7 novembre 2005
Durée : 2 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master 1 Informatique 2005-2006

Examen partiel

Typage

Exercice 1. Les définitions suivantes produisent des erreurs de type :

```
 1 let binome x y  =
 2   let
 3     rec fact = function n ->
 4       if n <= 0. then 1. else n *. fact (n -. 1.)
 5   in
 6     fact x / (fact y *. fact (x -. y)) ;;

 8 let rec somme_bornee (x,y) f acc =
 9   if x > y then acc else
10     somme_bornee x (y-1) f (acc + f y);;

12 let rec map f = function
13     [] -> ""
14   | t::q -> (f t)^(map f);;
```

Pour chacun des noms définis (`binome`, `somme_bornee`, `map`)

1. expliquer pour quelle raison l'erreur de type se produit,
2. corriger la définition, de façon qu'un erreur de type ne se produise pas,
3. après correction, en calculer le type.

Solution. On laisse Ocaml nous expliquer les erreurs de type qui se produisent :

```
# let binome x y  =
  let
    rec fact = function n ->
      if n < 0. then 1. else n *. fact (n -. 1.)
  in
    fact x / (fact y *. fact (x -. y));;
          Characters 113-119:
    fact x / (fact y *. fact (x -. y));;
    ^^^^^^
This expression has type float but is here used with type int
# let rec somme_bornee (x,y) f acc =
  if x > y then acc else
    somme_bornee x (y-1) f (acc + f y);;
    Characters 77-78:
      somme_bornee x (y-1) f (acc + f y);;
                   ^
This expression has type 'a but is here used with type 'a * 'a
# let rec map f = function
    [] -> ""
  | t::q -> (f t)^(map f);;
    Characters 56-63:
    | t::q -> (f t)^(map f);;
                     ^^^^^^^
This expression has type 'a list -> string but is here used with type string
```

On peut corriger comme il suit, en laissant Ocaml calculer les types :

```
# let binome x y  =
  let
    rec fact = function n ->
      if n < 0. then 1. else n *. fact (n -. 1.)
  in
    fact x /. (fact y *. fact (x -. y));;
          val binome : float -> float -> float = <fun>
# let rec somme_bornee (x,y) f acc =
  if x > y then acc else
    somme_bornee (x,(y-1)) f (acc + f y);;
    val somme_bornee : int * int -> (int -> int) -> int -> int = <fun>
# let rec map f = function
    [] -> ""
  | t::q -> (f t)^(map f q);;
    val map : ('a -> string) -> 'a list -> string = <fun>
```

□

Évaluation

Exercice 2.

– Expliquer ce que veut dire que une expression du langage Ocaml est un valeur.

Solution. Une expression est un valeur quand elle ne peut pas être ultérieurement évaluée. □

– Expliquer quand une expression fonctionnelle (c.-à-d. de type `'a -> 'b` où `'a` et `'b` peuvent être remplacés par des autres types) est un valeur.

Solution. Un expression fonctionnelle est un valeur quand son opérateur principal est un `fun` ou un `function` (ou abstraction). Par exemple

```
(fun x -> x + 1) : int -> int
```

est un valeur,

```
(fun y -> fun x -> x + y) 1 : int -> int
```

n'est pas un valeur. Dans ce deuxième cas, l'opérateur principal est une application : on peut donc appliquer la β-réduction. □

– Que se passe-t'il à la définition des noms `expr1` et `expr2` :

```
let expr1 = print_endline "1"; 2;;
let expr2 () = print_endline "1"; 2;;
```

Solution. L'évaluation donne

```
# let expr1 = print_endline "1"; 2;;
1
val expr1 : int = 2
# let expr2 () = print_endline "1"; 2;;
val expr2 : unit -> int = <fun>
```

On observe que `expr2` est une fonction car sa définition est équivalente à

```
let expr2 = fun () -> print_endline "1";2;;
```

Dans cette définition, l'expression `print_endline "1"` (de type `unit`) n'est pas évaluée car elle se trouve sous un `fun`. Elle ne produit donc pas des effets de bord. □

