Programmation fonctionnelle, OCaml #1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Programmation « impérative » : principe (rappel)

Modèle de calcul « impératif »

- Un programme est une suite d'instructions (modifiant un état)
- Le résultat d'un calcul est la valeur de l'état après exécution des instructions du programme
- (Exemple théorique canonique : les machines de Turing)

Langage de programmation impératif

- Langage dont la syntaxe & sémantique reflète une approche impérative du calcul

C : langage plutôt impératif

- (Quasiment) tout est une instruction
- Facile de modifier un état
  - Modification des objets *via* variables, pointeurs

Programmation « fonctionnelle » : principe

Modèle de calcul « fonctionnel »

- Un programme est une composition d'expressions
- Le résultat d'un calcul est la valeur en laquelle l'expression du programme s'évalue
- (Exemple théorique canonique : les λ-calcul)

Langage de programmation fonctionnel

- Langage dont la syntaxe & sémantique reflète une approche fonctionnelle du calcul

OCaml (votre nouveau langage préféré ?) : langage plutôt fonctionnel

- (Quasiment) tout est une expression
- On peut écrire (et exécuter) des programmes sans faire apparaître d'état
  - Pas de notion de `return` !

Table des matières

Ocaml : contexte

Langage de recherche… mais pas que !

- Développé à l'Inria ; illustre des concepts avancés en théorie des langages de programmation
- Utilisé (un peu) dans l'industrie
- Très adapté à certaines tâches (par ex. l'écriture de compilateurs)

Environnement de développement conseillé

- Éditeur de texte avec support pour le langage
- Version récente de OCaml (par ex. $\geq 5.*$)
- utop
- (Facultatif) opam

Ressources

- *Cf.* page du cours

OCaml : langage (entre autres) interprété

Interprétation

- L'exécution d'un programme écrit en OCaml ne nécessite pas de compilation (pas comme C, mais comme Python)
- Le programme (sous forme de code source) est directement *interprété* par un *interpréteur* (éventuellement *interactif*)
  - Un programme qui lit un programme et exécute celui-ci
  - Par exemple, l'*évalue*

Démo.

Interprétation v. Compilation

Interprétation

- Typiquement plus lent à l'exécution
  - Rajoute un intermédiaire ; plus difficile d'optimiser, tirer profit de la machine
- Plus *portable* (sur différentes architectures)
  - À condition de disposer d'un interpréteur !
- Techniquement plus facile / plus adapté à certains types de langages (par ex. fonctionnels !)

Compilation

- Typiquement (beaucoup) plus rapide à l'exécution
- Moins portable
  - Il faut compiler pour chaque architecture

(Plein de mondes intermédiaires)

- Interpréteurs de *byte-code* (Java, Python, aussi OCaml…)
- Compilation « à la volée »

En OCaml

Interprétation (interactive)

Via `ocaml` ou (mieux) `utop`

- On *peut* intégralement développer (faire son TP) à l'intérieur de l'interpréteur interactif
- Mais pas si pratique dès que les programmes sont un peu compliqués
- Pas de sauvegarde (pratique) !

`utop` + fichier

- Écrire le programme (y compris les fonctions de test !) dans un fichier `.ml`
- Charger le fichier avec `#use`
  - À recharger à chaque modification

Compilation vers *bytecode*

- `ocamlc`

Compilation vers code natif

- `ocamlopt`

Démo.

Table des matières

Expressions dans un langage fonctionnel (comme OCaml)

Typage

- Les expressions ont **un** type

Bestiaire d'expressions : littéraux

- Littéraux des types de base (`int`, `bool`)…

```
3
(3:int) (* on peut préciser le type par une *annotation* *)
false
```

- Littéraux de types construits (`list`, types produits etc.) (Plus tard)
- Fonctions

```
fun x -> x (* fonction qui quand appliquée à x s'évalue en x *)
```

Ici `fun` est un mot-clef ; ce n'est pas le nom de la fonction qui est anonyme

Dans tous les cas : s'évaluent en leur valeur
Démo.

Bestiaire d'expressions (suite)

Expression conditionnelle

Expression *composée* de la forme :

```
if be then et else ef
```

- `be` : expression de type `bool`, `et`, `ef` deux expressions de même type
- s'évalue en l'évaluation de `et` si `be` s'évalue à `true`, et en celle de `ef` sinon

L'expression entière a donc le même type que `et` et `ef`

Exemples :

```
if true then 3 else 4 (* s'évalue à 3 *)
((if false then 3 else 4):int) (* s'évalue à 4 *)
if true then true else 4 (* pas typable : illégal *)
```

Expression conditionnelle sans `else` ?

```
if be then et
```

- Si `be` : type valeur (et type) de `et`
- Sinon ??
  - Une solution : prendre une valeur par défaut : `()` pour un type par défaut : `unit`
  - Dans ce cas, `et` devra aussi avoir type `unit` ! (Plus à ce sujet, plus tard)

Bestiaire d'expressions (re-suite)

Expression conditionnelle sans `else` (suite)

```
if true then 3 (* pas typable : illégal *)
if true then () (* typable, mais ne fait rien d'intéressant *)
```

Démo.

