Analyzer 2.0

October 3, 2007

Contents

Chapter 1

Module `Print` : Printing functions using module `Format`

1.1 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 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

1.2 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 : ('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.

Chapter 2

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`.

Chapter 3

Module `Sette` : Sets over ordered types.

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 t
```

The type of sets over elements of type 'a.

```
type 'a tzz =
  | Emptyzz
  | Nodezz of 'a tzz * 'a * 'a tzz * int
```

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

```
val repr : 'a t -> 'a tzz
val obj : 'a tzz -> 'a t
val splitzz : 'a -> 'a tzz -> 'a tzz * bool * 'a tzz
```

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

```
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.

```
module Ord :
Set.OrderedType  with type t=elt
```

The ordering module used for this set module.

```
val repr : t -> elt Sette.tzz
val obj : elt Sette.tzz -> t
val splitzz : elt ->
  elt Sette.tzz -> elt Sette.tzz * bool * elt Sette.tzz
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`[3].

`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`[3], 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`[3].

```
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.

Chapter 4

Module Hashhe : Hash tables and hash functions.

Hash tables are hashed association tables, with in-place modification.
Modified by B. Jeannet: functions `map` and `print`.

Generic interface

`type ('a, 'b) t`

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

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

`Hashtbl.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`

`Hashtbl.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 `Hashtbl.remove 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`

`Hashtbl.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`

`Hashtbl.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`

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

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

Hashtbl.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
```

Hashtbl.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 Hashtbl.remove tbl x followed by Hashtbl.add tbl x y.

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

Hashtbl.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
```

Hashtbl.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
```

Hashtbl.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
```

Hashtbl.length tbl returns the number of bindings in tbl. Multiple bindings are counted multiply, so Hashtbl.length gives the number of times Hashtbl.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 (`(=)`, `Hashtbl.hash`) for comparing objects by structure, (`(fun x y -> compare x y = 0)`, `Hashtbl.hash`) for comparing objects by structure and handling `Pervasives.nan` correctly, and (`(==)`, `Hashtbl.hash`) for comparing objects by addresses (e.g. for or cyclic keys).

```
  end
```

The input signature of the functor `Hashtbl.Make`.

```
module type S =
  sig
    type key
    type 'a t
    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 `Hashtbl.Make`.

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

Functor building an implementation of the hashtable structure. The functor `Hashtbl.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
```

Hashtbl.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
```

Hashtbl.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.

Chapter 5

Module `Ilist` : Imbricated lists

The operations of this module have a functional semantics.

```
type 'a el =
  | Atome of 'a
```

Terminal case

```
  | List of 'a t
```

The element is recursively a list.

Type of list elements.

```
type 'a t =
  | Nil
```

Empty list

```
  | Cons of 'a el * 'a t
```

Non-empty list

Type of imbricated lists.

```
val cons : 'a el -> 'a t -> 'a t
```

Adding a new list element at the begining of the list

```
val atome : 'a -> 'a el
```

Create a list element from a single element.

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

Create a list element from a list.

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

Create a recursive list from a list

```
val hd : 'a t -> 'a el
```

Return the head of the list.

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

Return the tail of the list.

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

Return the ength of the list.

`val depth : 'a t -> int`

Return the (maximal) depth of the list.

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

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

Append two lists

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

Flatten the recursive list and converts it to a list

`val flatten : ?depth:int -> 'a t -> 'a t`

Flatten the recursive list, but only starting from the given depth. Defaut depth is 1.

- `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 t -> 'a t`

Recursively reverse the recursive list

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

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

Membership test.

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

Ordinary map function. The boolean value indicates whether the element is beginning a recursive list.

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

Ordinary iteration function. The boolean value indicates whether the element is beginning a recursive list.

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

Ordinary fold function, from left to right.

`val iter_rec : ('a -> 'a Sette.t array -> unit) -> 'a t -> unit`

Recursive iteration function, rather complex. `iter_rec f ilist` applies `f` to each atom of the recursive list, together with the array of enclosing components. When `f` is called with `f obj array`, `array` is the array of enclosing components to which `obj` belongs, from the deepest to the toplevel. Each component has been removed from the elements of the components of the level below.

Example: `iter_rec f [a;[b;c;d];e;[f]]` is equivalent to `f a [|{b,c,d,e,f}|]; f b [|{c,d};{a,e,f}|]; f c [|{b,d};{a,e,f}|]; f d [|{b,c};{a,e,f}|]; f e [|{a,b,c,d,f}|]; f f [|{};{a,b,c,d,e}|]; ()`.

