NFP 119 : Programmation Fonctionnelle

22 juin 2009

Il vous est recommandé de rédiger de façon concise et précise, tout en écrivant lisiblement. Il sera tenu compte de la présentation dans la note de la copie. Tous les documents sont autorisés. Le barème n'est donné qu'à titre indicatif

Rappel : fonctionnelles

- `List.assoc : 'a → 'a*'b list → 'b`. L'appel `List.assoc x l` prend une valeur x et une liste l formée de paires (a, b) et renvoie la valeur b qui correspond à la paire (x, b) dans la liste, et échoue si aucune paire avec x en première position n'est trouvée.
- `List.map f l :('a -> 'b) -> 'a list -> 'b list`. Prend en argument une fonction f, une liste l, et produit une nouvelle liste en appliquant f sur chacun des éléments de l.
- `List.mem x l : 'a list -> bool`. Prend en argument un élément x et une liste l et renvoie true si x se trouve dans l et false sinon.
- `List.for_all cond l : ('a -> bool) -> 'a list -> bool`. Prend en argument une fonction `cond` de test et une liste l. Renvoie true si tous les éléments de l vérifient la condition `cond`. Si au moins un des éléments de l ne vérifie pas cond, renvoie false. Remarque : la liste vide vérifie n'importe quelle condition.

Problème (12 points)

Exercice 1:

Un *contexte de typage* décrit les noms des types pour les variables d'un programme. Il s'agit d'une liste de paires (x, Ty) où x est un nom de variable et Ty est le nom de son type. Les noms des variables sont représentés par des `string` et leurs types par le type `nomType` donné plus bas, qui représente 2 sortes de types : les entiers et les booléens. La variable `c` est un exemple de contexte de typage pour deux variables, x et a.

```
type nomType = Int | Bool;;
let c = [("x", Int); ("a", Bool)];;
typeOfVar "a" c;;            (* doit renvoyer => Bool *)
```

1. Écrivez la fonction `typeOfVar` qui prend un contexte de typage et un nom de variable et qui retourne son type, ou qui échoue si la variable n'est pas dans le contexte.

   Réponse:

   ```
   let rec typeOfVar n c = match c
   with  []   -> failwith ("unboud␣variable␣"^n)
     | (x,t)::r -> if x=n then t else typeOfVar n r
   ```

2. Donner une nouvelle version **non récursive** de `typeOfVar`, utilisant cette fois la fonctionnelle `List.assoc`.
   Réponse:

```
let typeOfVarBis n c = List.assoc n c;;
```

□

Exercice 2:

À l'aide du type `expr` donné plus bas, nous définissons un petit langage d'expressions, permettant de représenter les expressions constantes entières, true, false, l'addition et le `if`. **Exemples** : la variable `e` correspond à l'expression `1 + 2` ; alors que `i` correspond à `if true then 1 else false` (**qui est une expression mal typée**).

```
type expr = Num of int              (* Nombres *)
          | CBool of bool           (* Boooleans *)
          | Sum of expr * expr      (* Addition *)
          | If of expr * expr * expr;; (* If *)

let e = Sum ((Num 1), (Num 2));; (* 1 + 2            *)
let i = If ( (CBool true),      (* if true then 1 else false *)
           (Num 1),
           (CBool false));;

typeOfExpr(e)               (* doit renvoyer => Int *)
typeOfExpr(i)               (* doit renvoyer => Bool *)
```

Ces expressions sont de type entier ou booléen. Nous voulons écrire une fonction `typeOfExpr` permettant de déterminer le nom du type (autrement dit, le `nameType` de la partie I) d'une expression ou de lever une exception si l'expression est mal typée.
Exemples : le type de `Sum( (Num 1), (Num 2))` sera `Int`, celui de `CBool(true)` sera `Bool`, alors que `Sum( (Num 1), (CBool true))` est mal typée et doit ainsi lever une exception.

Écrivez la fonction `typeOfExpr` qui prend une `expr` en argument et renvoie son `nomType` ou qui échoue si elle est mal typée.
Réponse:

```
let rec typeOfExpr e =
match e
with Num _ -> Int
  | CBool _ -> Bool
  | Sum (e1, e2) -> if (typeOfExpr e1) = Int &&
                    (typeOfExpr e2) = Int then Int else failwith"bad␣sum␣arguments"
  | If (e1, e2, e3) -> if (typeOfExpr e1) <> Bool
                       then failwith"if␣condition␣must␣be␣boolean"
                    else let t = (typeOfExpr e2) in
                        if (typeOfExpr e3) = t then t
                        else
                          failwith"if␣branches␣must␣be␣of␣same␣type";;
```

□

Exercice 3:

