Contents

1 Module Display_hasse_intf : Default specification for drawing with the DOT-utility.

```
module type DEFAULT_SPEC =
  sig
```

DOT-options (see "man dot")

```
    val name : string
    val label : string
    val size_x : float
    val size_y : float
    val ratio : float
    val rotation : float
    val center : bool
    val top_attr : string
```

Node attribute string for top nodes, e.g. "shape = box"

```
    val bot_attr : string
```

Node attribute string for bottom nodes

```
    val top_bot_attr : string
```

Node attribute string for top/bottom nodes

```
    val edge_attr : string
```

Edge attribute string, e.g. "color = blue"

```
  end
module type SPEC =
  sig
    include Display_hasse_intf.DEFAULT_SPEC
    type el
    type +'a node
    val pp_node_attr : Format.formatter ->
      el node -> unit
```

`pp_node_attr ppf node` prints attributes of `node` to the pretty-printer `ppf`.

```
  end
```

Specification for drawing Hasse-diagrams.

```
module type DISPLAY_HASSE =
  sig
    type pomap
    val fprintf : Format.formatter -> pomap -> unit
```

`fprintf ppf pm` prints partially ordered map `pm` to the pretty-printer `ppf`.

```
    val printf : pomap -> unit
```

`printf ppf pm` prints partially ordered map `pm` to `stdout`.

```
  end
```

Interface for drawing Hasse-diagrams.

2 Module `Display_hasse_impl` : Default specification than can be used by the pretty-printer for Hasse-diagrams.

Just `include` it into some module and override the defaults as required.

```
module DefaultSpec :
  sig
    include DEFAULT_SPEC
    val pp_node_attr : Format.formatter -> 'a -> unit
  end
```

Default specification than can be used by the pretty-printer for Hasse-diagrams. Just `include` it into some module and override the defaults as required.

```
module Make :
   functor (POMap :  Pomap_intf.POMAP) -> functor (Spec :  SPEC  with type (+'a) node
= 'a POMap.node) -> DISPLAY_HASSE  with type pomap = Spec.el POMap.pomap
```

Functor that generates a pretty-printer for Hasse-diagrams from a partially ordered map and a pretty-printer specification. See the `Display_hasse_intf.DISPLAY_HASSE`[1]-interface for documentation.

3 Module `Store_intf` : Specification of indices used to index elements in stores

```
module type INDEX =
  sig
```

`type t`

Type of indices

`type gen`

Type of index generators

`module Set :`

`Set.S  with type elt = t`

Efficient sets of indices

`module Map :`

`Map.S  with type key = t`

Efficient maps of indices

`val start : gen`

The start state of the index generator

`val next_ix : gen -> t`

`next_ix gen`
Returns the next index that generator `gen` will produce.

`val next : gen -> t * gen`

`next gen`
Returns the tuple of `(new_ix, new_gen)`, where `new_ix` is the next index and `new_gen` the next state of the index generator.

`val remove_ix : gen -> t -> gen`

`remove_ix gen ix`

Returns an updated index generator which is guaranteed to never return index `ix` or any other previously returned index.

`val int_of_ix : t -> int`

`int_of_ix ix` converts index `ix` to an integer.

Raises `Failure` if index out of range for machine integers.

`end`

`module type STORE =`
`sig`

`module Ix :`
`Store_intf.INDEX`

Index module used to index elements in the store

`type +'a t`

Type of stores

`val empty : 'a t`

The empty store

`val is_empty : 'a t -> bool`

`is_empty s`

Returns `true` if `s` is empty, `false` otherwise.

`val cardinal : 'a t -> int`

`cardinal s`

Returns the number of elements in `s`.

`val next_ix : 'a t -> Ix.t`

`next_ix s`

Returns the next index the store `s` will use to index a new element.

`val singleton : 'a -> Ix.t * 'a t`

`singleton el`

Returns the tuple `(ix, store)`, where `ix` is the index under which the only element `el` was stored, and `store` is the store containing `el`.

`val add : 'a -> 'a t -> Ix.t * 'a t`

`add el s`

Returns the tuple `(new_ix, new_store)`, where `new_ix` is the index under which the new element `el` was stored, and `new_store` is the new store containing `el`.

```
val find : Ix.t -> 'a t -> 'a
```

`find ix s`

Raises `Not_found` if index `ix` not bound.

