Programmation Fonctionnelle

C3P

Vincent Aranega
vincent.aranega@univ-lille.fr

Université de Lille

Table of contents

Table of Contents

C'est quoi la programmation fonctionnelle ?

Des idées ?

C'est quoi la programmation fonctionnelle ?

La programmation fonctionnelle est plus qu'un ensemble de principes. C'est un paradigme, une façon différente de penser. Il met l'accent sur le "quoi", par opposition au "comment", comme dans les paradigmes orientés objet et procéduraux.

C'est quoi la programmation fonctionnelle ?

- Composition de fonction pures
- On évite:
  - les états partagés
  - les données mutables
  - les effets de bord
- Déclaratif plutôt qu'impératif
- L'état de l'application passe par des fonctions
- Paradigme de programmation
  $\hookrightarrow$ il s'agit d'une façon de penser la construction d'un logiciel basée sur certains principes fondamentaux et définis

Quelles conséquences ?

En pratique

- On travaille par composition de fonctions
- On utilise essentiellement la récursivité
- Les données manipulées ne sont pas mutables

En résumé

La programmation fonctionnelle met en avant la définition, l'évaluation de **fonctions** et l'**immutabilité** des données.

Exemple – Calcul de 8! en style fonctionnel

```
let rec fact = function
| 0 -> 1
| n -> n * fact (n - 1)
in fact 8;;
```

Histoire de Caml

- Langage développé à l'Inria (équipes Formel, Cristal, puis Gallium)
- Recherche sur les systèmes de types:
  - Inférence de types, typage statique,
  - un programme bien typé $\rightarrow$ sûreté d'exécution,
  - accepter le plus de programmes corrects possibles, en rejetant tous les programmes incorrects.

1985 langage CAML

1990 Caml-light

1995 OCaml

2002 Création du consortium Caml (CEA, Citrix, Dassault, Esterel Technologies, Microsoft, ...)

Premiers exemples

```
# repeated "hello" 3;;
- : string = "hello hello hello"

# let average a b = (a +. b) /. 2.0;;
val average : float -> float -> float = <fun>

# average (average 4. 0.) 2.;;
- : float = 2.

# 3 = 3 && 4 <> 5;;
- : bool = true
```

Table of Contents

Typage fort

- Le typage d'une fonction est réalisé au moment de sa définition
- les conversions implicites de types sont interdites

```
# 1.0 + 2;;
Error: This expression has type float but an expression
    was expected of type int

# 1.0 +. float_of_int 2;;
- : float = 3.0

# "My first string " ^ 3;;
Error: This expression has type int but an expression was
    expected of type string

# "My first string " ^ (string_of_int 3);;
- : string = "My first string 3"
```

Inférence de types

Caml déduit tout seul le type d'une expression en fonction des opérateurs, des litéraux utilisées ainsi que des fonction connues dans l'environnement

Exemple – Fonctions anonymes

```
# fun x -> x + x ;;
- : int -> int = <fun>

# fun x -> x +. x ;;
- : float -> float = <fun>
```

Définitions globales

On attribut un nom à une valeur en utilisant l'instruction `let` (une définition).

Exemple – Définitions globales

```
# let n = 12;;
val n : int = 12

# let f = function x -> x + x;;
val f : int -> int = fun

# f n;;
- : int = 24
```

Les valeurs définies en utilisant `let` sont visible globalement

Définitions locales

On attribut un nom localement en utilisant l'instruction `let...in`.

Exemple – Définitions locales

```
# let n = 12 in n;;
- : int = 12

# n;;
Error: Unbound value n

# let f = function x -> x ** 2. in f 3.;;
- : float = 9
```

Définitions multiples

On peut effectuer plusieurs définitions simultanément en utilisant and.

