# Programmation Fonctionnelle: Références & mutabilitité (encore)

Adrien Durier

03 octobre 2023

# Mutabilité

On peut modifier un champ mutable avec `<-`.

```
type student = { number : int; mutable age : int}

let birthday e = e.age <- e.age + 1

let e = { number = 12134; age = 21}

let ()= birthday e; print_int e.age
```

Affiche 22.

# Références et mutabilité

Les références ne sont rien de plus que des enregistrements à un seul champ mutable. Moralement, c'est comme si les définitions suivantes avaient été effectuées :

```
type 'a ref = { mutable content : 'a }

let ref = fun v ->  { content = v }
let (:=) = fun r v -> r.contents <- v
let (!) = fun r -> r.contents
```

# Égalité de références

Ne pas confondre :

```
type t = { prenom : string; mutable age : int }
```

```
type s = { prenom : string; age : int ref }
```

L'égalité = ne compare pas les addresses, mais leur contenu.

```
# ref 5 = ref 5 ;;
- : bool = true
```

L'égalité "physique" == compare les adresses :

```
# ref 5 == ref 5 ;;
- : bool = false
```

# Références et fermetures

```
# let compteur i =
    let etat = ref i in
    fun () -> etat := !etat + 1; !etat ;;
val compteur : int -> unit -> int
# let f = compteur 0 ;;
val f : unit -> int = <fun>
# f () ;;
- : int = 1
# f () ;;
- : int = 2
# let g = compteur 10 ;;
val g : unit -> int = <fun>
# g() ;;
- : int = 11
```

- Une fermeture est une fonction dotée d'un état interne.
- La référence, constante, pointe vers cet état, changeant.
- Ici f et g ont chacune leur état.

# Références et fermetures

Ces fonctions n'allouent pas la mémoire au même moment :

```
let f = let x = ref 3 in fun () -> x
let g = fun () -> let x = ref 3 in x
```

Que contient b après :

```
# let a = f() ;;
# let b = f() ;;
# a := !a + 1 ;;
# !b ;;        (* b vaut 4 *)
```

Que contient d après :

```
# let c = g() ;;
# let d = g() ;;
# c := !c + 1 ;;
# !d ;;        (* d vaut 3 *)
```

# Références et variables de types monomorphes

La liste vide est une `'a list`, qui peut être utilisée comme une `int list` ou une `float list` sans problème, son type est polymorphe :

```
# let l = [] ;;
# l2 := 4 :: l ;;
# (function [] -> 0. | x :: _ -> x +. 0.5) l ;;
```

Mais la même tolérance envers une référence vers une liste vide conduirait à des bugs :

```
# let l = ref [] ;;
# l := 4 :: !l ;;
# (function [] -> 0. | x :: _ -> x +. 0.5) !l ;;
```

Une référence vers un type qui aurait pu être polymorphe `'a list` doit en fait être monomorphe `'_a list`.

# Références et variables de types monomorphes

```
# let l = ref []
val l : '_a list ref = ...
```

La variable de type monomorphe prendra, par la suite, une et une seule valeur de type, au fur et à mesure que de nouvelles contraintes apparaissent.

```
# l := 4 :: !l ;;
# l ;;
- : int list ref = ...
```

# Exercice

Pour chacune des expressions suivantes, dire si elles sont bien typées et, si oui, donner leur type :

```
let f1 x = !x
let f2 g x y = if g x then y x else g
let f3 h = let x = ref true in if h x then x:= false; !x
```