Programmation fonctionnelle

Examen du 12 janvier 2022
L2/L3 informatique – Université Côte d'Azur

Durée: 2 heures.

- Aucun document ni aucune machine ne sont autorisés. Un mémo est donné en fin de sujet.
- Les téléphones doivent être rangés.
- Les réponses sont à reporter sur les feuilles de réponse en fin de sujet. Ces feuilles sont à dégrafer du sujet et à rendre en fin d'épreuve sans agrafe. Vous pouvez déborder du cadre de réponses si besoin, sans écrire sur les cases réservées à la correction en haut du cadre.
- Les réponses doivent être écrites lisiblement, en respectant autant que possible l'interligne proposé. Le correcteur blanc et l'effaceur sont autorisés.
- Les questions sont indépendantes: une question peut être sautée, et une fonction `f` demandée à une question peut être utilisée pour définir une fonction `g` dans une question ultérieure même si la solution pour `f` n'a pas été trouvée.
- Le barème est une estimation basse, il pourra être revu à la hausse (note > 20 possible dans ce cas). Le nombre de points d'une question est proportionnel à l'estimation de la difficulté de la question.

Filter Map (8 points)

On s'intéresse à différentes formes de l'opération conjointe filter-map, qui sélectionne une partie du contenu d'un conteneur et renvoie un nouveau conteneur avec ce contenu transformé. Prenons d'abord un exemple avec une liste. On considère la fonction

```
list_filter_map :  ('a -> 'b option) -> 'a list -> 'b list
```

définie comme suit

```
let rec list_filter_map f l = match l with
| [] -> []
| hd::tl -> begin
      match f hd with
      | Some v -> v ::  list_filter_map f tl
      | None -> list_filter_map f tl
   end
```

Soit `f0` la fonction définie par

```
let f0 x = if x > 0 then Some(float x +.  0.5) else None
```

Alors `list_filter_map f0 [-2;0;2;-1;3]` renvoie `[2.5; 3.5]`.

Soit `f1` la fonction définie par `let f1 x = if x < 0 then Some(x*x) else None`.

Question 1 (0,5 pts) Que renvoie `list_filter_map f1 [-2;0;2;-1;3]`?

Question 2 (1 pts) Définissez en une ligne la fonction `list_map :  ('a -> 'b) -> 'a list -> 'b list` à partir de la fonction `list_filter_map` (pas de récurrence, pas droit à d'autres fonctions).

Indication: `let list_map f l = list_filter_map (fun x -> ...)  l`.

Question 3 (1,5 pts) Définissez en une ligne la fonction `list_filter :  ('a -> bool) -> 'a list -> 'a list` à partir de la fonction `list_filter_map` (pas de récurrence, pas droit à d'autres fonctions).

On considère le type suivant d'arbre binaire

```
type 'a bintree =
| Feuille of 'a
| Arbre of 'a bintree * 'a bintree
```

Question 4 (2 pts) Définissez une fonction `bintree_filter_map : ('a -> 'b option) -> 'a bintree -> 'b bintree option` analogue à `list_filter_map`, mais sur les arbres binaires: toutes les feuilles envoyées vers None par la fonction sont effacées, les autres sont remplacées par leur image par f (sans le Some).
Par exemple, si `a` est l'arbre `Arbre(Feuille 0, Feuille 1)`, `bintree_filter_map f0 a` renvoie `Some (Feuille(1.5))`, et `bintree_filter_map f1 a` renvoie `None`.

Question 5 (3 pts) Définissez une fonction `array_filter_map : ('a -> 'b option) -> 'a array -> 'b array` analogue à `list_filter_map`, mais sur les tableaux. Respectez la contrainte ci-dessous.

Contrainte: votre fonction ne doit manipuler que des tableaux: il est interdit d'utiliser des listes, la fonction `list_filter_map`, les fonctions `Array.of_list` et `Array.to_list`, etc. Vous n'avez pas non plus le droit de concaténer des tableaux avec `Array.concat`.