Application de fonctions

```
(fun x -> x) 3 (* notation préfixe ; pas de parenthèses nécessaires *)
3 * 4 (* « * » est une fonction en notation infixe *)
```

S'évaluent en… ce que fait la fonction

Donner des noms aux expressions : `let`-binding

C'est pratique de nommer les choses pour s'y retrouver…

« Soit $x \in \mathbb{N}$ valant 3 »
« Soit f une fonction qui… »

En OCaml : avec des `let`s

```
let x = 3
let f = fun x -> 3 * x
let c = if true then 3 else 4
```

Expression : *via* un nom

Une expression peut être un nom (connu dans l'environnement, qui a été précédemment lié globalement au résultat de l'évaluation d'une expression)

- A le type et la valeur en laquelle l'expression s'était évaluée

```
let a = 3
let b = 4 * a (* s'évalue à 12 *)
a * c (* erreur : c inconnu (*unbound*) *)
```

`let`s en cascade

Liaison globale : `let i = e`

- Associe l'évaluation de `e` au symbole `i`
  - Association valide pour toute la portée syntaxique/lexicale de `i` (pour nous : typiquement globale, hors *occultation* ; règles similaires qu'en C)
- N'est pas une expression, mais une *phrase* (techniquement, un « module-item »)

```
(let a = 3) + 2 (* erreur de syntaxe *)
```

Liaison locale par `let`-expression : `let i = e1 in e2`

- Associe l'évaluation de `e1` etc. localement dans le contexte de l'expression `e2`
- Est une expression (composée) qui s'évalue en `e2` (relativement à son contexte)
- Enchaînable en cascade !

```
((let a = 3 in a * a):int)
a  (* erreur : a inconnu *)
(let a = 3 in a * a) + 2
let a = 3 in
let b = 4 in
a * b
```

`let` : définition (ou pas) de variables ?

- Un `let` donne un nom à l'évaluation d'une expression
  - On évalue l'expression une fois
  - Puis associe sa valeur au nom
  - *E poi basta*
- Un `let` ne crée pas un objet en mémoire que l'on pourra ensuite modifier (contrairement à la déclaration d'une variable en C)
- Cela n'a pas de sens de « modifier » la valeur associée à un nom
- Mais on peut occulter un nom par un autre…

```
let a = 3
let b = a
let a = 4 (* a occulté la liaison précédente *)
b (* s'évalue à 3 *)
```

Interprétation en tant que variables

Les variables en OCaml sont immuables (non modifiables)
(Mais on peut quand même trouver de la mutabilité ailleurs)

Table des matières

Intermède : à propos des ; ;

- Pour évaluer une expression ou une phrase OCaml dans un interpréteur, il est nécessaire de les terminer par ; ;
- Ce n'est pas nécessaire dans un fichier ensuite chargé dans `utop` avec `#use`
- Mais de nouveau (partiellement) nécessaire si l'on veut compiler le programme…

Intermède : types `int` et `bool` et `unit`

`int`

- Entiers signés sur 63 bits
- 2^{63} valeurs (entre `min_int` et `max_int`)
- Dépassement de capacité bien définis, avec *wraparound*

`bool`

Deux valeurs : `true` et `false`

`unit`

Une valeur : `()`

Intermède : types de fonction

$t_1 \to t_2$

Une fonction d'un paramètre de type t_1 et s'évaluant en une valeur de type t_2 a type $t_1 \to t_2$ (se lit : « t_1 flèche t_2 »). C'est un type de fonction…

Exemple :

```
((fun x -> x + x):int -> int)
```

$t_1 \to t_2 \to t_3 \to \ldots$

Techniquement, une fonction OCaml n'a toujours qu'un seul paramètre

On peut cependant émuler des fonctions à nombre de paramètre (constant) quelconque (plus de détails un peu plus tard) ; leur type est alors $t_1 \to t_2 \to \cdots \to t_n \to t_r$ avec $t_1, \ldots, t_n$ les types des paramètres et t_r le type du résultat.