Exemple – Définitions locale multiple

```
# let x = 12 and y = 45 in x + y;;
- : int = 57
```

Attention, on ne peut pas utiliser une valeur avant qu'elle soit définie.

```
# let x = 12 and y = x + 3 in x + y;;
Error: Unbound value x

# let x = 12 in let y = x + 3 in x + y;;
- : int = 27
```

Fonctions

On définit une fonction par la syntaxe
`let fname arg = expr`

Exemple – Définitions et application de fonction

```
# let foo x = (int_of_float x) * 4;;
val foo : float -> int = <fun>

# foo 3.;;
- : int = 12

# let carre x =
    let ix = int_of_float x
    in ix * ix;;
val carre : float -> int = <fun>
```

Fonctions à plusieurs arguments

On définit une fonction avec plusieurs arguments par la syntaxe `let fname args = expr` ou `args` sont les arguments séparés par des espaces

Exemple – Définitions et application de fonction

```
# let bar x y = x + y;;
val bar : int -> int -> int = <fun>

# bar 3 4;;  (* Pas de parentheses pour les arguments *)
- : int = 7
```

L'application de `bar 3 4` est équivalent à `(bar 3) 4`

Exemple – Fonction partielle

```
# let temp = bar 3;;  (* Application partielle *)
val temp : int -> int = <fun>

# temp 4;;
- : int = 7
```

Conditions

Les conditions s'expriment à l'aide de la structure
`if ... then ... else ...`

```
# let n = 3;;
val n : int = 3

# let x = if n = 3 then "OK" else "KO";;
val x : string = "OK"
```

Remarque

- Les types de retours des branches du `if` doivent être obligatoirement du même type

Exercice

- Donnez la signature et ecrivez une fonction qui retourne la valeur absolue d'un nombre

Type tuple

Les membres d'un n-uples sont séparés par des virgules et représente une seule valeur

```
# (1, 2, 3)
- : int * int * int = 1, 2, 3

# ((1, 2), 3)
- : (int * int) * int = (1, 2), 3

# let f (x, y) = x + y;;
- : int * int -> int = <fun>
```

Fonctions recursives

Le mot clé `rec` indique la définition d'un objet récursif, c'est-à-dire un objet dont le nom intervient dans sa propre définition.

Exemple – Une fonction factorielle

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1);;
```

Exercices

- Donner la définition d'une fonction sommielle
- Donner la définition d'une fonction qui additionne tout les nombres pair compris entre 0 et l'argument

Polymorphisme et inférence de type

Une fonction polymorphe s'applique à tous les types. La notation `'a, 'b, 'c, ...` représente un type en particulier.

```
# let f a b = a;;
val f : 'a -> 'b -> 'a = <fun>

# let g a b = b;;
val g : 'a -> 'b -> 'b = <fun>

# let foo a b = 1 + (f a b);;
val g : int -> 'a -> int = <fun>
```

Fonctions – Exercices

Donner une fonctions qui correspondent aux types suivants

```
(int -> int) -> int
int -> (int -> int)
int -> int -> int
int -> (int -> int) -> int
```

Quels sont les types des fonctions suivantes?

```
# let foo f x y = f x y;;
# let foo f g x = g (f x);;
# let foo f g x = (f x) + (g x);;
```

Fonctions comme arguments de fonctions

Les fonctions sont des éléments de premier ordre en OCaml, ils peuvent être passés comme arguments d'autres fonctions

```
# let foo f x = (f (x + 0)) + 0;;
val foo : (int -> int) -> int -> int = <fun>

# let myfun x = x * 2;;
va myfun : int -> int = <fun>

# foo myfun 2;;
- : int = 4
```

Fonctions comme arguments de fonctions

- Quels intérêts voyez vous à l'utilisation de fonctions comme paramètres d'une autre ?

Fonctions comme arguments de fonctions

- Quels intérêts voyez vous à l'utilisation de fonctions comme paramètres d'une autre ?

- l'écriture de programmes plus concis
- l'écriture de programmes plus faciles à étendre/maintenir

