Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2013-2014

TP n°2 bis - Correction

Usages avancés des types

Exercice 1 [Variants polymorphes et variance]
Soit le code suivant :

```
module M : sig
  type ('a, 'b) t
end = struct
  type ('a, 'b) t = 'a * 'b
end
```

1. Est-ce que l'expression suivante est bien typée ? Pourquoi ?

   ```
   let f x = (x : ([ `A ], [ `B ]) M.t :> ([ `A | `C ], [ `B | `D ]) M.t)
   ```

2. Sans utiliser des types avancés, comment peut-on faire pour que cela type ?
3. Modifier le code afin de ne rien dévoiler de l'implémentation et que l'expression de la question 1 soit bien typée.
4. Corriger la signature du code suivant pour qu'il compile :

   ```
   module M : sig
     type (+'a, +'b) t
   end = struct
     type ('a, 'b) t = 'a -> 'b
   end
   ```

Correction :

1. Le code proposé pour `f` n'est pas bien typé car le type `M.t` est abstrait. Le compilateur ne peut pas vérifier que la relation de sous-typage demandée est bien vraie sans accéder à la définition (ou une annotation de variance). Du point de vue de l'endroit du programme où on se trouve, c'est-à-dire à l'extérieur de l'interface du module, le type pourrait très bien avoir été défini en interne comme

   ```
   type ('a, 'b) t = 'a -> 'b -> unit
   ```

   Dans ce cas, le sous-typage demandé serait complètement faux : une fonction qui accepte comme premier argument seulement `[ `A]` ne peut pas être considéré comme une fonction qui accepte `[`A | `B]`, puisqu'elle planterait sur l'entrée `` `B ``.

2. On peut avoir

   ```
   module M : sig
     type ('a, 'b) t = 'a * 'b
   end = struct
     type ('a, 'b) t = 'a * 'b
   end
   ```

   mais on perd tout l'intérêt d'avoir une interface...

3.
```
module M : sig
    type (+'a, +'b) t
  end = struct
    type ('a, 'b) t = 'a * 'b
  end
```
4.
```
module M : sig
    type (-'a, +'b) t
  end = struct
    type ('a, 'b) t = 'a -> 'b
  end
```

Exercice 2 [Types fantômes]
On veut écrire un module connection qui permet de se connecter en lecture seule ou lecture/écriture. Pour cela on va utiliser les types fantômes et les variants polymorphes.

Complétez le code suivant là où il ya des `...` :

```
module Connection : sig
  ...
  val connect_readonly : ...
  val connect : ...
  val status : ...
  val destroy : ...
end = struct
  type 'a t = int
  let count = ref 0
  let connect_readonly () = incr count; !count
  let connect () = incr count; !count
  let status c = c
  let destroy c = ()
end
```

Pour tester votre code vous pourrez essayer :

```
open Connection
open Printf

let () =
  let conn = connect_readonly () in
    printf "status = %d\n" (status conn);
    (*destroy conn; (* error *) *)
    let conn = connect () in
      printf "status = %d\n" (status conn); destroy conn
```

sachant que si vous décommentez `destroy conn` vous *devez* avoir une erreur.

Correction :

```
module Connection : sig
  type 'a t
  val connect_readonly : unit -> [`Readonly] t
```

```
  val connect : unit -> [‘Readonly|‘Readwrite] t
  val status : [>‘Readonly] t -> int
  val destroy : [>‘Readwrite] t -> unit
end = struct
  type ’a t = int
  let count = ref 0
  let connect_readonly () = incr count; !count
  let connect () = incr count; !count
  let status c = c
  let destroy c = ()
end

open Connection
open Printf

let () =
  let conn = connect_readonly () in
  printf "status = %d\n" (status conn);
  (*destroy conn; (* error *) *)
  let conn = connect () in
  printf "status = %d\n" (status conn);
  destroy conn
```

Exercice 3 [GADT]
On veut écrire une structure de données qui soit une liste pouvant contenir des entiers et des flottants avec une unique fonction `get` permettant de récupérer le premier entier (ou flottant).

1. Ecrire en utilisant un GADT le type kind dont on se servira pour indiquer à la fonction `get` le type que l'on veut récupérer.
2. Ecrire le type somme value permettant de contenir des entiers et des flottants
3. Ecrire le type list en utilisant un GADT.
4. Ecrire la fonction `get`
5. Ecrire une fonction `print` qui affiche le contenu d'une liste.
6. Testez vos fonctions avec les expressions suivantes :

```
let empty = Nil
let l1 = Cons ((Int 1), empty)
let l2 = Cons ((Float 2.), l1)
let () = print l2
let i = get l2 Int_kind
let f = get l2 Float_kind;;
```

Correction :

```
type _ kind = (* used for searching in the list *)
 | Int_kind : int kind
 | Float_kind : float kind

type value = (* box the values so we have runtime type information *)
 | Int of int
 | Float of float
```

```
type list = (* the universal list *)
 | Nil : list
 | Cons : value * list -> list

(* find the first value in the list of a given kind *)
let rec get : type a . list -> a kind -> a = fun l k ->
 match (k, l) with
   | (_, Nil) -> raise Not_found
   | (Int_kind, Cons (Int x, xs)) -> x
   | (Float_kind, Cons (Float x, xs)) -> x
   | (_, Cons (_, xs)) -> get xs k

(* print out list *)
let rec print = function
 | Nil -> print_newline ()
 | Cons ((Int x), xs) -> Printf.printf "%d " x; print xs
 | Cons ((Float x), xs) -> Printf.printf "%f " x; print xs

(* testing *)
let empty = Nil
let l1 = Cons ((Int 1), empty)
let l2 = Cons ((Float 2.), l1)
let () = print l2
let i = get l2 Int_kind
let f = get l2 Float_kind;;

(*
                                                        2.000000 1
type _ kind = Int_kind : int kind | Float_kind : float kind
type value = Int of int | Float of float
type list = Nil : list | Cons : value * list -> list
val get : list -> 'a kind -> 'a = <fun>
val print : list -> unit = <fun>
val empty : list = Nil
val l1 : list = Cons (Int 1, Nil)
val l2 : list = Cons (Float 2., Cons (Int 1, Nil))
val i : int = 1
val f : float = 2.
*)
```

At first glance you might think : Why does that need GADTs ? Why not simply use

```
type value = Int of int | Float of float
```

for this ?
Take a close look at the get funktion :

```
val get : list -> 'a kind -> 'a = <fun>
```

It does not return a value but directly the unboxed type. Because the value is unboxed you can directly use it and ocaml will detect if you screw up the type like in :

```
let () = Printf.printf "%d\n" (get l2 Float_kind);;
Error: This expression has type float but an expression was expected
       of type int
```