Partiel Logique, Déduction et Programmation
Master informatique 2004-2005

Durée : 2 heures
Documents autorisés

Examen partiel

Typage

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

```
# let rec somme  x y  = match x with
    0 -> y
  | _  -> 1 +. (somme (x -. 1)  y);;
# let rec mcd (x,y) = if (x mod y) = 0 then y else
  mcd y (x mod y) ;;
# let ajout c chaine = match chaine with
    ""  -> [c]
  | 'a'::q -> "a"^(ajout c q)
  | 'b'::q -> [c]@chaine
  |  t::q -> t::c::q ;;
```

Pour chacun des noms `somme`, `mcd` et `ajout` :

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. En calculer le type.

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

```
# let rec somme  x y  = match x with
    0 -> y
  | _  -> 1 +. (somme (x -. 1)  y);;
    Characters 57-58:
    | _  -> 1 +. (somme (x -. 1)  y);;
             ^
This expression has type int but is here used with type float
# let rec mcd (x,y) = if (x mod y) = 0 then y else
  mcd y (x mod y);;
  Characters 55-56:
    mcd y (x mod y);;
        ^
This expression has type int but is here used with type int * int
# let ajout c chaine = match chaine with
    ""  -> [c]
  | 'a'::q -> "a"^(ajout c q)
  | 'b'::q -> [c]@chaine
  |  t::q -> t::c::q;;
       Characters 59-65:
    | 'a'::q -> "a"^(ajout c q)
      ^^^^^^
This pattern matches values of type char list
but is here used to match values of type string
```

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

```
# let rec somme  x y  = match x with
    0. -> y
  | _  -> 1. +. (somme (x -. 1.)  y);;
```

```
    val somme : float -> float -> float = <fun>
# let rec mcd x y = if (x mod y) = 0 then y else
  mcd y (x mod y);;
  val mcd : int -> int -> int = <fun>
# let ajout c chaine = match chaine with
    []  -> [c]
  | 'a'::q -> 'a'::(ajout c q)
  | 'b'::q -> [c]@chaine
  |  t::q -> t::c::q;;
       val ajout : char -> char list -> char list = <fun>
```

□

Évaluation

Exercice 2. Donner deux exemples d'expressions ayant type `'a -> 'b` (où `'a` et `'b` peuvent être remplacés par des types concrètes). La première expression est une valeur, la deuxième n'est pas une valeur.

Solution.

```
1 : # fun f -> f(3);;
2 : # (fun x -> fun f -> f(x)) 3 ;;
```

□

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

```
( fun x -> fun f -> f(x + f(x)) )
( (fun g -> g(g(3))) (fun z -> z + 1) )
( fun w -> 1 )
```

Solution. Évaluation par VALEUR :

```
( fun x -> fun f -> f(x + f(x)) )
( (fun g -> g(g(3))) (fun z -> z + 1) )
( fun w -> 1 )
---->
( fun x -> fun f -> f(x + f(x)) )
( (fun z -> z + 1)((fun z -> z + 1) 3) )
( fun w -> 1 )
---->
( fun x -> fun f -> f(x + f(x)) )
( (fun z -> z + 1)(3 + 1 ) )
( fun w -> 1 )
---->
( fun x -> fun f -> f(x + f(x)) )
( (fun z -> z + 1) 4)
( fun w -> 1 )
---->
( fun x -> fun f -> f(x + f(x)) )
(4 + 1)
( fun w -> 1 )
---->
( fun x -> fun f -> f(x + f(x)) ) 5
( fun w -> 1 )
---->
fun f -> f(5 + f(5))
( fun w -> 1 )
```

```
---->
( fun w -> 1 )(5 + ( fun w -> 1 )(5))
---->
( fun w -> 1 )(5 + 1)
---->
( fun w -> 1 )(6)
---->
1
```

Évaluation par NOM :

```
( fun x -> fun f -> f(x + f(x)) )
( (fun g -> g(g(3))) (fun z -> z + 1) )
( fun w -> 1 )
---->
fun f -> f(( (fun g -> g(g(3))) (fun z -> z + 1) )  + f((fun g -> g(g(3))) (fun z -> z + 1) ))
( fun w -> 1 )
---->
( fun w -> 1 )(( (fun g -> g(g(3))) (fun z -> z + 1) )  + ( fun w -> 1)((fun g -> g(g(3))) (fun z -> z + 1
---->
1
```

□

Exercice 4. Considérer la définition suivante :

```
# let rec fix = fun () -> (print_string "*"; (fun x -> x + 1) (fix ()));;
```

1. Évaluer le type du nom `fix`.
2. Dire ce qui s'affiche à l'écran si l'on tape :

```
# fix;;
# fix ();;
```

Solution.

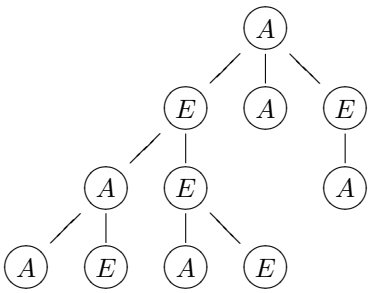
```
# let rec fix = fun () -> (print_string "*"; (fun x -> x + 1) (fix ()));;
val fix : unit -> int = <fun>
# fix;;
- : unit -> int = <fun>
# fix ();;
***************************** etc.
```

□

Types récursifs

Un jeu entre deux joueurs A(dam) et E(va) peut être représenté sous la forme d'un arbre :
– Les positions sont les noeuds de l'arbre.
– Chaque position est étiquetée par un joueur (le joueur dont est le tour à jouer).
– La position de départ est la racine de l'arbre.
– Un coup ou mouvement à partir d'une position consiste à choisir un fils : après le coup, le jeu se continuera de ce fils.
– Si un joueur ne peut pas choisir un fils (à cause que la position est une feuille) alors il perde.

