph. Otherwise, a `Failure` exception is raised.

`val replace_attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit`

Change the attribute of an existing vertex

`val replace_attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd -> unit`

Change the attribute of an existing hyperedge

```
val remove_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> unit
```

Remove the vertex from the graph, as well as all related hyperedges.

```
val remove_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> unit
```

Remove the hyperedge from the graph.

7.2.6 Iterators

```
val iter_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b Sette.t -> succ:'b Sette.t -> unit) -> unit
```

Iterates the function `f vertex attrvertex succhedges predhedges` to all vertices of the graph. `succhedges` (resp. `predhedges`) is the set of successor (resp. predecessor) hyperedges of the vertex

```
val iter_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> unit) -> unit
```

Iterates the function `f hedge attrhedge succvertices predvertices` to all hyperedges of the graph. `succvertices` (resp. `predvertices`) is the set of successor (resp. predecessor) vertices of the hyperedge

Below are the `fold` versions of the previous functions.

```
val fold_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b Sette.t -> succ:'b Sette.t -> 'h -> 'h) -> 'h -> 'h
val fold_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> 'h -> 'h) -> 'h -> 'h
```

Below are the `map` versions of the previous functions.

```
val map :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> 'cc) ->
  ('b -> 'd -> 'dd) -> ('e -> 'ee) -> ('a, 'b, 'cc, 'dd, 'ee) t
```

7.2.7 Copy and Transpose

```
val copy :
  ('a -> 'c -> 'cc) ->
  ('b -> 'd -> 'dd) ->
  ('e -> 'ee) ->
  ('a, 'b, 'c, 'd, 'e) t -> ('a, 'b, 'cc, 'dd, 'ee) t
```

Copy an hypergraph, using the given functions to duplicate the attributes associated to the elements of the graph. The vertex and hedge identifiers are copied using the identity function.

```
val transpose :
  ('a -> 'c -> 'cc) ->
  ('b -> 'd -> 'dd) ->
  ('e -> 'ee) ->
  ('a, 'b, 'c, 'd, 'e) t -> ('a, 'b, 'cc, 'dd, 'ee) t
```

Similar to `copy`, but hyperedges are reversed: successor vertices and predecessor vertices are exchanged.

7.2.8 Algorithms

```
val min : ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t
```

Return the set of vertices without predecessor hyperedges

```
val max : ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t
```

Return the set of vertices without successor hyperedges

Topological sort

```
val topological_sort :
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a list
```

Topological sort of the vertices of the hypergraph starting from a root vertex. The graph supposed to be acyclic. Any hyperedge linking two vertices (which are resp. predecessor and successor) induces a dependency. The result contains only vertices reachable from the given root vertex. If the dependencies are cyclic, the result is meaningless.

```
val topological_sort_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a list
```

Topological sort from a set of root vertices. The two first arguments are supposed to be yet unused vertex and hyperedge identifier.

Reachability and coreachability

The variants of the basic functions are similar to the variants described above.

```
val reachable :
  ?filter:('b -> bool) ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a Sette.t * 'b Sette.t
```

Returns the set of vertices and hyperedges that are *NOT* reachable from the given root vertex. Any dependency in the sense described above is taken into account to define the reachability relation. For instance, if one of the predecessor vertex of an hyperedge is reachable, the hyperedge is considered as reachable.

```
val reachable_multi :
  'a ->
  'b ->
  ?filter:('b -> bool) ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a Sette.t * 'b Sette.t
```

Strongly Connected Components and SubComponents

```
val cfc :
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a list list
```

Decomposition of the graph into Strongly Connected Components,

`cfc graph vertex` returns a decomposition of the graph. The exploration is done from the initial vertex `vertex`, and only reachable vertices are included in the result. The result has the structure `[comp1 comp2 comp3 ...]` where each component is defined by a list of vertices. The ordering of component correspond to a linearization of the partial order between the components.

```
val cfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a list list
```

idem, but from several initial vertices.

`cfc dummy_vertex dummy_hedge graph setvertices` returns a decomposition of the graph, explored from the set of initial vertices `setvertices`. `dummy_vertex` and `dummy_hedge` are resp. unused vertex and hyperedge identifiers.

```
val scfc :
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a -> (unit, 'a) Ilist.t
```

Decomposition of the graph into Strongly Connected Sub-Components,

`scfc graph vertex` returns a decomposition of the graph. The exploration is done from the initial vertex `vertex`, and only reachable vertices are included in the result. The result has the structure `[comp1 comp2 comp3 ...]` where each component is in turn decomposed into components. The third parameter can be used to assign a value to each component (sub-list).

```
val scfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> (unit, 'a) Ilist.t
```

idem, but from several initial vertices.

