Licence STS mention Informatique, semestre 5 — 2018–2019
Programmation fonctionnelle avancée — Durée : 2h
4 pages

Examen du 7 novembre 2018

Les notes de TD/TP (manuscrites ou tapées sur ordinateur) ainsi que les transparents de cours sont les seuls documents autorisés. Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse.

1 Typage

Question 1 Les fonctions suivantes (`f1`, `f2`, `f3` et `f4`) sont-elles bien typées ? Si oui, donner leur type, sinon préciser pourquoi.

```
let f1 (u, v , w) = if u < w then [v] else []

let rec f2 x y = f2 [y] ()

let rec f3 x = not (f3 (f3 (x+1)))

let rec f4 f x =
  match f x with
  | [] -> [""]
  | z :: s -> f4 f (x - 1)
```

2 Récursion terminale et CPS

Question 2 Donner une version récursive terminale de la fonction `f` ci-dessous.

```
let rec f l =
  match l with
  | [] -> 0
  | (x,y) :: s -> x * y + f s
```

Question 3 Soit le type `t` des arbres binaires et la fonction `add` ci-dessous. Réécrire la fonction `add` en CPS.

```
type t = E | N of int * t * t

let rec add t i =
  match t with
  | E -> E
  | N(x,g,d) -> N(x+i, add g i, add d i)
```

3 Programmation : chaînes de blocs (*blockchain*)

Le but de cette partie est de programmer une (forme simplifiée de) structure de données appelée « chaîne de blocs », ou *blockchain* en anglais. Une telle structure prend la forme d'une liste chaînée dont chaque élément est un bloc. Les blocs contiennent les trois informations suivantes :

- un indice qui donne le numéro du bloc, sachant que les blocs sont numérotés dans l'ordre à partir de 0 ;
- une empreinte numérique du bloc précédent dans la liste, cette empreinte étant calculée à l'aide d'une fonction de hachage (que l'on suppose donnée dans la suite du sujet) ;
- une empreinte numérique d'un ensemble des données associées à ce bloc (la manière dont cette empreinte est calculée est décrite un peu plus loin dans le sujet).

Les types des blocs et des chaînes de blocs sont définis en OCaml à l'aide respectivement des types `block` et `chain` ci-dessous.

```
type block = { index : int; prev_hash : int; merkel_root : int }
type chain = block list
```

Les blocs sont représentés par des enregistrements de type `block`. Le champ `index` contient l'indice du bloc, le champ `prev_hash` l'empreinte du bloc précédent et le champ `merkel_root` contient l'empreinte numérique des données. Une *blockchain* est représentée par une valeur de type `block list`.

Attention. Par convention (et pour simplifier l'ajout d'un nouveau bloc), les blocs seront stockés dans l'ordre *inverse* de leur numérotation, c'est-à-dire qu'une *blockchain* avec k blocs sera représentée par une liste de k blocs, avec le bloc d'indice $k-1$ en tête de liste.

Le premier bloc d'une *blockchain* (appelé *genesis*) est particulier puisqu'il n'a pas de bloc précédent et qu'il n'est associé à aucune donnée. Par conséquent, on fixe son index à 0 et les empreintes (du bloc précédent ou des données) à `-1`.

Question 4 Définir une valeur `genesis` pour représenter le bloc *genesis* d'une *blockchain*.

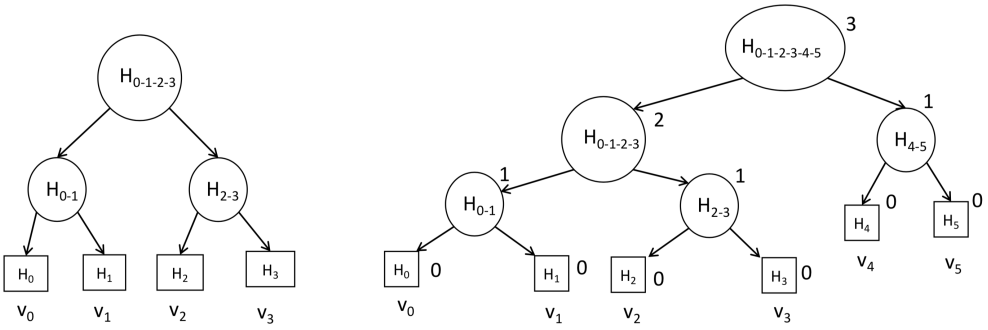
Dans la suite du sujet, on suppose que l'on dispose d'une fonction `hash` de type `'a -> int` pour calculer l'empreinte numérique de n'importe quelle valeur OCaml. En particulier, on peut calculer l'empreinte d'un bloc `b` en faisant `hash b`.