Indication: vous devez calculer d'abord la longueur du tableau résultat, puis l'allouer, et enfin le remplir. Vous pourrez par exemple écrire une fonction auxiliaire `count_some : 'a option array -> int` qui compte le nombre de cases d'un tableau d'options qui contiennent une valeur de la forme `Some x`. Vous pourrez aussi utiliser certaines fonctions du module Array de la librairie standard (entre autre `Array.map` et `Array.iter`).

Opérateurs binaires paresseux et réduction paresseuse (8 points)

Question 6 (1 pts) Qu'affiche le code ci-dessous?

```
let et a b = a && b
let _ = (print_string "a"; false) && (print_string "b"; false)
let _ = et (print_string "c"; false) (print_string "d"; false)
```

On se donne une très longue liste l de booléens tous égaux à false

```
let long_list = List.init 100_000_000 (fun _ -> false)
```

On observe que si on fait `List.fold_left (&&) true long_list` le calcul dure longtemps, alors que la réponse (false) est connue dès le premier élément de la liste. Cela est dû au fait que `(&&)` (autrement dit la fonction `et` ci-dessus) est une fonction comme une autre et non l'opérateur binaire paresseux `&&`, et que fold_left fait la réduction en applicant cette fonction à tous les éléments de la liste sans s'intéresser aux résultats intermédiaires qui permettraient de conclure et s'arrêter plus tôt.

Question 7 (2 pts) Définissez une fonction `andlist: bool list -> bool` qui prend en argument une liste de booléens et qui renvoie `true` si tous les booléens valent `true`, false sinon. Votre solution devra être plus efficace que la solution `let andlist l = List.fold_left (&&) true l` mentionnée ci-dessus.

Quand on fait un reduce (un fold_left) avec un opérateur binaire paresseux, il est donc possible de donner le résultat sans avoir à parcourir toute la liste. Par exemple, si on veut faire le produit d'une liste, on peut répondre 0 dès qu'on rencontre un 0 dans la liste. Dans la suite, on va donner une définition de ce qu'est un opérateur binaire paresseux et l'utiliser pour faire un reduce paresseux.

L'application partielle de `et` à `false`, autrement dit `(&&) false` ou `et false`, est une fonction qui renvoie toujours `false`, puisque `(&&) false x= false`. Il serait donc intéressant de ne pas représenter le résultat de cette application partielle par une fonction, mais par une constante. D'un autre côté, `(&&) true` est la fonction identité, puisque `(&&) true` x = x. Il faut donc parfois représenter l'application partielle d'un opérateur binaire paresseux par une fonction.

Afin de représenter le résultat de l'application partielle d'un opérateur binaire paresseux "abstrait", on se donne donc le type suivant

```
type 'a binop_partial_application =
| Result of 'a
| Transformation of ('a -> 'a)
```

On peut alors représenter un opérateur binaire paresseux par une fonction à un seul argument qui renvoie une application partielle.

```
type 'a lazy_binop = 'a -> 'a binop_partial_application
```

Par exemple, on peut représenter le résultat de l'opérateur binaire paresseux "et booleen" comme suit.

```
let lazy_and b = match b with
| false -> Result false
| true -> Transformation (fun b2 -> b2)
```

Question 8 (1 pts) Définissez de même l'opérateur binaire paresseux `lazy_product : int lazy_binop`. On notera que $0 \times m$ renvoie toujours 0, quel que soit m, donc l'application partielle $0 \times \ldots$ sera représentée par la constante 0.

Question 9 (1,5 pts) Définissez la fonction `fun_of_lazy_binop: 'a lazy_binop -> 'a -> 'a -> 'a` qui transforme un opérateur binaire paresseux en une fonction. Par exemple `fun_of_lazy_binop lazy_and` renvoie la fonction booléenne `(&&)`.

Question 10 (2,5 pts) Définissez la fonction `lazy_reduce: 'a lazy_binop -> 'a -> 'a list -> 'a` qui réduit paresseusement une liste à l'aide d'un opérateur binaire paresseux. Par exemple `lazy_reduce lazy_and true l` sera équivalent (en résultat et en temps de calcul) à `andlist l`. Plus généralement, `lazy_reduce plus zero l` devra renvoyer la "somme" (au sens de `plus` et `zero`) de l, qu'on pourrait calculer de manière moins efficace avec `List.fold_left (fun_of_lazy_binop plus) zero l`.

