NFP119 : corrigé de la feuille d'exercices 4

María-Virginia Aponte

26 mars 2010

Exercice 1

Nous donnons le développement des deux appels :

Appel `double_du_succ(g1,3)` :
`double_du_succ(g1,3)` ⇒ `2*f(y+1)` où f = g1 et y =3
⇒ `2* g1(3+1)`
⇒ `2* (g1 4)`
⇒ `2*` (`z*z` où z=4)
⇒ `2* 16` = 32

Appel `compare_res(f1,g1,2)` :
`compare_res(f1,g1,2)`
⇒ `f(x+2) > g(x)+2` où f=f1, g=g1, x=2
⇒ `f1(2+2) > g1(2)+2`
⇒ `f1(4) > g1(2)+2`
⇒ `4-2 > 2*2 + 2`
⇒ `2 > 6`
⇒ `false`

Exercice 2

Écrivez une fonctionnelle récursive `sigma` qui prend une fonction f et un entier n et calcule :

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) \ldots + f(n)$$

En utilisant cette fonctionnelle, calculez la somme des n premier entiers, et la somme des carrés de n premiers entiers.

```
# let rec sigma f n =
   if n <=0 then 0
   else (f n) + sigma f (n-1);;

# let somme_n n =
   let identite = (fun x -> x)
  in sigma identite n;;
val somme_n : int -> int = <fun>
```

```
# somme_n 4;;
- : int = 10
```

Une notation plus compacte pour `somme_n` :

```
# let somme_n n = sigma (fun x -> x) n;;
val somme_n : int -> int = <fun>

# somme_n 4;;
- : int = 10
```

Même chose pour `somme_carres_n` :

```
# let somme_carres_n n =  sigma (fun x -> x*x) n;;
val somme_carres_n : int -> int = <fun>

# somme_carres_n 3;;
- : int = 14
```

Voici une autre solution avec une notation encore plus compacte, mais un peu plus obscure :

```
# let somme_n = sigma (fun x -> x);;
val somme_n : int -> int = <fun>

# somme_n 4;;
- : int = 10

# let somme_carres_n = sigma (fun x -> x*x);;
val somme_carres_n : int -> int = <fun>

# somme_carres_n 3;;
- : int = 14
```

Exercice 3

```
(* question 1 *)

# let rec existe cond l =
match l
with [] -> false
|    a::reste -> cond a || existe cond reste;;
val existe : ('a -> bool) -> 'a list -> bool = <fun>

# existe (fun x -> x mod 2 =0) [1;3;5;6];;
- : bool = true

(* Version du module List *)

# List.exists;;
- : ('a -> bool) -> 'a list -> bool = <fun>

#  List.exists  (fun x -> x mod 2 =0) [1;3;5;6];;
- : bool = true
```

```
(* question 2 *)

# let rec pour_tous cond l =
match l
with [] -> false
|    a::reste -> cond a && pour_tous cond reste;;
val pour_tous : ('a -> bool) -> 'a list -> bool = <fun>

# pour_tous (fun x -> x mod 2 =0) [1;3;5;6];;
- : bool = false

(* Version du module List *)

# List.for_all;;
- : ('a -> bool) -> 'a list -> bool = <fun>

# List.for_all (fun x -> x mod 2 =0) [1;3;5;6];;
- : bool = false
```

Exercice 4

Dans cet exercice vous devez écrire des fonctions à l'aide des fonctionnelles vues en cours.

1. À l'aide de `filter` : écrire la fonction qui extrait d'une liste de dates toutes celles dont l'année est bissextile.

   ```
   type date = {jour:int; mois:int; annee: int};;

   let aujourd'hui = {jour=27; mois=2; annee = 2006};;

   let bissextile a =
     (a mod 4 =0) && (not (a mod 100 = 0) or (a mod 400 = 0));;

   let rec filter cond l =
     match l
    with []     -> []
     |  a::reste -> if (cond a) then a::(filter cond reste)
                    else filter cond reste;;

   (* La version du module List *)

   # List.filter;;
   - : ('a -> bool) -> 'a list -> 'a list = <fun>


   # let dates_bissextiles l = List.filter (fun d -> bissextile d.annee) l;;
   val dates_bissextiles : date list -> date list = <fun>
   ```

2. À l'aide de `map` : écrire la fonction qui extrait la liste de noms d'une liste d'employés, puis la fonction `remplace_tous` de l'exercice 4.

   ```
   # let remplace_tous a b  l = List.map (fun x -> if a=x then b else x) l;;
   val remplace_tous : 'a -> 'a -> 'a list -> 'a list = <fun>

   # type employe = {nom: string; numero: int};;
   type employe = { nom : string; numero : int; }
   ```

```
# let extrait_noms l = List.map (fun x -> x.nom) l;;
val extrait_noms : employe list -> string list = <fun>
```

Tester toutes ces fonctions en TP sur plusieurs exemples.

