Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2013-2014

TP n°2 - Correction

Usages avancés des types

Exercice 1

1. Parmi les deux expressions, laquelle est valide :

```
# let a = ‘B "foo" and b = ‘B 4;;
# let l = [‘B "foo"; ‘B 4];;
```

2. Quel est le type de `f` :

```
let f  (a:[< ‘A > ‘A]) = match a with ‘A -> 4;;
```

3. Restreignez le type de la liste :

```
let l = [‘A 4; ‘B "foo"];;
```

afin d'empêcher l'opération

```
let l’ = ‘C::l;;
```

4. Dans l'expression

```
‘A::[‘B];;
```

quel est le type de `‘A` ? Restreignez son type pour que l'opération de concaténation déclenche une erreur.

5. Soit la fonction `f` suivante :

```
# let f = function ‘A -> ‘B | _ -> ‘A;;
```

Restreignez le type de `f` afin de ne pouvoir utiliser comme argument de `f` que `‘A` ou `‘B`.

6. Grâce aux contraintes de type, écrivez deux fonction permettant de passer du type `[< ‘A]` au type `[‘A]`, et du type `[‘A]` au type `[> ‘A]`.

Correction :

1. La deuxième expression est incorrecte, car elle essaie de mélanger dans une même valeur deux constructeurs de même nom `‘B` mais avec des paramètres de type différent. Si OCaml autorisait ce mélange, on pourrait écrire

```
List.iter (function (‘B v) -> (* is ’v’ int or string ? *)) l
```

En contraste, la première définition manipule ces deux constructeurs de type différents, mais dans deux valeurs `a` et `b` distinctes, sans risque de confusion.

2. Le type de `f` est `[ ‘A ] -> int`. Il faut comprendre les deux contraintes comme une conjonction : les instances du type `[< ‘A > ‘A]` ont au moins le constructeur `‘A`, et au plus le constructeur `‘A` ; c'est exactement `[ ‘A ]`.

On retrouvera ces doubles contraintes dans des cas plus utiles, par exemple `[> ‘A < ‘A | ‘B ‘C]` peut s'instancier en les trois types clos `[ ‘A ]`, `[ ‘A | ‘B ]` et `[ ‘A | ‘B | ‘C ]`.

3. Il s'agit d'écrire

```
let (l : [ ‘A of int | ‘B of string ]) = ...
```

Notez qu'il n'est pas strictement nécessaire d'utiliser une annotation pour restreindre le type d'une variante polymorphe. On peut aussi la faire passer à travers un filtrage pour resteindre les constructeurs possibles :

```
# let a_or_b v = match v with
  | `A a -> v
  | `B b -> v;;
val a_or_b : ([< `A of 'b | `B of 'c ] as 'a) -> 'a = <fun>

# let l = List.map a_or_b [`A 4; `B "foo"];;
val l : [ `A of int | `B of string ] list = [`A 4; `B "foo"]
```

Le type de `` `A `` est `[> `A]` : tout type qui contient au moins le constructeur `` `A `` est accepté, et c'est pour cela que le mélange avec une liste de `[> `B]` est possible.

On peut restreindre le type en écrivant `` (`A : [`A]) `` par exemple pour empêcher l'ajout à la liste. On peut aussi explicitement indiquer des paramètres de type différent pour le constructeur `` `B ``.

4.
```
# (`A : [> `B of int ]) :: [`B];;
                             ^^
Error: This expression has type [> `B ]
       but an expression was expected of type [> `A | `B of int ]
       Types for tag `B are incompatible
```

5.
```
# let f : [< `A | `B ] -> _ = function
  | `A -> `B
  | _ -> `A;;
val f : [< `A | `B > `A ] -> [> `A | `B ] = <fun>
```

On peut remarquer que grâce à notre annotation, notre fonction accepte au plus les constructeurs `` `A `` ou `` `B ``, mais que le code du filtrage force le type entrée à accepter au moins `` `A ``. On rencontre une double contrainte comme mentionné dans la question 2.

6.
```
# let f `A : [ `A ] = `A;;;
val f : [< `A ] -> [ `A ] = <fun>
# let f (`A : [ `A ]) = `A;;
val f : [ `A ] -> [> `A ] = <fun>
```

Attention : dans le premier cas l'annotation concerne le type de retour, dans le second il s'agit du type de l'argument. Dans le cas général on peut écrire par exemple `let f (x : int) (y : bool) : char = ...` pour une fonction de type `int -> bool -> char`.

