

LIFPF – Programmation fonctionnelle

CM4

Ordre supérieur

Fermetures

Licence informatique UCBL – Printemps 2025

<https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md>

Ordre supérieur

Fonctions comme des valeurs

- Passer une fonction en argument
- Récupérer une fonction comme résultat
- Placer une fonction dans une variable
- Remarque : naturel en λ -calcul.

Passer une fonction en argument

Exemple : List.map

```
let rec map (f: 'a -> 'b) (l: 'a list): 'b list =  
  match l with  
  | [] -> []  
  | x :: l' -> (f x) :: map f l'
```

- Permet de généraliser certains comportements, par exemple les parcours de structure.
- Également utilisé pour gérer des événements : la fonction est passée en argument à un "gestionnaire" d'événement qui l'appellera quand l'événement survient.

Fonction comme résultat

Exemple : create_gt

Fonction qui prend un `int` et renvoie la fonction qui renvoie `true` si son argument est plus grand que cet `int` :

```
let create_gt (x: int): int -> bool =  
  fun y -> x < y
```

Fonction dans une variable

Exemple :

```
let filtre_pairs: int list -> int list =  
  let pair x = x mod 2 = 0 in  
  fun l -> List.filter pair l
```

Différentes notation pour les fonctions avec let

Notations équivalentes :

```
let f (x1: t1) (x2: t2): tr = ...
```

```
let f (x1: t1): t2 -> tr = fun x2 -> ...
```

```
let f: t1 -> t2 -> tr = fun x1 -> fun x2 -> ...
```

Application partielle

- Fonctions à plusieurs arguments sont souvent des fonctions qui prennent le premier argument et renvoient une fonction qui prend les autres arguments et calcule le résultat.
- On peut donc passer uniquement les k premiers arguments à une fonction qui en attend n . On obtient la fonction qui attend les $n - k$ arguments restants.
- C'est ce qu'on appelle une application *partielle* de fonction.

Exemple d'application partielle

Avec List.filter

```
let cmp x y = x < y;;  
let sup_3 (l:int list): int list =  
    List.filter (cmp 3) l
```


Exemple d'application partielle

Avec List.filter

```
let cmp x y = x < y;;  
let sup_3 (l:int list): int list =  
    List.filter (cmp 3) l
```

Même exemple, mais sans l

```
let cmp x y = x < y;;  
let sup_3: int list -> int list =  
    List.filter (cmp 3)
```

Fonctions de manipulation de fonctions

Avec l'ordre supérieur, on peut facilement écrire des fonctions utilitaires :

`apply ((@) dans la Stdlib)`

```
let apply (f: 'a -> 'b) (x: 'a): 'b = f x
```

`(@)` est associatif à droite, on ainsi a :

```
f @@ g @@ x = f (g x)
```

`inverse_apply ((|>) dans la Stdlib)`

```
let inverse_apply (x: 'a) (f: 'a -> 'b): 'b = f x
```

`(|>)` est associatif à gauche, on ainsi a :

```
x |> f |> g = g (f x)
```

Fonctions de manipulation de fonctions - suite

Fun.flip

```
let flip (f: 'a -> 'b -> 'c): 'b -> 'a -> 'c =  
    fun x -> fun y -> f y x
```

compose

```
let compose (f: 'b -> 'c) (g: 'a -> 'b): 'a -> 'c =  
    fun x -> f (g x)
```

n-uplets et fonctions à plusieurs arguments

Deux manières d'utiliser plusieurs valeurs en argument :

- En une fois avec un n-uplet
- Avec plusieurs arguments

Exemple : add

Avec une paire :

```
let add (p: int * int): int =  
  match p with  
  | (x, y) -> x + y
```

Avec deux arguments :

```
let add (x: int) (y: int): int = x + y
```

(Dé)curryfication

Curryfier : transformer une fonction qui prend un n -uplet en fonction à n arguments

curry2

```
let curry2: ('a * 'b -> 'c) -> 'a -> 'b -> 'c =  
  fun f -> fun x -> fun y -> f (x,y)
```

uncurry2

```
let uncurry2: ('a -> 'b -> 'c) -> 'a * 'b -> 'c =  
  fun f -> fun p ->  
    match p with  
    | (x, y) -> f x y
```