Returns the element stored under index `ix`.

```
val update : Ix.t -> 'a -> 'a t -> 'a t
```

`update ix el s` rebinds index `ix` in store `s` to point to `el`, and returns the resulting store. The previous binding disappears. New indices resulting from further adds are guaranteed to have higher indices.

Raises `Not_found` if index `ix` not bound.

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

`remove ix s` removes the binding of index `ix` of store `s`, and returns the resulting store.

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

`iter f s` applies `f` to all stored elements in store `s`. The order in which elements are passed to `f` is unspecified.

```
val iteri : (Ix.t -> 'a -> unit) -> 'a t -> unit
```

`iter f s` applies `f` to all indexes and their related elements in store `s`. The order in which elements are passed to `f` is unspecified.

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

`map f s`

Returns a store with all elements in `s` mapped from their original value to the codomain of `f`. Only the elements are passed to `f`. The order in which elements are passed to `f` is unspecified.

```
val mapi : (Ix.t -> 'a -> 'b) -> 'a t -> 'b t
```

`mapi f s` same as `map`, but function `f` also receives the index associated with the elements.

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

`fold f s a` computes `(f eN ... (f e1 a) ...)`, where `e1 ... eN` are the elements of all bindings in store `s`. The order in which the bindings are presented to `f` is unspecified.

```
val foldi : (Ix.t -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

`foldi f s a` same as `fold`, but function `f` also receives the index associated with the elements.

```
val to_list : 'a t -> (Ix.t * 'a) list
```

`to_list s` converts `s` to an association list of indices and elements.

```
val choose : 'a t -> Ix.t * 'a
```

`choose s`
Raises `Not_found` if `s` is empty.
Returns a tuple `(ix, x)`, where `ix` is the index of some unspecified value `x` in store `s`.

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

`filter p s`
Returns the store of all elements in `s` that satisfy `p`.

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

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

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

`eq_classes eq s`
Returns a list of tuples `(el, ec)`, where `el` is the only kind of element as identified by the equivalence relation `eq` stored in the equivalence class (store) `ec` under each index. Every such equivalence class is unique and maximal with respect to `s`, and the original indices of the elements are preserved in each class.

```
val get_ix_map : 'a t -> 'a Ix.Map.t
```

`get_ix_map s`
Returns a map of all indices mapped to their respective elements in store `s`.

```
end
```

Interface to stores

4 Module `Store_impl` : Implementation of stores

Implementation of stores

```
module IntIx :
   Store_intf.INDEX
module Make :
   functor (Ix :  Store_intf.INDEX) -> Store_intf.STORE
module IntStore :
   Store_intf.STORE
```

5 Module `Pomap_intf` : Specification of a partial order relation

```
module type PARTIAL_ORDER =
  sig
```

```
    type el
```

Element type

```
    type ord =
      | Unknown
      | Lower
      | Equal
      | Greater
    val compare : el ->
      el -> ord
  end
module type POMAP =
  sig
```

Modules and types

```
    module Store :
    Store_intf.STORE
```

Store module used to store nodes of the partially ordered map.

```
    type key
```

Type of map keys

```
    type +'a node
```

Type of nodes in the partially ordered map

```
type +'a pomap
```

Type of partially ordered maps

```
type 'a add_find_result =
  | Found of Store.Ix.t * 'a node
  | Added of Store.Ix.t * 'a node * 'a pomap
```

Type of result originating from an `add_find` operation

Map-constructors

```
val empty : 'a pomap
```

The empty partially ordered map.

```
val singleton : key -> 'a -> 'a pomap
```

`singleton k el`
Returns a partially ordered map containing as only binding the one from `k` to `el`.

Information on maps

```
val is_empty : 'a pomap -> bool
```

`is_empty pm` tests whether partially ordered map `pm` is empty.

```
val cardinal : 'a pomap -> int
```

`cardinal pm`
Returns the number of elements in `pm`.

Adding and removing

```
val add : key ->
  'a -> 'a pomap -> 'a pomap
```

`add k el pm`
Returns a partially ordered map containing the same bindings as `pm`, plus a binding of `k` to `el`. If `k` was already bound in `pm`, its previous binding disappears.

```
val add_node : 'a node ->
  'a pomap -> 'a pomap
```

`add_node node pm`
Returns a partially ordered map containing the same bindings as `pm` plus a binding as represented by `node`. If the associated key already existed in `pm`, its previous binding disappears.