```
let s_long long_fun x callback = callback (long_fun x);;
let cb res = print_string ("Computation is finished " ^
    (string_of_int res));;

s_long factorial 1000000000 cb;;
```

Fonctions anonymes

Exemple – Fonction anonyme

```
# fun x -> x + 1;;
-: int -> int = <fun>

# let f = fun x -> x + 1;;
val f : int -> int = <fun>

# fun x -> fun y -> x + y;;
-: int -> int -> int = <fun>

# let g = fun x -> fun y -> x + y;;
val g : int -> int -> int = <fun>
```

Remarque

- Le typage est associatif à droite, donc
  int -> (int -> int) est équivalent à int -> int -> int

Passage de fonction anonyme en paramètre

Exemple – Fonction anonyme

```
# let decorate f x pre post =
    let _ = pre x
    in let res = f x
       in post x res;;
val decorate : ('a -> 'b) -> 'a -> ('a -> 'c) -> ('a ->
    'b -> 'd) -> 'd = <fun>

# decorate (fun x -> x + 1) 6
    (fun x ->
      print_endline ("args: " ^ string_of_int x))
    (fun x res ->
      let _ = print_endline ("res: " ^ (string_of_int
          res))
      in res);;
args: 6
res: 7
- : int = 7
```

Notes sur les fonctions

Nous avons toujours manipulé des fonctions dites curryfiées, c'est à dire des fonctions qui peuvent se décomposer en plusieurs fonctions unaires. La curryfication d'une fonction permet son application partielle.

```
# (fun x y -> x + y) 4;;   (* Fonction curryfie *)
- : int -> int = <fun>

# (fun (x, y) -> x + y) 4;;  (* Fonction non curryfie *)
Error: This expression has type int but an expression was
    expected of type int * int
```

Intérêts de la Curryfication

- Définition d'une fonction f qui prend 'a et 'b et renvoie 'c
- Deux façon de faire:
  - Sans curryfication f : 'a * 'b -> 'c
    - f a un seul paramètre (un couple)
    - f (a,b) est de type 'c
  - Avec curryfication f : 'a -> 'b -> 'c
    - f a deux paramètres
    - f a b est de type 'c
    - f a est de type 'b -> 'c

```
# (+);;
- : int -> int -> int = <fun>

# let plus2 = (+) 2;;
plus2 : int -> int = <fun>

# (+) 2 5;;  (* ((+) 2) 5 *)
- : int = 7
```

Type Liste

Les listes sont des collections d'éléments du même type et sont construites par la syntaxe `[e1; e2; e3; ...; en]`.

```
# [1; 2; 3]
- : int list = [1; 2; 3]

# [[1; 2]; [3]];;
- : int list list = [[1; 2]; [3]]

# [(1, [2]); (3, [5; 6])];;
- : (int * int list) list = [(1, [2]); (3, [5; 6])]
```

Fonctions et opérateurs sur les listes

Deux fonctions de base sont utilisés pour récupérer la tête de la liste et son reste

```
# let l = [12; 99; 37];;
val l : int list = [12; 99; 37]

# List.hd l;;
- : int = 12

# List.tl (List.tl l);;
- : int list = [37]
```

Les opérateurs `::` and `@` permettent de concaténer un élément à une liste ou de concaténer une liste à une autre

```
# let l = 12 :: 99 :: 37 :: [];;
val l : int list = [12; 99; 37]

# [1; 3; 4] @ l;;
- : int list = [1; 3; 4; 12; 99; 37]
```

Listes – Exercices

Exercices

- Écrivez une fonction qui calcule la longueur d'une liste
- Écrivez une fonction qui vérifie si un élément appartient à une liste

Filtrage par motifs

- Nouvelle forme qui permet de se rapprocher de l'écriture mathématique
- Nécessaire d'écrire toute les formes d'arguments que la fonction est suceptible de traiter
- OCaml sélectionne la première forme qui correspond et exécute l'expression associée

Exemple – Syntaxe du filtrage de motif

```
match expr with
| motif1 -> expr1
| motif2 -> expr2
| ................
| motifn -> exprn
```

Filtrage par motifs

Exemple – Retour sur la factorielle

```
# let rec fact n = match n with
| 0 -> 1
| n -> n * fact (n - 1);;
val fact : int -> int = <fun>
```

Si aucun motif n'est reconnu, une exception `Match_failure` est levée et les motifs non complets lèvent un warning.

