Devoir d'Informatique n°4

30 mars 2024

*
* *

Durée de l'épreuve : 3 heures et 30 minutes

L'usage de tout dispositif électronique est interdit.

Consignes

Pour répondre à une question, il vous est permis de réutiliser le résultat d'une question antérieure même si vous n'avez pas réussi à établir ce résultat. En particulier, vous pouvez réutiliser les fonctions introduites dans une question dans les questions suivantes.

Vous devrez traiter les questions de programmation dans le langage OCaml. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier la correction ou la terminaison de cette fonction, ou de la commenter. Cependant, il est conseillé d'expliquer l'intention de votre code, surtout si celui-ci est long.

Vous attacherez la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si vous repérez ce qu'il vous semble être une erreur d'énoncé, signalez-la sur votre copie et poursuivez la composition en expliquant vos éventuelles prises d'initiative.

Le sujet comporte trois parties indépendantes, pour un total de 29 points. La première partie, sur 6 points, est constituée de questions portant sur la programmation OCaml et sur les fonctions récursives. La deuxième partie, sur 14 points, porte sur l'étude d'un type d'arbre binaire non-équilibré mais permettant tout de même des opérations efficaces en moyenne. La troisième partie porte sur l'étude d'un système de calcul fonctionnel, et de la représentation dans ce système des entiers et autres types de base.

Exercices (6 points)

Q1. Pour chaque type suivant, écrire la définition d'une expression OCaml de ce type :

a) `('a -> (int * 'a))list`

b) `'a option -> ('a -> 'b)-> 'b option`

c) `'a -> 'b -> ('c -> 'a)`

d) `('a list -> 'b)-> 'b`

Q2. Donnez la valeur des expressions suivantes :

a)
```
let x = 3 in
let f =
  let a = 2 in
  fun y -> a*y + x
in (fun t -> let x = 0 in f 11) 195
```

b)
```
let rec f l a = match l with
  | (a, b)::q -> b
  | (x, b)::q -> f q a
  | [] -> 0
in f [("x", 1); ("y", 2)] "y"
```

Q3. Donner le type des fonctions suivantes :

a)
```
let g x y z =
  if y x then z ([] :: x)
  else 0
```

b)
```
let h a b =
  match a b with
  | ([], x) -> x :: b
  | (x :: q, y :: p) -> ((x-y) :: q) :: b
  | _ -> failwith "erreur"
```

Q4. On définit les fonctions suivantes sur les listes :

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: q -> x :: append q l2

let rec map f l =
  match l with
  | [] -> []
  | x :: q -> f x :: map f q
```

Montrer par induction structurelle sur l_1 que pour toutes listes l_1, l_2 et pour toute fonction f, on a :

`map f (append l1 l2)= append (map f l1)(map f l2)`

Q5. Écrire une fonction récursive terminale `rev` permettant de renverser l'ordre d'une liste. Donner sa complexité.

Q6. Écrire une fonction `rev_map` récursive terminale telle que `rev_map f l = rev (map f l)` pour toute fonction f et pour toute liste l, en montrant soigneusement que c'est bien le cas.

Arbres croissants (14 points)

Adapté du sujet d'informatique X-ENS 2014

Dans ce problème, on considère des arbres binaires, qui sont définis inductivement comme suit :

— L'arbre vide, noté V, est un arbre binaire

— Pour g, d deux arbres binaires et $x \in \mathbb{N}$, $N(g, x, d)$ est un arbre binaire.

La taille d'un arbre t, notée $|t|$, est définie inductivement de la manière suivante :

— $|V| = 1$

— Pour g, d deux arbres binaires et $x \in \mathbb{N}$, $|N(g, x, d)| = 1 + |g| + |d|$

La hauteur d'un arbre t, notée $h(t)$, est définie inductivement de la manière suivante :

— $h(V) = 0$ (*attention : un arbre vide est de hauteur 0 et pas -1)*

— $h(N(g, x, d)) = 1 + \max(h(g), h(d))$

Q1. Définir une fonction inductive **occ** telle que pour un arbre t et un entier y, $\mathbf{occ}(y, t)$ donne le nombre d'occurrences de y dans t.

L'ensemble des éléments d'un arbre t est l'ensemble des éléments $y \in \mathbb{N}$ pour lesquels $\mathbf{occ}(y, t) > 0$.

On se donne le type `arbre` OCaml suivant :