Question 5 Écrire une fonction `add_block`, de type `int -> chain -> chain`, telle que `add_block mr bc` ajoute à la *blockchain* `bc` un nouveau bloc contenant un champ `merkel_root` initialisé avec `mr`.

On dit qu'une *blockchain* est intègre si (1) ses blocs sont bien numérotés dans l'ordre, (2) l'empreinte numérique contenu dans un bloc est bien celle du bloc précédent et (3) le premier bloc est *genesis*.

Question 6 En utilisant l'itérateur `List.fold_left`, écrire une fonction `check_integrity`, de type `chain -> bool`, qui vérifie l'intégrité d'une *blockchain*.

Arbres de Merkel. Nous nous intéressons maintenant au calcul de l'empreinte numérique des données associées à un bloc. Étant donnée une liste `[v0; v1; ...; vn]` de valeurs (d'un type quelconque), l'empreinte de cette liste est calculée à l'aide d'un arbre binaire particulier appelé *arbre de Merkel*. Cet arbre est construit de la manière suivante. Chaque feuille i contient l'empreinte de la valeur `vi` de la liste et chaque nœud contient l'empreinte numérique de la somme des empreintes de ses deux fils. Par exemple, l'arbre de Merkel obtenu pour une liste de 4 valeurs `[v0; v1; v2; v3]` est représenté dans la figure gauche ci-dessous, où $\texttt{H}_0$ représente l'empreinte de `v0`, $\texttt{H}_{0\text{-}1}$ l'empreinte de $\texttt{H}_0 + \texttt{H}_1$ et l'empreinte $\texttt{H}_{0\text{-}1\text{-}2\text{-}3}$ est celle de $\texttt{H}_{0\text{-}1} + \texttt{H}_{2\text{-}3}$.



Les arbres binaires de Merkel ont également la propriété que les *sous-arbres gauches* sont toujours *complets*, c'est-à-dire que tous les niveaux de ces arbres sont remplis. Ainsi, un nœud de niveau n aura nécessairement 2^{n-1} feuilles dans son arbre gauche. Par exemple, l'arbre de Merkel construit pour la liste `[v0; v1; v2; v3; v4; v5]` aura la forme présentée par l'arbre à droite dans la figure ci-dessus, pour lequel nous avons indiqué le niveau de chaque nœud. C'est l'empreinte stockée à la racine de l'arbre de Merkel qui est alors enregistrée dans le champ `merkel_root` d'un bloc.

On définit le type des arbres de Merkel de la manière suivante :

```
type merkel = E | N of int * merkel * merkel * int
```

Les arbres vides sont représentés par le constructeur `E`. Un arbre non-vide est un nœud `N(x, g, d, lvl)` où `x` est une empreinte numérique, `g` et `d` les sous-arbres gauches et droits et `lvl` est le niveau du nœud. Par convention, une feuille est donc représentée par un nœud avec deux sous-arbres vides.

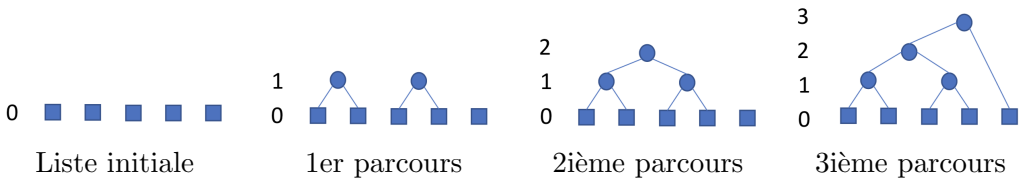
Question 7 Dessiner l'arbre de Merkel (avec les niveaux) construit à partir de la liste `[v0; v1; ...; v6]`.