7.2.9 Printing

```
val print :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  (Format.formatter -> 'd -> unit) ->
  (Format.formatter -> 'e -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
```

Print a graph in textual format on the given formatter, using the given functions to resp. print: vertices (`'a`), hedges (`'b`), vertex attributes (`'c`), hedge attributes (`'d`), and the user information (`'e`).

```
val print_dot :
  ?style:string ->
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle:string ->
  ?fvertexstyle:('a -> string) ->
  ?fhedgestyle:('b -> string) ->
  ?title:string ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'a -> 'c -> unit) ->
  (Format.formatter -> 'b -> 'd -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
```

Output the graph in DOT format on the given formatter, using the given functions to resp print:

- vertex identifiers (in the DOT file)
- hedge identifiers (in the DOT file).

  BE CAUTIOUS. as the DOT files vertices and hedges are actually nodes, the user should take care to avoid name conflicts between vertex and hedge names.
- vertex attributes.

  BE CAUTIOUS: the output of the function will be enclosed bewteen quotes. If ever the output contains line break, or other special characters, it should be escaped. A possible scheme to do this is to first output to `Format.str_formatter` with a standard printing function, then to escape the resulting string and to output the result. This gives something like:

  `print_attrvertex Format.str_formatter vertex attr;`

  `Format.pp_print_string fmt (String.escaped (Format.flush_str_formatter ()));`.
  Concerning the escape function, you may use `String.escaped`, which will produce center justified line breaks, or `Print.escaped` which allows also to choose between center, left and right justified lines.
- hedge atributes (same comment as for vertex attributes).

The optional arguments allows to customize the style. The default setting corresponds to:

```
print_dot ~style="ranksep=0.1; size=\"7,10\";"
~titlestyle="shape=ellipse,style=bold,style=filled,fontsize=20"
~vertexstyle="shape=box,fontsize=12" ~hedgestyle="shape=ellipse,fontsize=12"
~title="" ....
```

7.3 Parameter module for the functor version

```
module type T =
  sig
```

```
type vertex
```

Type of vertex identifiers

```
type hedge
```

Type of hyperedge identifiers

```
val vertex_dummy : vertex
```

A dummy (never used) value for vertex identifiers (used for the functions `XXX_multi`)

```
val hedge_dummy : hedge
```

A dummy (never used) value for hyperedge identifiers (used for the functions `XXX_multi`)

```
module SetV :
Sette.S  with type elt=vertex
```

Set module for vertices

```
module SetH :
Sette.S  with type elt=hedge
```

Set module for hyperedges

```
module HashV :
Hashhe.S  with type key=vertex
```

Hash module with vertices as keys

```
module HashH :
Hashhe.S  with type key=hedge
```

Hash module with hyperedges as keys

```
end
```

7.4 Signature of the functor version

All functions have the same signature as the polymorphic version, except the functions `XXX_multi` which does not need any more a dummy value of type `vertex` (resp. `hedge`).

```
module type S =
  sig

    type vertex
    type hedge
    val vertex_dummy : vertex
    val hedge_dummy : hedge
    module SetV :
    Sette.S  with type elt=vertex
    module SetH :
    Sette.S  with type elt=hedge
    module HashV :
    Hashhe.S  with type key=vertex
    module HashH :
    Hashhe.S  with type key=hedge
    val stdcompare : (vertex, hedge) SHGraph.compare
    type ('a, 'b, 'c) t
```

Type of hypergraphs, where

- 'a : information associated to vertices
- 'b : information associated to hedges
- 'c : user-information associated to an hypergraph

```
    val create : int -> 'c -> ('a, 'b, 'c) t
    val clear : ('a, 'b, 'c) t -> unit
    val is_empty : ('a, 'b, 'c) t -> bool
```

7.4.1 Statistics

```
    val size_vertex : ('a, 'b, 'c) t -> int
    val size_hedge : ('a, 'b, 'c) t -> int
    val size_edgevh : ('a, 'b, 'c) t -> int
    val size_edgehv : ('a, 'b, 'c) t -> int
    val size : ('a, 'b, 'c) t -> int * int * int * int
```

7.4.2 Information associated to vertives and edges

```
val attrvertex : ('a, 'b, 'c) t -> vertex -> 'a
val attrhedge : ('a, 'b, 'c) t -> hedge -> 'b
val info : ('a, 'b, 'c) t -> 'c
```