```
1 type arbre = V | N of arbre * int * arbre
```

Structure d'arbre croissant

On dit qu'un arbre t est un **arbre croissant** si $t = V$ ou bien si $t = N(g, x, d)$ et g et d sont croissants et x est strictement inférieur à tous les éléments de g et de d.

Q2. Dessiner un arbre croissant de hauteur au plus 3 contenant les éléments de l'intervalle $[\![1, 7]\!]$.

Q3. Écrire une fonction `minimum: arbre -> int option` renvoyant le plus petit élément d'un arbre croissant sous forme d'une option : `None` si l'arbre est vide et `Some y` si `y` est le plus petit élément.

Q4. Écrire une fonction `est_croissant: arbre -> bool` renvoyant un booléen indiquant si un arbre binaire a la structure d'arbre croissant. On garantira une complexité en $\mathcal{O}(|t|)$.

Q5. Montrer qu'il y a exactement $n!$ arbres croissants à n nœuds étiquetés par $1, 2, \ldots, n$ (chaque nœud étant étiqueté par un entier distinct).

Opérations sur les arbres croissants

On définit la fonction **fusion** inductivement comme suit :

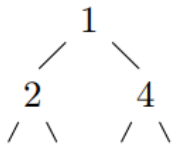
— $\mathbf{fusion}(V, a) = a$ pour tout arbre croissant a

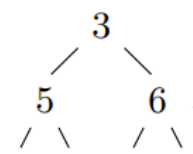
— $\mathbf{fusion}(a, V) = a$ pour tout arbre croissant a

— $\mathbf{fusion}(N(g_1, x_1, d_1), N(g_2, x_2, d_2)) = N(\mathbf{fusion}(d_1, N(g_2, x_2, d_2)), x_1, g_1)$ si $x_1 \leq x_2$

— $\mathbf{fusion}(N(g_1, x_1, d_1), N(g_2, x_2, d_2)) = N(\mathbf{fusion}(d_2, N(g_1, x_1, d_1)), x_2, g_2)$ sinon.

Note importante : dans la troisième (resp. quatrième) ligne de cette définition, on a sciemment échangé les sous-arbres g_1 et d_1 (resp. g_2 et d_2). Les avantages de ce choix apparaîtront dans les parties suivantes du problème.

Q6. Donner le résultat de la fusion des arbres croissants 

et 

.

Q7. Montrer que pour t_1, t_2 deux arbres croissants, $\mathbf{fusion}(t_1, t_2)$ est un arbre croissant, et que pour tout $y \in \mathbb{N}$, $\mathbf{occ}(y, \mathbf{fusion}(t_1, t_2)) = \mathbf{occ}(y, t_1) + \mathbf{occ}(y, t_2)$.

On suppose dans la suite que la fonction `fusion: arbre -> arbre -> arbre` a été implémentée en OCaml.

Q8. Déduire de `fusion` une fonction `ajoute: int -> arbre -> arbre` permettant d'ajouter un élément à un arbre croissant. Autrement dit, pour t un arbre croissant et $y \in \mathbb{N}$, en notant $t' = \mathbf{ajoute}(y, t)$, t' est un arbre croissant tel que $\mathbf{occ}(y, t') = \mathbf{occ}(y, t) + 1$ et $\mathbf{occ}(x, t') = \mathbf{occ}(x, t)$ pour tout $x \neq y$.

