

# OCaml: le noyau fonctionnel

David Delahaye

Faculté des Sciences  
[David.Delahaye@lirmm.fr](mailto:David.Delahaye@lirmm.fr)

Licence L3 2019-2020

# De l'importance de la programmation fonctionnelle

## John Carmack (id Software)

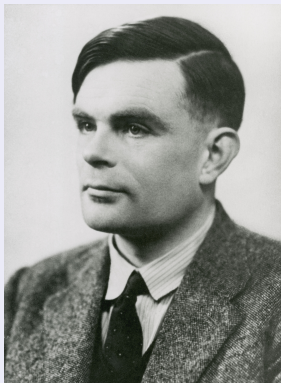
- *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*



## Machines de Turing et programmation impérative

Machines de Turing :

- Fondement théorique des ordinateurs modernes et de la programmation impérative ;
- Bande  $\equiv$  mémoire adressable en lecture/écriture avec un programme stocké ;
- Automate  $\equiv$  microprocesseur.



## $\lambda$ -calcul et programmation fonctionnelle

$\lambda$ -calcul :

- $\lambda x.M$  = fonction anonyme avec le paramètre formel  $x$  et le corps  $M$  (abstraction) ;
- $M N$  = appel de la fonction  $M$  avec le paramètre effectif  $N$  (application) ;
- Règle de calcul ( $\beta$ -réduction) :  $(\lambda x.M) N \rightarrow_{\beta} M[x := N]$ .



# Thèse de Church-Turing

## Équivalence des machines de Turing et du $\lambda$ -calcul (Turing, 1937)

- Une fonction est calculable par une machine de Turing, si et seulement si elle est calculable en utilisant le  $\lambda$ -calcul.

## Thèse de Church-Turing

- Une fonction calculable par n'importe quelle méthode effective de calcul est aussi calculable par une machine de Turing.

## En d'autres termes

- Tous les langages de programmation généralistes ont le même pouvoir de calcul et sont donc équivalents du point de vue de la calculabilité.

# Mais les langages de programmation ne sont pas nés égaux

## Différents pouvoirs d'expression

- Différentes représentations des données ;
- Différents modèles d'exécution ;
- Différents mécanismes d'abstraction.

## D'autres caractéristiques désirables

- Sûreté de l'exécution ;
- Efficacité de l'exécution ;
- Maintenabilité du code.

# La programmation fonctionnelle en pleine ascension

## Une citation d'un des pionniers de FORTRAN

- John Backus, Turing lecture, 1978 : *Can programming be liberated from the von Neumann style ?*
- *Functional programs deal with structured data, ... do not name their arguments, and do not require the complex machinery of procedure declarations ...*

## Besoin d'une plus grande sûreté de fonctionnement des programmes

- Notamment dans le domaine des applications critiques ;
- Il est plus facile de prouver la correction de programmes fonctionnels (purs) que de programmes impératifs.

## Des fonctions tout autour de nous

- Java 1.8 a introduit les lambda expressions ;
- C++ version 11 a introduit les lambda expressions.

# Nous allons apprendre OCaml !

## Histoire d'OCaml

- 1978 : langage ML (Milner) ;
- 1980 : projet Inria Formel (Huet) ;
- 1985 : « Categorical Abstract Machine » (Cousineau, Curien, Mauny) ;
- 1987 : première release de Caml (Suarez) ;
- 1988-1992 : Caml prend de l'ampleur (Mauny, Weis) ;
- 1990-1991 : machine Zinc, Caml Light (Leroy, Doligez) ;
- 1995 : ajout des modules, Caml Special Light (Leroy) ;
- 1996 : ajout des objets, Objective Caml (Vouillon, Rémy) ;
- 2000 : merge avec la branche Objective Label (Guarrigue) ;
- 2011 : le nom devient définitivement OCaml.



# Les « plus » d'OCaml

## Spécificités

- Typage statique et inférence de type ;
- Typage fort (correct vis-à-vis de la sémantique) :
  - ▶ Un programme qui compile ne fera aucune erreur de type à l'exécution ;
  - ▶ Robin Milner : *Well-typed programs cannot go wrong*.
- Polymorphisme et ordre supérieur ;
- Types structurés :
  - ▶ Types tuples ;
  - ▶ Types concrets ;
  - ▶ Enregistrements.
- Langage modulaire :
  - ▶ Objets ;
  - ▶ Modules.
- « Garbage collector ».

