Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2013-2014

TP n°2

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]`.

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).

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>
   ```