Exercice 3. Donner les étapes de l'évaluation par valeur ET de l'évaluation par nom de l'expression suivante :

```
(fun x -> fun y -> x)
2
( (fun f -> f 3) (fun x -> x +1) )
```

Solution. Évaluation par VALEUR :

```
(fun x -> fun y -> x)
2
( (fun f -> f 3) (fun x -> x +1) )
---->
(fun y -> 2)
( (fun f -> f 3) (fun x -> x + 1) )
---->
(fun y -> 2)
( (fun f -> f 3) (fun x -> x + 1) )
---->
(fun y -> 2)
( (fun x -> x + 1) 3 )
---->
(fun y -> 2)
(3  + 1)
---->
(fun y -> 2)
4
---->
2
```

Évaluation par NOM :

```
(fun x -> fun y -> x)
2
( (fun f -> f 3) (fun x -> x +1) )
---->
(fun y -> 2)
( (fun f -> f 3) (fun x -> x + 1) )
---->
2
```

□

Fonctions récursives

Exercice 4. Considérez la fonction `append` définie par :

```
let rec append l1 l2 = match l1 with
    [] -> l2
  | t::q -> t::(append q l2);;
```

Démontrer les egalités suivantes :

– `append l1 [] = l1`,

Solution. Nous allons demontrer cette proprieté par induction sur la longeur de la liste `l1`.

– Si `l1 = []`, alors

```
append l1 [] = append [] [] -> l2 = [] = l1
```

– Si `l1 = t : :q`, alors

```
append l1 [] = append t::q [] -> t::(append q [])
```

Car la longueur de la liste `q` est strictement plus petite que celle le la liste `t : :q`, par hypothese d'induction

```
append q [] -> q
```

et donc

```
append l1 [] -> t::(append q []) -> t::q = l1 .
```

□

– `append (append l1 l2) l3 = append l1 (append l2 l3)`.

Solution. Nous allons demontrer cette proprieté par induction sur la longeur de la liste `l1`.

– Si `l1 = []`, alors

```
append (append l1 l2) l3 = append (append [] l2 ) l3
          -> append l2 l3 -> l4
append l1 (append l2 l3) -> append l1 l4
          =  apend [] l4 -> l4
```

– Si `l1 = t : :q`, alors

```
append (append l1 l2) l3 =  append (append t::q l2) l3
   -> append t::(append q l2) l3
   -> t::(append (append q l2) l3)
append l1 (append l2 l3) = append t::q (append l2 l3)
   -> t::(append q (append l2 l3))
```

Car la longueur de la liste `q` est strictement plus petite que celle le la liste `t : :q`, par hypothese d'induction

```
append (append q l2) l3 = append q (append l2 l3)
```

et donc

```
append (append l1 l2) l3
  = t::(append (append q l2) l3) = t::(append q (append l2 l3))
  = append l1 (append l2 l3) .
```

□

Conseil : utiliser l'induction sur la longueur de la liste `l1`.

Types récursifs

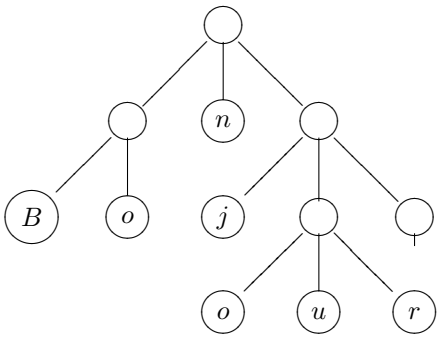
Considérez le type récursif `arbre` avec son itérateur générique `arbre_iter` :

```
 1  type arbre = Feuille of char | Noeud of arbre list ;;

 3  let rec arbre_iter funvar funcoller premier funnoeud = function
 4      Feuille c -> funvar c
 5    | Noeud fils ->
 6        let
 7          recur = arbre_iter funvar funcoller premier funnoeud
 8        in
 9          funnoeud (
10            List.fold_left funcoller premier
11                       (List.map recur fils));;
```

Un tel arbre a un nombre arbitraire fini de fils et peut stocker l'information, des caractères, sur les feuilles.