```
# match 4 with
  | 0 -> true
  | 1 -> false ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
2
Exception: Match_failure ("//toplevel//", 1, 0).
```

Exemples

- Fonctions `et` et `ou` à partir d'entiers

Filtrage par motifs avec garde

Il est possible d'ajouter une garde sur un motif avec le mot clef when

```
# let foo n = match n with
| 0 | 1 | 2 -> 1  (* plusieurs motifs sur une ligne *)
| n when n < 50 -> 2
| n -> n;;
val foo : int -> int = <fun>
```

Filtrage par motifs sur les tuples

Il est possible d'exprimer des motifs par rapport à des tuples et à ses valeurs

Exemple – Filtrage sur tuples

```
# let add x = match x with
| ((0, 0), (1, 1)) -> 99
| ((0, 0), (1, _)) -> 50
| ((a, b), r) -> let (c, d) = r
                 in a + b + c + d;;
val add : (int * int) * (int * int) -> int = <fun>
```

Filtrage par motifs sur les listes

Il est possible d'exprimer des motifs par rapport à des listes et à leur tête et reste

Exemple – Filtrage sur liste

```
# let is_len2 x = match x with
| [] -> false
| [_] -> false
| [_; _] -> true
| _ -> false;;
val is_len2 : 'a list -> bool = <fun>

# let rec is_even_len x = match x with
| [] -> true
| a::b::r -> is_even_len r
| a::r -> false;;
val is_even_len : 'a list -> bool = <fun>
```

Filtrage par motifs sur les listes – Exercices

- Retour sur la fonction `longeur` et `appartient`
- Écrivez une fonction `gen_list` qui génère une liste des n premiers éléments
- Écrivez une fonction `maximum` qui retourne le plus grand élément d'une liste

Les types somme

Les types somme permettent d'énumérer les valeurs que peuvent prendre nos variables

Exemple – Définition de type somme

```
# type montype = A | B;;
type montype = A | B

# let a = A ;;
val a : montype = A

# type calcexpr = Negative of int | Addition of int *
    int;;
type calcexpr = Negative of int | Addition of int * int;;

# let e1 = Negative 3 ;;
val e1 : calcexpr = Negative 3

# let e2 = Addition (3, 4) ;;
val e1 : calcexpr = Addition (3, 4)
```

Filtrage de type somme

Il est aussi possible de filtrer sur les constructeurs d'un type somme

Exemple – Définition de type somme

```
# type calcexpr = Negative of int | Addition of int *
    int;;
type calcexpr = Negative of int | Addition of int * int

# let eval expr = match expr with
| Negative x -> - x
| Addition (a, b) -> a + b;;
val eval : calcexpr -> int = <fun>

# eval (Negative 4);;
- : int = -4

# eval (Addition (1, 2));;
- : int = 3
```

Type somme récursif

Exemple – Définition de type somme

```
# type calcexpr =
| Valeur of int
| Negative of calcexpr
| Addition of calcexpr * calcexpr;;
type calcexpr = Valeur of int | ...

