Programmation Fonctionnelle Avancée

Séance 0 : types primitifs, modules, fonctions, boucles, filtrage, types algébriques

Alexandros Singh

Université Paris 8

19 septembre 2023

- Les fonctions à la base de tout ! Les fonctions sont des *valeurs* :

```
# let f x = x+2
  let g x = 5*x;;
val f : int -> int = <fun>
val g : int -> int = <fun>
```

- Application

```
# f 3;;
- : int = 5
```

- Composition

```
# f(g(3));;
- : int = 17
```

- Système de types : annotations permettant de raisonner sur les fonctions (détecter les erreurs, optimiser)

```
# #show (+);;
external ( + ) : int -> int -> int = "%addint"
# 3 + "spam" ;;
Error: This expression has type string but an expression was expected of type
         int
```

- OCaml a été créé en 1996 et est géré et principalement maintenu par l'INRIA.
- Langage de programmation multi-paradigme : fonctionnel, impératif, orienté objet
- Typage inféré : annotation automatique des expressions

```
# let f x = x+2
val f : int -> int = <fun>
```

- Typage statique et fort : détection des erreurs de typage à la compilation, pas des conversions implicites

- Toplevel/REPL : `ocaml`
  (voir aussi `utop` pour une version plus performante : édition interactive, tab-complétion, coloration syntaxique ...)
- Compilateur bytecode : `ocamlc`
- Interprétateur de bytecode : `ocamlrun`
- Compilateur code natif : `ocamlopt`
- Gestionnaire de paquet : `opam`
  - lister les packages disponibles : `opam list -a`
  - Installer un paquet (par exemple `utop`) : `opam install utop`
  - Mettre à jour la liste des paquets : `opam update`
  - Mettre à jour les paquets installés : `opam upgrade`

Dans le toplevel :

```
# print_string "Hello world!\n";;
Hello world!
- : unit = ()
```

`unit` : un type spécial

Utilisé par des expressions qui “n’ont pas de valeur” et qui ne sont utilisées que pour leurs effets (de bord).

Hello world ! (cont.)

Dans un fichier `helloworld.ml` :

```
1 print_string "Hello world!\n"
```

Compilation et exécution :

```
$ocamlc helloworld.ml -o hello
$ls
hello  helloworld.cmi  helloworld.cmo  helloworld.ml
$ ./hello
Hello world!
```

Hello world ! (cont.)

Chargement du fichier dans le toplevel :

```
# #use "helloworld.ml";;
Hello World!
- : unit = ()
```

Ou pour démarrer le toplevel avec le fichier chargé :

```
$ ocaml -init helloworld.ml
OCaml version 4.14.0
Enter #help;; for help.

Hello World!
```

- L'usage basique de `let` est :
  `let ident = expr`
- Où `ident` est un identifiant valide :
  `ident ::= (a...z|_){letter|0...9| _|´}`

  voir https://v2.ocaml.org/manual/expr.html#let-binding
- Liaisons multiples à l'aide de `and` :

```
# let x = 10 and y = 20 in (x+y);;
- : int = 30
```

Module d'OCaml : séquence de déclarations et d'expressions (entre autres) **à évaluer**.
Les déclarations (lier des noms à des valeurs) se font via l'opérateur `let` :

```
1 let x = 5
2 let y = 20
3 let f x = x+y
4
5 let toPrint = string_of_int(f(3))^"\n"^string_of_int(x)^"\n"
6
7 let sideEffect = print_string(toPrint)
```

Devinez la sortie (le cas échéant) !

Module d'OCaml : séquence de déclarations et d'expressions (entre autres) **à évaluer**.
Les déclarations (lier des noms à des valeurs) se font via l'opérateur `let` :

```
1 let x = 5
2 let y = 20
3 let f x = x+y
4
5 let toPrint = string_of_int(f(3))^"\n"^string_of_int(x)^"\n"
6
7 let sideEffect = print_string(toPrint)
```

Devinez la sortie (toutes les expressions sont évaluées, la dernière induit des effets de bord) !

```
23
5
```

Types inférés : pas nécessaire de les déclarer explicitement.
Cependant, nous pouvons le faire ! Pouvez-vous deviner le reste des types ?

```
1 let (x : int) = 5
2 let (y : int) = 20
3 let f x = x+y
4
5 let toPrint = string_of_int(f(3))^"\n"^string_of_int(x)^"\n"
6
7 let sideEffect = print_string(toPrint)
```

Types inférés : pas nécessaire de les déclarer explicitement.
Cependant, nous pouvons le faire ! Pouvez-vous deviner le reste des types ?

```
let (x : int) = 5
let (y : int) = 20
let f (x : int) : int = x+y

let toPrint : string = string_of_int(f(3))^"\n"^string_of_int(x)^"\n"

let sideEffect : unit = print_string(toPrint)
```

Currying

Fonctions prenant des tuples :

```
# let f (x,y) = x+y;;
val f : int * int -> int = <fun>
 # f (10,5);;
- : int = 15
```

sont équivalentes aux fonctions à arguments multiples :

```
# let f x y = x+y;;
val f : int -> int -> int = <fun>
# f 10 5;;
- : int = 15
```

Ceci est formalisé par la notion de *currying* : $A \times B \to C \cong A \to B \to C$.[1]

1. Aucun lien avec  , nommé d'après le logicien Haskell Curry.

Portée (scope)

La portée (scope) d'une liaison est la partie d'un programme où la liaison est valide.