7.4.3 Membership tests

```
val is_vertex : ('a, 'b, 'c) t -> vertex -> bool
val is_hedge : ('a, 'b, 'c) t -> hedge -> bool
```

7.4.4 Successors and predecessors

```
val succhedge : ('a, 'b, 'c) t -> vertex -> SetH.t
val predhedge : ('a, 'b, 'c) t -> vertex -> SetH.t
val succvertex : ('a, 'b, 'c) t -> hedge -> vertex array
val predvertex : ('a, 'b, 'c) t -> hedge -> vertex array
val succ_vertex : ('a, 'b, 'c) t -> vertex -> SetV.t
val pred_vertex : ('a, 'b, 'c) t -> vertex -> SetV.t
```

7.4.5 Adding and removing elements

```
val add_vertex : ('a, 'b, 'c) t -> vertex -> 'a -> unit
val add_hedge :
  ('a, 'b, 'c) t ->
  hedge ->
  'b -> pred:vertex array -> succ:vertex array -> unit
val replace_attrvertex : ('a, 'b, 'c) t -> vertex -> 'a -> unit
val replace_attrhedge : ('a, 'b, 'c) t -> hedge -> 'b -> unit
val remove_vertex : ('a, 'b, 'c) t -> vertex -> unit
val remove_hedge : ('a, 'b, 'c) t -> hedge -> unit
```

7.4.6 Iterators

```
val iter_vertex :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetH.t -> succ:SetH.t -> unit) ->
  unit
val iter_hedge :
  ('a, 'b, 'c) t ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> unit) ->
  unit
val fold_vertex :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetH.t -> succ:SetH.t -> 'g -> 'g) ->
  'g -> 'g
val fold_hedge :
```

```
  ('a, 'b, 'c) t ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> 'g -> 'g) ->
  'g -> 'g
```

7.4.7 Copy and Transpose

```
val map :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> 'aa) ->
  (hedge -> 'b -> 'bb) -> ('c -> 'cc) -> ('aa, 'bb, 'cc) t
val copy :
  (vertex -> 'a -> 'aa) ->
  (hedge -> 'b -> 'bb) ->
  ('c -> 'cc) -> ('a, 'b, 'c) t -> ('aa, 'bb, 'cc) t
val transpose :
  (vertex -> 'a -> 'aa) ->
  (hedge -> 'b -> 'bb) ->
  ('c -> 'cc) -> ('a, 'b, 'c) t -> ('aa, 'bb, 'cc) t
```

7.4.8 Algorithms

```
val min : ('a, 'b, 'c) t -> SetV.t
val max : ('a, 'b, 'c) t -> SetV.t
val topological_sort :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> vertex -> vertex list
val topological_sort_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> SetV.t -> vertex list
val reachable :
  ?filter:(hedge -> bool) ->
  ('a, 'b, 'c) t ->
  vertex -> SetV.t * SetH.t
val reachable_multi :
  ?filter:(hedge -> bool) ->
  ('a, 'b, 'c) t ->
  SetV.t -> SetV.t * SetH.t
val cfc :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> vertex -> vertex list list
val cfc_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> SetV.t -> vertex list list
val scfc :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t ->
  vertex -> (unit, vertex) Ilist.t
val scfc_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t ->
  SetV.t -> (unit, vertex) Ilist.t
```

7.4.9 Printing

```
val print :
  (Format.formatter -> vertex -> unit) ->
  (Format.formatter -> hedge -> unit) ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  Format.formatter -> ('a, 'b, 'c) t -> unit
```

```
val print_dot :
  ?style:string ->
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle:string ->
  ?fvertexstyle:(vertex -> string) ->
  ?fhedgestyle:(hedge -> string) ->
  ?title:string ->
  (Format.formatter -> vertex -> unit) ->
  (Format.formatter -> hedge -> unit) ->
  (Format.formatter -> vertex -> 'a -> unit) ->
  (Format.formatter -> hedge -> 'b -> unit) ->
  Format.formatter -> ('a, 'b, 'c) t -> unit
```

```
end
```

7.5 Functor

```
module Make :
functor (T : T) -> S  with type vertex=T.vertex and type hedge=T.hedge and module SetV=T.SetV
and module SetH=T.SetH and module HashV=T.HashV and module HashH=T.HashH
```