Q9. Déduire de `fusion` une fonction `supprime_min: arbre -> arbre` telle que pour t un arbre croissant non vide dont l'élément minimal est m, en notant $t' = \mathbf{supprime\_min}(t)$, t' est un arbre croissant tel que $\mathbf{occ}(m, t') = \mathbf{occ}(m, t) - 1$ et $\mathbf{occ}(x, t') = \mathbf{occ}(x, t)$ pour tout $x \neq m$.

Q10. Soient $x_1, \ldots, x_n$ des entiers. On définit les arbre croissants $(t_i)_{i \in [\![0,n]\!]}$ par :

— $t_0 = V$

— Pour $0 \leq i < n, t_{i+1} = \mathbf{fusion}(t_i, N(V, x_{i+1}, V))$.

Écrire une fonction `ajouts_successifs: int list -> arbre` telle que `ajouts_successifs [xn; ...; x1]` renvoie t_n.

Q11. Avec les mêmes notations, donner, pour tout $n \in \mathbb{N}^*$, des valeurs $x_1, \ldots x_n$ telles que t_n est un arbre croissant de hauteur au moins égale à $\frac{n}{2}$.

On garde les mêmes notations. On étudie la hauteur de t_n dans le cas où pour $i \leq n$, $x_i = i$, i.e. lorsque t_n est construit à partir de la séquence d'entiers $1, 2, \ldots, n$.

Q12. Dessiner $t_1, t_2, t_3, \ldots t_7$.

On dit qu'un arbre binaire A est équilibré à gauche si tout sous-arbre non-vide de A, en notant p son nombre de noeuds, a $\lceil \frac{p-1}{2} \rceil$ noeuds à gauche et $\lfloor \frac{p-1}{2} \rfloor$ noeuds à droite. On se propose de montrer que pour tout $n \in \mathbb{N}$, $\mathcal{P}(n)$: "t_n est équilibré à gauche". On raisonne par récurrence sur $n \in \mathbb{N}$.

Q13. Justifier que la propriété est vraie pour $n = 0$.

Q14. On suppose $\mathcal{P}(n)$ pour un certain $n \in \mathbb{N}$, afin de montrer l'hérédité. Montrer par induction sur les sous-arbres de t_n que pour tout sous-arbre A de t_n, **fusion**$(A, N(V, x_{n+1}, V)$ est équilibré à gauche.

On se propose d'utiliser le résultat précédent afin de montrer que la hauteur des arbres $(t_n)_{n\in\mathbb{N}}$ est logarithmique en leur taille.

Q15. On fixe $n \in \mathbb{N}$. En raisonnant par induction sur les sous-arbres de t_n, montrer que la hauteur de t_n est $\lfloor \log_2(n+1) \rfloor$.

Analyse

On dit qu'un noeud $N(g, x, d)$ est **lourd** si $|g| < |d|$, et qu'il est **léger** sinon. On définit le potentiel d'un arbre t, noté $\Phi(t)$, comme le nombre total de nœuds lourds qu'il contient.

On se propose d'écrire une fonction OCaml `potentiel: arbre -> int` calculant le potentiel d'un arbre en $\mathcal{O}(|t|)$. On propose une première solution :

