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

TP n°3

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

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`.

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)

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.