

Programmation Fonctionnelle: Typage

Adrien Durier

13 septembre 2023

Typage

Pourquoi le typage statique ?

- OCaml est fortement et statiquement typé.

```
# 4 + 3 ;;  
- : int = 7  
  
# 4. +. 3. ;;  
- : float = 7.  
  
# 4. + 3 ;;
```

Error: This expression has **type float** but an expression was expected

Pourquoi le typage statique ?

- OCaml est fortement et statiquement typé.

```
# 4 + 3 ;;  
- : int = 7  
# 4. +. 3. ;;  
- : float = 7.  
# 4. + 3 ;;
```

Error: This expression has **type float** but an expression was expected



- Erreurs coûteuses :
 - Exemple de la NASA : Mars Climate Orbiter (1999)
 - Erreur de conversion unité : newton-seconde et livre-seconde

Pourquoi le typage statique ?

- OCaml est fortement et statiquement typé.

```
# 4 + 3 ;;  
- : int = 7  
# 4. +. 3. ;;  
- : float = 7.  
# 4. + 3 ;;
```

Error: This expression has **type float** but an expression was expected

- Erreurs coûteuses :
 - Exemple de la NASA : Mars Climate Orbiter (1999)
 - Erreur de conversion unité : newton-seconde et livre-seconde
- Le typage revient à la mode !
 -  Rust
 -  TypeScript

Types simples

Expressions de type `int` : les entiers

```
# 4 + 1 - 2 * 2 ;;  
- : int = 1  
# 5 / 2 ;;  
- : int = 2  
# 1_000_005 mod 2 ;;  
- : int = 1  
# max_int + 1 ;;  
- : int = -4611686018427387904
```

- `int` représente les entiers compris entre -2^{62} et $2^{62} - 1$ (sur une machine 64 bits)
- opérations sur ce type : `+`, `-`, `*`, `/` (division entière), `mod` (reste de la division) etc.

Types simples

Expressions de type **float** : les nombres à virgule flottante

```
# 4.3e4 +. 1.2 *. -2.3 ;;  
- : float = 42997.24  
# 5. /. 2. ;;  
- : float = 2.5  
# 1. /. 0. ;;  
- : float = infinity  
# 0. /. 0. ;;  
- : float = nan
```

- les types int et float sont disjoints, la moitié de vos bugs seront des points (.) manquants.
- opérations sur ce type : +. -. *. /. sqrt cos etc.
- on passe d'un entier à un flottant à l'aide de la fonction float_of_int et inversement avec truncate

Types simples : Booléens

```
# false || true ;;  
- : bool = true  
# 3 <= 1 ;;  
- : bool = false  
# not (0=2) && 1>=3 ;;  
- : bool = false  
# if 2<0 then 2.0 else (4.6 *. 1.2) ;;  
- : float = 5.52
```

- les constantes sont true (vrai) et false (faux)
- les opérations sur ce type not (non), && (et) et || (ou)
- les opérateurs de comparaison (=, <, >, <=, >=) retournent des valeurs booléennes
- dans une conditionnelle de la forme

if exp_1 then exp_2 else exp_3

l'expression exp_1 doit être de type bool.

Expressions de type `char` : les caractères

```
# 'a' ;;  
- : char = 'a'  
# int_of_char 'a' ;;  
- : int = 97  
# char_of_int 100 ;;  
- : char = 'd'
```

- les caractères sont encadrés par deux apostrophes '
- la fonction `int_of_char` renvoie le code ASCII d'un caractère (et inversement pour la fonction `char_of_int`).

Types simples

Expressions de type `string` : les chaînes de caractères

```
# "hello" ;;  
- : string = "hello"  
# "" ;;  
- : string = ""  
# "hello".[1] ;;  
- : char = 'e'  
# string_of_int 123 ;;  
- : string = "123"
```

- ces valeurs sont encadrées par deux guillemets `"`
- l'opérateur `^` concatène des chaînes
- l'opération `.[i]` accède au *i*ème caractère d'une chaîne (le premier caractère est à l'indice 0)

Types simples

```
# ( ) ;;  
- : unit = ()  
# Print.printf "bonjour\n" ;;  
bonjour  
- : unit = ()
```

- unit représente les expressions qui font uniquement des effets de bord
- une seule valeur a ce type, elle est notée ()
- c'est l'équivalent du type void en C

Typage Avancé

```
let u = (1, 2)

let f (x, y, z) w = if x < y then w + 10 else z

let rec division n m =
  if n < m then (0, n)
  else
    let (q,r) = division (n - m) m in
    (q + 1, r)

let v =
  ("Durand", "Jacques",
   ("2 rue J.Monod", "Orsay Cedex", 91893),
   (10,03,1967), "0130452637", "0645362738")
```

Enregistrements

```
type complexe = { re : float; im : float }
```

```
let v = { re = 1.3; im = 0.9 }
```

```
let add z1 z2 =  
  { re = z1.re +. z2.re;  
    im = z1.im +. z2.im }
```

```
let conjugue z = { z with im = -. z.im }
```

```
type t = V | N of int * t

let l1 = N(1, N(2, N(3, V)))

let rec f l =
  match l with
  | V -> 0
  | N(x, s) -> x + f s

let r = f l1
```

Inférence de type et polymorphisme

```
let f1 x f = if x > 0 then f x else "OK" ;;  
val f1 : int -> (int -> string) -> string = <fun>
```


Inférence de type et polymorphisme

```
let f1 x f = if x > 0 then f x else "OK" ;;  
val f1 : int -> (int -> string) -> string = <fun>
```

```
let f2 x = x ;;  
val f2 : 'a -> 'a = <fun>
```

Inférence de type et polymorphisme

```
let f1 x f = if x > 0 then f x else "OK" ;;  
val f1 : int -> (int -> string) -> string = <fun>
```

```
let f2 x = x ;;  
val f2 : 'a -> 'a = <fun>
```

```
let f3 f g x = f (g x) ;;  
val f3 : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```