Département d'Informatique
LICENCE 2ème année
Mardi 24 octobre 2006

Travaux Dirigés de Programmation Fonctionnelle

«Une véritable théorie des types»

I Règles de typage

1. Parenthèser autant que possible ces expressions de type :
   (a) `int * float -> float`
   (b) `int -> float * float`
   (c) `int -> float -> float`
   (d) `int * float * float`
2. Enlever les parenthèses inutiles dans ces expressions de type :
   (a) `(int * float) -> float`
   (b) `int * (float -> float)`
   (c) `(int -> float) * float`
   (d) `int -> (float * float)`
   (e) `(int -> float) -> float`
   (f) `int -> (float -> float)`
   (g) `int * (float * float)`
   (h) `(int * float) * float`
3. Associer correctement ces valeurs (a)-(d) à ces types (1)-(4) :
   (a) `(0,function x -> x+.0.) ; ;`
   (b) `function (x,y) -> if x=0 then y else 0. ; ;`
   (c) `function x -> function y -> if x=0 then y else 0. ; ;`
   (d) `function x -> (x 0)+.0. ; ;`

   (1) `int * float -> float`
   (2) `(int -> float) -> float`
   (3) `int * (float -> float)`
   (4) `int -> float -> float`
4. Dérouler l'algorithme d'inférence de type vu en cours (construction et résolution du système d'équations) pour trouver le type de ces fonctions :
   (a) `let fa = function x -> function y -> x+y ; ;`
   (b) `let fb = function x -> (x 0)+0 ; ;`
   (c) `let fc = function x -> function y -> (y x)+x ; ;`
   (d) `let rec fd = function x -> if x=0 then 0 else (fd (x-1)) ; ;`
5. Même question que 4. pour les fonctions suivantes :

(a) `let ga = function x -> function y -> x ; ;`
(b) `let gb = function x -> (x 0) ; ;`
(c) `let gc = function x -> function y -> (y x) ; ;`
(d) `let gd = function x -> function y -> if y=0 then (ga x 0)`
`else (ga 1 true) ; ;`

II Fonctions classiques sur des types algébriques

1. **Fonctions sur les listes**. Rappeler la définition du type des listes polymorphes en utilisant les constructeurs `Cons` et `Nil`. Définir les fonctions suivantes, d'abord en utilisant la définition classique, puis en utilisant les notations prédéfinies en Ocaml pour les listes : `longueur` (longueur d'une liste), `tete` (premier élément d'une liste), `reste` (la liste sans son premier élément), `nieme` (n-ième élément d'une liste en numérotant les éléments à partir de 0), `inverse` (liste obtenue en renversant une liste : le 1er élément devient le dernier, le 2ème devient l'avant-dernier, ...), `concat` (concaténation de 2 listes). Préciser le type de chaque fonction.
2. **Fonctions sur les arbres**. Rappeler la définition du type des arbres binaires polymorphes à valeurs aux noeuds en utilisant les constructeurs `Vide` (arbre vide) et `Racine` (enracinement de 2 sous-arbres). Définir les fonctions suivantes : `recherche` (recherche d'un élément dans un arbre : la fonction retourne vrai si l'élément est dans l'arbre, faux sinon), `taille` (nombre d'éléments dans un arbre), `parcours_prof` (parcours d'un arbre en profondeur d'abord : Etant donné un arbre, construire la liste de ces éléments en suivant l'ordre de parcours en profondeur d'abord. Lors de ce parcours, on descend le plus profondément dans l'arbre avant de se déplacer en largeur), `parcours_larg` (parcours de l'arbre en largeur d'abord : même chose que précédemment en suivant l'ordre de parcours en largeur d'abord).

III Expressions arithmétiques

1. Définir le type des *expressions arithmétiques* sur les entiers naturels. On utilisera la définition suivante : Une expression arithmétique est soit la notation d'un entier naturel, soit l'addition de 2 expressions arithmétiques, soit la multiplication de 2 expressions arithmétiques.
2. Donner un exemple de valeur du type défini en 1.
3. Définir une fonction qui calcule le résultat d'une expression arithmétique.