```
let rec taille t = match t with
  | V -> 1
  | N(g, x, d) -> taille g + taille d + 1

let rec potentiel_0 t = match t with
  | V -> 0
  | N(g, x, d) -> (if taille g < taille d then 1 else 0) + potentiel g + potentiel d
```

Q16. Donner une famille d'arbres où la fonction `potentiel_0` s'exécute en $\Theta(|t|^2)$, en justifiant soigneusement.

Q17. Proposer une solution de calcul du potentiel en temps linéaire en la taille de l'arbre.

On définit le **coût** de la fusion de deux arbres t_1, t_2, noté $C(t_1, t_2)$, comme étant le nombre d'appels récursifs à la fonction **fusion** effectués lors du calcul de **fusion**(t_1, t_2). En particulier, $C(V, t) = C(t, V) = 0$.

Q18. Soient t_1, t_2 deux arbres croissants et $t =$ **fusion**(t_1, t_2). En raisonnant par induction sur (t_1, t_2), Montrer que :

$$C(t_1, t_2) \leq \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log(|t_1|) + \log(|t_2|))$$

Q19. Soient $x_1, \ldots, x_n$ des entiers. On définit les arbre croissants $(t_i)_{i\in[\![0,n]\!]}$ par :

- $t_0 = V$
- Pour $0 \leq i < n, t_{i+1} =$ **fusion**$(t_i, N(V, x_{i+1}, V))$.

Montrer que le coût total des fusions effectuées lors de cette construction de t_n est en $\mathcal{O}(n \log n)$.

Q20. Montrer que dans la construction de la question précédente, une des opérations **fusion** peut avoir un coût au moins égal à $\frac{n}{2}$ (on exhibera des valeurs $x_1, \ldots x_n$ le mettant en évidence). Justifier alors la notion de complexité **amortie** logarithmique pour la fusion de deux arbres croissants.

Q21. Soit t_0 un arbre croissant contenant n nœuds, c'est-à-dire de taille $2n+1$. On construit les n arbres croissants $t_1, \ldots, t_n$ par $t_{i+1} =$ **fusion**(g_i, d_i), où $t_i = N(g_i, x_i, d_i)$. En particulier, on a $t_n = V$. Montrer que le coût total de cette construction est $\mathcal{O}(n \log n)$.

Q22. En utilisant les arbres croissants, écrire une fonction `tri: int list -> int list` permettant de trier une liste d'entiers en $\mathcal{O}(n \log n)$.

Lambda-calcul (9 points)

Dans cette partie, **on interdit l'usage de l'intégralité des éléments d'OCaml**, excepté :

— les applications de fonctions ;

— les définitions non récursives de fonctions.

Tout le reste (addition, soustraction, if-then-else, fonctions récursives, constantes, ...) devient interdit. Dans le sujet, certains blocs de codes contiendront tout de même ces éléments, mais ce sera uniquement pour préciser ou donner des exemples sur le comportement d'une fonction : il vous est **interdit** de les utiliser sur votre copie.

Par ailleurs, tous les types sont aussi interdits, sauf les types flèches polymorphes, c'est à dire ceux de la forme `'a -> 'a`, `'a -> ('a -> 'b)-> 'b`, etc... Les types de base comme `int` et `float`, les types listes, les tuples et les types somme disparaissent intégralement.

Ce sous-ensemble d'OCaml, extrêmement basique, s'appelle le **lambda-calcul**. On appelle **lambda-terme** toute expression de ce système. Voici 4 exemples de définitions de lambda-termes :

```
let id x = x (* type de id: 'a -> 'a *)
let app f x = f x (* type de app: ('a -> 'b) -> 'a -> 'b *)
let id2 = app id id (* type de id2: 'a -> 'a *)
let ech f = fun x y -> f y x (* type de ech: ('a -> 'b -> 'c) -> 'b -> 'a -> 'c *)
```

Bien que ce système puisse sembler faible, il est Turing-complet, c'est à dire aussi puissant que n'importe quel langage de programmation. On s'intéresse à l'étude de ce système, en particulier on souhaite y encoder les booléens, les entiers, les couples et les listes.

Booléens de Church

Définition 1. On dit qu'un lambda-terme b est un **booléen** de Church si l'une des deux conditions suivantes est réalisée :

— Pour tous lambda-termes x, y, `b x y = x`

— Pour tous lambda-termes x, y, `b x y = y`

Dans le premier cas, on dit que b est un booléen de Church vrai, dans le deuxième cas on dit que c'est un booléen de Church faux.

Par exemple, voici deux manières simples d'écrire les deux booléens de Church :

```
let vrai = fun x y -> x
let faux = fun x y -> y
```

On souhaite écrire des fonctions qui simulent sur les booléens de Church les opérations booléennes classiques comme "et", "ou" et "non", ainsi qu'un opérateur if-else.

Q1. Considérons la fonction :

```
let mystere b = fun x y -> b y x
```

Donner la valeur de :

(a) `vrai 3 4` (b) `faux 3 4` (c) `mystere vrai 3 4` (d) `mystere faux 3 4`

Q2. Si `b` est un booléen de Church, qu'est `mystere b` ?

Q3. Écrire une fonction `si` telle que `si b x y` vaut `x` si `b` est un booléen de Church vrai, et `y` si `b` est un booléen de Church faux.

Q4. Écrire deux fonctions `et` et `ou` réalisant les opérations booléennes correspondantes sur les booléens de Church. Par exemple, si `b1` est un booléen de Church vrai et que `b2` est un booléen de Church faux, alors `et b1 b2` est un booléen de Church faux, et `ou b1 b2` est un booléen de Church vrai.

Entiers de Church

Définition 2. On dit qu'un lambda terme `n` est un **entier de Church** s'il existe $N \in \mathbb{N}$ tel que pour tout lambda terme `x` et pour toute fonction `f`, `n f x = f ( f ( ... f (f x)... ))` avec N applications de `f`. On dit alors que `n` **représente** N.

Ainsi, un entier $N \in \mathbb{N}$ sera représenté par la fonction qui prend en entrée f une fonction et renvoie $f^{\circ N}$, i.e. f composée N fois avec elle-même. Voici deux exemples d'entiers de Church :