Exemple :

```
((fun x y -> x + y):int -> int -> int)
```

Intermède : types de fonction (bis)

Polymorphisme

Certaines fonctions peuvent accepter des arguments de *plusieurs types possibles* : elles sont polymorphes

Exemple :

```
let f = ((fun x -> x):'a -> 'a)
f 3 (* s'évalue à (3:int) *)
f true (* s'évalue à (true:bool) *)
```

Le type `'a -> 'a` (se lit « α flèche α ») se comprend comme celui d'une fonction qui pour un argument de type α quelconque s'évalue en une valeur du même type α

Plus de détails un peu plus tard

Inférence de type

On peut remarquer que (contrairement au C) il n'est pas *nécessaire* de typer les paramètres des fonctions en OCaml

Plus de détails un peu plus tard

Intermède : quelques fonctions usuelles en OCaml

Opérations sur les entiers `int`

- `+`, `-`, `*`, `/`, `mod`

Opérations de comparaison (sur les entiers, mais *pas que*)

- `<`, `>`, `<=`, `>=`
- Attention : `=` : égalité
  - `==` existe aussi mais avec un sens différent, et HP !
- Attention : `<>` : inégalité
  - `!=` existe aussi mais avec un sens différent, et HP !

Opérations sur les booléens `bool`

- `&&`, `||` : paresseux (comme en C)
- `not`
- Attention : `!` existe aussi mais avec un sens **totalement différent** (mais c'est au programme, *cf.* plus tard)

Fonctions usuelles (suite)

+, = etc. sont des fonctions

On peut les appeler (par défaut) en notation infixe :

```
3 + 4
false && ((1/0) = 1)
```

Mais aussi comme n'importe quelle fonction (en notation préfixe) en les parenthésant :

```
(+) 3 4 (* 7 *)
(&&) false ((=) ((/) 1 0) 1) (* false *)
```

Ceci permet aussi d'avoir accès aux fonctions en tant que telles :

```
+ (* Error: Syntax error *)
((+):int -> int -> int) (* _fine_ *)
((&&):bool -> bool -> bool)
((=):'a -> 'a -> bool)
```

Intermède : point sur la syntaxe des fonctions

Syntaxe allégée pour le nommage

Les fonctions sont des littéraux : on peut les lier à des noms normalement (comme ci-dessus) :

```
let add = fun x y -> x + y
```

Il existe aussi une syntaxe allégée (pratique) :

```
let add x y = x + y
```

Syntaxe générale d'une fonction

On a :

```
fun <params> -> ef
```

où `ef` est l'expression définissant la fonction, où tout identifiant apparaissant dans `<params>` est remplacé par l'(évaluation des) arguments à l'évaluation de la fonction

- Typiquement `ef` est une expression composée (`if`, `let in`, …)

Intermède : exemples syntaxiques de fonctions

```
fun x -> 12 (* ignore son argument, s'évalue au littéral 12 *)

fun x -> y (* s'évalue à l'expression nommée par l'identifiant y,
              qui doit être connu pour que la fonction soit bien
              définie (par ex. lié globalement plus haut). *)

fun x y -> if x > y then x else y

fun a b c -> let b2 = b*b in
             let discr = b2 - 4*a*c in
             if discr < 0 then false else true

fun x -> fun y -> x + y (* ce qui se passe réellement quand on
                           a une fonction à « plusieurs »
                           paramètres *)
```

Exercice : inférence de types

Table des matières

Que peut on calculer avec tout ça ?

- Pas (encore) de répétitions possibles (`while`, `for`…)
  - Pas facilement/naturellement exprimable comme une expression
- On peut faire des calculs sur des entiers, enrichis de tests

Par ex. :

```
fun x y z ->
    if 2*x > y then
        (if 3*x > z then x else z)
    else if 4*y > z then y else z
```

Déjà bien (?), mais quand-même limité…

Mécanisme fondamental pour la répétition : fonctions récursives

Fonction récursive

Une fonction est dite récursive si elle s'appelle elle-même

- Possible seulement pour une fonction que l'on a nommé (globalement ou localement, *via* un `let`)
- Mais il faut aussi annoncer l'intention d'établir une définition récursive, *via* le mot-clef supplémentaire `rec`

Démo.

Intérêt

Les fonctions récursives (combinées aux tests) permettent de répéter un calcul un nombre de fois dépendant d'un paramètre : alternative aux boucles

Fonctions récursives : pourquoi/quand ça marche ?