Les liaisons au toplevel sont globales :

```
# let x = 5;;
val x : int = 5
# let f y = y + x;;
val f : int -> int = <fun>
# f 3;;
- : int = 8
```

Scoping

Les liaisons les plus étroites sont prioritaires :

```
# let g x = x + 3;;
val g : int -> int = <fun>
# g 3;;
- : int = 6
# (x+3);; (* dans le contexte extérieur x est toujours 5 *)
- : int = 8
```

Variables locales à l'aide de "`let-in`" :

```
# let x = 20 in (x+3);;
- : int = 23
# (x+3);; (* dans le contexte extérieur x est toujours 5 *)
- : int = 8
```

Scoping

Les fonctions OCaml sont de “première classe”, on peut les lier localement comme n'importe quelle autre valeur :

```
# let my_add x y = x+y in my_add 5 10;;
- : int = 15
# my_add;; (* hors de portée *)
Error: Unbound value my_add
```

```
# let f x =
    let g x = 2*x in
      (g x) + (g x);;
val f : int -> int = <fun>
# f 5;;
- : int = 20
```

Récursion

- Dans l'expression `let ident = e2` est hors de portée dans `e1`.
- Dans `let ident = e1 in e2` la portée de `ident` est `e2`.

Pour que la portée de `ident` inclue `e1`, nous utilisons `let rec` :

```
# let rec factorial n =
  if n = 0 then 1
  else n * (factorial (n-1));;
val factorial : int -> int = <fun>
# factorial 10;;
- : int = 3628800
# let rec triangular_num n =
  if n = 0 then 0
  else n + (triangular_num (n-1))
  in triangular_num 10;;
- : int = 55
```

Récursion Terminale

Une fonction à récursivité terminale est une fonction où la dernière instruction à être évaluée est l'appel récursif.

```
let rec triangular_num n =
    if n = 0 then 0
    else n + triangular_num (n-1)

let triangular_num_tr n =
    let rec tn_helper n acc =
        if n = 0 then acc
        else (tn_helper[@tailcall]) (n-1) (acc+n) in
    tn_helper n 0
```

Astuce : utilisez l'annotation [@tailcall] pour que le compilateur vérifie si un appel est optimisé pour le tailcall.

Les fonctions définies l'une en termes de l'autre sont appelées **mutuellement récursives**.

```
# let rec even x =
      if x = 0 then true else odd (x-1)
  and odd x =
      if x = 0 then false else even (x-1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4;;
- : bool = true
# even 5;;
- : bool = false
# odd 4;;
- : bool = false
# odd 5;;
- : bool = true
```

Les motifs nous permettent de sélectionner des structures de données d'une forme spécifique :

```
# let isZero n = match n with
    | 0 -> print_string "l'entrée est nulle\n"
    | _ -> print_string "l'entrée est non nulle\n";;
val isZero : int -> unit = <fun>
# isZero 0;;
l'entrée est nulle
- : unit = ()
# isZero 5;;
l'entrée est non nulle
- : unit = ()
```

Les motifs sont vérifiés dans l'ordre, le motif `_` concorde avec n'importe quelle valeur.

L'exemple précédent est équivalent à :

```
# let isZero = function
    | 0 -> print_string "l'entrée est nulle\n"
    | _ -> print_string "l'entrée est non nulle\n";;
val isZero : int -> unit = <fun>
# isZero 0;;
l'entrée est nulle
- : unit = ()
# isZero 5;;
l'entrée est non nulle
- : unit = ()
```

Motifs, `match` et `function`

Les motifs (à l'exception de _) peuvent être utilisés pour lier des valeurs à des identifiants :

```
# let isZeroF = function
    | 0 -> print_string "0=0"
    | i -> print_string ((string_of_int i)^"!=0\n");;
val isZeroF : int -> unit = <fun>
# isZeroF 5;;
5!=0
- : unit = ()
```

Nous pouvons aussi faire `match` des arguments spécifiques :

```
let greetName its_morning name = match its_morning with
    | true -> print_string ("Bonjour "^name^" !\n")
    | false -> print_string ("Bonsoir "^name^" !\n")
```

