Programmation Fontionnelle Avancée

Correction de l'examen du Jeudi 12 Janvier 2017, réalisée par un étudiant pendant ses révisions. Des erreurs ont pu s'y glisser.

Exercice 1

1. ```
   module type ORDER = sig
     type t
     val leq : t -> t -> bool
     val equal : t -> t -> bool
   end

   module Int : ORDER with type t = int = struct
     type t = int
     let leq = (<=)
     let equal = (=)
   end
   ```

   We need to expose the fact that endpoint is equal to `Int.t` (or more generally, `Endpoint.t`, where `Endpoint` is the argument to the functor). One way of doing this is through a *sharing constraint*, which allows you to tell the compiler to expose the fact that a given type is equal to some other type.

   The result of this expression is a new signature that's been modified so that it exposes the fact that `type` defined inside of the module type is equal to `type'` whose definition is outside of it.

From *Real World Ocaml* > Functors > Sharing Constraints.

2. ```
   module type INTERVAL = sig
     exception NoOverlap
     type element
     type t
     val create : element -> element -> t
     val equal : t -> t -> bool
     val mem : element -> t -> bool
     val before : t -> t -> bool
     val intersection : t -> t -> t
     val union : t -> t -> t
   end

   module MakeInterval (Endpoint : ORDER)
          : (INTERVAL with type element = Endpoint.t) = struct
     exception NoOverlap
     type element = Endpoint.t
     type t = Empty | Interval of Endpoint.t * Endpoint.t

     let compare a b =
       if Endpoint.equal a b then 0 else if Endpoint.leq a b then -1 else 1
     let min x y = if compare x y <= 0 then x else y
     let max x y = if compare x y >= 0 then x else y

     let create low high =
       if compare low high > 0 then Empty
       else Interval (low, high)

     let mem x t =
       match t with
       | Empty -> false
       | Interval (l, h) ->
          compare x l >= 0 && compare x h <= 0

     let equal t1 t2 =
       match t1, t2 with
   ```

```
    | Empty, Empty -> true
    | Interval (l1, h1), Interval (l2, h2)
        when 0 = compare l1 l2 && 0 = compare h1 h2 -> true
    | _ -> false

  let before t1 t2 =
    match t1, t2 with
    | Empty, _ | _, Empty -> true (* ad-hoc *)
    | Interval (_, h1), Interval (l2, _) -> compare h1 l2 < 0

  let intersection t1 t2 =
    match t1, t2 with
    | Empty, _ | _, Empty -> Empty
    | Interval (l1, h1), Interval (l2, h2) ->
       if before t1 t2 || before t2 t1 then raise NoOverlap
       else create (max l1 l2) (min h1 h2)

  let union t1 t2 =
    match t1, t2 with
    | Empty, Empty -> Empty
    | Empty, t | t, Empty -> t
    | Interval (l1, h1), Interval (l2, h2) -> create (min l1 l2) (max h1 h2)
end

module I = MakeInterval(Int)
I.(mem 42 (intersection (create 51 81) (create 40 60)))
```

3. ```
   module type SET = sig
     type element
     type t
     val interval : element -> element -> t
     val union : t -> t -> t
     val mem : element -> t -> bool
     val equal : t -> t -> bool
   end

   module MakeSet (Endpoint : ORDER) = struct

     module Interval = MakeInterval(Endpoint)
     type element = Interval.t
     type t = Interval.t list

     let interval l1 h1 = [Interval.create l1 h1]

     let rec union set set' =
       let rec aux set set' acc =
         match set, set' with
         | [], [] -> acc
         | [], s | s, [] -> List.rev acc @ s
         | h :: t, h' :: t' ->
            if Interval.before h h' then
              aux t set' (h :: acc)
            else if Interval.before h' h then
              aux set t' (h' :: acc)
            else                    (* a non-empty intersection *)
              let new_interval = Interval.union h h' in
              let new_set = union [new_interval] t in
              let left = union new_set t in
              aux left t' acc
       in
       aux set set' []
   ```
   ```

```
    let mem x set =
      let rec aux x' = function
        | [] -> false
        | h :: t ->
           if Interval.mem x h then true
           else if Interval.before x' h then aux x' t
           else false
      in
      aux (Interval.create x x) set

    (* Il n'est pas clair ici si on parle d'égalité structurelle sur les
       ensembles et les intervalles, ou d'égalité logique dans le sens
       ou deux ensembles sont égaux ssi l'union de leurs intervalles est
       égale.  Il me semble qu'il est impossible d'implémenter l'égalité
       logique sans pouvoir décomposer un intervalle, ce qui n'est pas
       permis par les interfaces. *)
    let rec equal set set' =
      try
        List.for_all2 (fun s s' -> Interval.equal s s') set set'
      with Invalid_argument _ -> false
  end
```

Exercice 2

```
type 'a stream = 'a cell Lazy.t
and 'a cell = Cons of 'a * 'a stream

let rec take n s =
  if n = 0 then []
  else match s with lazy (Cons (x, s')) -> x :: take (n - 1) s'
```

1. ```
   let rec from n = lazy (Cons (n, from (n+1)))
   let numbers = from 1
   ```

2. ```
   let sum s =
     let rec aux s n =
       match s with
       | lazy (Cons (x, s)) -> lazy (Cons (n + x, aux s (n + x)))
     in
     aux s 0

   let x = take 10 (sum numbers)
   ```

