**Département d'Informatique**
*LICENCE 2ème année*
*Mardi 14 octobre 2008*

# 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`  (c) `int -> float -> float`
   (b) `int -> float * float`  (d) `int * float * float`

2. Enlever les parenthèses inutiles dans ces expressions de type :

   (a) `(int * float) -> float`  (e) `(int -> float) -> float`
   (b) `int * (float -> float)`  (f) `int -> (float -> float)`
   (c) `(int -> float) * float`  (g) `int * (float * float)`
   (d) `int -> (float * float)`  (h) `(int * float) * float`

3. Associer correctement les valeurs (a)-(d) aux types (1)-(4) :

   (1) `int * float -> float`  (a) `(0,function x -> x+.0.) ; ;`
   (2) `(int -> float) -> float`  (b) `function (x,y) -> if x=0 then y else 0. ; ;`
   (3) `int -> float -> float`  (c) `function x -> (x 0)+.0. ; ;`
   (4) `int * (float -> float)`  (d)
   ```
   function x ->
       function y -> if x=0 then y else 0. ; ;
   ```

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 pour les fonctions polymorphes 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 En noir et blanc

1. Définir le type `palette` contenant trois teintes : blanc, gris et noir, susceptibles d'être mélangées et éclaircies.
2. Définir une fonction `melanger` qui retourne la couleur obtenue en mélangeant deux couleurs (ex. si on mélange du blanc et du noir, on obtient du gris).
3. Définir les fonctions `eclaircir` (resp. `assombrir`) qui, étant donnée une couleur, donne une couleur plus claire (resp. plus foncée).

## III Expressions arithmétiques

1. Définir le type `expr_arith` 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 `expr_arith`.
3. Définir une fonction qui, étant donnée une valeur du type `expr_arith`, retourne le résultat du calcul de cette valeur.

## IV Types algébriques classiques

1. **Listes polymorphes**. Définir le type des listes polymorphes en utilisant les constructeurs `Nil` et `Cons`. Utiliser le type `list` prédéfini en Ocaml pour définir les fonctions `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. **Arbres binaires polymorphes**. Définir le type des arbres binaires polymorphes valués en chaque noeud. Définir les fonctions `recherche` (recherche d'une valeur dans un arbre : la fonction retourne vrai si la valeur est dans l'arbre, faux sinon), `taille` (nombre de valeurs dans un arbre), `parcours_prof` (parcours d'un arbre en profondeur d'abord : construire la liste des valeurs de l'arbre ordonnées selon l'ordre du parcours en profondeur d'abord.). Préciser le type de chaque fonction.