Université Paris Diderot — Programmation Fonctionnelle Avancée
M1 Informatique — Année 2013-2014

TP n°3 - Correction

Combinateurs

Exercice 1 En utilisant uniquement les combinateurs map, `List.fold_left` et `List.fold_right`, écrivez les fonctions suivantes :

- somme des éléments d'une liste d'entiers
- `List.filter`
- `List.length`
- Compter le nombre d'occurences d'un élément dans une liste

Correction :

```
let comp f g = fun x -> f (g x);;
let ( |> ) = comp;;

let lsum l = List.fold_left (fun x y -> x+y) 0 l;;

(* non tail-recursive *)
let lfilter p l =
  List.fold_right (fun x res -> if p x then x::res else res) l [] ;;

(* tail-recursive, two traversals to preserve order *)
let lfilter p l =
  List.fold_left (fun res x -> if p x then x::res else res) [] l |> List.rev;;

(* two traversals *)
let llength l = (lsum |> (List.map (fun _ -> 1))) l;;
(* one traversal *)
let lfilter l = List.fold_left (fun len _ -> len+1) 0 l;;

let lcount x l = (llength |> (lfilter (fun z -> z=x))) l;;
```

Exercice 2 Dans cet exercice, on essaie d'écrire un tri fusion en utilisant des combinateurs de destruction et constrution de liste. La structure d'ensemble du tri ne change pas :

```
let rec fusion_sort li = match li with
  | [] | [_] -> li
  | _::_::_ ->
    let l1, l2 = split li in
    merge (fusion_sort l1) (fusion_sort l2)
```

1. Définir (sans récursion, avec un `fold`) une fonction `split : 'a list -> 'a list * 'a list` répartissant les éléments d'une liste en deux listes de taille proches. Indice : on n'est pas obligé de mettre à gauche tous les éléments du début de la liste, et à droite tous ceux de la fin.

2. Définir (avec recursion) le combinateur classique `unfold : ('a -> ('b * 'a) option) -> 'a -> 'b list`
3. Définir (sans récursion) `merge` en utilisant `unfold`.

Correction :

```
let split li =
  let split_and_swap (left, right) x = (right, x::left) in
  List.fold_left split_and_swap ([], []) li

(* version simple *)
let rec unfold step state = match step state with
  | None -> []
  | Some (elem, new_state) -> elem :: unfold step new_state

(* version tail-rec *)
let unfold step state =
  let rec unfold state acc = match step state with
    | None -> List.rev acc
    | Some (elem, new_state) -> unfold new_state (elem::acc)
  in unfold state []

(* we first write a recursive version of 'merge' to understand
   the problem; this isn't the expected answer *)

let rec merge xs ys = match xs, ys with
  | [], [] -> []
  | x::xs, [] -> x :: merge xs ys

    (* remark: in the case above we could return x::xs directly,
       but producing one element at a time will make using 'unfold'
       easier *)

    (* warning: this pattern has given the already-used name 'xs' to
       the tail of the left list, hiding the previous 'xs' designating
       the whole left list; on the contrary, 'ys' still denotes the
       whole right input list, exactly as we need to call
       'merge xs ys'.

       All cases of this pattern-matching use this kind of tricks. It
       may not be the most readable to beginners, but it's good that
       you encounter this style which is used by some programmers. *)

  | [], y::ys -> y :: merge xs ys
  | x::xs, y::_ when x < y -> x :: merge xs ys
  | x::_, y::ys -> y :: merge xs ys

(* version using unfold, without direct recursion *)
let merge xs ys =
  let step (xs, ys) = match xs, ys with
    | [], [] -> None
    | x::xs, [] -> Some (x, (xs, ys))
    | [], y::ys -> Some (y, (xs, ys))
    | x::xs, y::_ when x < y -> Some (x, (xs, ys))
    | x::_, y::ys -> Some (y, (xs, ys))
```

```
  in unfold step (xs, ys)

let rec fusion_sort li = match li with
  | [] | [_] -> li
  | _::_::_ ->
    let l1, l2 = split li in
    merge (fusion_sort l1) (fusion_sort l2)

let test = fusion_sort [5;2;3;6;1;7]
(* [1; 2; 3; 5; 6; 7] *)
```

Exercice 3 Le but de cet exercice est de réfléchir à l'efficacité des programmes construits comme des séquences de transformations simples, tels que

```
let filter_count p data =
  filter p data |> map (fun _ -> 1) |> fold_left (+) 0
```

par rapport à une version écrite en une passe, telle que

```
let filter_count_inline p data =
  fold_left (fun count x -> count + if p x then 1 else 0) 0 data
```

On définit le type de séquences

```
type 'a seq = unit -> 'a option
```

Une séquence est une fonction qui renvoie un nouveau élément à chaque fois qu'on l'appelle (il y a un effet de bord qui modifie son état en interne), et qui finit par renvoyer `None` quand il n'y a plus d'élément à fournir.

1. Écrire une fonction `of_list : 'a list -> 'a seq`.
2. Écrire une fonction `map : ('a -> 'b) -> 'a seq -> 'b seq`.
3. Écrire une fonction `filter : ('a -> bool) -> 'a seq -> 'a seq`.
4. Écrire une fonction `fold : ('a -> 'b -> 'b) -> 'a seq -> 'b -> 'b`.
5. Quel est le coût en mémoire de `filter_count` quand la fonction est définie comme utilisant des listes ? Le coût de `filter_count_inline`. Quels sont ces coûts si on utilise plutôt les fonctions sur les séquences précédemment définies ?
6. *Optionnel (car difficile) :* redéfinir les fonctions sur les séquences en utilisant le type suivant, qui permet de ne pas utiliser d'effets de bord.