7.6 Compare interface

```
module Compare :
  sig
```

```
val attrvertex :
  ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd
```

```
val attrhedge :
  ('a, 'b) SHGraph.compare -> ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'e
```

```
val is_vertex :
  ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> bool
```

```
val is_hedge :
  ('a, 'b) SHGraph.compare -> ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> bool
```

```
val succhedge :
  ('a, 'b) SHGraph.compare ->
  ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'c Sette.t
```

```
val predhedge :
  ('a, 'b) SHGraph.compare ->
  ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'c Sette.t
```

```
val succvertex :
  ('a, 'b) SHGraph.compare ->
```

```
    ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'c array
  val predvertex :
    ('a, 'b) SHGraph.compare ->
    ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'c array
  val succ_vertex :
    ('a, 'b) SHGraph.compare ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a Sette.t
  val pred_vertex :
    ('a, 'b) SHGraph.compare ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a Sette.t
  val add_vertex :
    ('a, 'b) SHGraph.compare ->
    ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd -> unit
  val add_hedge :
    ('a, 'b) SHGraph.compare ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph ->
    'b -> 'd -> pred:'a array -> succ:'a array -> unit
  val replace_attrvertex :
    ('a, 'b) SHGraph.compare ->
    ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd -> unit
  val replace_attrhedge :
    ('a, 'b) SHGraph.compare ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'b -> 'd -> unit
  val remove_hedge :
    ('a, 'b) SHGraph.compare -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'b -> unit
  val remove_vertex :
    ('a, 'b) SHGraph.compare -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> unit
  val topological_sort :
    ('a, 'b) SHGraph.compare ->
    ?priority:'b SHGraph.priority ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a -> 'a list
  val topological_sort_multi :
    ('a, 'b) SHGraph.compare ->
    'a ->
    'b ->
    ?priority:'b SHGraph.priority ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a Sette.t -> 'a list
  val reachable :
    ('a, 'b) SHGraph.compare ->
    ?filter:('b -> bool) ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a -> 'a Sette.t * 'b Sette.t
  val reachable_multi :
    ('a, 'b) SHGraph.compare ->
    'a ->
    'b ->
    ?filter:('b -> bool) ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a Sette.t -> 'a Sette.t * 'b Sette.t
  val cfc :
    ('a, 'b) SHGraph.compare ->
    ?priority:'b SHGraph.priority ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a list list
  val cfc_multi :
```

```
    ('a, 'b) SHGraph.compare ->
    ?priority:'b SHGraph.priority ->
    'a -> 'b -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a Sette.t -> 'a list list
  val scfc :
    ('a, 'b) SHGraph.compare ->
    ?priority:'b SHGraph.priority ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> (unit, 'a) Ilist.t
  val scfc_multi :
    ('a, 'b) SHGraph.compare ->
    'a ->
    'b ->
    ?priority:'b SHGraph.priority ->
    ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a Sette.t -> (unit, 'a) Ilist.t
  val print :
    ('a, 'b) SHGraph.compare ->
    (Format.formatter -> 'a -> unit) ->
    (Format.formatter -> 'b -> unit) ->
    (Format.formatter -> 'c -> unit) ->
    (Format.formatter -> 'd -> unit) ->
    (Format.formatter -> 'e -> unit) ->
    Format.formatter -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> unit
  val min :
    ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a Sette.t
  val max :
    ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a Sette.t
end
```

Chapter 8

Module `Ilist` : Imbricated lists

The operations of this module have a functional semantics.

```
type ('a, 'b) el =
  | Atome of 'b
```

Terminal case

```
  | List of ('a, 'b) t
```

The element is recursively a list, with an attribute of type `'a`.

Type of list elements

```
type ('a, 'b) t = 'a * ('a, 'b) el list
```

Type of imbricated lists. `'a` is the type of attributes associated to lists, and `'b` the type of elements.

```
val cons : ('a, 'b) el -> ('a, 'b) t -> ('a, 'b) t
```

Adding a new list element at the begining of the list

```
val atome : 'b -> ('a, 'b) el
```

Create a list element from a single element.

```
val list : 'a -> ('a, 'b) el list -> ('a, 'b) el
```

Create a list element from a list.

```
val of_list : 'a -> 'b list -> ('a, 'b) t
```

Create a recursive list from a regular list

```
val to_list : ('a, 'b) t -> 'b list
```

Create a regular list from a recursive list. Order is preserved but imbrication is lost (as in `Ilist.flatten`[8]).

- `to_list [[a;b];c;[d;e]] = [a;b;c;d;e]`

```
val hd : ('a, 'b) t -> ('a, 'b) el
```

Return the head of the list.

```
val tl : ('a, 'b) t -> ('a, 'b) t
```

Return the tail of the list.

```
val length : ('a, 'b) t -> int
```

Return the ength of the list.

```
val depth : ('a, 'b) t -> int
```

Return the (maximal) depth of the list.