```
val remove : key ->
  'a pomap -> 'a pomap
```

```
remove k pm
```

Returns a map containing the same bindings as `pm` except for the node with key `k`.

```
val remove_node : 'a node ->
  'a pomap -> 'a pomap
```

```
remove_node node pm
```

Returns a map containing the same bindings as `pm` except for the one with the key of `node`.

```
val remove_ix : Store.Ix.t -> 'a pomap -> 'a pomap
```

```
remove_ix ix pm
```

Raises `Not_found` if `ix` does not index any node.

Returns a map containing the same bindings as `pm` except for the node indexed by `ix`.

```
val take : key ->
  'a pomap ->
  Store.Ix.t * 'a node * 'a pomap
```

```
take k pm
```

Raises `Not_found` if there is no binding for `key`.

Returns a tuple `(ix, node, map)`, where `ix` is the index of the `node` associated with key `k` in `pm`, and `map` is `pm` without this element.

```
val take_ix : Store.Ix.t ->
  'a pomap ->
  'a node * 'a pomap
```

```
take_ix ix pm
```

Raises `Not_found` if `ix` does not index any node.

Returns a tuple `(n, m)`, where `n` is the node associated with index `ix`, and `m` is a map without this element.

```
val add_find : key ->
  'a -> 'a pomap -> 'a add_find_result
```

`add_find k el pm` similar to `add`, but if the binding did already exist, then `Found (ix, node)` will be returned to indicate the index and node under which key `k` is bound. Otherwise `Added (new_ix, new_pm)` will be returned to indicate that `k` was bound under new index `new_ix` in the partially ordered map `new_pm`.

```
val add_fun : key ->
  'a -> ('a -> 'a) -> 'a pomap -> 'a pomap
```

`add_fun k el f pm` similar to `add`, but if the binding already existed, then function `f` will be applied to the previously bound data. Otherwise the binding will be added as in `add`.

```
val mem : key -> 'a pomap -> bool
```

`mem k pm`

Returns `true` if `pm` contains a binding for key `k` and `false` otherwise.

```
val mem_ix : Store.Ix.t -> 'a pomap -> bool
```

`mem el pm`

Returns `true` if `pm` contains a binding for data element `el` and `false` otherwise.

```
val find : key ->
  'a pomap -> Store.Ix.t * 'a node
```

`find k pm`

Raises `Not_found` if no such binding exists.

Returns a tuple `(ix, node)`, where `ix` is the index of key `k` and `node` its associated node in map `pm`.

```
val find_ix : Store.Ix.t -> 'a pomap -> 'a node
```

`find_ix ix pm`

Raises `Not_found` if such a node does not exist.

Returns the node associated with index `ix` in map `pm`.

```
val choose : 'a pomap -> Store.Ix.t * 'a node
```

`choose pm`

Raises `Not_found` if `pm` is empty.

Returns a tuple `(ix, node)`, where `ix` is the index of the `node` of some unspecified element in `pm`.

```
val filter : (Store.Ix.t -> 'a node -> bool) ->
  'a pomap -> 'a pomap
```

`filter p pm`

Returns the map of all elements in `pm` that satisfy `p`.

```
val partition :
  (Store.Ix.t -> 'a node -> bool) ->
  'a pomap ->
  'a pomap * 'a pomap
```

`partition p pm`

Returns a pair of maps `(pm1, pm2)`, where `pm1` is the map of all the elements of `pm` that satisfy the predicate `p`, and `pm2` is the map of all the elements of `pm` that do not satisfy `p`.

Iterators

```
val iter : ('a node -> unit) -> 'a pomap -> unit
```

`iter f pm` applies `f` to all bound nodes in map `pm`. The order in which the nodes are passed to `f` is unspecified. Only current bindings are presented to `f`: bindings hidden by more recent bindings are not passed to `f`.

```
val iteri : (Store.Ix.t -> 'a node -> unit) ->
  'a pomap -> unit
```

`iteri f pm` same as `Pomap_intf.POMAP.iter`[5], but function `f` also receives the index associated with the nodes.

```
val map : ('a node -> 'b) ->
  'a pomap -> 'b pomap
```

`map f pm`

Returns a map with all nodes in `pm` mapped from their original value to identical nodes whose data element is in the codomain of `f`. The order in which nodes are passed to `f` is unspecified.