# Un système mature

## Un ensemble riche d'outils de développement

- `ocaml` : boucle interactive (« toplevel ») ;
- `ocamlc` : compilateur bytecode (code portable) ;
- `ocamlopt` : compilateur natif (AMD64, IA32, Power PC, ARM) ;
- `opam` : gestionnaire de paquets OCaml ;
- `js_of_ocaml` : compilateur vers JavaScript (applications Web).

## Installation

- Pour l'essayer : Try OCaml, <https://try.ocamlpro.com/> ;
- Pour l'installer : <https://ocaml.org/>.

# Plan du cours

## 3 semaines de cours

- ❶ *Noyau fonctionnel* ;
- ❷ Objets simples (héritage simple et multiple, sous-typage) ;
- ❸ Objets avancés (types ouverts, contraintes, « self-types »).

# Expressions

## Dans la boucle interactive

*OCaml version 4.06.1*

```
# 1 + 1;;  
- : int = 2  
# 2.5 *. 3.7;;  
- : float = 9.25  
# true || false;;  
- : bool = true  
# char_of_int 65;;  
- : char = 'A'  
# "Bonjour" ^ "HLIN603!";;  
- : string = "BonjourHLIN603!"  
# [1; 2; 3; 4; 5];;  
- : int list = [1; 2; 3; 4; 5]
```

# Attention au typage !

## Quelques exemples d'erreurs de typage

```
# 42 + "HLIN603";;
```

*Error: This expression has **type** string but an expression was expected **of type** int*

```
# 1 + 1.;;
```

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

```
# 1 +. 1.;;
```

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

```
# 1 + (int_of_float 1.);;
```

```
- : int = 2
```

```
# (float_of_int 1)+.1.;;
```

```
- : float = 2.
```

# Types primitifs

## Types (avec opérations et modules correspondants)

Type	Opérations	Modules
int (31/63 bits)	+, -, *, /, mod, abs	Pervasives
float (64 bits)	+, -, *, /, **, sqrt, exp, log, cos, sin, tan	Pervasives
bool	not, &&,   , =, <>, <, >, <=, >=, ==, !=	Pervasives
char	int_of_char, char_of_int	Pervasives, Char
string	^	Pervasives, String
list	@	Pervasives, List

# Expressions

## Conditionnelles

```
# if 1<2 then 6 + 7 else 67 / 23;;
```

```
- : int = 13
```

```
# if 6=8 then 1 else 77.5;;
```

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

```
# (if 6 = 3 + 3 then 3 < 4 else 8 > 7) && 67.8 > 33.1;;
```

```
- : bool = true
```

```
# if (if 1 = 1 then 2 = 2 else 4.0 > 3.2) then 2 < 3  
else 3 < 2;;
```

```
- : bool = true
```

# Expressions

## Fonctions (anonymes) et applications

```
# fun x → x + 1;;  
- : int → int = <fun>  
# (fun x → x + 1) 1;;  
- : int = 2  
# fun x y → x + y;;  
- : int → int → int = <fun>  
# (fun x y → x + y) 1 1;;  
- : int = 2
```



# Expressions

## Déclarations locales (simples et imbriquées)

```
# let x = 4 + 5 in 2 * x;;
```

```
– : int = 18
```

```
# x;;
```

```
Error: Unbound value x
```

```
# let x = 4 in
```

```
  let y = x + 1 in
```

```
    let z = 2 * y in z;;
```

```
– : int = 10
```

```
# let x = 4 in
```

```
  let y = x + 1 in
```

```
    let x = 2 * y in x;;
```

```
– : int = 10
```

# Expressions

## Déclarations locales (simultanées)

```
# let x = 1
  and y = 1 in x + y;;
- : int = 2
# let x = 1
  and y = x in y;;
Error: Unbound value x
```

# Déclarations globales

## Modification de l'environnement

```
# let x = 2 + 3;;  
val x : int = 5  
# x;;  
- : int = 5  
# let y = 2 * x;;  
val y : int = 10  
# y;;  
- : int = 10  
# let x = 42;;  
val x : int = 42  
# x;;  
- : int = 42  
# let x = true;;  
val x : bool = true  
# x;;  
- : bool = true
```

# Déclarations globales

## Fonctions

```
# let succ = fun x → x + 1;;  
val succ : int → int = <fun>  
# succ 1;;  
- : int = 2  
# let succ x = x + 1;;  
val succ : int → int = <fun>  
# succ 1;;  
- : int = 2  
# let add x y = x + y;;  
val add : int → int → int = <fun>  
# add 1 1;;  
- : int = 2  
# let add x = x + y;;  
Error: Unbound value y
```

