Examen de Programmation Comparée

Mars 2010 – Durée : 2h30

Seuls documents autorisés : cours, TPs, projets.

Livres interdits — Échange de documents interdit — Ordinateurs et téléphones interdits

1 Types et valeurs en Haskell

1. En Haskell, quel est le type et la valeur de l'expression suivante : `4`
2. En Haskell, quel est le type de l'expression suivante : `\x -> x+1`
3. En Haskell, quel est le type et la valeur de l'expression suivante :
```
[ x*x | x <- [1, -2, -3, 4], x>0 ]
```
4. En Haskell, quel est le type et la valeur de l'expression suivante :
```
bidule 44 [1,2,3,4]
  where bidule x l | x > length l = map (2*) l
                   | True = l ++ l
```
5. En Haskell, quel est le type et la valeur définie par :
```
z (Just f) (Just x) = Just (f x)
z _        _        = Nothing
```

2 Listes infinies en Haskell

1. Définir en Haskell la liste de tous les entiers
2. Définir en Haskell une fonction qui prend un prédicat (fonction booléenne) en entrée et renvoie la liste de tous les entiers qui satisfont ce prédicat.

3 Classes de types

1. Définir en Haskell la classe des types disposant d'une fonction `map`.
2. Instancier cette classe pour le type des listes (sans utiliser la fonction `map` prédéfinie).
3. Définir un type arbre binaire avec des valeurs uniquement aux feuilles. Instancier votre classe pour ce type.

4 CPS

On définit, en OCaml :

```
# let rec f x = if x = 0. then 1. else x *. g (x -. 1.)
  and g x = if x = 0. then 1. else f (x -. 1.) /. x;;
val f : float -> float = <fun>
val g : float -> float = <fun>
```

Écrire en OCaml une version CPS des fonctions `f` et `g`. C'est-à-dire des fonctions qui prennent en paramètre supplémentaire une continuation. (On ne demande pas de faire la transformation CPS systématique vue en cours, qui donnerait des résultat probablement très compliqués).

5 Monade des listes

En utilisant la monade des listes vue en cours (que vous n'avez pas besoin de redéfinir), écrire en OCaml une fonction `f : int list -> int list -> int list` telle que `f l1 l2` est la liste des `a+b`, où `a` est un élément de `l1` *positif ou nul*, et `b` est un élément de `l2` quelconque.

6 λ-calcul

Réduire faiblement le λ-terme suivant (écrit avec la syntaxe d'OCaml), d'abord en appel par valeur de gauche à droite, puis en appel par nom, puis en appel par nécessité. Indiquez pour chaque cas le nombre de règles de β-réduction appliquées.

Vous écrirez chaque étape de la β-réduction (en précisant le nom des règles), mais vous pouvez abréger en n'écrivant que les redex concernés et les parties qui changent, et en donnant des noms à des sous-termes.

```
(fun x y z -> match x y with Left u -> x y | Right v -> z)
  ((fun y x -> x) (fun x -> (fun y -> x) 1))
  (Left 4)
  ((fun t -> t) (Left 4))
```

Attention : il s'agit d'un seul terme, écrit sur plusieurs lignes (application d'une fonction à trois arguments).

7 Lwt et les mammouths

Le but de cet exercice est d'observer l'évolution d'une population de mammouths dans la steppe orientale en période pré-glaciaire. En cette période, la neige recouvre le sol et les mammouths se nourrissent d'arbustes, qui sont très rares. Les mammouths doivent donc aller le plus vite possible d'un arbuste à l'autre pour les manger et pouvoir survivre. Plus il est rapide, plus il mangera et donc plus il survivra longtemps.

Un mammouth est donc caractérisé par le temps qu'il met pour aller d'un arbuste à l'autre, qui sera constant tout au long de sa vie. Un mammouth a aussi un niveau d'énergie, initialement à 1000, et qui doit rester positif sinon le mammouth meurt. Manger un arbuste augmente l'énergie de 100. Sinon l'énergie décroît de 10 tous les dixièmes de seconde.

1. Écrire une fonction `mammouth` qui prend en paramètre le nom du mammouth (une chaîne de caractère) et lance deux threads lwt : l'un va faire décroître l'énergie de 10 tous les dixièmes de seconde, et l'autre augmente l'énergie de 100 à intervalle régulier (passé en paramètre à la fonction `mammouth`). On affichera le nom et l'énergie du mammouth à chaque fois qu'il mange et les deux threads ne poursuivront leur travail que si l'énergie est positive.
2. En fait le nombre d'arbustes n'est pas fixe. Un thread crée un arbuste à intervalle de temps régulier, et une fonction permet de connaître le nombre courant d'arbustes. Maintenant, le temps entre deux repas n'est plus fixe mais proportionnel au nombre d'arbustes.
   Implémentez ces fonctionnalités (sans réécrire la fonction `mammouth` entièrement : indiquez juste les changements).
3. Implémentez un thread Lwt qui crée des nouveaux mammouths. Le temps entre chaque naissance est inversement proportionnel au nombre de mammouths en vie. Indiquez les modifications à apporter à la fonction `mammouth` si nécessaire.