Exercice 5

Fonctionnelles sur les listes

1. Écrire la fonction `est_trie` qui prend en argument une liste de n'importe quel type et teste si elle est triée dans l'ordre croissant.

```
# let rec est_trie l =
match l
with []  -> true
  |  [_] -> true
  |  x::y::reste -> x<= y && est_trie (y::reste);;
val est_trie : 'a list -> bool = <fun>

# est_trie [1;2;6;9];;
- : bool = true

# est_trie [9;3;4];;
- : bool = false
```

2. Modifier cette fonction afin d'obtenir une fonctionnelle `est_trie_gen` qui prend en argument une liste de n'importe quel type et et une fonction qui correspond à un test d'ordre entre deux éléments de la liste. Cette fonction de test doit être de type `'a -> 'a -> bool`. La fonction `est_trie_gen` teste si la liste est triée selon l'ordre passé en argument.

```
# let rec est_trie_gen testOrdre l =
match l
with []  -> true
  |  [_] -> true
  |  x::y::reste -> testOrdre(x, y) && est_trie_gen  testOrdre (y::reste);;
val est_trie_gen : ('a * 'a -> bool) -> 'a list -> bool = <fun>
```

3. A l'aide le fonction précédente, écrire les fonctions :
   - la fonction qui teste si une liste est triée dans l'odre décroissant,

```
# let est_trie_decroissant l =
 let  testDecroissant (x,y) = x>=y
 in est_trie_gen  testDecroissant l;;
val est_trie_decroissant : 'a list -> bool = <fun>

# est_trie_decroissant  [9;5;3];;
- : bool = true

# est_trie_decroissant [1;3;5;6];;
- : bool = false
```

   - la fonction qui teste si une liste des dates est triée dans l'odre croissant.

```
# let dates_triees l =
 let  testDate (x,y) =
     x.annee < y.annee ||
    (x.annee = y.annee && x.mois < y.mois) ||
    (x.annee = y.annee && x.mois = y.mois && x.jour < y.jour)
```

```
 in est_trie_gen  testDate l;;
val dates_triees : date list -> bool = <fun>

# let ld = [{ jour = 14; mois = 1; annee =2005 };
            { jour = 17; mois = 3; annee =2005 };
            { jour = 1; mois = 3; annee =2007 }];;

dates_triees  ld;;
- : bool = true
```

– la fonction qui teste si une liste de vols est triée par l'ordre croissant des prix et décroissant des disponibilités.

Exercice 7

Exercice d'un ancien sujet

Partie I

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 répresente 2 sortes de types : les entiers et les booléans. 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. Ecrivez 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.
2. Donner une nouvelle version **non récursive** de `typeOfVar`, utilisant cette fois la fonctionnelle `List.assoc`.

Partie II

A l'aide du type `expr` donné plus bas, nous définissons un petit langage d'expressions, permettant de représenter les expressions constantes entierès, 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 booléean. 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.

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

Partie III

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"]      *)
```

Vous devez réécrire la fonction `typeOfExpr` de manière à prendre en argument supplémentaire un contexte de typage.

1. Ecrivez la nouvelle fonction `typeOfExpr:(string * nameType) list -> expr -> nameType`.
2. Ecrivez 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"]`.
3. On voudrait écrire une fonction qui determine 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". Ecrire 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.

Correction

```
(* Partie I *)

 (* 1 *)
  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 *)

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

(* Partie II *)

   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";;

(* Partie III *)

   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";;

(* Question 4 *)
   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);;


(* Question 5 *)

  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
;;

listUnbound [] e ;;
listUnbound ["x",Int] e;;
listUnbound ["y",Int] i;;
```

Exercice 8

Deux exercices d'un ancien sujet

Partie I
Correction :

```
(* Question 1 *)

 let afficheTable (Table d) =
   let rec affiche d c mess cm =
   match d
   with [] -> ()
    |   (Section (title,ss))::r ->
              let num = cm^(string_of_int c) in
              let s = mess^"section " in
              let entete = num^s^title in
              print_string entete; print_newline();
              affiche ss 1 " sub" ("  "^num^".");
              affiche r (c+1) mess cm
    |   (FinalSect title)::r ->
              let num = cm^(string_of_int c) in
              let s = mess^"section " in
              let entete = num^s^title in
              print_string entete; print_newline();
              affiche r (c+1) mess cm
   in affiche d 1 " " "";;
```

Pour la question 2 on construit la numérotation récursivement avec une méthode similaire à celle employée par la fonction ci-dessus.

Partie II
Corrections

```
(* Question 1 *)
  let rec chercheUnMot m indx = match indx
  with     []     -> []
    |  (t,l)::r -> if List.mem m l then t::(chercheUnMot m r)
                   else chercheUnMot m r;;

chercheUnMot "Egypte" i;;

(* Question 2 *)

let rec chercheTousLesMots lm indx = match indx
  with     []     -> []
    |  (t,lt)::r -> if List.for_all (fun x -> List.mem x lt) lm
                              then t::(chercheTousLesMots lm r)
                   else chercheTousLesMots lm r;;

chercheTousLesMots ["programmation";"informatique"] i;;

(* Question 3 *)

  let pertinenceDe lm (t,lt) =
```

```
    let rec compte lm =
    match lm
    with  []    -> 0
      |  m::r  -> if List.mem m lt then 1+ (compte r)
                  else compte r
    in compte lm;;

pertinenceDe ["Egypte"; "dieux"] ("Voyager en Egypte",["Egypte"; "voyages"]);;

(* Question 4 *)

let lePlusPertinent lm indx =
 let pesageDocs = List.map (fun (t,lt) -> (t, pertinenceDe lm (t,lt))) indx in
 let rec maxDoc p =
 match p
 with [] -> failwith  "lePlusPertinent"
  |   [(t,_) as x] -> x
  |   (t,n)::r  -> let (tr,maxr) = maxDoc r in
                   if n>maxr then (t,n) else (tr,maxr)
  in fst(maxDoc pesageDocs);;

lePlusPertinent  ["programmation";"informatique"; "Java"] i;;
```