- `depth [] = 0`
- `depth [a;b;c] = 1`
- `depth [[a];b] = 2`

```
val append :
  combine:('a -> 'a -> 'a) ->
  ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

Append two lists

```
val flatten : ?depth:int -> ('a, 'b) t -> ('a, 'b) t
```

Flatten the recursive list, only starting from the given

- `flatten [] = []`
- `flatten [a;[b;[c];d];e;[f]] = [a;b;c;d;e;f]`
- `flatten ~depth:2 [a;[b;[c];d];e;[f]] = [a;[b;c;d];e;[f]]`
- `flatten ~depth:3 [a;[b;[c];d];e;[f]] = [a;[b;[c];d];e;[f]]`

```
val rev : ('a, 'b) t -> ('a, 'b) t
```

Recursively reverse the recursive list

- `rev [a;[b;[c];d];e;[f]] = [[f];e;[d;[c];b];a]`

```
val mem : 'b -> ('a, 'b) t -> bool
```

Membership test.

```
val exists : ('a -> 'b -> bool) -> ('a, 'b) t -> bool
```

Existence test

```
val map :
  ('a -> 'c) ->
  (bool -> 'a -> 'b -> 'd) -> ('a, 'b) t -> ('c, 'd) t
```

Ordinary map function

```
val iter : (bool -> 'a -> 'b -> unit) -> ('a, 'b) t -> unit
```

Ordinary iteration function for atoms

```
val fold_left : ('c -> bool -> 'a -> 'b -> 'c) -> 'c -> ('a, 'b) t -> 'c
```

Ordinary fold function for atoms, from left to right.

```
val fold_right : (bool -> 'a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

Ordinary fold function for atoms, from right to left.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstexp:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastexp:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Printing function.

Chapter 9

Module Sette : Sets over ordered types (extension of standard library module and polymorphic variant)

Modified by B. Jeannet to get a generic type and a few additions (like conversions form and to maps and pretty-printing).

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

Modified by B. Jeannet to get a generic type and a few additions (like conversions form and to maps and pretty-printing).

```
type 'a set =
  | Empty
  | Node of 'a set * 'a * 'a set * int
```

Meant to be internal, but exporting needed for Mappe.maptoset.

```
type 'a t = 'a set
```

The type of sets over elements of type 'a.

```
val empty : 'a t
```

The empty set.

```
val is_empty : 'a t -> bool
```

Test whether a set is empty or not.

```
val mem : 'a -> 'a t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val add : 'a -> 'a t -> 'a t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val singleton : 'a -> 'a t
```

`singleton x` returns the one-element set containing only `x`.

```
val remove : 'a -> 'a t -> 'a t
```

remove x s returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged.

`val union : 'a t -> 'a t -> 'a t`

`val inter : 'a t -> 'a t -> 'a t`

`val diff : 'a t -> 'a t -> 'a t`

Union, intersection and set difference.

`val compare : 'a t -> 'a t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : 'a t -> 'a t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : 'a t -> 'a t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : ('a -> unit) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

`val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

Raises `Not_found` if no fount

Returns the computed accumulator

`val for_all : ('a -> bool) -> 'a t -> bool`

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

`val exists : ('a -> bool) -> 'a t -> bool`

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

`val filter : ('a -> bool) -> 'a t -> 'a t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

`val partition : ('a -> bool) -> 'a t -> 'a t * 'a t`

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

`val cardinal : 'a t -> int`

Return the number of elements of a set.

`val elements : 'a t -> 'a list`

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

`val min_elt : 'a t -> 'a`

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : 'a t -> 'a`

Same as `min_elt`, but returns the largest element of the given set.

```
val choose : 'a t -> 'a
```

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
```

```
module type S =
  sig
```

```
type elt
```

The type of the set elements.

```
type t
```

The type of sets.

```
val repr : t -> elt Sette.set
val obj : elt Sette.set -> t
module Ord :
Set.OrderedType  with type t=elt
```

The ordering module used for this set module.

```
val empty : t
```

The empty set.

```
val is_empty : t -> bool
```

Test whether a set is empty or not.

```
val mem : elt -> t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val add : elt -> t -> t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val singleton : elt -> t
```

`singleton x` returns the one-element set containing only `x`.

