Programmation Fonctionnelle Avancée

Contrôle des connaissances 2009/2010
Master Informatique - Institut Galilée - Paris 13
J.-V. Loddo

Documents de cours et de TP autorisés
Durée 3h
Le barème est donné à titre indicatif pour les trois parties

Première partie

Compréhension de code OCaml et typage (8 points)

Q1. Nous rappelons que les opérateurs de comparaison (=) et (<) sont *infixes*, *polymorphes* et ont le *même* type :

```
# (=) ;;
- : 'a -> 'a -> bool = <fun>
# (<) ;;
- : 'a -> 'a -> bool = <fun>
```

Pour chaque expression suivante, dire si l'expression est correcte et, le cas échéant, donner le type. En revanche, justifiez la réponse si vous considérez l'expression non typable.

(*a*) `['d';'a';'d';'a']`
(*b*) `('d','a','d','a')`
(*c*) `('d',12,"hello",[3.14;6.28])`
(*d*) `('d','a','d','a') = ('d','o','d','o')`
(*e*) `['d';'a';'d'] = ['d';'o';'d';'o']`
(*f*) `('d','a','d') = ('d','o','d','o')`
(*g*) `let x="hello" in ['d';'a';x] < ['d';'o';x;'o']`
(*h*) `let x="hello" in ('d','a',x) < ('d','o',x)`
(*i*) `(=) 3`
(*j*) `(<) [3;5]`

Q2. Écrire une expression dont le type soit :

(*a*) `'a -> ('a list * 'a)`
(*b*) `('a list list * string) -> 'a`

Q3. Étant donné la définition de type :

```
type answer = Yes | No ;;
```

et sachant que la fonction `abs` renvoie la valeur absolue de son argument :

```
# abs ;;
- : int -> int = <fun>
```

(*a*) quel est le résultat du programme suivant :

```
let eq = fun x y -> (x=y) ;;
let f x y = if (eq x y) then Yes else No ;;
let eq = fun x y -> (abs x) = (abs y) in
  f 3 (-3) ;;
```

(*b*) y aurait-il une différence observable avec le programme suivant ?

```
let eq = fun x y -> (abs x) = (abs y) in
  let f x y = if (eq x y) then Yes else No in
    f 3 (-3) ;;
```

(*c*) et avec le programme suivant ?

```
let eq = ref (fun x y -> (x=y)) ;;
let f x y = if (!eq x y) then Yes else No ;;
eq := (fun x y -> (abs x) = (abs y));;
f 3 (-3) ;;
```

Justifier les réponses (les réponses non motivées ne seront pas prises en compte).

Q4. Sachant que la fonction `uncurry` transforme une fonction à plusieurs arguments en fonction à un seul argument de type paire :

```
# let uncurry f = function (x,y) -> f x y ;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

en rappelant le type des fonctions suivantes du module standard `List` :

```
# List.combine ;;
- : 'a list -> 'b list -> ('a * 'b) list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

et en rappelant que la fonction `combine` transforme deux listes `[x0;y0;z0;..]` et `[x1;y1;z1;..]` en liste de couples `[(x0,x1);(y0;y1);(z0;z1);...]`, répondre aux questions suivantes :

(*a*) quel est le type de la fonction `foo` ainsi définie :

```
let foo f x y = List.map (uncurry f) (List.combine x y) ;;
```

(*b*) qu'est-ce que calcule la fonction `foo`

(*c*) faire un exemple d'utilisation de `foo`, c'est-à-dire donner les arguments et le résultat attendu en utilisant l'addition sur les entiers `(+)` comme premier argument.

Deuxième partie

Typage formel (4 points)

Q5. Supposons de définir un langage fonctionnel typé statiquement. Supposons que la règle logique définissant (calculant) le type d'une application fonctionnelle soit écrite formellement de la façon suivante :

$$\frac{\Gamma \vdash e_0 : t_1 \to t_2 \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash e_0\, e_1 : t_2}$$

et codée de la façon suivante :

```
let rec type_of_expr env = function
 ...
 | Apply (e0,e1) ->
     let t0 = type_of_expr env e0 in
     (match t0 with
      | TArrow (t1,t2) -> ...
      | _ -> failwith ‘‘bad application’’
      )
 ...
```

(*a*) comment vous écririez la règle logique définissant le *while* classique des langages de programmation impératifs ?

$$\frac{?}{\Gamma \vdash \texttt{while}\ e_0\ \texttt{do}\ e_1\ : ?}$$

(*b*) comment vous écririez le code OCaml correspondant à votre règle ?

```
let rec type_of_expr env = function
 ...
 | While (e0,e1) ->  ?
 ...
```

Troisième partie

Programmation (8 points)

Q6. Le langage OCaml est distribué avec une librairie standard qui contient un module `Array` permettant la définition et le traitement des tableaux. Son comportement est cependant impératif, ce qui peut être deviné en observant la signature du module et en particulier le type de la fonction `set` :

```
module Array : sig
    val init       :  int -> (int -> 'a) -> 'a array       (* création *)
    val length     :  'a array -> int                      (* longueur *)
    val get        :  'a array -> int -> 'a                (* lecture  *)
    val set        :  'a array -> int -> 'a -> unit        (* écriture *)
    ...
 end
```

Le résultat de `Array.set a i x` est un effet de bord (`unit`) représentant la modification "sur place" à la valeur `x` de la cellule `i` du tableau `a`. Nous voulons implémenter un module similaire mais persistent (fonctionnel) que nous appellerons `PArray`. Donc, nous allons définir une fonction `set` d'un autre type :

```
    val set        :  'a array -> (index * 'a) list -> 'a array
```

(où le type `index` est juste un alias : `type index = int`) qui construira un résultat sans modifier son argument. Ce résultat sera identique au tableau donné à l'exception des mises à jours indiquées par le second argument, qui est une liste de couples (*index,valeur*). Exemple :

```
# let a = PArray.create 10 (fun i -> 0) ;;
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
# let b = PArray.set a [(1,42);(9,84)] ;;
val b : int array = [|0; 42; 0; 0; 0; 0; 0; 0; 0; 84|]
# a;;
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
# PArray.rev b ;;
- : int array = [|84; 0; 0; 0; 0; 0; 0; 0; 42; 0|]
# b;;
- : int array = [|0; 42; 0; 0; 0; 0; 0; 0; 0; 84|]
```

Travail à réaliser : écrire un foncteur `PArray_completion` tel que le module `PArray` puisse être défini comme application :

```
module PArray = PArray_completion (PArray_kernel) ;;
```

et de façon que la signature de `PArray` soit exactement (ni plus, ni moins) la signature `PArray_signature` définie ainsi :

```
module type PArray_kernel_signature =
 sig
   type size  = int
   type index = int
   val create : size -> (index -> 'a) -> 'a array
   val length : 'a array -> size
   val    get : 'a array -> index -> 'a
 end
module type PArray_signature =
 sig
   include PArray_kernel_signature
   val set  : 'a array -> (index * 'a) list -> 'a array
   val map  : (index -> 'a -> 'b) -> 'a array -> 'b array
   val copy : 'a array -> 'a array
   val rev  : 'a array -> 'a array
 end
```

Suggestion : sans que cela soit nécessaire, songez à utiliser la fonction `List.iter` pour programmer `set` :

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Q7. Extrait de la documentation officielle OCaml du module `List` :

`val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

`val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.rev_map2 f l1 l2` gives the same result as `List.rev (List.map2 f l1 l2)`, but is tail-recursive and more efficient.

Questions :

(a) programmer `map2`

(b) programmer `rev_map2` de façon récursive terminale