Correction :

1. La seconde (une liste doit contenir des variants polymorphes cohérents)
2. `val f : [ `A] -> int = <fun>`
3.
```
# let (l:[`A of int | `B of string ] list) = [`A 4; `B "foo"];;
val l : [ `A of int | `B of string ] list = [`A 4; `B "foo"]
# let l' = `C::l;;
Error: This expression has type [ `A of int | `B of string ] list
       but an expression was expected of type [> `C ] list
       The first variant type does not allow tag(s) `C
```

4. `` `A `` est de type `` [>`A] ``. En restreignant son type a lui même on obtient une errreur :

```
# (`A:[`A])::[`B];;
Error: This expression has type [> `B ]
       but an expression was expected of type [ `A ]
       The second variant type does not allow tag(s) `B
```

5. `# let (f:[`A|`B] -> [> `A | `B ]) = function `A -> `B | _ -> `A;;`
   `val f : [ `A | `B ] -> [> `A | `B ] = <fun>`

6.
```
let f `A = (`A:[`A]);;
let g (`A:[`A]) = `A;;
```

Exercice 2 On veut associer aux nombres flottants de OCaml un signe. Pour cela on utilise les variants polymorphes de façon à capturer plus d'erreurs.

Soit les fonctions suivantes permettant de créer zero, et des nombres positifs et négatifs à partir de flottants :

```
let zero = (`Nil, 0.0)
let positive flo = if flo > 0. then (`Pos, flo) else invalid_arg "positive"
let negative flo = if flo < 0. then (`Neg, flo) else invalid_arg "negative"
```

1. Ecrivez les fonctions `to_float`, `sqrt`, `exp`, et `log` de telle façon à ce qu'il ne soit pas possible d'écrire l'expression suivante (car sqrt est défini uniquement sur les nombres positifs ou nuls) :

   ```
   # let sqrt_log x = sqrt (log x)
   Error: This expression has type [> `Neg | `Nil | `Pos ] * float
          but an expression was expected of type [< `Nil | `Pos ] * float
          The second variant type does not allow tag(s) `Neg
   ```

2. Essayez d'écrire le même code sans utiliser les variants polymorphes. Peut-on arriver au même résultat pour `sqrt_log` ?
3. On veut maintenant vraiment écrire le code de `sqrt_log` (on pourra utiliser le type Option).

Correction :

1.
```
let to_float = snd

let sqrt = function `Nil, _ -> zero | `Pos, x -> (`Pos, sqrt x)

let exp t = (`Pos, exp (to_float t))

let log (`Pos, x) =
  let r = log x in
    if r = 0. then zero
    else if r < 0. then (`Neg, x) else (`Pos, x)

let sqrt_log x = sqrt (log x)
```

2. Si on définit :

   ```
   type floatsign = Nil | Pos of float | Neg of float
   ```

   et des fonctions du style :

   ```
   let sqrt = function
     | Nil-> zero
     | Pos  x -> Pos (sqrt x)
     | Neg x -> failwith "sqrt n'accepte que des flottants positifs"
   ```

   on n'arrive pas à capturer l'erreur précédente au moment du typage.

3.
```
type nonneg = [`Nil | `Pos]

let sqrt_log x =
  match log x with
```

```
  | `Neg, _ -> None
  | #nonneg, _ as y -> Some (sqrt y)
```

On obtient le type :

```
val sqrt_log : [ `Pos ] t -> [> `Nil | `Pos ] t option
```

Exercice 3 1. Ecrivez une fonction `f` qui convertit un couple en un couple avec un type différent :

```
f : [ `A ] * [ `B ] -> [ `A | `C ] * [ `B | `D ]
```

2. Soit le code suivant :

```
module M : sig
    type 'a t
    val embed : 'a -> 'a t
  end = struct
    type 'a t = 'a
    let embed x = x
end
```

On remarque que l'expression suivante n'est pas généralisée :

```
# M.embed [];;
- : '_a list M.t = <abstr>
```

Cependant `'a t` est utilisé uniquement en position positive. Modifiez le code ci-dessus afin d'obtenir :

```
# M.embed [];;
- : 'a list M.t = <abstr>
```

Correction :

1. `let f x = (x : [ `A] * [`B ] :> [ `A | `C ] * [ `B | `D ]);;`
2. Il suffit de rajouter un + devant `'a t`.