Par exemple, le suivant est un jeu entre A et E, où le joueur E possède une stratégie gagnante :



(1)

Exercice 5.

1. En Caml, définir un type `jeu` qui code les jeux en forme d'arbre.
2. En utilisant la syntaxe ainsi définie, écrire une expression Caml ayant type `jeu` qui code le jeu (1).

Exercice 6.

1. Écrire une fonction `est_gagnant`, ayant type `jeu -> bool`, qui retourne la valeur `true` si et seulement si le joueur E a une stratégie gagnante dans le jeu passé en paramètre.
2. Définir un type `strategie` pour coder les stratégies gagnantes dans une jeu. Modifier la fonction `est_gagnant` pour obtenir une fonction `chercher_strategie` de type `jeu -> strategie` : cette fonction lèvera une exception si E n'a pas de stratégies gagnantes, ou bien retournera un objet `s : strategie` représentant une stratégie gagnante pour E.

Exercice 7.

1. Définir une fonction `iter_jeu` (iterateur générique sur les jeux) dont on s'en servira sous la forme

   ```
   iter_jeu funjoueur premier coller j
   ```

   Dans son complexe, cette expression aura un type générique `'a`. Le types des paramètres sont :
   - `j : jeu`,
   - `funjoueur` a deux paramètres, un joueur et un objet de type `'b`, et elle retourne un objet de type `'a`,
   - `premier : 'b` et `coller : 'b -> 'a -> 'b` expliquent comme traiter les résultats obtenus de façon récursive sur les fils.
2. Se servir de l'iterateur générique `iter_jeu` pour donner une deuxième définition de la fonction `est_gagnante`.

Solution. On regroupe les solution des exercices 4-5-6 dans le programme suivant :

jeu.ml

```
 1 : type joueur = Adam | Eva;;
 2 : type jeu = Noeud of joueur*jeu list;;
 3 :
 4 : let jeuno1 =
 5 :   Noeud(Adam,[
 6 :           Noeud(Eva,[
 7 :                   Noeud(Adam,[
 8 :                           Noeud(Adam,[]);
 9 :                           Noeud(Eva,[])
10 :                         ]);
11 :                   Noeud(Eva,[
12 :                           Noeud(Adam,[]);
13 :                           Noeud(Eva,[])
14 :                         ])
15 :                ]);
16 :           Noeud(Adam,[]);
17 :           Noeud(Eva,[Noeud(Adam,[])])
18 :         ]);;
19 :
20 : let rec est_gagnant = function
21 :     Noeud(Eva,fils) ->
22 :       List.exists est_gagnant fils
23 :   | Noeud(Adam,fils) ->
24 :       List.for_all est_gagnant fils;;
25 :
26 : type strategie = Padam of strategie list | Peva of int*strategie;;
27 :
28 : exception NotFound;;
29 :
30 : let rec chercher_strategie =
31 :   let rec
32 :     strat_chercher n = function
33 :         [] -> raise NotFound
34 :       | t::q -> try (n,chercher_strategie t) with
35 :             NotFound -> strat_chercher (n + 1) q
36 :   in
37 :     function
38 :         Noeud(Eva,fils) ->
39 :           let
40 :             (i,s) = strat_chercher 0 fils
41 :           in
42 :             Peva (i,s)
43 :       |
44 :         Noeud(Adam,fils) ->
45 :          Padam (List.map chercher_strategie fils);;
46 :
47 : let rec iter_jeu funjoueur premier coller =
48 :   function Noeud(joueur,fils) ->
49 :     let
50 :       recur j = iter_jeu funjoueur premier coller j
51 :     in
52 :     let
53 :       autrecoller x j = coller x (recur j)
54 :     in
55 :       funjoueur joueur (List.fold_left autrecoller premier fils);;
56 :
57 : let autre_est_gagnante =
58 :   let
59 :     premier = []
60 :   and
61 :     coller l x = x::l
62 :   and
63 :     funjoueur joueur liste_de_bool =
64 :     match joueur with
65 :         Eva -> List.exists (fun x -> x) liste_de_bool
66 :       | Adam ->  List.for_all (fun x -> x) liste_de_bool
67 :   in
68 :     iter_jeu funjoueur premier coller
69 : ;;
```

□

Programmation

Exercice 8. On rappelle d'abord l'algorithme `quicksort` (« tri vite » ?). On se donne un liste d'objets à trier :
- on choisit un de ces objets, on l'appelle pivot.
- on met tous qui est plus petit que le pivot dans une liste, on met tous qui est plus grand que le pivot dans une autre liste.
- on trie, de façon récursive, les deux listes ainsi obtenues, et on les met l'une à coté de l'autre dans le bon ordre, possiblement séparées par le pivot.

Écrire une implémentation de cet algorithme en Caml (expliciter si vous utilisez le langage Ocaml ou le langage Camllight). La fonction `quicksort` aura le type `('a -> 'a -> bool) -> 'a list -> 'a list`. Le premier argument est une fonction de comparaison sur les objets de type `'a`, le deuxième argument est la liste à trier.

Solution (en Ocaml).

quicksort.ml

```
 1 : (* Tri vite (quick sort)  *)
 2 :
 3 : let rec quicksort comparaison = function
 4 :     [] -> []
 5 :   | pivot::q ->
 6 :       let
 7 :         l1 = List.filter (fun x -> comparaison x pivot) q
 8 :       and
 9 :         l2 = List.filter (fun x -> not (comparaison x pivot) ) q
10 :       in
11 :         (quicksort comparaison l1) @
12 :         [pivot] @ (quicksort comparaison l2) ;;
13 :
14 : quicksort (<=) [3;2;5;6;2;88;5;2;4];;
```

□