```
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
```

Printing function.

Chapter 6

Module SHGraph : Oriented hypergraphs

6.1 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.

6.2 Generic (polymorphic) interface

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

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 -> 'a -> ('b, 'c, 'd, 'e, 'a) 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 ?

6.2.1 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)`

6.2.2 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`

6.2.3 Membership tests

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

6.2.4 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

6.2.5 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 predecssor and successor vertices should already exist in the graph. Otherwise, a `Failure` exception is raised.

```
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.

6.2.6 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 -> 'f -> 'f) -> 'f -> 'f
val fold_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> 'f -> 'f) -> 'f -> 'f
```

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

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

6.2.7 Copy and Transpose

```
val copy :
  ('a -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) 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 -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) t
```

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

6.2.8 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 : ('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 -> ('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.

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

Variant of the previous function, where the Boolean function `f hedge succ` tells whether the given dependency should be taken into account or not in the sort.

Reachability and coreachability

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

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

Strongly Connected Components and SubComponents

```
val cfc : ('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 -> ('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 cfc_filter_multi :
  'a ->
  'b ->
  ('a, 'b, 'c, 'd, 'e) t -> ('b -> bool) -> 'a Sette.t -> 'a list list
```

idem, but with a filtering of dependencies

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

idem, but with a priority of dependencies.

The priority function `p:'b -> int` filters out hyperedges `b` such that `p b<0`, and explores first hyperedges `b` with the highest priority.

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

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

idem, but from several initial vertices.

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

idem, but with a filtering of dependencies

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

idem, but with a priority of dependencies.

6.2.9 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 :
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle: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 the provided `escaped` (see below) 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 ~titlestyle="shape=ellipse,style=bold,style=filled,fontsize=20"
~vertexstyle="shape=box,fontsize=12" ~hedgestyle="shape=ellipse,fontsize=12"
~title="" ....
```

```
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 righjt justified lines, instead of center justified lines.

6.3 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
```

6.4 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
    module SetV :
    Sette.S  with type elt=vertex
    module SetH :
    Sette.S  with type elt=hedge
    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 -> 'a -> ('b, 'c, 'a) t
    val clear : ('a, 'b, 'c) t -> unit
    val is_empty : ('a, 'b, 'c) t -> bool
```

6.4.1 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
```

6.4.2 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
```

6.4.3 Membership tests

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

6.4.4 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
```

6.4.5 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 remove_vertex : ('a, 'b, 'c) t -> vertex -> unit
val remove_hedge : ('a, 'b, 'c) t -> hedge -> unit
```

6.4.6 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 -> 'd -> 'd) ->
  'd -> 'd
val fold_hedge :
  ('a, 'b, 'c) t ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> 'd -> 'd) ->
  'd -> 'd
val map :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetH.t -> succ:SetH.t -> 'd) ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> 'e) ->
  ('c -> 'f) -> ('d, 'e, 'f) t
```

6.4.7 Copy and Transpose

```
val copy :
  (vertex -> 'a -> 'b) ->
  (hedge -> 'c -> 'd) ->
  ('e -> 'f) -> ('a, 'c, 'e) t -> ('b, 'd, 'f) t
val transpose :
  (vertex -> 'a -> 'b) ->
  (hedge -> 'c -> 'd) ->
  ('e -> 'f) -> ('a, 'c, 'e) t -> ('b, 'd, 'f) t
```