Nous allons ajouter les noms de variables aux expressions `expr`, ce qui nous permettra de représenter des expressions telles que `1 + x`. Pour typer ce genre d'expression on doit connaître le type de ces variables. On va donc utiliser les contextes de typage de la partie I. Voici la nouvelle définition du type `expr`.

```
type expr = Num of int                (* Nombres *)
          | CBool of bool             (* Boooleans *)
          | Var of string             (* Variables *)
          | Sum of expr * expr        (* Addition *)
          | If of expr * expr * expr;; (* If     *)

let e = Sum ((Num 1), (Var "x"));; (* 1 + x        *)
let i = If ( (Var "b"),           (* if b then 1+x else y *)
           (Sum ((Num 1), (Var "x"))),
           (Var "y"));;

typeOfExpr c e                (* doit renvoyer => Int *)
typeOfExpr [("x",Bool)] e     (* doit lever une exception *)
varsInExpr i                  (* => renvoit ["b";"x";"y"] *)
listUnbound [("x",Int)] e     (* => renvoit []       *)
listUnbound [("y",Int)] i     (* => renvoit [("x";"b"] *)
```

1. Écrivez la nouvelle fonction `typeOfExpr`, de manière à prendre en argument supplémentaire un contexte de typage (`typeOfExpr :(string * nameType) list -> expr -> nameType`).

   Réponse:

   ```
   let rec typeOfExpr c e =
   match e
   with Num _ -> Int
     | CBool _ -> Bool
     | Var x -> typeOfVar x c
     | Sum (e1, e2) -> if (typeOfExpr c e1) = Int &&
                         (typeOfExpr c e2) = Int then Int else failwith"bad␣sum␣arguments"
     | If (e1, e2, e3) -> if (typeOfExpr c e1) <> Bool
                            then failwith"if␣condition␣must␣be␣boolean"
                          else let t = (typeOfExpr c e2) in
                              if (typeOfExpr c e3) = t then t
                              else
                                failwith"if␣branches␣must␣be␣of␣same␣type";;
   ```

2. Écrivez la fonction `varsInExpr` qui prend en argument une expression et qui renvoie en résultat la liste de tous les noms de variables qui apparaissent dans l'expression. Par exemple, pour l'expression `i` plus haut, elle doit retourner la liste `["b"; "x"; "y"]`.

   Réponse:

   ```
   let rec varsOfExpr e =
   match e
   with Num _ -> []
     | CBool _ -> []
   ```

```
    | Var x -> [x]
    | Sum (e1, e2) -> (varsOfExpr e1) @ (varsOfExpr e2)
    | If (e1, e2, e3) -> (varsOfExpr e1) @ (varsOfExpr e2) @(varsOfExpr e3);;
```

3. On voudrait écrire une fonction qui détermine si une expression contient des variables non définies (absentes) dans son contexte de typage. Par exemple, `e` ne peut pas être typée dans le contexte `[(Var "w",Int)]` car il ne contient pas la variable "x".
   Écrivez la fonction `unboundVarsInExpr` qui prend un contexte de typage c, une expression e, et donne la liste de toutes les variables de e qui ne sont pas définies dans c. Vous **devez utiliser** la fonction `varsInExpr` de la question précédente. On peut écrire une solution **non récursive** avec les fonctionnelles List.filter, List.map et List.mem. Attention : la solution **sans** ces fonctionnelles est récursive et beaucoup plus longue.
   Réponse:

```
  let listUnbound c e =
    let v = varsOfExpr e
    and vc = List.map fst c in
    List.filter (fun x -> not (List.mem x vc)) v
;;
```

□

Exercices (8pts)

Exercice 4 (2pts):

Soit le fonction `pourtout` suivante :

```
let rec filtre f l =
 match l with
  | [] -> true
  | e::l' -> if f e then e::(filtre f l') else filtre f l'
```

Démontrez la propriété suivante : $\forall$ `l`, `|f l|=|l|`, où `|f l|` et `|l|` désignent les longueurs des listes `f l` et `l`.
Réponse: Par induction sur `l`*, facile.*

□

Exercice 5 (4pts):

Soit `g` la fonction suivante :

```
let rec g (x,y,l) =
 if x <= 0 || y <= 0 then []
 else
  if x >= y then
    match l with
    | [] -> g ((x-1),(y-1),[x;y])
    | e::l' -> e::e+x::e+y:: g(x,y,l')
 else g(y,x,l)
```

1. (2pts) Démontrez que la fonction `g` termine.

   Réponse: On définit la composition lexicographique de l'ordre $<_{\mathbb{N}}$ et de l'ordre sur les tailles de liste, sur le 2ème et le 3ème argument de g (dans cet ordre).

   `g (x-1) (y-1) [x;y]` *décroît bien sur le 2ème argument.* `(g x y l')` *décroît bien sur le 3ème argument alors que le 2ème ne change pas.* `(g y x l)` *décroît bien sur le 2ème argument car cet appel récursif est effectué lorsque* `x<y`.

2. (2pts, plus dur) Démontrez que $\forall$ `l`, `y`, `x`,`|g x y l|` est un multiple de 3.

   Réponse: On procède par induction forte en utilisant l'ordre donné à la question précédente.

□

Exercice 6 (2pts):

Soit l'ensemble E défini par induction comme suit :

$$\begin{array}{rcl} B_E & = & \{0, 3\} \\ \Omega_B & = & \{\omega : (E \times E) \to E\} \\ & & \text{où } \omega(i, j) = 3i + j. \end{array}$$

1. Démontrez que tous les éléments de E sont divisibles par 3. Autrement dit démontrez la propriété suivante : $\forall n \in E, \exists i \in \mathbb{N}, n = 3i$.

   Réponse: Par induction sur n.

□