Université Paris Diderot — 1re année de master d'informatique

Examen de Programmation Fonctionnelle Avancée

Mai 2010 – Durée 3h

Seuls documents autorisés : cours, TPs, projets.
Livres interdits — Échange de documents interdit — Ordinateurs et téléphones interdits
Pour les exercices en OCaml, vous avez le droit d'utiliser des références et des exceptions.

1 λ-calcul

Réduire faiblement les λ-termes suivants (écrit avec la syntaxe d'OCaml), en utilisant les stratégies précisées :

1. en appel par valeur, de gauche à droite : `(fun x y -> x) 5`
2. en appel par nom : `(fun x y -> x) (fun t -> t)`
3. en appel par nom : `(fun x y z -> z x) 3 ((fun x -> x) 4) (fun t u -> t)`
4. en appel par valeur, de droite à gauche : `(fun x y z -> z x) 3 ((fun x -> x) 4) (fun t u -> t)`
5. en appel par valeur, de gauche à droite : `((fun x y -> y x) (fun t -> t)) (fun u v -> u 3)`
6. en appel par valeur, de gauche à droite :
   `match Right (fun x -> x) with Left y -> y 4 | Right z -> z z`
7. en appel par valeur, de gauche à droite : `(fun x y -> x y) y z`

Vous écrirez chaque étape de la β-réduction (en précisant le nom des règles).

2 Types et valeurs en Haskell

1. En Haskell, donner la valeur de l'expression suivante. Donner un type possible pour cette valeur (pas forcément le plus général).

```
[ (x, y) | x <- [-1, 2], y <- ['a', 'b'], x<0 ]
```

2. En Haskell, quel est le type et la valeur de l'expression suivante :

```
f "abc"
  where f [] = []
        f (a:l) = 0:f l
```

3. En Haskell, quel est le type et la valeur définie par :

```
f = \s -> case s of Left x -> x
                    _ -> 1
```

3 Classes de types

1. Définir en Haskell la classe des types disposant d'une fonction de hashage vers les entiers.
2. Instancier cette classe pour les entiers (dans ce cas-là la fonction de hashage est l'identité).

4 Monade des continuations

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 des fonctions `f` et `g` qui utilise la monade des continuations. Donner un exemple d'utilisation.

5 Réels en précision infinie

Les ordinateurs travaillent généralement avec une approximation des nombres réels appelé « nombres à virgule flottante » (type `float` en Caml), qui sont en général codés sur 64 bits. Dans cet exercice nous allons chercher à travailler sur une représentation exacte des réels, en utilisant la paresse.

Un réel peut toujours s'écrire sous la forme $s \times m \times b^e$, où
- s est le signe, égal à -1 ou 1,
- m est appelé la *mantisse*, qui est :
  - soit zéro
  - soit un nombre à virgule avec un seul chiffre non nul avant la virgule
- b est la base (généralement 2 pour les ordinateurs et 10 pour nous)
- et e est l'exposant (un entier relatif).

Exemple : $-5{,}432101.10^{-7}$

Pour représenter un nombre réel en précision infinie, nous pouvons coder l'exposant comme une liste finie de chiffres et la mantisse dans une liste paresseuse, (éventuellement infinie).

1. Écrire en OCaml **et** en Haskell un type permettant de représenter les réels de cette façon en base 10.
2. Définir le nombre $1/3$ en OCaml **et** en Haskell en utilisant cette représentation.
3. Écrire en OCaml **ou** en Haskell une fonction permettant de convertir un réel utilisant cette représentation en un nombre à virgule flottante du système. On supposera que l'exposant n'est pas trop grand et l'on se limitera à 16 chiffres significatifs au maximum.

En fait les opérations arithmétiques sont difficiles à coder (voir impossibles) avec cette représentation. Par exemple lors d'une addition de deux réels, on ne peut pas connaître un chiffre sans avoir calculé tous les chiffres suivants...

6 Lwt

Pour améliorer vos performances, vous avez décidé de mettre à jour le logiciel de votre cerveau pour le remplacer par une version implémentée par vous-même en OCaml avec Lwt. Le but de cet exercice est de commencer cette implémentation.

La première chose à faire est de créer un thread qui va donner l'ordre au cœur de battre à intervalle régulier. Pour donner l'ordre au cœur de faire un battement, il suffit d'appeler la fonction
`battre : unit -> unit`
En fait le temps entre deux battements doit être modifiable de l'extérieur. Il faudra donc définir une fonction `set_heart_period : float -> unit` qui pourra être appelée par un autre thread Lwt pour la modifier.

Votre programme n'utilise pas du tout de threads préemptifs.

1. Implémenter la fonction `set_heart_period` et le thread qui fait battre le cœur. Écrivez le programme principal, qui (pour l'instant) ne fait que lancer le thread qui fait battre le cœur.
2. Quel type de communication entre threads utilisez-vous pour transférer la valeur représentant la période du cœur ? Avez-vous besoin de verrous (mutex, sémaphores...) ? Expliquez pourquoi.