# let expr = Negative (Addition (Negative (Valeur 3),
    Valeur 6));;
val expr : calcexpr = Negative ... <fun>
```

- Donner la définition de la nouvelle version de `eval`

Type enregistrement

Le type enregistrement permet de décrire des données structurées à l'image des structures en C

Exemple – Définition de type enregistrement

```
# type user = {name: string; birth: int * int * int};;
type user = name : string; birth : int * int * int;

# let u1 = {name = "User 1"; birth = (15, 1, 1950)};;
val u1 : user = name = "User 1"; birth = (15, 1, 1950)

# u1.name;;
- : string = "User 1"
```

Les champs d'un type enregistrement ne sont pas mutables, si on veut modifier le champ d'un `user`, il faut en créer un nouveau

```
# let u1 = {name = u1.name; birth = (16, 1, 1950)};;
val u1 : user = name = "User 1"; birth = (15, 1, 1950)
```

Filtrage de type enregistrement

Il est possible de filter les types enregistrement et d'appliquer aussi le filtrage d'autres types pour les champs de l'enregistrement

Exemple – Filtrage de type enregistrement

```
# let build_century user =
  match user with
| {name = name; birthday = (_, _, d)} when d < 1970
  -> name ^ " est super vieux"

| {name = name; birthday = (_, _, d)} when d < 1980
  -> name ^ "est assez vieux"

| {name = name; birthday = date}
  -> let _, _, d = date
     in name ^ " a " ^ (string_of_int (2019 - d));;
val build_century : user -> string = <fun>
```

Table of Contents

Test unitaire avec OUnit

OUnit est un framework pour l'écriture de tests en OCaml. Le workflow de base pour l'utilisation d'OUnit est le suivant :

- Écrire une fonction dans un fichier `f.ml` (il pourrait y avoir d'autres fonctions)
- Écrire les tests unitaires pour cette fonction dans un fichier séparé `f_test.ml`
- Construire et lancer `f_test.byte` pour exécuter les tests unitaire

Test unitaire avec OUnit – Exemple

```
let rec sum = function
  | []    -> 0
  | x::xs -> x + sum xs;;
```

```
open OUnit2
open Sum

let tests = "test suite for sum" >::: [
  "empty"  >:: (fun _ -> assert_equal 0 (sum []));
  "one"    >:: (fun _ -> assert_equal 1 (sum [1]));
  "onetwo" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
];;

let _ = run_test_tt_main tests;;
```

```
$ ocamlbuild -pkgs oUnit sum_test.byte
$ ./sum_test.byte
```

Table of Contents

Pile à partir d'une liste

Les questions à se poser ?

- Comment représenter le système ?
- Quels sont les états du système à conserver ?
- Quelles sont les opérations à effectuer sur le système ?
  - Quels sont leurs signatures ?

Pile à partir d'une liste

Les questions à se poser ?

- Comment représenter le système ?
- Quels sont les états du système à conserver ?
- Quelles sont les opérations à effectuer sur le système ?
  - Quels sont leurs signatures ?

Exemple – Signatures

```
empty_stack : 'a list
is_empty    : 'a list -> bool
push        : 'a -> 'a list -> 'a list
pop         : 'a list -> 'a list
peek        : 'a list -> 'a
```

Pile à partir d'une liste – Tests

Exemple – Test

```
open OUnit2
open Pile

let pushpeek = "test suite for push and peek" >::: [
  "empty" >:: (fun _ -> assert_raises
    (Failure "Empty stack") (fun ()-> peek empty_stack))
        ;

  "one elt" >:: (fun _ -> assert_equal
    'a' (peek (push 'a' empty_stack)));

  "two elts" >:: (fun _ -> assert_equal
    'b' (peek (push 'b' (push 'a' empty_stack))));

  "stack state" >:: (fun _ -> assert_equal
    ['a'; 'b'; 'c']
    (push 'a' (push 'b' (push 'c' empty_stack))));
]
let _ = run_test_tt_main pushpeek
```

Pile à partir d'une liste – Tests

Exemple – Test

```
let pop_push = "test suite for push and pop" >::: [
  "pop empty is failure" >:: (fun _ -> assert_raises
  (Failure "Empty stack") (fun () -> pop empty_stack));