En maths

- Suites définies par récurrence :
  - Si u_0 est défini
  - Si à partir de u_n on sait définir u_{n+1} (Ou : si l'on peut définir u_n en connaissant u_{n-1})
  - Alors u_n est défini pour tout $n \in \mathbb{N}$
- Preuves par récurrence
  - Si $\mathcal{P}(0)$
  - Si $\mathcal{P}(n) \rightarrow \mathcal{P}(n+1)$ (Ou : si l'on peut prouver $\mathcal{P}(n > 0)$ en supposant $\mathcal{P}(n-1)$)
  - Alors $\mathcal{P}(n)$ est vrai pour tout $n \in \mathbb{N}$ *axiomatiquement*

Fonction récursive

Exemple simple sur les naturels. Si :

- La valeur en laquelle la fonction s'évalue est définie quand son argument vaut zéro (cas de base)
- La valeur [...] pour un argument $n > 0$ est définie en fonction de la valeur [...] pour $n - 1$ (cas récursifs)
- Alors la valeur [...] est définie pour tout $n \geq 0$

Fonctions récursives (suite)

Exemples canoniques

```
(* similaire à une récurrence simple *)
let rec fact n = if n = 0 then 1 else n * (fact (n - 1))

(* similaire à une récurrence forte d'ordre deux *)
let rec fib n =
    if n = 0 then 0 else
    if n = 1 then 1 else fib (n - 1) + fib (n - 2)
```

Exemple d'exécution

Démo. (`#trace` dans *utop*)

- Les appels de fonction (récursifs ou non) s'empilent dans l'ordre d'émission ; se dépilent dans l'ordre de fin d'exécution
- Comme (usuellement) en C !

Fonctions récursives (suite)

Bonne définition (informellement)

Une fonction récursive est bien définie si la récursion termine toujours : on ne peut pas avoir de suite infinie d'appels récursifs qui ne tombent pas sur un cas de base

Plus de détails plus tard (second semestre ?)

Pas que sur des entiers !

Les fonctions récursives peuvent être définies sur autre chose que des entiers : elles généralisent (mathématiquement) les suites/preuves par récurrence

Exemples… très bientôt

Fonctions récursives : conseils de rédaction

Les fonctions récursives complexes peuvent avoir

- De nombreux cas de base
- De nombreux cas récursifs

Les cas de base sont souvent les plus simples à écrire (et de toutes façons nécessaires) : mieux vaut commencer par eux

Fonctions récursives : terminaison

Preuves de terminaison

Les fonctions récursives s'assimilent (superficiellement) à des boucles `while`

- Leur terminaison doit être prouvée

Ceci peut se faire en utilisant une méthode similaire aux variants ; on identifie une quantité entière :

- (Typiquement) minorée en cas d'appel récursif
- Qui décroît strictement à chaque appel récursif

On peut généraliser ces arguments à des quantités qui ne sont pas entières (*cf.* second semestre)

Exercice

Prouvez la terminaison de :

```
let rec myst x = if x < 2 then 0 else 1 + myst (x / 2)
```

(Que fait cette fonction ?)

Fonctions récursives : dépassement de pile et récursion terminale

Stack overflow

En principe, en général :

- Chaque appel de fonction empile un contexte sur la pile d'exécution
- La pile est petite
- Trop d'appels récursifs peuvent mener à un débordement :(

Exemple :

```
(* precondition: a >= 0 *)
let rec add a b = if a = 0 then b else 1 + (add (a - 1) b)
```

Récursion terminale

Informellement, un appel récursif est dit terminal s'il est une dernière expression à être évaluée avant terminaison de la fonction

Exemple :

```
(* precondition: a >= 0 *)
let rec add' a b = if a = 0 then b else add' (a - 1) (b + 1)
```

Fonctions récursives : récursion terminale et *tail-call optimisation*

Tail-call optimisation

On peut mécaniquement transformer une fonction récursive terminale en une fonction non récursive utilisant une boucle `while` (éventuellement dans un autre langage)

- Les langages fonctionnels implémentent (généralement) cette optimisation
- Dans ce cas, les fonctions récursives terminales ne causent pas de débordement de pile

(On peut aussi mécaniquement transformer toute fonction en une fonction récursive terminale par exemple avec des *continuations*, mais c'est complètement )

Caveats

- La récursion terminale est techniquement hors-programme
- À l'écrit, ce n'est pas nécessaire voire fortement déconseillé
  - Les versions récursives terminales sont souvent plus compliquées et moins lisibles
- À l'écrit, on suppose implicitement que la pile d'exécution est de taille illimitée
  - La récursion terminale ne résout aucun problème : il n'y a pas de problème
  - (Au passage : ne pas écrire une fonction récursivement par peur de déborder la pile n'est pas une bonne raison / un bon *prétexte*)