3. ```
   let decimate k s =
     let rec aux k' = function
       | lazy (Cons (x, s)) ->
          if k' = 0 then
            match s with
            | lazy (Cons (x', s')) -> lazy (Cons (x', aux (k - 1) s'))
          else lazy (Cons (x, aux (k' - 1) s))
     in
     aux k s

   let x = take 10 (decimate 2 numbers)
   ```

4. ```
   let moessner n =
     let rec aux n s =
       if n = 0 then
         match s with
         | lazy (Cons (x, s)) -> lazy (Cons (x, s))
       else aux (n - 1) (sum (decimate n s))
     in
   ```

```
  aux (n - 1) numbers

let x = take 3 (moessner 4)
```

5. Comptez vous-mêmes.

Exercice 3

1. ```
   type ('a, _) sequence =
     | Nil : ('a, unit) sequence
     | Cons : 'a * ('a, 'l) sequence -> ('a, 'l * unit) sequence

   let x = Nil
   (* val x : ('a, unit) sequence = Nil *)
   let y = Cons (1, Cons (2, Nil))
   (* val y : (int, (unit * unit) * unit) sequence = Cons (1, Cons (2, Nil)) *)
   ```

2. ```
   type ('a, 'h) bintree =
     | Leaf : ('a, unit) bintree
     | Node : ('a, 'h) bintree * 'a * ('a, 'h) bintree -> ('a, 'h * unit) bintree

   let tx = Node (Node (Leaf, 1, Leaf), 3, Node (Leaf, 2, Leaf))
   (* val tx : (int, (unit * unit) * unit) bintree *)
   let tx' = Node (Leaf, 42, Node (Leaf, 1337, Leaf))
   (* Error: This expression has type (int, 'a * unit) bintree
    *        but an expression was expected of type (int, unit) bintree
    *        Type 'a * unit is not compatible with type unit *)
   ```

   bonus:

   ```
   let rec depth : type a h. (a, h) bintree -> int = function
     | Leaf -> 0
     | Node (tl, _, _) -> 1 + depth tl
   (* val depth : ('a, 'h) bintree -> int = <fun> *)
   let d = depth tx
   (* val d : int = 2 *)
   ```

3. ```
   let rec treemap : type a b h. (a -> b) -> (a, h) bintree -> (b, h) bintree =
     fun f t ->
     match t with
     | Leaf -> Leaf
     | Node (tl, v, tr) -> Node (treemap f tl, f v, treemap f tr)
   (* val treemap : ('a -> 'b) -> ('a, 'h) bintree -> ('b, 'h) bintree = <fun> *)
   let t = treemap (fun x -> x + 1) tx
   (* val t : (int, (unit * unit) * unit) bintree =
    *   Node (Node (Leaf, 2, Leaf), 4, Node (Leaf, 3, Leaf)) *)
   ```

4. ```
   let rec treemap2 : type a b c h.
                           (a -> b -> c) ->
                           (a, h) bintree -> (b, h) bintree -> (c, h) bintree =
     fun f t1 t2 ->
     match t1, t2 with
     | Leaf, Leaf -> Leaf
     | Node (tl, v, tr), Node (tl', v', tr') ->
        Node (treemap2 f tl tl', f v v', treemap2 f tr tr')
   (* val treemap2 :
    *   ('a -> 'b -> 'c) ->
    *   ('a, 'h) bintree -> ('b, 'h) bintree -> ('c, 'h) bintree = <fun> *)
   let t = treemap2 (fun x y -> x * y) tx tx
   (* val t : (int, (unit * unit) * unit) bintree =
    *   Node (Node (Leaf, 1, Leaf), 9, Node (Leaf, 4, Leaf)) *)
   ```

5. ```
   let rec shrink : type a h.
                           (a -> a -> a) -> (a, h) bintree -> (a, h) sequence =
   ```
   ```

```
  fun f t1 ->
  match t1 with
  | Leaf -> Nil
  | Node (tl, v, tr) -> Cons (v, shrink f (treemap2 f tl tr))
(* val shrink : ('a -> 'a -> 'a) -> ('a, 'h) bintree -> ('a, 'h) sequence = <fun> *)
let ty = Node (Node (Node (Leaf, 4, Leaf), 2, (Node (Leaf, 5, Leaf))),
               1,
               Node (Node (Leaf, 6, Leaf), 3, (Node (Leaf, 7, Leaf))))
(* val ty : (int, ((unit * unit) * unit) * unit) bintree *)
let t = shrink (+) ty
(* val t : (int, ((unit * unit) * unit) * unit) sequence =
 *   Cons (1, Cons (5, Cons (22, Nil))) *)
```

6. ```
   let collect_levels : type a h. (a, h) bintree -> (a list, h) sequence =
     fun t -> shrink (fun a b -> a @ b) (treemap (fun x -> [x]) t)
   (* val collect_levels : ('a, 'h) bintree -> ('a list, 'h) sequence = <fun> *)
   let t = collect_levels ty
   (* val t : (int list, ((unit * unit) * unit) * unit) sequence =
    *   Cons ([1], Cons ([2; 3], Cons ([4; 6; 5; 7], Nil))) *)
   ```
   ```