Exercice 5. Écrire une expression OCaml correspondant à l'arbre suivant :



Solution.

```
30  Noeud [ Noeud [Feuille 'B'; Feuille 'o'];
31          Feuille 'n';
32          Noeud [ Feuille 'j';
33                  Noeud [ Feuille 'o' ; Feuille 'u'; Feuille 'r' ];
34                  Noeud [];
35                ]

37        ];;
```

□

Exercice 6. En utilisant l'itérateur générique `arbre_iter`
- écrire une fonction `no_feuilles : arbre -> int` qui compte le nombre des feuilles dans un arbre.
- écrire une fonction `frontiere : arbre -> string` qui produit la chaîne de caractères placés sur les feuilles dans l'ordre de gauche à droite. (On pourra supposer qu'une fonction `char_to_string : char -> string`, transformant un caractère en chaîne de caractères, est définie).

Par exemple :

```
# let a = Noeud [Feuille 'a'; Noeud [Feuille 'c'; Feuille 'b']];;
no_feuilles a;;
- : int = 3
# frontiere a;;
- : string = "acb"
```

Solution. Voici le code :

```
 3  let no_feuilles a =
 4    let
 5      funvar = fun x -> 1
 6    and
 7      (funcoller,premier) = ((+),0)
```

```
 8    and
 9      funnoeud = fun x -> x
10    in
11      arbre_iter funvar funcoller premier funnoeud a;;

13  let char_to_string = String.make 1 ;;

15  let frontiere a =
16    let
17      funvar = char_to_string
18    and
19      (funcoller,premier) = ((^),"")
20    and
21      funnoeud = fun x -> x
22    in
23      arbre_iter funvar funcoller premier funnoeud a;;
```

□

Programmation

Exercice 7. On peut représenter un nombre positif en notation binaire par une suite d'entiers contenant seulement des 0 ou des 1. Le bit de poids plus faible sera en tête de la liste. Par exemple, les listes

```
[1;0;1], [1;1;0;1;0]
```

représenteront les nombres 5 et 11 respectivement. Le but de l'exercice est d'écrire une fonction `sommebin` qui calcule la (représentation binaire de la) somme de deux listes.
Plus précisément, on implémentera (le style de programmation sera fonctionnel) les fonctions décrites dans l'interface suivante :

```
(* On lèvera cette exception
   si une liste n'est pas la représentation
   binaire d'un nombre, à savoir si elle
   contient des entier différents de 0 et 1 *)
exception Nombre_non_binaire

(* Transformer une liste de 1,0 en entier  *)
val to_int : int list -> int

(* Transformer un entier en sa représentation binaire *)
val from_int : int -> int list

(* Calcule la représentation binaire
   de la somme [i] + l,
   où i in {0,1}
   et l est la représentation binaire d'un nombre
*)
val carry : int * int list -> int list

(* Calcule la représentation binaire
   de la somme [i] + l1 + l2,
   où i in {0,1}
   et l1,l2 sont des représentations binaires de deux nombres.
   On se sert pas des fonctions to_int et from_int
*)
val sommebinc : int * int list * int list -> int list

(* Calcule la représentation binaire
   de la somme  l1 + l2,
   où l1,l2 sont des représentations binaires de deux nombres.
   On se sert pas des fonctions to_int et from_int
*)
val sommebin : int list -> int list -> int list
```

Solution.

```
exception Nombre_non_binaire;;

let rec to_int  = function
    [] -> 0
  | 0::q -> 2 * to_int q
  | 1::q -> 1 + 2 * to_int q
  | _ -> raise Nombre_non_binaire;;

let rec from_int = function
    0 -> []
  | n -> (n mod 2)::from_int (n/2);;
```

```
let rec carry = function
    (0,l) -> l
  | (1,[]) -> [1]
  | (1,t::q) ->
      (1-t)::carry (t,q)
  | _ -> raise Nombre_non_binaire;;

let rec sommebinc = function
    (c,[],ps) | (c,ps,[]) -> carry (c,ps)
  | (c,t1::q1,t2::q2) ->
      ((c + t1 + t2) mod 2):: sommebinc ((c + t1 + t2) / 2,q1,q2);;

let sommebin n1 n2 = sommebinc (0,n1,n2) ;;
```

□