   ```
   type 'a seq = Seq : 'st * ('st -> ('a * 'st) option) -> 'a seq
   ```

   (On pourra méditer aux troublantes ressemblances avec le type de `unfold` à l'exercice précédent)

Correction :

```
type 'a seq = unit -> 'a option

let of_list li =
  let state = ref li in
  fun () -> match !state with
    | [] -> None
    | x::xs -> state := xs; Some x
```

```
let map f seq = fun () -> f (seq ())

let filter p seq =
  let rec find_next () =
    match seq () with
    | None -> None
    | Some v -> if p v then Some v else find_next ()
  in fun () -> find_next ()

let rec fold f seq acc = match seq () with
  | None -> acc
  | Some x -> fold f seq (f x acc)



(* question difficile *)
type 'a seq = Seq : 'b * ('b -> ('a * 'b) option) -> 'a seq

let of_list li = Seq (li, function
                          | [] -> None
                          | x::xs -> Some (x, xs))

let rec map f (Seq (state, next)) =
  Seq (state, fun st -> match next st with
                        | None -> None
                        | Some (x, st) -> Some (f x, st))

let rec filter p (Seq (state, next)) =
  let rec find_next st =
    match next st with
    | None -> None
    | Some (v, st) -> if p v then Some (v, st) else find_next st
  in Seq (state, find_next)

let fold f (Seq (state, next)) acc =
  let rec fold state acc = match next state with
    | None -> acc
    | Some (v, state) -> fold state (f v acc)
  in fold state acc
```

Exercice 4 On rappelle qu'une fonction h est un homomorphisme de liste s'il existe un opérateur associatif $\oplus$ avec élément neutre e tel que :

$$h\ [] = e$$

$$h(l1@l2) = (h\ l1) \oplus (h\ l2)$$

Démontrez le théorème suivant :

Théorème 1 *Une fonction h est un homomorphisme de liste ssi il un existe un couple de fonctions f et g tel que h peut s'écrire :*

$$h = (reduce\ g) \circ (map\ f)$$

On rappelle les différentes égalités que l'on a à notre disposition :

— $(reduce\ g)\ [] = e$

— $(reduce\ g)\ [a] = a$

— $(reduce\ g)\ (l1@l2) = g\ (reduce\ g\ l1)\ (reduce\ g\ l2)$ pour des listes non vides.

Correction : Le sens plus facile est de vérifier que les fonctions de la forme $h = (reduce\ g) \circ (map\ f)$ sont bien des homomorphismes – en définissant $\oplus$ comme étant précisément g, d'élément neutre e.

$$\begin{array}{ll} & ((reduce\ g) \circ (map\ f))[] \\ = & reduce\ g\ (map\ f\ []) \\ = & reduce\ g\ [] \\ = & e \end{array}$$

$$\begin{array}{ll} & ((reduce\ g) \circ (map\ f))(l1@l2) \\ = & reduce\ g\ (map\ f\ (l1@l2)) \\ = & reduce\ g\ (map\ f\ l1@map\ f\ l2) \\ = & g\ (reduce\ g\ (map\ f\ l1))\ (reduce\ g\ (map\ f\ l2)) \\ = & (reduce\ g \circ map\ f)\, l1 \oplus (reduce\ g \circ map\ f)\, l2 \end{array}$$

L'autre sens est plus délicat : à partir d'un homomorphisme $(h, \oplus, e)$, il faut deviner les fonctions f et g qui conviennent. En s'inspirant du sens précédent, on peut prendre $\oplus$ pour g. Il reste alors à définir

$$f(x) := h\ [x]$$

On peut alors prouver par induction sur les séquences que pour toute séquence l, $((reduce \oplus) \circ (map\ f))\ l$ est bien égal à $h\ l$.

— Si l est vide, on a bien

$$\begin{array}{ll} & ((reduce \oplus) \circ (map\ f))\ [] \\ = & reduce \oplus\ (map\ f\ []) \\ = & reduce \oplus\ [] \\ = & e \\ = & h\ [] \end{array}$$

— Si l est de la forme $x{::}xs$, ou encore $[x]@xs$, on a

$$\begin{array}{ll} & ((reduce \oplus) \circ (map\ f))\ ([x]@xs) \\ = & reduce \oplus\ (map\ f\ ([x]@xs)) \\ = & reduce \oplus\ ([f\ x]@map\ f\ xs) \\ = & (reduce \oplus [f\ x]) \oplus (reduce \oplus\ (map\ f\ xs)) \\ = & (f\ x) \oplus (reduce \oplus \circ map\ f)\ xs \end{array}$$

On a alors $f\ x$ égal à $h\ [x]$ par définition de f, et comme xs est une liste strictement plus petite que l on peut utiliser l'hypothèse d'induction, c'est-à-dire que $(reduce \oplus \circ map\ f)xs$ est égal à $h\ xs$. On a donc

$$\begin{array}{ll} & ((reduce \oplus) \circ (map\ f))\ (x{::}xs) \\ = & \ldots \\ = & (f\ x) \oplus (reduce \oplus \circ map\ f)\ xs \\ = & h\ [x] \oplus h\ xs \\ = & h([x]@xs) \\ = & h\ (x{::}xs) \end{array}$$