Matrices persistantes (4 points)

On se donne la signature suivante pour des matrices de flottants persistantes.

```
module type PERSISTANT_MATRIX = sig
   type t
   val init:  int -> int -> (int -> int -> float) -> t
   val dim:  t -> (int * int)
   val get:  t -> int -> int -> float
   val set:  t -> int -> int -> float -> t
end
```

La fonction `init n m f` renvoie une nouvelle matrice de n lignes et m colonnes dont la case d'indice (i,j) contient `f i j`. La fonction `dim mat` renvoie (n, m) si la matrice `mat` contient n lignes et m colonnes. La fonction `get mat i j` renvoie le contenu de la case d'indice (i,j). La fonction `set mat i j x` renvoie une nouvelle matrice dont le contenu est celui obtenu en remplaçant le contenu de la case d'indice (i,j) de mat par x. Attention, cette dernière fonction ne modifie pas `mat` (persistance) mais renvoie une nouvelle matrice.

Question 11 (2 pts) On se propose d'implémenter les matrices par des tableaux de tableaux. Complétez le module ci-dessous.

```
module ArrayBasedMatrix:  PERSISTANT_MATRIX = struct
   type t = float array array
   let init n m f = ...
end
```

On s'intéresse maintenant à une représentation adaptée aux "matrices creuses" (contenant beaucoup de zéros). On représente une matrice par la liste des cases qui ne contiennent pas zéro. Toutes les cases non listées contiennent donc implicitement 0. Par exemple, la matrice pleine de zéros est représentée par la liste vide `[]`. Chaque case est représentée par un triplet `(i, j, v)`, où (i,j) désigne les coordonnées de la case et v la valeur de cette case. Par exemple, la matrice $\begin{pmatrix} -1 & 0 \\ 0 & 2 \end{pmatrix}$ est représentée par la liste `[(0,0,-1.); (1,1,2.)]`.

Question 12 (2 pts) Complétez le module ci-dessous.

```
module SparseMatrix:  PERSISTANT_MATRIX = struct
   type t = {
     width:  int;
     height:  int;
     non_null_cells:  (int * int * float) list
   }
   let init n m f = ...
end
```

Memo sur les listes

```
utop # List.filter;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
utop # List.filter (fun x -> x > 0) [-1;0;1;2;-3;4];;
- : int list = [1; 2; 4]
utop # List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
utop # List.map (fun x -> x*x) [-1;0;1;2;-3;4];;
- : int list = [1; 0; 1; 4; 9; 16]
```

Memo sur les arbres

```
utop # let rec hauteur arbre = match arbre with
| Feuille _ -> 0
| Arbre(filsg, filsd) -> 1 + max (hauteur filsg) (hauteur filsd);;
val hauteur : 'a bintree -> int = <fun>
```

Memo sur les tableaux et les boucles for

```
utop # Array.make;;
- : int -> 'a -> 'a array = <fun>
utop # Array.make 10 'a';;
- : char array = [|'a'; 'a'; 'a'; 'a'; 'a'; 'a'; 'a'; 'a'; 'a'; 'a'|]
utop # Array.init;;
- : int -> (int -> 'a) -> 'a array = <fun>
utop # let arr = Array.init 10 (fun i -> 9-i);;
val arr : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
utop # for i=0 to 9 do arr.(i)<-i done;;
- : unit = ()
utop # arr;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

Memo sur les fonctions d'ordre supérieur d'itération dans les tableaux

```
utop # let acc = ref 0;;
val acc : int ref = {contents = 0}
utop # Array.iter (fun v -> acc := !acc + v) arr;;
- : unit = ()
utop # !acc;;
- : int = 45
utop # let arr2 = Array.copy arr;;
val arr2 : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]

utop # Array.iteri;;
- : (int -> 'a -> unit) -> 'a array -> unit = <fun>
utop # Array.iteri (fun i v -> arr2.(9-i) <- v) arr;;
- : unit = ()
utop # arr2;;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
utop # arr;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```