Master d'Informatique - Année 2011-2012
Examen de Programmation Fonctionnelle Avancée

Tous documents autorisés

Durée: 2h45

Deuxième Session: 26 Juin 2012

Zippers (7 points)

Exercice 1 (Visite d'une structure à l'aide d'un zipper) *Considérons la structure de donnée polymorphe suivante, qui décrit des arbres dont les noeuds peuvent avoir deux ou trois fils:*

```
type 'a tree23 = Empty | Bin of 'a * 'a tree23 * 'a tree23
               | Ter of 'a * 'a tree23 * 'a tree23 * 'a tree23;;
```

1. *dériver le type d'un zipper pour cette structure:*
2. *écrivez les fonctions permettant de se déplacer dans la structure: en haut, en bas à droite, en bas à gauche, en bas au milieu.*

Homomorphismes d'arbre (7 points)

Exercice 2 (Remplacer les feuilles d'un arbre par leur minimum) *Nous nous intéressons à des opération sur des arbres binaires d'entiers définis comme suit:*

```
type tree = L of int | N of tree * tree;;
```

Nous voulons programmer une fonction replace_min qui prend un arbre binaire t et produit un autre arbre binaire, ayant la même structure, mais dans lesquel toutes les feuilles contiennent comme valeur le plus petit entier présent dans l'arbre t.

Par exemple:

```
# replace_min (N (N (L 3, L 4), N (L 1, L 2)));;
- : tree = N (N (L 1, L 1), N (L 1, L 1))
```

- *écrivez une fonction get_min : tree -> int qui trouve la plus petite valeur contenue dans les feuilles d'un arbre*
- *écrivez une fonction replace : int -> tree -> tree qui remplace toutes les feuilles d'un arbre donnée avec une valeur entière donnée*

On peut maintenant écrire `let replace_min t = let m = get_min t in replace m t`; *mais cette solution nous oblige à visiter l'arbre deux fois: une pour trouver le minimum, et une pour effectuer le remplacement: nous pouvons faire mieux.*

Une fonction h est un homomorphisme d'arbre d'entiers si ils existent un opérateur leaf : int -> 'a et un opérateur $\oplus$: 'a -> 'a -> 'a tels que:

$$\begin{aligned} h(L\ n) &= leaf\ n \\ h(N(t_1, t_2)) &= (h\ t_1) \oplus (h\ t_2) \end{aligned}$$

- *écrivez une fonction tree_hom : (int -> 'a, 'a -> 'a -> 'a) qui implémente un homomorphisme d'arbre d'entiers*
- *reécrivez la fonction get_min : tree -> int en utilisant tree_hom*
- *reécrivez la fonction replace : int -> tree -> tree en utilisant tree_hom*
- *en modifiant la définition de replace, écrivez une fonction will_replace : tree -> (int -> tree), toujours en utilisant tree_hom, qui prend un arbre et retourne une fonction qui prend un entier et produit un arbre ayant la même structure, mais avec toutes les feuilles contenant cet entier*
- *fusionnez les opérations utilisées dans get_min et will_replace pour écrire une fonction will_replace_min_of_tree: tree -> (int -> tree) * int, en utilisant tree_hom une seule fois*

On peut définir la fonction recherchée, qui visite l'arbre une seule fois, simplement comme

```
let replace_min t = let (gt,m) = will_replace_min_of_tree t in gt m;;
```

Evaluation paresseuse (7 points)

Exercice 3 (Méthode de Newton et listes paresseuses) *La méthode de Newton pour trouver une solution d'une équation $f(x) = 0$, quand la fonction f est différentiable avec dérivée f', est la suivante:*

- *choisir une valeur $x + 0$*
- *construire la séquence des valeurs $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$*
- *s'arréter quand $f(x_i)$ est égal à 0.*

Cette méthode ne termine pas toujours: l'équation pourrait ne pas avoir une solution, la séquence des $f(x_i)$ pourrait ne pas converger, ou converger à l'infini et dans les deux cas on se retrouve avec une sequence infinie de valeurs.

On souhaite representer cette séquence potentiellement infinie avec une liste paresseuse, et définir des critères d'arrêt qui opèrent directement sur cette liste paresseuse.

Pour faire cela:

- *définissez un type floatseq des listes paresseuses de flottants*
- *écrivez une fonction récursive newton : (float -> float) -> (float -> float) -> float -> floatseq telle que newton f f' x donne la séquence (possiblement infinie) des x_i de la méthode de Newton pour la fonction f, avec dérivée f', à partir du point initial x.*
- *écrivez une fonction take: int -> floatseq -> floatseq qui retourne une liste paresseuse contenant les premiers n éléments de la liste paresseuse passée en paramètre*
- *écrivez une fonction converge: float -> floatseq -> floatseq telle que converge e fs retourne une liste paresseuse de flottants dont les deux derniers éléments diffèrent de moins que e*
- *enfin, écrivez une fonction qui permet d'arreter le calcul soit au but d'un nombre n d'itérations, soit quand la différence de deux valeurs consécutifs est inférieure à e, dès qu'une des deux conditions se produit.*