Motifs, `match` et `function`

Les motifs sont *linéaires*, les variables apparaissent exactement une fois :

```
# let is_eq = function
    | x, x -> true
    | x, y -> false;;
Error: Variable x is bound several times in this matching
```

et peuvent inclure des *gardes* :

```
# let is_eq = function
    | (x : int), (y : int) when x = y -> true
    | _ -> false;;
val is_eq : int * int -> bool = <fun>
# is_eq (5,5);;
- : bool = true
# is_eq (5,10);;
- : bool = false
```

Types produits

Un type de produit p est construit à partir des facteurs a,b,c,... comme suit :

```
type p = a * b * c * ...
```

Les données de type p sont des tuples (ou des records comme nous le verrons plus tard) dont les composants sont respectivement de type a,b,c,....

```
# type coord = float * float;;
type coord = float * float
# let x : coord = (3.141, 1.202);;
val x : coord = (3.141, 1.202)
```

Types sommes

Les types de sommes sont construits à l'aide de constructeurs de données, qui prennent zéro ou plusieurs arguments, comme suit :

```
type s = Const_1 of a | Const_2 of b | ...
```

où les `a, b,...` sont des types. Les valeurs de types somme sont créés à l'aide des constructeurs et, le cas échéant, d'un élément du type correspondant `a,b,....`.

```
# type day = Lun | Mar | Mer | Jeu | Ven | Sam | Dim;;
type day = Lun | Mar | Mer | Jeu | Ven | Sam | Dim
# let x = Lun;;
val x : day = Lun
```

En mélangeant les constructeurs des types produit (*) et somme (|), nous obtenons des **types de données algébriques** :

```
# type shape =
    (* carré paramétré par côté *)
    | Square of float
    (* triangle paramétré par la base et hauteur *)
    | Triangle of float * float;;
# Square 5.;;
- : shape = Square 5.
# Triangle (1.,3.);;
- : shape = Triangle (1., 3.)
```

Les types de données algébriques utilisés avec `match`, permet de définir des fonctions de manière très expressive :

```
# let area = function
    | Square side -> side *. side
    | Triangle (base, height) -> (base *. height) /. 2.
;;
val area : shape -> float = <fun>
# area (Square 5.);;
- : float = 25.
# area (Triangle (1.,1.));;
- : float = 0.5
```

OCaml permet d'utiliser des données mutables via des *références* :

```
# let x = ref 0;;
val x : int ref = {contents = 0}
# !x;; (* déréférencement via l'opérateur ! *)
- : int = 0
# x := 5;; (* affectation via l'opérateur := *)
- : unit = ()
# !x;;
- : int = 5
```

Il permet également de réaliser des boucles telles que :

```
# let x = ref 0;;
val x : int ref = {contents = 0}
# for n = 0 to 9 do
    x := !x + 1
  done;;
- : unit = ()
# while !x > 0 do
    print_string ((string_of_int !x)^" ");
    x := !x - 1
  done;
  print_string "\n";;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

Lisez la documentation !

Sous https ://v2.ocaml.org/docs/index.fr.html, vous trouverez

- Des tutoriels OCaml
- Documentation pour l'API OCaml
- Documentation des outils (ocaml, ocamlc, etc.)

Il est **très important** que vous vous familiarisiez avec eux !
En particulier, la page API (`https://v2.ocaml.org/api/index.html`) sera votre meilleure amie pour ce cours ! Il contient des informations sur tous les modules fournis avec OCaml (y compris `Int`, `Float`, `String`, etc) avec lesquels je vous conseille vivement de vous familiariser !

Supposons que nous voulions imprimer une valeur booléenne. Nous ne pouvons pas faire :

```
# print_string true;;
Error: This expression has type bool but an expression
was expected of type
         string
```

comme dans certains langages à typage dynamique !
En effet, `print_string` a come type :

```
# print_string;;
- : string -> unit = <fun>
```

Nous avons donc besoin d'une fonction qui convertit un `bool` en `string` !

Utilisation de la documentation de l'API

Nous pouvons consulter la bibliothèque standard pour voir si une telle fonction existe déjà. Si c'est le cas, il devrait se trouver sous `https://v2.ocaml.org/api/Bool.html`. Effectivement, nous y trouvons :

Converting

```
val to_int : bool -> int
```

`to_int b` is `0` if `b` is `false` and `1` if `b` is `true`.

```
val to_float : bool -> float
```

`to_float b` is `0.` if `b` is `false` and `1.` if `b` is `true`.

```
val to_string : bool -> string
```

`to_string b` is `"true"` if `b` is `true` and `"false"` if `b` is `false`.

```
val seeded_hash : int -> bool -> int
```

A seeded hash function for booleans, with the same output value as `Hashtbl.seeded_hash`. This function allows this module to be passed as argument to the functor `Hashtbl.MakeSeeded`.

La page de l'API dispose également d'une fonction de recherche *très pratique* :