6.4.8 Algorithms

```
val min : ('a, 'b, 'c) t -> SetV.t
```

```
  val max : ('a, 'b, 'c) t -> SetV.t
  val topological_sort : ('a, 'b, 'c) t -> vertex -> vertex list
  val topological_sort_multi : ('a, 'b, 'c) t -> SetV.t -> vertex list
  val topological_sort_filter_multi :
    ('a, 'b, 'c) t ->
    (hedge -> bool) -> SetV.t -> vertex list
  val reachable : ('a, 'b, 'c) t ->
    vertex -> SetV.t * SetH.t
  val reachable_multi : ('a, 'b, 'c) t ->
    SetV.t -> SetV.t * SetH.t
  val reachable_filter_multi :
    ('a, 'b, 'c) t ->
    (hedge -> bool) ->
    SetV.t -> SetV.t * SetH.t
  val cfc : ('a, 'b, 'c) t -> vertex -> vertex list list
  val cfc_multi : ('a, 'b, 'c) t -> SetV.t -> vertex list list
  val cfc_filter_multi :
    ('a, 'b, 'c) t ->
    (hedge -> bool) -> SetV.t -> vertex list list
  val cfc_priority_multi :
    ('a, 'b, 'c) t ->
    (hedge -> int) -> SetV.t -> vertex list list
  val scfc : ('a, 'b, 'c) t -> vertex -> vertex Ilist.t
  val scfc_multi : ('a, 'b, 'c) t -> SetV.t -> vertex Ilist.t
  val scfc_filter_multi :
    ('a, 'b, 'c) t ->
    (hedge -> bool) -> SetV.t -> vertex Ilist.t
  val scfc_priority_multi :
    ('a, 'b, 'c) t ->
    (hedge -> int) -> SetV.t -> vertex Ilist.t
```

6.4.9 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 :
    ?titlestyle:string ->
    ?vertexstyle:string ->
    ?hedgestyle: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
```

6.5 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
```

Chapter 7

Module `MkFixpoint` : Fixpoint analysis of an equation system

7.1 Introduction

This module provides a generic engine for computing iteratively the solution of a fixpoint equation on a lattice.

The functor takes as argument an hypergraph module representing the system of equation. In this graph, vertices corresponds to unknown and oriented hyperedges to functions. It is assumed that hyperedges have unique destination vertex, and that associated functions are strict in each of their arguments: a bottom value in one of the argument implies that the result is empty.

Most functions of the module delivered by the functor (module type `S`) take as argument an object of type `('abstract,'arc) manager`, where `'abstract` is the type of abstract values and `'arc` the type of information associated to hyperedges and computed by the fixpoint analysis. The manager of type `('abstract,arc) manager` defines operations on abstract values, meaning of hyperedges, printing functions, and various options.

Real functions to be applied in the equation system are indexed by the type `hedge` of the graph. The result of a function application (`manager.apply`) is first a user-defined object of type `'arc` attached to the hyperedge, and an abstract value of type `'abstract` that will be joined with the current reachable value fo the destination vertex.

7.2 Module type for the fixpoint engine

```
module type S =
  sig
```

```
    module Graph :
    SHGraph.S
```

7.2.1 Datatypes

Manager

```
    type ('a, 'b) manager = {
      mutable bottom : Graph.vertex -> 'a ;
```

Create a bottom value

```
      mutable canonical : Graph.vertex -> 'a -> unit ;
```

Make an abstract value canonical

```
  mutable is_bottom : Graph.vertex -> 'a -> bool ;
```

Emptiness test

```
  mutable is_leq : Graph.vertex -> 'a -> 'a -> bool ;
```

Inclusion test

```
  mutable join : Graph.vertex -> 'a -> 'a -> 'a ;
  mutable join_list : Graph.vertex -> 'a list -> 'a ;
```

Binary and n-ary join operation

```
  mutable widening : Graph.vertex -> 'a -> 'a -> 'a ;
```

Apply widening at the given point, with the two arguments

```
  mutable apply : Graph.hedge -> 'a array -> 'b * 'a ;
```

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 arc_init : Graph.hedge -> 'b ;
```

Initial value for arcs

```
  mutable get_init : Graph.vertex -> 'a ;
```

Return the initial value associated to the given vertex

```
  mutable print_abstract : Format.formatter -> 'a -> unit ;
  mutable print_arc : Format.formatter -> 'b -> unit ;
  mutable print_vertex : Format.formatter -> Graph.vertex -> unit ;
  mutable print_hedge : Format.formatter -> Graph.hedge -> unit ;
  mutable widening_first : bool ;
```

When to apply widening from first non bottom value.