  "one element pop empty" >:: (fun _ -> assert_equal
    true (is_empty (pop (push 'a' empty_stack))));
]
let _ = run_test_tt_main pop_push
```

Pile à partir d'une liste

Exemple – Implementation

```
let empty_stack = []

let is_empty stack = stack = []

let push value stack = value::stack

let pop stack =
  match stack with
| [] -> failwith "Empty stack"
| a::b -> b

let peek stack =
  match stack with
| [] -> failwith "Empty stack"
| a::b -> a
```

Pile à partir d'un type somme

- On a besoin de définir un nouveau type

```
type 'a stack = EmptyStack
              | Node of 'a * 'a stack;;
```

Pile à partir d'un type somme – Signature

- Quelles sont les signatures des méthodes précédentes ?

Pile à partir d'un type somme – Signature

- Quelles sont les signatures des méthodes précédentes ?

```
empty_stack : 'a stack
is_empty    : 'a stack -> bool
push        : 'a -> 'a stack -> 'a stack
pop         : 'a stack -> 'a stack
peek        : 'a stack -> 'a
```

Pile à partir d'un type somme – Implementation

Exemple – Implementation

```
let empty_stack = EmptyStack;;

let is_empty a = a = EmptyStack;;

let push v s = Node(v, s);;

let peek s = match s with
  EmptyStack -> failwith "Empty stack"
| Node(v, s) -> v;;

let pop s = match s with
  EmptyStack -> failwith "Empty stack"
| Node(v, s) -> s;;
```

Dictionnaire

- Comment représenter le système ?
- Quels sont les états du système à conserver ?
- Quelles sont les opérations à effectuer sur le système ?
  - Quels sont leurs signatures ?

Dictionnaire

- Comment représenter le système ?
- Quels sont les états du système à conserver ?
- Quelles sont les opérations à effectuer sur le système ?
  - Quels sont leurs signatures ?

Exemple – Signatures

```
empty_dict: ('a * 'b) list
lookup    : 'a -> ('a * 'b) list -> 'b
insert    : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list
```

Dictionnaire

Exemple – Implementation

```
let empty_dict = [];;

let rec lookup key dict =
  match dict with
| [] -> failwith "Unknown key"
| (k, v)::r when key = k -> v
| a::b -> lookup key b;;

let example_dict  = [(1, "toto"); (4, "titi")]

let rec insert key value dict =
  match dict with
| [] -> [(key, value)]
| (k, v)::b when k = key -> (k, value)::b
| a::b -> a::(insert key value b);;
```

Table of Contents

Exemple en LISP

Exemple – Factorielle

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))) ) )
```

Exemple – Somme et longueur de liste

```
(defun sum (list)
  (if (null list)
    0
    (+ (car list) (sum(cdr list))) ))

(defun len (list)
  (if list
    (1+ (len (cdr list)))
    0))

(defun average (list)
  (/ (sum list) (len list)))
```

Exemple en Erlang

Exemple – Factorielle

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

Exemple – Somme et longueur de liste

```
sum([H|T]) -> H + sum(T);
sum([]) -> 0.

len([_|T]) -> 1 + len(T);
len([]) -> 0.

average(X) -> sum(X) / len(X).
```

Exemple en Haskell

Exemple – Factorielle

```
sum :: (Num a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Exemple – Somme et longueur de liste

```
sum :: (Num a) => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs

len :: (Num b) => [a] -> b
len [] = 0
len (_:xs) = 1 + len xs

average :: (Num a) => [a] -> a
average x -> (sum x) / (len x)
```

Ressources

- Documentation OCaml
  `https://caml.inria.fr/pub/docs/manual-ocaml/index.html`
- Real world OCaml
  `https://dev.realworldocaml.org/toc.html`
- OUnit user guide
  `http://ounit.forge.ocamlcore.org/api-ounit/index.html`
- Functional Thinking
  `http://nealford.com/functionalthinking.html`