Université de Paris, UFR Informatique — Programmation Fonctionnelle Avancée
M1 Info — Année 2020-2021

Examen session 1

Lundi, 10 Mai 2021

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints et rangés. Le temps à disposition est de 2.5 heures.

Les exercices doivent être rédigés en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prédéfinies (sauf indication contraire), et/ou les fonctions des questions précédentes.

Cet énoncé a 4 pages.

Exercice 1 Dans cet exercice, on cherche à représenter des *séquences* d'éléments, comme lors des TP sur les `blist` et `qblist`. Cette fois-ci, outre l'accès indexé (`nth`) et l'ajout d'un élément à gauche de la séquence (`cons`), on souhaite également que l'ajout d'un élément à droite de la séquence (`snoc`) soit possible en temps logarithmique.

On considère les arbres binaires suivants, et la fonction `size` comptant les éléments :

```
type 'a tree =
 | Leaf
 | Node of 'a * 'a tree * 'a tree

let rec size = function Leaf->0 | Node(x,g,d)->1+size g+size d
```

Une séquence va ici être encodée par un `'a tree` de la façon suivante :

- La séquence vide est encodée par `Leaf`
- Une séquence non-vide $a_0, a_1, ..., a_n$ est encodée par `Node(`a_0`,g,d)` avec l'élément de tête a_0 à la racine, et où récursivement le sous-arbre gauche `g` encode les éléments en position impaire $a_1, a_3, ...$ et le sous-arbre droit `d` encode les éléments en position paire non-nulle $a_2, a_4, ...$

Par exemple, la séquence $0, 1, 2, 3, 4, 5$ est encodée par l'arbre suivant :

```
let seq5 = Node(0, Node(1, Node(3,Leaf,Leaf), Node(5,Leaf,Leaf)),
                   Node(2, Node(4,Leaf,Leaf), Leaf))
```

On notera en particulier que dans un arbre encodant une séquence, tout noeud `Node(x,g,d)` satisfait `size g = size d` ou `size g = 1 + size d`. Cet invariant assure en particulier un bon équilibrage de l'arbre, en particulier la hauteur de l'arbre est un logarithme du nombre d'éléments. Cet invariant sera supposé valide pour les fonctions suivantes recevant des arbres en arguments.

1. Écrire une fonction `nth : 'a tree -> int -> 'a` telle que `(nth t i)`=a_i lorsque `t` encode la séquence $a_0, a_1, ..., a_n$. Par exemple `nth seq5 4 = 4`. Cette fonction lèvera l'exception de votre choix si `i` n'est pas une position valide dans la séquence. Votre code devra être linéaire en la hauteur de l'arbre `t`.

2. Écrire une fonction `split : 'a list -> 'a list * 'a list` répartissant alternativement les éléments de la liste d'entrée dans les deux listes de sorties. Par exemple `split [0;1;2] = [0;2],[1]`. En déduire une fonction `of_list : 'a list -> 'a tree` convertissant une liste d'éléments en l'arbre codant la séquence correspondante. Par exemple `of_list [0;1;2;3;4;5] = seq5`.
3. Écrire une fonction `mix : 'a list -> 'a list -> 'a list` qui entremêle les deux listes d'entrée de façon réciproque à `split`. Par exemple `mix [0;2] [1] = [0;1;2]`. En déduire une fonction `to_list : 'a tree -> 'a list` produisant la liste correspondant à la séquence codée par un arbre. Par exemple `to_list seq5 = [0;1;2;3;4;5]`.
4. Écrire une fonction `cons : 'a -> 'a tree -> 'a tree` réalisant l'ajout d'un élément à gauche dans la séquence codée par l'arbre d'entrée. En particulier, on a `(cons x t) = of_list (x :: to_list t)` mais votre réponse devra procéder directement sur l'arbre, sans détour par des listes, et devra être linéaire en la hauteur de l'arbre `t`.
5. Écrire une fonction `snoc : 'a tree -> 'a -> 'a tree` réalisant l'ajout d'un élément à droite dans la séquence codée par l'arbre d'entrée. En particulier, on a `(snoc t x) = of_list (to_list t @ [x])` mais votre réponse devra procéder directement sur l'arbre, sans détour par des listes. Vous pouvez ici utiliser librement la fonction fournie `size`. En négligeant pour l'instant le coût des appels à `size`, votre code devra être linéaire en la hauteur de l'arbre `t`.
6. Indiquer en quelques lignes comment adapter la représentation des séquences pour qu'on ait accès en temps constant à la longueur d'une séquence, puis comment ajuster alors la fonction `snoc` pour que sa complexité totale (toutes sous-fonctions incluses) soit logarithmique en le nombre d'éléments de la séquence.

Exercice 2 Les requêtes suivantes sont exécutées une après l'autre. Dire pour chaque définition (let) si elle est bien typée. Si vous pensez que non donner une explication brève (quelque mots), si vous pensez que oui donner le type inféré par OCaml.