```
val mapi : (Store.Ix.t -> 'a node -> 'b) ->
  'a pomap -> 'b pomap
```

`mapi f pm` same as `Pomap_intf.POMAP.map`[5], but function `f` also receives the index associated with the nodes.

```
val fold : ('a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`fold f pm a` computes `(f nN ... (f n1 a) ...)`, where `n1 ... nN` are the nodes in map `pm`. The order in which the nodes are presented to `f` is unspecified.

```
val foldi : (Store.Ix.t -> 'a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`foldi f pm a` same as `Pomap_intf.POMAP.fold`[5], but function `f` also receives the index associated with the nodes.

```
val topo_fold : ('a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`topo_fold f pm a` computes `(f nN ... (f n1 a) ...)`, where `n1 ... nN` are the nodes in map `pm` sorted in ascending topological order. Slower than `fold`.

```
val topo_foldi :
  (Store.Ix.t -> 'a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`topo_foldi f pm a` same as `Pomap_intf.POMAP.topo_fold`[5], but function `f` also receives the index associated with the nodes.

```
val topo_fold_ix : (Store.Ix.t -> 'a -> 'a) -> 'b pomap -> 'a -> 'a
```

topo_fold_ix f pm a same as Pomap_intf.POMAP.topo_fold[5], but function f only receives the index associated with the nodes.

```
val rev_topo_fold : ('a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

rev_topo_fold f pm a computes (f nN ... (f n1 a) ...), where n1 ... nN are the nodes in map pm sorted in descending topological order. Slower than fold.

```
val rev_topo_foldi :
  (Store.Ix.t -> 'a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

rev_topo_foldi f pm a same as Pomap_intf.POMAP.rev_topo_fold[5], but function f also receives the index associated with the nodes.

```
val rev_topo_fold_ix : (Store.Ix.t -> 'a -> 'a) -> 'b pomap -> 'a -> 'a
```

rev_topo_fold_ix f pm a same as Pomap_intf.POMAP.rev_topo_fold[5], but function f only receives the index associated with the nodes.

```
val chain_fold : ('a node list -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

chain_fold f pm a computes (f cN ... (f c1 a) ...), where c1 ... cN are the ascending chaines of nodes in map pm. Only useful for small maps, because of potentially exponential complexity.

```
val chain_foldi :
  ((Store.Ix.t * 'a node) list -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

chain_foldi f pm a same as Pomap_intf.POMAP.chain_fold[5], but function f receives chains including the index associated with the nodes.

```
val rev_chain_fold : ('a node list -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

rev_chain_fold f pm a computes (f cN ... (f c1 a) ...), where c1 ... cN are the descending chaines of nodes in map pm. Only useful for small maps, because of potentially exponential complexity.

```
val rev_chain_foldi :
  ((Store.Ix.t * 'a node) list -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

rev_chain_foldi f pm a same as Pomap_intf.POMAP.rev_chain_fold[5], but function f receives chains including the index associated with the nodes.

Set-like map-operations

```
val union : 'a pomap ->
  'a pomap -> 'a pomap
```

union pm1 pm2 merges pm1 and pm2, preserving the bindings of pm1.

```
val inter : 'a pomap ->
  'a pomap -> 'a pomap
```

inter pm1 pm2 intersects pm1 and pm2, preserving the bindings of pm1.

```
val diff : 'a pomap ->
  'a pomap -> 'a pomap
```

diff pm1 pm2 removes all elements of pm2 from pm1.

Node-creators and accessors

```
val create_node : key ->
  'a -> Store.Ix.Set.t -> Store.Ix.Set.t -> 'a node
```

create_node k el sucs prds
Returns a node with key k, data element el, successors sucs and predecessors prds.

```
val get_key : 'a node -> key
```

get_key n
Returns the key associated with node n.

```
val get_el : 'a node -> 'a
```

get_el n
Returns the data element associated with node n.

```
val get_sucs : 'a node -> Store.Ix.Set.t
```

get_sucs n
Returns the successors associated with node n.

```
val get_prds : 'a node -> Store.Ix.Set.t
```

get_prds n
Returns the predecessors associated with node n.

```
val set_key : 'a node -> key -> 'a node
```

set_key n k sets the key of node n to k and returns new node.

```
val set_el : 'a node -> 'a -> 'a node
```

set_el n el sets the data element of node n to el and returns new node.

