UNIVERSITÉ DE STRASBOURG

Département d'Informatique
LICENCE 2ème année
Vendredi 11 mars 2011

Interrogation écrite de Programmation Fonctionnelle

Durée : 2h. Aucun document autorisé. Toutes les fonctions sont à écrire en OCAML. Ne pas utiliser d'éléments impurs !

I Récursivité

(Dans cet exercice, on utilisera la notation prédéfinie pour les listes.)

1. (Rendez la monnaie !). Définir une fonction `change` qui détermine une façon de payer une somme d'argent à l'aide d'une liste de valeurs de billets disponibles. On supposera que cette liste est donnée suivant un ordre décroissant. On lèvera une exception en cas d'impossibilité de paiement. Exemple :

   ```
   change 1280 [500;200;100;50;10] = [500;500;200;50;10;10;10]
   ```

   signifie que l'on peut payer 1280 euros avec 2 billets de 500, 1 de 200, 1 de 50 et 3 de 10.

2. (Intervalle). Définir une fonction `intervalle` qui, étant donnés deux entiers n et m tels que $n \leq m$, retourne la liste des entiers successifs de n à m. Exemple :

   ```
   intervalle 3 6 = [3; 4; 5; 6]
   ```

3. (Adjonction en tête). Définir une fonction `adjoint` qui, étant donnés un élément x et une liste de listes l, renvoie la liste de toutes les listes obtenues en ajoutant x au début des listes de l. Exemple :

   ```
   adjoint 1 [[2;3];[2];[3];[]] = [[1; 2; 3]; [1; 2]; [1; 3]; [1]]
   ```

4. (Ensemble des parties). Définir une fonction `ensemble_des_parties` qui, étant donnée une liste l, renvoie l'ensemble des parties de l. On se servira de la fonction `adjoint` de la question précédente. Exemples :

   ```
   ensemble_des_parties [2;3] = [[2; 3]; [2]; [3]; []]

   ensemble_des_parties [1;2;3] = [[1; 2; 3]; [1; 2]; [1; 3]; [1];
                                   [2; 3]; [2]; [3]; []]
   ```

5. (Insertions d'un nouvel élément dans une liste). Définir une fonction nommée `insertions` qui, étant donnés un élément x et une liste l, renvoie la liste de toutes les listes obtenues en insérant x à une position quelconque dans l. Exemple :

```
insertions 4 [1;2;3] = [[4; 1; 2; 3]; [1; 4; 2; 3];
                        [1; 2; 4; 3]; [1; 2; 3; 4]]
```

II Algorithme de typage

Dérouler l'algorithme de typage vu en Cours et en TD pour trouver le type de cette fonction : `let twice = function f -> function x -> f (f x);;`. (On remplira 3 colonnes : une pour l'ensemble $\mathcal{H}$ des hypothèses, une pour l'ensemble $\mathcal{E}$ des équations entre types et une pour l'ensemble $\mathcal{T}$ des expressions à typer).

III Types structurés

On définit en OCaml le type `'a arbre` des arbres binaires polymorphes avec des valeurs de type `'a` aux noeuds :

```
type 'a arbre =
   Vide
 | Noeud of 'a arbre * 'a * 'a arbre;;
```

1. Écrire une fonction `eclate` qui, étant donné un arbre t dont les valeurs aux noeuds sont des couples, construit un couple d'arbre $(t1, t2)$ tels que $t1$ (resp. $t2$) est obtenu à partir de t en masquant, en chaque noeud, les deuxièmes (resp. premières) composantes des couples. Le type de `eclate` sera `('a * 'b) arbre -> 'a arbre * 'b arbre`.
2. Écrire une fonction `renverse` qui, étant donné un arbre t dont les valeurs sont des couples, renvoie l'arbre obtenu à partir de t en renversant tous les couples. Le type de `renverse` sera `('a * 'b) arbre -> ('b * 'a) arbre`.
3. Écrire une fonctionnelle `map_arbre` qui, étant donné une fonction f et un arbre t, renvoie l'arbre obtenu en appliquant f sur chaque valeur de t. Le type de `map_arbre` sera `('a -> 'b) -> 'a arbre -> 'b arbre`.
4. Réécrire `renverse` en utilisant `map_arbre`.