```
val remove : elt -> t -> t
```

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

```
val union : t -> t -> t
```

Set union.

`val inter : t -> t -> t`

Set intersection.

`val diff : t -> t -> t`

Set difference.

`val compare : t -> t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : t -> t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : t -> t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : (elt -> unit) -> t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

`val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

`val for_all : (elt -> bool) -> t -> bool`

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

`val exists : (elt -> bool) -> t -> bool`

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

`val filter : (elt -> bool) -> t -> t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

`val partition : (elt -> bool) -> t -> t * t`

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

`val cardinal : t -> int`

Return the number of elements of a set.

`val elements : t -> elt list`

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Ord.compare`, where `Ord` is the argument given to `Sette.Make`[9].

`val min_elt : t -> elt`

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : t -> elt`

Same as `Sette.S.min_elt`[9], but returns the largest element of the given set.

```
    val choose : t -> elt
```

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> elt -> unit) ->
      Format.formatter -> t -> unit
  end
```

Output signature of the functor `Sette.Make`[9].

```
module Make :
functor (Ord :  Set.OrderedType) -> S  with type elt = Ord.t and module Ord=Ord
```

Functor building an implementation of the set structure given a totally ordered type.

```
module Compare :
  sig

    val split :
      ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t * bool * 'a Sette.t
```

Meant to be internal, but exporting needed for Mappe.maptoset.

```
    val add : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t
    val mem : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> bool
    val remove : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t
    val union : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val inter : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val diff : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val equal : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> bool
    val compare : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> int
    val subset : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> bool
    val filter : ('a -> 'a -> int) -> ('a -> bool) -> 'a Sette.t -> 'a Sette.t
    val partition :
      ('a -> 'a -> int) -> ('a -> bool) -> 'a Sette.t -> 'a Sette.t * 'a Sette.t
  end
```

Chapter 10

Module `Hashhe` : Hash tables and hash functions (extension of standard library module)

Hash tables are hashed association tables, with in-place modification.

Modified by B. Jeannet: functions `map`, `copy` and `print`.

```
type ('a, 'b) hashtbl
```

```
type 'a compare = {
  hash : 'a -> int ;
  equal : 'a -> 'a -> bool ;
}
```

Generic interface

```
type ('a, 'b) t = ('a, 'b) hashtbl
```

The type of hash tables from type `'a` to type `'b`.

```
val create : int -> ('a, 'b) t
```

`create n` creates a new, empty hash table, with initial size `n`. For best results, `n` should be on the order of the expected number of elements that will be in the table. The table grows as needed, so `n` is just an initial guess.

```
val clear : ('a, 'b) t -> unit
```

Empty a hash table.

```
val add : ('a, 'b) t -> 'a -> 'b -> unit
```

`add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashhe.remove`[10] `tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists.)

```
val copy : ('a, 'b) t -> ('a, 'b) t
```

Return a copy of the given hashtable.

```
val find : ('a, 'b) t -> 'a -> 'b
```

`find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

```
val find_all : ('a, 'b) t -> 'a -> 'b list
```

`find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

```
val mem : ('a, 'b) t -> 'a -> bool
```

`mem tbl x` checks if `x` is bound in `tbl`.

```
val remove : ('a, 'b) t -> 'a -> unit
```

`remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

```
val replace : ('a, 'b) t -> 'a -> 'b -> unit
```

`replace tbl x y` replaces the current binding of `x` in `tbl` by a binding of `x` to `y`. If `x` is unbound in `tbl`, a binding of `x` to `y` is added to `tbl`. This is functionally equivalent to `Hashhe.remove`[10] `tbl x` followed by `Hashhe.add`[10] `tbl x y`.

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

`iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

`fold f tbl init` computes `(f kN dN ... (f k1 d1 init)...)`, where `k1 ... kN` are the keys of all bindings in `tbl`, and `d1 ... dN` are the associated values. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val map : ('a -> 'b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
```

`map f tbl` applies `f` to all bindings in table `tbl` and creates a new hashtable associating the results of `f` to the same key type. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val length : ('a, 'b) t -> int
```

`length tbl` returns the number of bindings in `tbl`. Multiple bindings are counted multiply, so `length` gives the number of times `iter` calls it first argument.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Functorial interface

```
module type HashedType =
  sig
```

```
type t
```

The type of the hashtable keys.

```
val equal : t -> t -> bool
```

The equality predicate used to compare keys.

```
val hash : t -> int
```

A hashing function on keys. It must be such that if two keys are equal according to `equal`, then they have identical hash values as computed by `hash`. Examples: suitable (`equal`, `hash`) pairs for arbitrary key types include (`(=)`, `Hashhe.HashedType.hash`[10]) for comparing objects by structure, (`(fun x y -> compare x y = 0)`, `Hashhe.HashedType.hash`[10]) for comparing objects by structure and handling `Pervasives.nan` correctly, and (`(==)`, `Hashhe.HashedType.hash`[10]) for comparing objects by addresses (e.g. for or cyclic keys).

```
end
```