L'architecture du logiciel de votre cerveau est en fait en deux parties[1] : l'une appelée *volonté*, donne des ordres à la deuxième, appelée *centre moteur* (qui gère les mouvements). On ne s'intéresse dans cet exercice qu'à l'implémentation du centre moteur.

Le centre moteur reçoit des commandes de la volonté à l'aide de la fonction :

```
read_action : unit -> action Lwt.t
```

Le type `action` étant défini par[2] :

```
type action =
| Marche of float (* duree *) * float (* vitesse *)
| Parle of string (* mots a prononcer *)
```

La commande `Marche` demande de marcher pendant un certain temps et à une certaine vitesse. La commande `Parle` sert à activer votre synthétiseur vocal (les cordes vocales). On supposera écrites les deux fonctions suivantes qui implémentent ces actions :

```
marche : float -> float -> unit Lwt.t
parle : string -> unit Lwt.t
```

3. Implémenter la boucle principale qui écoute la volonté et déclenche les actions. On doit bien sûr pouvoir parler en marchant.
4. En fait une commande `Marche` peut arriver alors que la précédente n'est pas terminée. Dans ce cas, les deux actions doivent être séquentialisées (avoir lieu l'une après l'autre). Modifier le programme pour avoir ce comportement, en utilisant juste des fonctionnalités de base de Lwt (pas de structure de données de file ou similaires). (Il faudrait faire pareil pour `Parle` mais ce n'est pas demandé).
5. Le type action a en fait deux autres constructeurs : `Pause_Marche` et `Recommence_Marche`. Quand on reçoit le premier, il faut terminer les actions de marche en cours, mais les actions de marche reçues ensuite sont mises en attente jusqu'à ce que l'on reçoive `Recommence_Marche`. Implémenter ceci en utilisant `Lwt_unix.wait`.

7 Functionnal Parsing

On souhaite écrire une fonction `scan : 'a t -> string -> 'a option` qui prend comme premier argument un *format* (de type `'a t`) et en deuxième argument une chaîne de caractères qu'on souhaite analyser. Celle-ci devra se comporter de façon similaire à `scanf` :

- `scan (int ** bool) "42 true "` s'évaluera en `Some (42, true)`
- `scan ((char ** bool) ** int) "u true 32 "` s'évaluera en `Some (('u', true), 32)`
- `scan ((int ** bool) ** int) "666 true true "` s'évaluera en `None` (échec)

Les différentes valeurs lues sont supposées être terminées par une espace, de façon à rendre l'analyse déterministe. Les constructeurs de formats auquels on s'interesse sont :

```
val char : char t
val bool : bool t
val int : int t
val ( ** ) : 'a t -> 'b t -> ('a * 'b) t
val many : 'a t -> 'a list t
```

1. pour des raisons essentiellement historiques
2. Dans la réalité il est beaucoup plus compliqué

On propose de considerer les formats comme des cas particuliers de « parsers » dont le type est le suivant :

```
type 'a t =
  | Fail
  | Done of 'a
  | Left of (char -> 'a t)
```

Le constructeur `Fail` représente un parser qui échoue tout le temps, `Done x` représente un parser qui retourne la valeur `x` sans nécessiter la lecture de caractères, et la construction `Left f` représente un parser qui nécessite la lecture d'un caractère pour retourner (éventuellement) une valeur. La fonction `f` pourra être appliquée au premier caractère de l'expression à analyser, le nouveau parser obtenu par cette application devra être utilisé pour analyser le reste de l'expression.

L'exécution d'un format sur une chaîne donnée est faite de cette façon :

```
let scan format str =
  let rec aux i parser =
    (* i designe la position de lecture courante dans la chaine str *)
    match parser with
      | Fail -> None
      | Done a ->
          (* on verifie qu'il ne reste rien a lire *)
          if i = String.length str
          then Some a
          else None
      | Left f ->
          (* on verifie qu'il reste quelque chose a lire *)
          if i < String.length str
          then aux (i+1) (f str.[i])
          else None
    in
    aux 0 format
```

Question 1 Implémentez les formats basiques qui reconnaissent les chaînes de caractères adéquates, terminées par une espace.

- `char : char t`
- `bool : bool t`
- `int : int t`

Question 2 Le type des parsers qui a été proposé peut être vu comme une monade. Implémentez les fonctions `return` et `bind` qui permettent de composer plusieurs parsers.

Question 3 Implémentez les constructeurs de format `( ** )` et `many` qui découlent assez facilement des deux fonctions évoquées à la question précédente.

Note : Le parser `many p` analyse une liste de valeurs reconnues par le parser `p`. Dans la chaîne à analyser toute liste est supposée précédée par sa longueur :

- `scan (many bool) "3 true false false "` s'évaluera en `Some [true; false; false]`
- `scan (many (int ** int)) "2 43 67 12 14 "` s'évaluera en `Some [(43, 67); (12, 14)]`
- `scan (many char) "12 a b c "` s'évaluera en `None` (trop peu d'éléments)