```
(* entier de Church représentant 3 *)
let trois = fun f x -> f (f (f x));;
assert (trois (fun x -> x + 1) 0 = 3);; (* ajouter 1 trois fois <-> ajouter 3 *)

(* entier de Church représentant 2 *)
let deux f x = f (f x);;
assert (deux (fun x -> x + 1) 4 = 6);;
```

Q5. On suppose que l'on dispose d'un lambda-terme `quatre`, dont on pense qu'il représente 4. En utilisant la fonction `fun x -> x * 2` et `assert`, écrire deux tests permettant de tester `quatre`.

Q6. Recopier et compléter le code suivant permettant de définir un entier de Church représentant 0 :

```
let zero = fun f x -> ...
```

Q7. Recopier et compléter le code suivant permettant de définir une fonction `succ`, telle que si `n` est un entier de Church représentant $N \in \mathbb{N}$, `succ n` est un entier de Church représentant $N + 1$:

```
let succ n =
  fun f x -> ...
```

Q8. Écrire une fonction `plus`, telle que si `n` est un entier de Church représentant $N \in \mathbb{N}$ et `m` un entier de Church représentant $M \in \mathbb{N}$, alors `plus n m` est un entier de Church représentant l'entier $n + m$.

Q9. Définir une fonction `fois` réalisant la multiplication de deux entiers de Church.

Q10. On considère la fonction suivante :

```
let mystere2 n m = m n;;
```

Si `n` représente un entier N et `m` un entier M, déterminer l'entier représenté par `mystere2 n m`, en justifiant votre réponse.

On représente les couples en utilisant le principe des **couples de Church**, comme suit :

```
let couple x y = fun f -> f x y
```

Le terme `fun f -> f x y` représente donc le couple (x, y).

Q11. Si `c` est un couple de Church représentant le couple (x, y), que vaut `c vrai` ? Et `c faux` ?

Q12. Écrire une fonction `echanger` telle que si `c` représente le couple (x, y) alors `echanger c` représente le couple (y, x).

On définit la fonction prédécesseur sur les entiers naturels classiques comme étant :

$$\begin{array}{rccl} & \mathbb{N} & \to & \mathbb{N} \\ \textbf{pred} : & n & \mapsto & \begin{cases} 0 \text{ si } n = 0 \\ n - 1 \text{ sinon} \end{cases} \end{array}$$

Nous allons utiliser les couples pour implémenter le prédécesseur sur les entiers de Church. On considère la fonction suivante :

```
1 let progress f c = couple (f (c vrai)) (c vrai)
```

Q13. Si `c` représente le couple (x, x), et f est une fonction quelconque, quel couple est représenté par `progress f c` ? Et par `progress f (progress f c)` ?

Q14. En déduire une fonction `pred` implémentant le prédécesseur sur les entiers de Church.

Q15. Écrire une fonction `est_zero` telle que si `n` représente un entier N, alors `est_zero n` est un booléen de Church, vrai si et seulement si $N = 0$.

Q16. En déduire une fonction `inf` qui implémente l'opérateur de comparaison $\leq$ sur les entiers de Church, puis une fonction `egal` qui implémente l'opérateur d'égalité $=$ sur les entiers de Church.

Listes de Church

Comme pour les booléens, les entiers et les couples, on souhaite trouver une représentation des listes. Plus précisément, on souhaite construire :

- Un lambda-terme `vide` représentant la liste vide
- Un lambda-terme `cons` tel que `cons x q` représente la liste $x :: q$
- Un lambda-terme `est_vide` testant si une liste est vide. Cette fonction devra renvoyer un booléen de Church.
- Deux lambda-termes `tete` et `queue` renvoyant la tête et la queue d'une liste supposée non-vide.

Q17. (*Question ouverte*) Proposer une représentation des listes en donnant les cinq lambda-termes demandés. Les traces de recherche seront évaluées.