Recodage du `let`

La construction

$$\text{let } x = e1 \text{ in } e2$$

peut être recodée avec une fonction anonyme :

$$(\text{fun } x \rightarrow e2) \ e1$$

- pas besoin de `let` en λ -calcul.
- on voit bien que la portée de `x` est limitée à `e2`

Portée et fonctions retournées

Une fonction renvoyée

```
let f x =  
  fun y -> x + y  
in  
let g = f 3 in  
g 5
```

- Lors de l'exécution de ce code on peut imaginer que `g` vaut `fun y -> x + y` car le `fun` n'est pas évalué à ce moment là.
- Cependant `x` n'est plus visible au moment de l'appel `g 5`.
- On ne constate cependant pas de problème dans l'interpréteur...

Et en λ -calcul ?

Exemple précédent traduit en λ -calcul

$$(\lambda f.(\lambda g.(g\ 5)\ (f\ 3))\ \lambda x.\lambda y.(x + y))$$

Évaluation :

$$\begin{array}{ll}
 \overset{f}{\rightsquigarrow} & (\lambda g.(g\ 5)\ (\lambda x.\lambda y.(x + y)\ 3)) \\
 \overset{x}{\rightsquigarrow} & (\lambda g.(g\ 5)\ \lambda y.(3 + y)) \\
 \overset{g}{\rightsquigarrow} & (\lambda y.(3 + y)\ 5) \\
 \overset{y}{\rightsquigarrow} & (3 + 5) \\
 \overset{+}{\rightsquigarrow} & 8
 \end{array}$$

OK en λ -calcul car x remplacé dans le corps du λy .

Comment fait OCaml ?

Fermetures

- OCaml ne remplace pas les variables par leur valeur, mais accède à celles-ci dans l'environnement d'exécution.
- Lors de l'"exécution" d'un `fun` il n'y a pas vraiment d'exécution de code : OCaml crée une structure qui référence le code de la fonction créée
- Pour éviter les problèmes de non définition de variables, OCaml va également effectuer une capture de la valeur des variables référencées dans le code de la fonction.
- Lors de l'appel à la fonction, OCaml va reconstituer l'environnement capturé lors du `fun`
- La référence à la fonction avec les valeurs de variables capturées est appelée **fermeture**

Dessin au tableau de l'exécution précédente

Quelques fonctions sur les structures

On considère une structure `'a s` que l'on peut parcourir, par exemple `'a list` ou encore `'a arbre`.

- `map: ('a -> 'b) -> 'a s -> 'b s`
`map f` transforme tous les éléments de `s` en utilisant `f`
- `flatten: 'a s s -> 'a s` transforme une structure de structure en une structure simple
- `fold_left: ('b -> 'a -> 'b) -> 'b -> 'a s -> 'b`
`fold_left f e` parcourt la structure obtenant les éléments `x1, x2, ...` et calcule `f ... (f (f e x1) x2) ... xn`
- `fold_right: ('a -> 'b -> 'b) -> 'a s -> 'b -> 'b`
 similaire à `fold_left`, mais calcule `f x1 (f x2 ... (f xn e) ...)`.

Quelques fonctions sur les structures

On considère une structure `'a s` que l'on peut parcourir, par exemple `'a list` ou encore `'a arbre`.

- `filter: ('a -> bool) -> 'a s -> 'a s`
`filter f` crée une structure contenant tous les éléments `e` de la structure initiale pour lesquels `f e` s'évalue à `true`.
- `for_all: ('a -> bool) -> 'a s -> bool`
`for_all f` renvoie `true` si pour tous les éléments `e` de la structure, `f e` s'évalue à `true`.
- `exists: ('a -> bool) -> 'a s -> bool`
`exists f` renvoie `true` si on peut trouver un élément `e` de la structure, `f e` s'évalue à `true`.
- `mem: 'a -> 'a s -> bool`
`mem e` renvoie `true` si `e` est dans la structure.

Enchaînement de transformations

L'opérateur (`|>`) est utile pour exprimer des transformations de structure quand on le combine avec des fonctions sur les structures partiellement appliquées.

Enchaînement de transformations

```
let y = f 5
in ma_liste
  |> List.map (fun x -> [x; x*x; 2*x+y])
  |> List.flatten
  |> List.filter (fun x -> x mod 3 = 0)
  |> List.fold_left (fun acc x -> acc * x) 1
```