Interface for module Hashset

1. This module implements imperative sets as hash tables. Operations like union, intersection or difference are not provided, since there is no way to implement them easily (i.e. more easily than iterating over sets).

2. Generic interface

type $\alpha\ t$
(* The type of sets. Elements have type α. *)

val *create* : $int \to \alpha\ t$
(* *Hashset.create* n creates a new, empty set. For best results, n should be on the order of the expected number of elements that will be in the set. The internal structure grows as needed, so n is just an initial guess. *)

val *clear* : $\alpha\ t \to unit$
(* Empty a set. *)

val *add* : $\alpha\ t \to \alpha \to unit$
(* *Hashset.add* s x adds x into the set s. *)

val *copy* : $\alpha\ t \to \alpha\ t$
(* Return a copy of the given set. *)

val *mem* : $\alpha\ t \to \alpha \to bool$
(* *Hashset.mem* s x checks if x belongs to s. *)

val *remove* : $\alpha\ t \to \alpha \to unit$
(* *Hashset.remove* s x removes x from s. It does nothing if x does not belong to s. *)

val *cardinal* : $\alpha\ t \to int$
(* *Hashset.cardinal* s returns the cardinal of s. *)

val *iter* : $(\alpha \to unit) \to \alpha\ t \to unit$
(* *Hashset.iter* f s applies f to all elements in s. *)

val *fold* : $(\alpha \to \beta \to \beta) \to \alpha\ t \to \beta \to \beta$
(* *Hashset.fold* f s *init* computes (f *eN* ... (f *e1* *init*)...), where *e1* ... *eN* are the elements in s. The order in which the elements are passed to f is unspecified. *)

3. Functorial interface

```
module type HashedType =
  sig
    type t
      (* The type of the elements. *)
    val equal : t -> t -> bool
      (* The equality predicate used to compare elements. *)
    val hash : t -> int
```

(∗ A hashing function on elements. It must be such that if two elements are equal according to *equal*, then they have identical hash values as computed by *hash*. Examples: suitable (*equal*, *hash*) pairs for arbitrary element types include ((=), !Hashset.hash) for comparing objects by structure, and ((≡), !Hashset.hash) for comparing objects by addresses (e.g. for mutable or cyclic keys). ∗)

```
  end
```

The input signature of the functor !Hashset.Make.

```
module type S =
  sig
    type elt
    type t
    val create : int -> t
    val clear : t -> unit
    val copy : t -> t
    val add : t -> elt -> unit
    val remove : t -> elt -> unit
    val mem : t -> elt -> bool
    val cardinal : t -> int
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> α -> α) -> t -> α -> α
  end
```

(∗ The output signature of the functor !Hashset.Make. ∗)

```
module Make (H : HashedType) : S with type elt = H.t
```

(∗ Functor building an implementation of the hashtable structure. The functor *Hashset.Make* returns a structure containing a type *elt* of elements and a type *t* of hash sets. 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. ∗)

Module Hashset

4. Sets as hash tables. Code adapted from ocaml's Hashtbl.
We do dynamic hashing, and resize the table and rehash the elements when buckets become too long.

```
type α t =
  { mutable size : int; (* number of elements *)
    mutable data : α list array } (* the buckets *)

let create initial_size =
  let s = min (max 1 initial_size) Sys.max_array_length in
  { size = 0; data = Array.make s [] }

let clear h =
  for i = 0 to Array.length h.data − 1 do
    h.data.(i) ← []
  done;
  h.size ← 0

let copy h =
  { size = h.size;
    data = Array.copy h.data }

let resize hashfun tbl =
  let odata = tbl.data in
  let osize = Array.length odata in
  let nsize = min (2 × osize + 1) Sys.max_array_length in
  if nsize ≠ osize then begin
    let ndata = Array.create nsize [] in
    let rec insert_bucket = function
        [] → ()
      | key :: rest →
          insert_bucket rest; (* preserve original order of elements *)
          let nidx = (hashfun key) mod nsize in
          ndata.(nidx) ← key :: ndata.(nidx) in
    for i = 0 to osize − 1 do
      insert_bucket odata.(i)
    done;
    tbl.data ← ndata;
  end

let add h key =
  let i = (Hashtbl.hash key) mod (Array.length h.data) in
  let bucket = h.data.(i) in
  if ¬ (List.mem key bucket) then begin
```

```
    h.data.(i) <- key :: bucket;
    h.size <- succ h.size;
    if h.size > Array.length h.data lsl 1 then resize Hashtbl.hash h
  end

let remove h key =
  let rec remove_bucket = function
      [] ->
        []
    | k :: next ->
        if k = key
        then begin h.size <- pred h.size; next end
        else k :: remove_bucket next in
  let i = (Hashtbl.hash key) mod (Array.length h.data) in
  h.data.(i) <- remove_bucket h.data.(i)

let mem h key =
  List.mem key h.data.((Hashtbl.hash key) mod (Array.length h.data))

let cardinal h =
  let d = h.data in
  let c = ref 0 in
  for i = 0 to Array.length d - 1 do
    c := !c + List.length d.(i)
  done;
  !c

let iter f h =
  let d = h.data in
  for i = 0 to Array.length d - 1 do
    List.iter f d.(i)
  done

let fold f h init =
  let rec do_bucket b accu =
    match b with
      [] ->
        accu
    | k :: rest ->
        do_bucket rest (f k accu) in
  let d = h.data in
  let accu = ref init in
  for i = 0 to Array.length d - 1 do
```

```
    accu := do_bucket d.(i) !accu
  done;
  !accu
```

Functorial interface

```
module type HashedType =
  sig
    type t
    val equal : t → t → bool
    val hash : t → int
  end

module type S =
  sig
    type elt
    type t
    val create : int → t
    val clear : t → unit
    val copy : t → t
    val add : t → elt → unit
    val remove : t → elt → unit
    val mem : t → elt → bool
    val cardinal : t → int
    val iter : (elt → unit) → t → unit
    val fold : (elt → α → α) → t → α → α
  end

module Make(H : HashedType) : (S with type elt = H.t) =
  struct
    type elt = H.t
    type set = elt t
    type t = set
    let create = create
    let clear = clear
    let copy = copy

    let safehash key = (H.hash key) land max_int

    let rec mem_in_bucket key = function
      | [] → false
      | x :: r → H.equal key x ∨ mem_in_bucket key r
```

```
let add h key =
  let i = (safehash key) mod (Array.length h.data) in
  let bucket = h.data.(i) in
  if ¬ (mem_in_bucket key bucket) then begin
    h.data.(i) ← key :: bucket;
    h.size ← succ h.size;
    if h.size > Array.length h.data lsl 1 then resize safehash h
  end

let remove h key =
  let rec remove_bucket = function
        [] →
          []
    | k :: next →
          if H.equal k key
          then begin h.size ← pred h.size; next end
          else k :: remove_bucket next in
  let i = (safehash key) mod (Array.length h.data) in
  h.data.(i) ← remove_bucket h.data.(i)

let mem h key =
  mem_in_bucket key h.data.((safehash key) mod (Array.length h.data))

let cardinal = cardinal

let iter = iter
let fold = fold
end
```