1. ```
   let x1 = [ 0; [1;2]; [[1;2];[3;4]] ]
   ```
2. ```
   let x2 = [ `I 0; `L [1;2]; `LL [[1;2];[3;4]] ]
   ```
3. ```
   let emballer x = match x with
     | `I y -> `L [y]
     | `L y -> `LL [y]
   ```
4. ```
   let deuxfois x = emballer (emballer x)
   ```
5. ```
   let deballer x = match x with
     | `LL y -> `L (List.hd y)
     | `L y -> `I (List.hd y)
   ```
6. ```
   let inout x = deballer (emballer x)
   ```
7. ```
   type ill = [ `I of int | `L of int list | `LL of int list list ]
   let mix1 (x:ill) = match x with
     | `I _ -> emballer x
     | `L _ -> emballer x
     | `LL _ -> deballer x
   ```
8. ```
   let mix2 (x:ill) = match x with
     | `I y -> emballer (`I y)
     | `L y -> emballer (`L y)
     | `LL y -> deballer (`LL y)
   ```

Exercice 3 Dans cet exercice vous allez implémenter des automates cellulaires unidimensionnelles, c'est-à-dire des automates qui travaillent sur une ligne de valeurs booléennes, au lieu des automates cellulaires habituels qui travaillent sur une grille de valeurs.

Dans un premier temps, la ligne sera infinie d'un seul côté, elle sera représentée par un flot de valeurs booléennes. Soit le code OCaml suivant :

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

let stream_head (lazy (Cons(head,rest))) = head
```

1. Quel est le type de la fonction `stream_head` ?
2. Définir un flot `false_stream` dont tous les éléments sont **`false`**, ainsi que le flot `initial_stream` qui commence sur la valeur **`true`**, et dont tous les éléments suivants sont **`false`**.

Un *automate cellulaire* est une liste de triplets de valeurs booléennes. Par exemple,

```
let auto2 = [
    (false,false,true);
    (true,false,false);
  ]
```

Si s est un flot et n un entier naturel on note s_n le n-ème élément de s. L'élément s_{n-1} est le *voisin gauche* de s_n, et s_{n+1} est le *voisin droit* de s_n. Si s est un flot de booléens, son *successeur* s', par rapport à un automate cellulaire a, est définie comme suit : pour tout n, s'_n est `true` si l'automate a contient le triplet (l, s_n, r) où l est le voisin gauche de s_n et r est le voisin droit de s_n, sinon s'_n est `false`. Puisqu'il n'y a pas d'élément s_{-1} on fournit aussi une valeur par défaut qui joue le rôle du voisin gauche du premier élément.

Par exemple, soit s le flot `initial_stream` défini précédemment :

`true false false false false false` ...

Ici, comme dans la suite, nous notons les flots comme une séquence infinie d'éléments séparés par des espaces. Le successeur de ce flot par rapport à l'automate `auto2` et la valeur de défaut **`false`** est

`false true false false false false` ...

Le n-ème successeur de s est obtenu en calculant n fois le successeur de s. Par exemple on obtient les successeurs suivant de s :

n	n-ème successeur
0	`true false false false false false` ...
1	`false true false false false false` ...
2	`true false true false false false` ...

3. Écrire une fonction `successor_stream` qui prend en arguments une valeur v par défaut, un automate cellulaire a, et un flot s, et qui retourne le flot successeur de s par rapport à a et v.

4. Écrire une fonction `generation_stream` qui prend en arguments un entier n et un automate cellulaire a, et renvoie le n-ème successeur du stream `initial_stream` par rapport à l'automate a et la valeur par défaut **`false`**.

On passe maintenant aux lignes infinies des deux côtés. Une telle ligne a une position d'origine. À partir de cette origine, il y a une infinité d'éléments à gauche et une infinité d'éléments à droite. On représente une ligne par un triplet consistant en le flot des éléments à gauche de l'origine, la valeur à l'origine, et le flot des éléments à droite de l'origine. Le flot à gauche contient donc les éléments de la ligne lus à partir de l'origine de droite à gauche. Par exemple, la ligne suivante, où l'origine est soulignée :

... `false true true false false` <u>`true`</u> `true false false` ...

est représentée par ce triplet :

— `false false true true false` ...
— `true`
— `true false false` ...

5. Définir un flot `initial_line` qui a **`true`** à son origine, et dont tous les autres éléments sont **`false`**.

Pour une ligne (l, o, r), on calcule le successeur par application d'un automate cellulaire comme dans le cas des flots, à ceci près que :

— le voisin gauche de l'origine est l_0 et son voisin droit est r_0 ;
— les voisins dans r sont calculé comme avant, et le voisin gauche de r_0 est l'origine ;
— le voisin gauche de l_n est l_{n+1} et son voisin droit est l_{n-1} quand $n > 0$, et le voisin droit de l_0 est l'origine.

On peut donc calculer le successeur d'une ligne en calculant le successeur du flot droit comme auparavant, et en calculant le successeur du flot gauche en faisant attention à inverser l'ordre des éléments. Le cas de l'origine est à traiter à part.

Indication : Pour calculer le successeur du flot des éléments à gauche de l'origine on peut utiliser la fonction `successor_stream` de la Question 3 avec un automate modifié.

6. Définir une fonction `successor_line` qui prend en arguments un automate cellulaire a et une ligne l, et qui retourne la ligne successeur de l par rapport à et a.