```
  mutable widening_start : int ;
```

First widening step

```
  mutable widening_freq : int ;
```

widening every n steps

```
  mutable widening_descend : int ;
```

Maximum nb. of descending steps

```
  mutable print_analysis : bool ;
  mutable print_step : bool ;
  mutable print_state : bool ;
  mutable print_postpre : bool ;
}
```

Iteration strategies

```
type strategy_vertex = {
  mutable vertex : Graph.vertex ;
  mutable hedges : Graph.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 strategy = 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.

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`.

Result

```
type stat = {
  time : float ;
  iterations : int ;
  descendings : int ;
}
```

statistics at the end of the analysis

```
type ('a, 'b) output = ('a, 'b, stat) Graph.t
```

result of the analysis

Internal datatype

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

7.2.2 Printing functions

```
val print_strategy_vertex :
  ('a, 'b) manager ->
  Format.formatter -> 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 `(vertex,[list of hedges],boolean)`.

```
val print_strategy :
  ('a, 'b) manager ->
  Format.formatter -> 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_graph :
  ('a, 'b) manager ->
  Format.formatter -> ('a, 'b) graph -> unit
```

Prints internal graph.

print_graph man print_arc fmt graph prints an object of (abstract) type ('abstract,'arc) graph, using print_arc for printing the information associated to hyperedges.

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

Prints statistics

```
val print_output :
  ('a, 'b) manager ->
  Format.formatter -> ('a, 'b) output -> unit
```

Prints the result of an analysis.

print_graph man print_arc fmt graph prints an object of type 'arc output, using print_arc for printing the information associated to hyperedges.

7.2.3 Main Functions

```
val init :
  ('a, 'b) manager ->
  ('c, 'd, 'e) Graph.t -> Graph.SetV.t -> ('a, 'b) graph
```

init manager inputgraph sinit creates an internal graph and initializes it.

- Only the structure of inputgraph is used.
- sinit is the set of vertices with non-bottom initial values. Initial values are obtained by calling the function manager.get_init on these vertices.

```
val fixpoint : ('a, 'b) manager ->
  ('a, 'b) graph -> strategy -> unit
```

fixpoint manager graph strategy computes a fixpoint (or a postfixpoint in case of widening) of the system of equation represented by graph. The iteration order (and set of widening points) is specified by the argument strategy.

```
val descend : ('a, 'b) manager ->
  ('a, 'b) graph -> strategy -> bool
```

descend manager graph sinit strategy performs several descending steps, depending on the option manager.widening_descend. strategy is used only for the ordering of connex components and the iteration ordering inside them.

```
val analysis :
  ('a, 'b) manager ->
  ('c, 'd, 'e) Graph.t ->
  Graph.SetV.t -> strategy -> ('a, 'b) graph
```

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

```
val analysis_guided :
  ?depth:int ->
  ?priority:(Graph.hedge -> int) ->
  ?modify_strategy:(strategy -> strategy) ->
  ('a, 'b) manager ->
  ('c, 'd, 'e) Graph.t -> Graph.SetV.t -> ('a, 'b) graph
```

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

```
val output_of_graph : ('a, 'b) graph -> ('a, 'b) output
```

Getting the result of the analysis.

7.2.4 Utility functions

```
val strategy_default :
  ?depth:int ->
  ?priority:(Graph.hedge -> int) ->
  ('a, 'b, 'c) Graph.t -> Graph.SetV.t -> strategy
```

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

- depth: to apply the recursive strategy of Bourdncle'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)`.
- 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, wher eone analysis a subgraph of the equation graph.

```
end
```

7.3 Functor

```
module Make :
functor (Graph :  SHGraph.S) -> S  with module Graph = Graph
```

Index