The input signature of the functor `Hashhe.Make`[10].

```
module type S =
  sig
    type key
    type 'a t = (key, 'a) Hashhe.hashtbl
    module Hash :
    Hashhe.HashedType  with type t=key
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val map : (key -> 'a -> 'b) -> 'a t -> 'b t
    val length : 'a t -> int
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> key -> unit) ->
      (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
  end
```

The output signature of the functor `Hashhe.Make`[10].

```
module Make :
functor (H : HashedType) -> S  with type key = H.t
```

Functor building an implementation of the hashtable structure. The functor `Make` returns a structure containing a type `key` of keys and a type `'a t` of hash tables associating data of type `'a` to keys of type `key`. The operations perform similarly to those of the generic interface, but use the hashing and equality functions specified in the functor argument `H` instead of generic equality and hashing.

The polymorphic hash primitive

```
val hash : 'a -> int
```

`hash x` associates a positive integer to any value of any type. It is guaranteed that if `x = y` or `Pervasives.compare x y = 0`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

```
val hash_param : int -> int -> 'a -> int
```

`hash_param n m x` computes a hash value for `x`, with the same properties as for `hash`. The two extra parameters `n` and `m` give more precise control over hashing. Hashing performs a depth-first, right-to-left traversal of the structure `x`, stopping after `n` meaningful nodes were encountered, or `m` nodes, meaningful or not, were encountered. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of `m` and `n` means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters `m` and `n` govern the tradeoff between accuracy and speed.

```
val stdcompare : 'a compare
module Compare :
  sig

    val resize : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> unit
    val add : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b -> unit
    val remove : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> unit
    val find : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b
    val find_all : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b list
    val replace :
      'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b -> unit
    val mem : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> bool
  end
```

Chapter 11

Module Print : Printing functions using module Format

11.1 Printing functions for standard datatypes (lists,arrays,...)

In the following functions, optional arguments `?first`, `?sep`, `?last` denotes the formatting instructions (under the form of a `format` string) issued at the beginning, between two elements, and at the end.

The functional argument(s) indicate(s) how to print elements.

```
val list :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
```

Print a list

```
val array :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a array -> unit
```

Print an array

```
val pair :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) -> Format.formatter -> 'a * 'b -> unit
```

Print a pair

```
val option :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a option -> unit
```

Print an optional element

```
val hash :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
```

```
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) Hashtbl.t -> unit
```

Print an hashtable

```
val weak :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a Weak.t -> unit
```

Print a weak pointer array

11.2 Useful functions

```
val string_of_print : (Format.formatter -> 'a -> unit) -> 'a -> string
```

Transforms a printing function into a conversion-to-string function.

```
val print_of_string : ('a -> string) -> Format.formatter -> 'a -> unit
```

Transforms a conversion-to-string function to a printing function.

```
val sprintf :
  ?margin:int -> ('a, Format.formatter, unit, string) Pervasives.format4 -> 'a
```

Better `sprintf` function than `Format.sprintf`, as it takes the same kind of formatters as other `Format.Xprintf` functions.

```
val escaped : ?linebreak:char -> string -> string
```

Escape a string, replacing line breaks by `linebreak` (default `'\n'`). When used for DOT output, `'\l'` and `'\r'` produces respectively left or right justified lines, instead of center justified lines.

Chapter 12

Module `Time` : Small module to compute the duration of computations

```
val wrap_duration : float Pervasives.ref -> (unit -> 'a) -> 'a
```

`wrap_duration duration f` executes the function `f` and stores into `!duration` the time spent in `f`, in seconds. If `f` raises an exception, the exception is transmitted and the computed duration is still valid.

```
val wrap_duration_add : float Pervasives.ref -> (unit -> 'a) -> 'a
```

Similar to `wrap_duration`, but here the time spent in `f` is added to the value `!duration`.