```
val set_sucs : 'a node -> Store.Ix.Set.t -> 'a node
```

`set_sucs n sucs` set the successors of node `n` to `sucs` and returns new node.

```
val set_prds : 'a node -> Store.Ix.Set.t -> 'a node
```

`set_prds n prds` set the predecessors of node `n` to `prds` and returns new node.

Map-accessors

```
val get_nodes : 'a pomap -> 'a node Store.t
```

`get_nodes pm`
Returns the store of nodes associated with partially ordered map `pm`. This store represents the Hasse-graph of the nodes partially ordered by their keys.

```
val get_top : 'a pomap -> Store.Ix.Set.t
```

`get_top pm`
Returns the set of node indices of nodes that are greater than any other node in `pm` but themselves.

```
val get_bot : 'a pomap -> Store.Ix.Set.t
```

`get_bot pm`
Returns the set of node indices of nodes that are lower than any other node in `pm` but themselves.

Operations over equivalences of data elements

```
val remove_eq_prds : ('a -> 'a -> bool) -> 'a pomap -> 'a pomap
```

`remove_eq_prds eq pm`
Returns a map containing the same bindings as `pm` except for nodes whose non-empty predecessors all have the same data element as identified by `eq`.

```
val fold_eq_classes :
  ('a -> 'a -> bool) ->
  ('a -> 'a pomap -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`fold_eq_classes eq f pm a` factorizes `pm` into maximal equivalence classes of partial orders: all bindings in each class have equivalent data elements as identified by `eq` and are connected in the original Hasse-diagram. This function then computes `(f ec_elN ecN ... (f ec_el1 ec1 a))`, where `ec1 ... ecN` are the mentioned equivalence classes in unspecified order, and `ec_el1 ... ec_elN` are their respective common data elements.

```
val fold_split_eq_classes :
  ('a -> 'a -> bool) ->
  ('a -> 'a pomap -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`fold_split_eq_classes eq f pm a` same as `Pomap_intf.POMAP.fold_eq_classes`[5], but the equivalence classes are split further so that no element of other classes would fit between its bottom and top elements. It is unspecified how non-conflicting elements are assigned to upper or lower classes!

```
val preorder_eq_classes : ('a -> 'a -> bool) ->
  'a pomap -> 'a pomap list
```

`preorder_eq_classes eq pm`
Returns a preordered list of equivalence classes, the latter being defined as in `fold_split_eq_classes`.

```
val topo_fold_reduced :
  ('a -> 'a -> bool) ->
  ('a node -> 'b -> 'b) ->
  'a pomap -> 'b -> 'b
```

`topo_fold_reduced eq f pm a` computes `(f nN ... (f n1 a) ...)`, where `n1 ... nN` are those nodes in map `pm` sorted in ascending topological order, whose data element is equivalent as defined by `eq` to the one of lower elements if there are no intermediate elements that violate this equivalence.

Unsafe operations - USE WITH CAUTION!

```
val unsafe_update : 'a pomap ->
  Store.Ix.t -> 'a node -> 'a pomap
```

`unsafe_update pm ix node` updates the node associated with node index `ix` in map `pm` with `node`. The Hasse-diagram associated with the partially ordered map `pm` may become inconsistent if the new node violates the partial order structure. This can lead to unpredictable results with other functions!

```
val unsafe_set_nodes : 'a pomap ->
  'a node Store.t -> 'a pomap
```

`unsafe_set_nodes pm s` updates the node store associated with map `pm` with `s`. This assumes that `s` stores a consistent Hasse-diagram of nodes.

```
val unsafe_set_top : 'a pomap -> Store.Ix.Set.t -> 'a pomap
```

`unsafe_set_top pm set` updates the index of top nodes in map `pm` with `set`. This assumes that the nodes referenced by the node indices in `set` do not violate the properties of the Hasse-diagram of `pm`.

```
val unsafe_set_bot : 'a pomap -> Store.Ix.Set.t -> 'a pomap
```

`unsafe_set_bot pm set` updates the index of bottom nodes in map `pm` with `set`. This assumes that the nodes referenced by the node indices in `set` do not violate the properties of the Hasse-diagram of `pm`.

```
end
```

Interface to partially ordered maps

6 Module `Pomap_impl` : Implementation of partially ordered maps

Implementation of partially ordered maps

```
module Make :
   functor (PO : Pomap_intf.PARTIAL_ORDER) -> POMAP  with type key = PO.el
```