Interface for module Trie

1. This module implements *tries*. Given a map M over an arbitrary type $M.key$, the following functor constructs a new map over type $M.key\ list$.

module type S = sig
  type key
  type $+\alpha\ t$
  val $empty$: $\alpha\ t$
  val is_empty : $\alpha\ t \rightarrow bool$
  val add : $key \rightarrow \alpha \rightarrow \alpha\ t \rightarrow \alpha\ t$
  val $find$: $key \rightarrow \alpha\ t \rightarrow \alpha$
  val $remove$: $key \rightarrow \alpha\ t \rightarrow \alpha\ t$
  val mem : $key \rightarrow \alpha\ t \rightarrow bool$
  val $iter$: $(key \rightarrow \alpha \rightarrow unit) \rightarrow \alpha\ t \rightarrow unit$
  val map : $(\alpha \rightarrow \beta) \rightarrow \alpha\ t \rightarrow \beta\ t$
  val $mapi$: $(key \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha\ t \rightarrow \beta\ t$
  val $fold$: $(key \rightarrow \alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha\ t \rightarrow \beta \rightarrow \beta$
  val $compare$: $(\alpha \rightarrow \alpha \rightarrow int) \rightarrow \alpha\ t \rightarrow \alpha\ t \rightarrow int$
  val $equal$: $(\alpha \rightarrow \alpha \rightarrow bool) \rightarrow \alpha\ t \rightarrow \alpha\ t \rightarrow bool$
end

module $Make(M : S)$: (S with type $key = M.key\ list$)

Module Trie

2. Id

3. A trie is a tree-like structure to implement dictionaries over keys which have list-like structures. The idea is that each node branches on an element of the list and stores the value associated to the path from the root, if any. Therefore, a trie can be defined as soon as a map over the elements of the list is given.

module type S = sig
  type key
  type $+\alpha\ t$
  val $empty$: $\alpha\ t$
  val is_empty : $\alpha\ t \rightarrow bool$
  val add : $key \rightarrow \alpha \rightarrow \alpha\ t \rightarrow \alpha\ t$
  val $find$: $key \rightarrow \alpha\ t \rightarrow \alpha$

```
  val remove : key → α t → α t
  val mem : key → α t → bool
  val iter : (key → α → unit) → α t → unit
  val map : (α → β) → α t → β t
  val mapi : (key → α → β) → α t → β t
  val fold : (key → α → β → β) → α t → β → β
  val compare : (α → α → int) → α t → α t → int
  val equal : (α → α → bool) → α t → α t → bool
end

module Make (M : S) = struct
```

4. Then a trie is just a tree-like structure, where a possible information is stored at the node (α *option*) and where the sons are given by a map from type *key* to sub-tries, so of type α *t* *M.t*. The empty trie is just the empty map.

```
  type key = M.key list

  type α t = Node of α option × α t M.t

  let empty = Node (None, M.empty)

  let is_empty = function
    | Node (None, m1) → M.is_empty m1
    | _ → false
```

5. To find a mapping in a trie is easy: when all the elements of the key have been read, we just inspect the optional info at the current node; otherwise, we descend in the appropriate sub-trie using *M.find*.

```
  let rec find l t = match (l, t) with
    | [], Node (None, _) → raise Not_found
    | [], Node (Some v, _) → v
    | x :: r, Node (_, m) → find r (M.find x m)

  let rec mem l t = match (l, t) with
    | [], Node (None, _) → false
    | [], Node (Some _, _) → true
    | x :: r, Node (_, m) → try mem r (M.find x m) with Not_found → false
```

6. Insertion is more subtle. When the final node is reached, we just put the information (*Some v*). Otherwise, we have to insert the binding in the appropriate sub-trie t'. But it may not exists, and in that case t' is bound to an empty trie. Then we get a new sub-trie t'' by a recursive insertion and we modify the branching, so that it now points to t'', with *M.add*.

```
let add l v t =
  let rec ins = function
    | [], Node (_, m) → Node (Some v, m)
    | x :: r, Node (v, m) →
        let t' = try M.find x m with Not_found → empty in
        let t'' = ins (r, t') in
        Node (v, M.add x t'' m)
  in
  ins (l, t)
```

7. When removing a binding, we take care of not leaving bindings to empty sub-tries in the nodes. Therefore, we test wether the result t' of the recursive call is the empty trie *empty*: if so, we just remove the branching with $M.remove$; otherwise, we modify it with $M.add$.

```
let rec remove l t = match (l, t) with
  | [], Node (_, m) → Node (None, m)
  | x :: r, Node (v, m) →
      try
        let t' = remove r (M.find x m) in
        Node (v, if is_empty t' then M.remove x m else M.add x t' m)
      with Not_found →
        t
```

8. The iterators *map*, *mapi*, *iter* and *fold* are implemented in a straigthforward way using the corresponding iterators $M.map$, $M.mapi$, $M.iter$ and $M.fold$. For the last three of them, we have to remember the path from the root, as an extra argument *revp*. Since elements are pushed in reverse order in *revp*, we have to reverse it with *List.rev* when the actual binding has to be passed to function f.

```
let rec map f = function
  | Node (None, m) → Node (None, M.map (map f) m)
  | Node (Some v, m) → Node (Some (f v), M.map (map f) m)

let mapi f t =
  let rec maprec revp = function
  | Node (None, m) →
      Node (None, M.mapi (fun x → maprec (x :: revp)) m)
  | Node (Some v, m) →
      Node (Some (f (List.rev revp) v), M.mapi (fun x → maprec (x :: revp)) m)
  in
  maprec [] t
```

```
let iter f t =
  let rec traverse revp = function
    | Node (None, m) →
        M.iter (fun x → traverse (x :: revp)) m
    | Node (Some v, m) →
        f (List.rev revp) v; M.iter (fun x t → traverse (x :: revp) t) m
  in
  traverse [] t

let rec fold f t acc =
  let rec traverse revp t acc = match t with
    | Node (None, m) →
        M.fold (fun x → traverse (x :: revp)) m acc
    | Node (Some v, m) →
        f (List.rev revp) v (M.fold (fun x → traverse (x :: revp)) m acc)
  in
  traverse [] t acc

let compare cmp a b =
  let rec comp a b = match a, b with
    | Node (Some _, _), Node (None, _) → 1
    | Node (None, _), Node (Some _, _) → −1
    | Node (None, m1), Node (None, m2) →
        M.compare comp m1 m2
    | Node (Some a, m1), Node (Some b, m2) →
        let c = cmp a b in
        if c ≠ 0 then c else M.compare comp m1 m2
  in
  comp a b

let equal eq a b =
  let rec comp a b = match a, b with
    | Node (None, m1), Node (None, m2) →
        M.equal comp m1 m2
    | Node (Some a, m1), Node (Some b, m2) →
        eq a b ∧ M.equal comp m1 m2
    | _ →
        false
  in
  comp a b
end
```

Index