Interface for module Hashcons

1. Hash tables for hash consing.

Hash consed values are of the following type *hash_consed*. The field *tag* contains a unique integer (for values hash consed with the same table). The field *hkey* contains the hash key of the value (without modulo) for possible use in other hash tables (and internally when hash consing tables are resized). The field *node* contains the value itself.

Hash consing tables are using weak pointers, so that values that are no more referenced from anywhere else can be erased by the GC.

```
type α hash_consed = private {
  hkey : int;
  tag : int;
  node : α }
```

2. Generic part, using ocaml generic equality and hash function.

```
type α t

val create : int → α t
  (** create n creates an empty table of initial size n. The table will grow as needed. *)
val clear : α t → unit
  (** Removes all elements from the table. *)
val hashcons : α t → α → α hash_consed
  (** hashcons t n hash-cons the value n using table t i.e. returns any existing value in t
  equal to n, if any; otherwise, allocates a new one hash-consed value of node n and returns
  it. As a consequence the returned value is physically equal to any equal value already hash-
  consed using table t. *)
val iter : (α hash_consed → unit) → α t → unit
  (** iter f t iterates f over all elements of t. *)
val stats : α t → int × int × int × int × int × int
  (** Return statistics on the table. The numbers are, in order: table length, number of
  entries, sum of bucket lengths, smallest bucket length, median bucket length, biggest bucket
  length. *)
```

3. Functorial interface.

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

```
module type S =
  sig
    type key
    type t
    val create : int → t
    val clear : t → unit
    val hashcons : t → key → key hash_consed
    val iter : (key hash_consed → unit) → t → unit
    val stats : t → int × int × int × int × int × int
  end

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

Module Hashcons

4. Hash tables for hash-consing. (Some code is borrowed from the ocaml standard library, which is copyright 1996 INRIA.)

```
type α hash_consed = {
  hkey : int;
  tag : int;
  node : α }

let gentag =
  let r = ref 0 in
  fun () → incr r; !r

type α t = {
  mutable table : α hash_consed Weak.t array;
  mutable totsize : int; (* sum of the bucket sizes *)
  mutable limit : int; (* max ratio totsize/table length *)
}

let create sz =
  let sz = if sz < 7 then 7 else sz in
  let sz = if sz > Sys.max_array_length then Sys.max_array_length else sz in
  let emptybucket = Weak.create 0 in
  { table = Array.create sz emptybucket;
    totsize = 0;
    limit = 3; }
```

```
let clear t =
  let emptybucket = Weak.create 0 in
  for i = 0 to Array.length t.table - 1 do t.table.(i) <- emptybucket done;
  t.totsize <- 0;
  t.limit <- 3

let fold f t init =
  let rec fold_bucket i b accu =
    if i >= Weak.length b then accu else
      match Weak.get b i with
        | Some v -> fold_bucket (i+1) b (f v accu)
        | None -> fold_bucket (i+1) b accu
  in
  Array.fold_right (fold_bucket 0) t.table init

let iter f t =
  let rec iter_bucket i b =
    if i >= Weak.length b then () else
      match Weak.get b i with
        | Some v -> f v; iter_bucket (i+1) b
        | None -> iter_bucket (i+1) b
  in
  Array.iter (iter_bucket 0) t.table

let count t =
  let rec count_bucket i b accu =
    if i >= Weak.length b then accu else
      count_bucket (i+1) b (accu + (if Weak.check b i then 1 else 0))
  in
  Array.fold_right (count_bucket 0) t.table 0

let next_sz n = min (3 * n/2 + 3) (Sys.max_array_length - 1)

let rec resize t =
  let oldlen = Array.length t.table in
  let newlen = next_sz oldlen in
  if newlen > oldlen then begin
    let newt = create newlen in
    newt.limit <- t.limit + 100; (* prevent resizing of newt *)
    fold (fun d () -> add newt d) t ();
    t.table <- newt.table;
    t.limit <- t.limit + 2;
  end
```

```
and add t d =
  let index = d.hkey mod (Array.length t.table) in
  let bucket = t.table.(index) in
  let sz = Weak.length bucket in
  let rec loop i =
    if i >= sz then begin
      let newsz = min (sz + 3) (Sys.max_array_length - 1) in
      if newsz <= sz then
        failwith "Hashcons.Make: hash bucket cannot grow more";
      let newbucket = Weak.create newsz in
      Weak.blit bucket 0 newbucket 0 sz;
      Weak.set newbucket i (Some d);
      t.table.(index) <- newbucket;
      t.totsize <- t.totsize + (newsz - sz);
      if t.totsize > t.limit * Array.length t.table then resize t;
    end else begin
      if Weak.check bucket i
      then loop (i+1)
      else Weak.set bucket i (Some d)
    end
  in
  loop 0

let hashcons t d =
  let hkey = Hashtbl.hash d in
  let index = hkey mod (Array.length t.table) in
  let bucket = t.table.(index) in
  let sz = Weak.length bucket in
  let rec loop i =
    if i >= sz then begin
      let hnode = { hkey = hkey; tag = gentag (); node = d } in
      add t hnode;
      hnode
    end else begin
      match Weak.get_copy bucket i with
        | Some v when v.node = d ->
            begin match Weak.get bucket i with
              | Some v -> v
              | None -> loop (i+1)
            end
        | _ -> loop (i+1)
```