Index

accumulate_vertex, 24
add, 46, 48, 50, 51, 53, 54
add_active_hedges, 26
add_hedge, 32, 38, 41
add_vertex, 32, 38, 41
analysis, 25–27
analysis_dyn, 15
analysis_guided, 14
analysis_std, 14
append, 44
arc, 21
array, 55
atome, 43
attr, 21
attrhedge, 32, 38, 40
attrvertex, 32, 38, 40

cardinal, 47, 49
cfc, 34, 39, 41
cfc_multi, 35, 39, 42
choose, 48, 50
clear, 31, 37, 51, 53
Compare, 40, 50, 54
compare, 30, 47, 49–51
cons, 43
copy, 33, 39, 51, 53
create, 31, 37, 51, 53

depth, 44
descend, 24
descend_strategy, 24
diff, 47, 49, 50
dot_graph, 22

el, 43
elements, 47, 49
elt, 48
empty, 46, 48
equal, 47, 49, 50, 53
equation, 12, 19
equation_of_graph, 14, 27
escaped, 56
exists, 44, 47, 49

filter, 47, 49, 50
find, 51, 53, 54
find_all, 51, 53, 54
Fixpoint, 10
fixpoint, 27
FixpointDyn, 27
FixpointGuided, 26
FixpointStd, 23
FixpointType, 17
flatten, 44
fold, 47, 49, 52, 53
fold_hedge, 33, 39
fold_left, 44
fold_right, 44
fold_vertex, 33, 38
for_all, 47, 49

graph, 22, 31
graph_of_equation, 15, 28

Hash, 53
hash, 53, 54, 56
hash_param, 54
HashedType, 52
HashH, 37
Hashhe, 51
hashtbl, 51
HashV, 36, 37
hd, 43
hedge, 36, 37
hedge_dummy, 36, 37
hedge_n, 31

Ilist, 43
ilist_map_condense, 21
info, 22, 32, 38
infodyn, 21
init, 23, 27
inter, 47, 49, 50
is_empty, 31, 37, 46, 48
is_hedge, 32, 38, 40
is_tvertex, 23
is_vertex, 32, 38, 40
iter, 44, 47, 49, 52, 53
iter_hedge, 33, 38
iter_vertex, 33, 38

key, 53

length, 44, 52, 53
list, 43, 55

Make, 40, 50, 54
make_strategy_default, 13, 20
make_strategy_iteration, 20
manager, 12, 18
map, 33, 39, 44, 52, 53
max, 34, 39, 42
max_elt, 47, 49
mem, 44, 46, 48, 50, 52–54
min, 34, 39, 42
min_elt, 47, 49

obj, 48
of_list, 43
option, 55
Ord, 48
output, 13, 21
output_of_graph, 25

pair, 55
partition, 47, 49, 50
pred_vertex, 32, 38, 41
predhedge, 32, 38, 40
predvertex, 32, 38, 41
Print, 55
print, 35, 40, 42, 45, 48, 50, 52, 53
print_arc, 22
print_attr, 22
print_dot, 35, 40
print_graph, 22
print_info, 22
print_of_string, 56
print_output, 15, 21
print_stat, 15, 21
print_stat_iteration, 21
print_stat_iteration_ilist, 21
print_strategy, 15, 20
print_strategy_iteration, 20
print_strategy_vertex, 15, 20
print_workingsets, 22
priority, 30
process_strategy, 24
process_toplevel_strategy, 24
process_vertex, 24
propagate, 27
propagate_vertex, 24

reachable, 34, 39, 41
reachable_multi, 34, 39, 41
remove, 46, 48, 50, 52–54
remove_hedge, 33, 38, 41
remove_vertex, 33, 38, 41
replace, 52–54
replace_attrhedge, 32, 38, 41
replace_attrvertex, 32, 38, 41
repr, 48
resize, 54
rev, 44

S, 37, 48, 53
scfc, 35, 39, 42
scfc_multi, 35, 39, 42
set, 46
SetH, 36, 37
Sette, 46
SetV, 36, 37
SHGraph, 30
singleton, 46, 48
size, 31, 37
size_edgehv, 31, 37
size_edgevh, 31, 37
size_hedge, 31, 37
size_vertex, 31, 37
split, 50
sprintf, 56
stat, 13, 21
stat_iteration, 13, 21
stat_iteration_merge, 21
stdcompare, 31, 37, 54
strategy, 12, 19
strategy_iteration, 12, 19
strategy_vertex, 12, 19
string_of_print, 56
subset, 47, 49, 50
succ_vertex, 32, 38, 41
succhedge, 32, 38, 40
succvertex, 32, 38, 41

T, 36
t, 31, 37, 43, 46, 48, 51, 53
Time, 57
tl, 43
to_list, 43
topological_sort, 34, 39, 41
topological_sort_multi, 34, 39, 41
transpose, 33, 39
treach_of_tvertex, 23

union, 47, 48, 50
update_workingsets, 23

vertex, 36, 37
vertex_dummy, 36, 37
vertex_n, 31

weak, 56
wrap_duration, 57
wrap_duration_add, 57