Question 8 Écrire une fonction `value`, de type `merkel -> int`, telle que `value t` renvoie la valeur de hash contenue dans le nœud racine de l'arbre de Merkel `t`. Cette fonction lèvera l'exception `Not_found` pour un arbre vide.

Question 9 Écrire une fonction `make_leaf`, de type `'a -> merkel`, telle que `make_leaf v` crée un arbre de Merkel réduit à une feuille en calculant l'empreinte de `v` avec `hash`.

Question 10 Écrire une fonction `make_node`, de type `merkel -> merkel -> merkel`, telle que `make_node g d` renvoie l'arbre de Merkel construit à partir des sous-arbres (gauche) `g` et (droit) `d`, supposés non-vides.

Pour créer un arbre de Merkel à partir d'une liste `l` d'arbres de Merkel `[t`$_0$`;...;t`$_n$`]`, on parcourt `l` de gauche à droite et on construit une nouvelle liste en fusionnant les arbres deux à deux pour former un nouvel arbre de niveau $max(\texttt{level}(\texttt{t}_i), \texttt{level}(\texttt{t}_{i+1})) + 1$, avec i un indice pair. On recommence l'opération jusqu'à obtenir un seul arbre. Par exemple, en partant d'une liste avec 5 feuilles (de niveau 0), on construit l'arbre de la manière suivante (les niveaux sont indiqués à gauche des figures) :

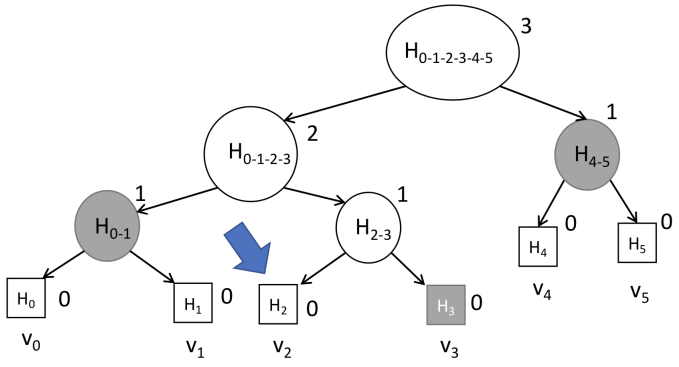


Question 11 Écrire une fonction `fusion`, de type `merkel list -> merkel list`, telle que `fusion l` fusionne deux à deux les arbres de Merkel de la liste `l` comme indiqué ci-dessus.

Question 12 En utilisant la fonction précédente, écrire une fonction `merkel_of_list`, de type `merkel list -> merkel`, telle que `merkel_of_list l` fabrique un arbre de Merkel en appliquant l'algorithme décrit ci-dessus. Cette fonction renverra l'arbre `E` si `l` est vide.

Question 13 En utilisant la fonction précédente, écrire une fonction `make_merkel`, de type `'a list -> merkel`, qui fabrique un arbre de Merkel à partir d'une liste de valeurs.

Les arbres de Merkel permettent facilement de vérifier qu'une valeur `v` est associée à un bloc (qui ne contient que l'empreinte de la racine). Pour cela, il suffit de renvoyer la liste des empreintes contenues dans les nœuds frères des nœuds parcourus depuis la racine de l'arbre jusqu'à la position de la feuille contenant l'empreinte de `v`. Cette liste, appelée *preuve de Merkel*, permet alors de refaire le calcul complet de la racine de l'arbre et vérifier qu'il est égal à la valeur stockée dans le champ `merkel_root` du bloc. Par exemple, la preuve de Merkel pour la feuille $\texttt{v}_2$ de l'arbre ci-dessous sera la liste `[H`$_{4-5}$`;H`$_{0-1}$`;H`$_3$`]`. En effet, on a bien $\texttt{H}_{0-1-2-3-4-5} = \texttt{hash}(\texttt{H}_{4-5} + \texttt{hash}(\texttt{H}_{0-1} + \texttt{hash}(\texttt{hash}(v2) + H_3))$.



Question 14 Écrire une fonction `proof_of_merkel`, de type `merkel -> int -> int list`, telle que `proof_of_merkel t i` renvoie la preuve de Merkel de la feuille `i` d'un arbre de Merkel `t`.