```
    end
  in
  loop 0

let stats t =
  let len = Array.length t.table in
  let lens = Array.map Weak.length t.table in
  Array.sort compare lens;
  let totlen = Array.fold_left ( + ) 0 lens in
  (len, count t, totlen, lens.(0), lens.(len/2), lens.(len - 1))
```

Functorial interface

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

module type S =
  sig
    type key
    type t
    val create : int → t
    val clear : t → unit
    val hashcons : t → key → key hash_consed
    val iter : (key hash_consed → unit) → t → unit
    val stats : t → int × int × int × int × int × int
  end

module Make(H : HashedType) : (S with type key = H.t) = struct

  type key = H.t

  type data = H.t hash_consed

  type t = {
    mutable table : data Weak.t array;
    mutable totsize : int; (* sum of the bucket sizes *)
    mutable limit : int; (* max ratio totsize/table length *)
  }

  let emptybucket = Weak.create 0
```

```
let create sz =
  let sz = if sz < 7 then 7 else sz in
  let sz = if sz > Sys.max_array_length then Sys.max_array_length else sz in
  {
    table = Array.create sz emptybucket;
    totsize = 0;
    limit = 3;
  }

let clear t =
  for i = 0 to Array.length t.table - 1 do
    t.table.(i) ← emptybucket
  done;
  t.totsize ← 0;
  t.limit ← 3

let fold f t init =
  let rec fold_bucket i b accu =
    if i ≥ Weak.length b then accu else
    match Weak.get b i with
    | Some v → fold_bucket (i + 1) b (f v accu)
    | None → fold_bucket (i + 1) b accu
  in
  Array.fold_right (fold_bucket 0) t.table init

let iter f t =
  let rec iter_bucket i b =
    if i ≥ Weak.length b then () else
    match Weak.get b i with
    | Some v → f v; iter_bucket (i + 1) b
    | None → iter_bucket (i + 1) b
  in
  Array.iter (iter_bucket 0) t.table

let count t =
  let rec count_bucket i b accu =
    if i ≥ Weak.length b then accu else
    count_bucket (i + 1) b (accu + (if Weak.check b i then 1 else 0))
  in
  Array.fold_right (count_bucket 0) t.table 0

let next_sz n = min (3 × n/2 + 3) (Sys.max_array_length - 1)
```

```
let rec resize t =
  let oldlen = Array.length t.table in
  let newlen = next_sz oldlen in
  if newlen > oldlen then begin
    let newt = create newlen in
    newt.limit ← t.limit + 100; (* prevent resizing of newt *)
    fold (fun d () → add newt d) t ();
    t.table ← newt.table;
    t.limit ← t.limit + 2;
  end

and add t d =
  let index = d.hkey mod (Array.length t.table) in
  let bucket = t.table.(index) in
  let sz = Weak.length bucket in
  let rec loop i =
    if i ≥ sz then begin
      let newsz = min (sz + 3) (Sys.max_array_length − 1) in
      if newsz ≤ sz then
        failwith "Hashcons.Make: hash bucket cannot grow more";
      let newbucket = Weak.create newsz in
      Weak.blit bucket 0 newbucket 0 sz;
      Weak.set newbucket i (Some d);
      t.table.(index) ← newbucket;
      t.totsize ← t.totsize + (newsz − sz);
      if t.totsize > t.limit × Array.length t.table then resize t;
    end else begin
      if Weak.check bucket i
      then loop (i + 1)
      else Weak.set bucket i (Some d)
    end
  in
  loop 0

let hashcons t d =
  let hkey = H.hash d in
  let index = hkey mod (Array.length t.table) in
  let bucket = t.table.(index) in
  let sz = Weak.length bucket in
  let rec loop i =
    if i ≥ sz then begin
```

```
        let hnode = { hkey = hkey; tag = gentag (); node = d } in
        add t hnode;
        hnode
      end else begin
        match Weak.get_copy bucket i with
        | Some v when H.equal v.node d ->
            begin match Weak.get bucket i with
              | Some v -> v
              | None -> loop (i + 1)
            end
        | _ -> loop (i + 1)
      end
    in
    loop 0

  let stats t =
    let len = Array.length t.table in
    let lens = Array.map Weak.length t.table in
    Array.sort compare lens;
    let totlen = Array.fold_left ( + ) 0 lens in
    (len, count t, totlen, lens.(0), lens.(len/2), lens.(len - 1))

end
```