# Déclarations globales

## Application partielle

```
# add 1;;  
- : int → int = <fun>  
# let succ = add 1;;  
val succ : int → int = <fun>  
# succ 1;;  
- : int = 2
```

# Déclarations globales

## Fonctions polymorphes

```
# let id x = x;;  
val id : 'a → 'a = <fun>  
# id 1;;  
- : int = 1  
# id true;;  
- : bool = true  
# let cons_last x l = l @ [x];;  
val cons_last : 'a → 'a list → 'a list = <fun>  
# cons_last 5 [1; 2; 3; 4];;  
- : int list = [1; 2; 3; 4; 5]  
# cons_last 'e' ['a'; 'b'; 'c'; 'd'];;  
- : char list = ['a'; 'b'; 'c'; 'd'; 'e']
```

# Déclarations globales

## Ordre supérieur

```
# let comp f g x = f (g x);;  
val comp : ('a → 'b) → ('c → 'a) → 'c → 'b = <fun>  
# let succ x = x + 1;;  
val succ : int → int = <fun>  
# let sqr x = x * x;;  
val sqr : int → int = <fun>  
# comp sqr succ 1;;  
- : int = 4
```

# Déclarations globales

## Fonctions récursives

```
# let rec fact n =  
  if n = 0 then 1  
  else n * fact (n - 1);;  
val fact : int → int = <fun>  
# fact 3;;  
- : int = 6  
# let rec size l =  
  if l = [] then 0  
  else 1 + size (List.tl l);;  
val size : 'a list → int = <fun>  
# size [1; 2; 3; 4; 5];;  
- : int = 5
```



# Déclarations globales

## Fonctions sur les listes

```
# let rec size l =  
  match l with  
  | [] → 0  
  | e :: tl → 1 + (size tl);;  
val size : 'a list → int = <fun>  
# size [1; 2; 3; 4; 5];;  
# let rec size = function  
  | [] → 0  
  | _ :: tl → 1 + (size tl);;  
val size : 'a list → int = <fun>  
# size [1; 2; 3; 4; 5];;  
- : int = 5
```

# Déclarations globales

## Fonctions sur les listes

```
# let rec incr_list = function
  | [] → []
  | e :: tl → (e + 1) :: (incr_list tl);;
val incr_list : int list → int list = <fun>
# incr_list [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
# let rec map f = function
  | [] → []
  | e :: tl → (f e) :: (map f tl);;
val map : ('a → 'b) → 'a list → 'b list = <fun>
# map (fun x → x + 1) [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

# Types concrets (types sommes)

## Définition et valeurs des types concrets

```
# type number =  
  | Int of int  
  | Float of float;;  
type number = Int of int | Float of float
```

```
# let x = Int 1;;  
val x : number = Int 1  
# let y = Float 1.5;;  
val y : number = Float 1.5
```

```
# Int 1.5;;  
Error: This expression has type float but an expression  
      was expected of type int
```

```
# Float 1;;  
Error: This expression has type int but an expression was  
      expected of type float
```

# Types concrets (types sommes)

## Fonctions sur les types concrets

```
# let l = [x; y];;  
val l : number list = [Int 1; Float 1.5]  
# let float_of_number = function  
| Int n → float_of_int n  
| Float f → f;;  
val float_of_number : number → float = <fun>  
# let rec sum_number_list = function  
| [] → 0.  
| e :: tl → (float_of_number e) +.  
              (sum_number_list tl);;  
val sum_number_list : number list → float = <fun>  
# sum_number_list l;;  
- : float = 2.5
```

# Types concrets (types sommes)

## Types concrets récurifs polymorphes

```
# type 'a tree =  
  | Leaf of 'a  
  | Node of 'a * 'a tree * 'a tree;;  
type 'a tree = Leaf of 'a  
             | Node of 'a * 'a tree * 'a tree  
# let t = Node (1, Leaf 2, Node (3, Leaf 4, Leaf 5));;  
val t : int tree = Node (1, Leaf 2,  
                          Node (3, Leaf 4, Leaf 5))
```

## Types concrets (types sommes)

### Types concrets rékursifs polymorphes

```
# let rec depth = function
  | Leaf _ → 1
  | Node (_, l, r) → 1 + (max (depth l) (depth r));;
val depth : 'a tree → int = <fun>
# depth t;;
- : int = 3
```