Programmation Fonctionnelle

Emmanuel Chailloux

24 juin 2020

Plan

Plan du cours 3:

- traits fonctionnels et structures linéaires
  - fonctionnelles, itérateurs,
  - polymorphisme paramétrique
  - récursivité terminale
  - structures linéaires : pile, dictionnaire, file
- modules simples et compilation séparée
  - modules simples
  - compilation séparée

Représentation des valeurs fonctionnelles (1)

Les valeurs fonctionnelles sont représentées par des fermeture.

Une fermeture est une fonction `function` avec son environnement lexical, c'est-à-dire l'ensemble des variables libres (non liées à une déclaration locale ou au nom d'un paramètre, ou au nom dans un motif) nécessaires à son évaluation.

```
# let f = function a -> function b ->
    let c = a + b in
      function x -> x + c    ;;
val f : int -> ( int -> ( int -> int ) )  = <fun>
# let g = f 5 10 ;;
val g : int -> int = <fun>
# g 1 ;;
- : int = 16
# g 10 ;;
- : int = 25
```

Représentation des valeurs fonctionnelles (2)

Les fermetures sont généralement codées par un couple représentant l'environnement et le code de la fonction.

Dans l'exemple précédent la valeur fonctionnelle g a dans son environnement la liaison de la variable c qui vaut 15.

A chaque fois que g est appelée, son corps est évalué et la valeur associée à c est recherchée dans l'environnement.

La création de valeurs fonctionnelles alloue l'espace d'une fermeture, principalement l'environnement et un pointeur de code de la fonction. Il n'y a pas de génération de code ! ! !

```
# let h = f 6 20 ;;
val h : int -> int = <fun>
# h 1 ;;
- : int = 27
# h 10 ;;
- : int = 36   let h = f 6 20
```

Application partielle (1)

Les déclaration suivantes sont équivalentes :

```
1  let f = fun x y -> x + y
2  let g = function x -> function y -> x + y
```

Bien que l'on puisse considérer toutes les fonctions comme des fonctions à un seul paramètre pouvant retourner une valeur fonctionnelle, le compilateur OCaml effectue une optipmisation (car il n'y a pas de calculs entre les deux `function`, quand on appelle `f` ou g avec deux arguments :

```
1  # f 2 3;;
2  - : int = 5
```

Il n'y a pas la création d'une fermeture `(f 2)` appliquée ensuite à 3. On parle alors Cela évite de l'allocation mémoire et du calcul inutile.

Application partielle (2)

Par contre dans :

```
# let h = g 2 ;;
  val h : int -> int = <fun>
```

il y a bien la création d'une fermeture :

```
env    [ x = 2]
code : function y -> x + y
```

qui peut ensuite être appelée :

```
# h 3;;
- : int = 5
# List.map h [1; 2; 3] ;;
- : int list = [3; 4; 5]
# List.map ((fun x y -> x + y) 2) [1; 2; 3];;
- : int list = [3; 4; 5]
```

rev : création d'une liste inverse

l'application `(rev [x0; x1; ..; xn])` a pour valeur `[xn; ..; x1; x0]`.

On a :

- `(rev [])` a pour valeur `[]`
- la liste `[xn; ..;x1]` est la valeur de `rev [x1. ..; xn]`
- la liste `[xn; ..;x1; x0]` est la valeur de la concaténation de `[xn; ..;x1]` et de `[x0]`

La fonction rev satisfait donc les deux équations :

$$\begin{cases} (rev\ []) & = & [] \\ (rev\ (x :: xs)) & = & (revxs)\ @\ (x :: []) \end{cases}$$

en voici une définition directe :

```
1 let rec rev (xs:'a list) : 'a list =  match xs with
2 | [] -> []
3 | x::xs -> (rev xs) @ [x]
```

rev : trace de la version naïve

```
(rev [x1; x2; x3; x4]) :
= (rev [x2; x3; x4]) @ (x1::[])
= ((rev [x3; x4]) @ (x2::[])) @ (x1::[])
= (((rev (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])
= ((((rev []) @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])
= ((([] @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])

= (((x4::[]) @ (x3::[])) @ (x2::[])) @ (x1::[])
= ((x4::([] @ (x3::[]))) @ (x2::[])) @ (x1::[])
= ((x4::(x3::[])) @ (x2::[])) @ (x1::[])
= (x4::((x3::[]) @ (x2::[]))) @ (x1::[])
= (x4::x3::([] @ (x2::[]))) @ (x1::[])
= (x4::x3::(x2::[])) @ (x1::[])
= x4::((x3::(x2::[])) @ (x1::[]))
= x4::x3::((x2::[]) @ (x1::[]))
= x4::x3::x2::([] @ (x1::[]))
= x4::x3::x2::(x1::[]
```

rev : autres versions

en utilisant un accumulateur:

```
1  let rec rev_aux (xs:'a list) (r:'a list) = match xs with
2    | [] -> r
3    | x::xs -> (rev_aux xs (x::r))
4
5  let rev (xs:'a list) : 'a list =
6    (rev_aux xs [])
```

avec rev_append (module List) :

```
1  let rec rev_append l1 l2 =
2    match l1 with
3    |  [] -> l2
4    | a :: l -> rev_append l (a :: l2)
5
6  let rev l = rev_append l []
```

Schémas d'itération

Le module `List` contient également nombre de fonctionnelles qui correspondent à des schémas génériques de traitement ou d'exploration de listes.

- Schéma d'application : `List.map : ('a -> 'b) -> 'a list -> 'b list` telle que `(List.map f [x1; ..; xn])` a pour valeur la liste `[(f x1); ..; (f xn)]`.
- Schéma de filtrage : `List.filter : ('a -> bool) -> 'a list -> 'a list` telle que `(List.filter p xs)` donne la liste des éléments `xi` de `xs` pour lesquels `(p xi)` vaut `true`.
- Schémas d'accumulation :
  `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` telle que `(List.fold_right f [x1; ..; xn] a)` donne la valeur de l'expression `(f x1 .. (f xn a)..)`.
  `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que `(List.fold_left f a [x1; ..; xn])` donne la valeur de l'expression `(f (.. (f a x1) ..) xn)`.

Exemples d'itérateurs (1)

- `map` :

```
# let rec map (f : 'a -> 'b) (xs : 'a list) : ('b list) =
   match xs with
   | [] -> []
   | x::xs -> (f x)::(map f xs)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `filter` :

```
# let rec filter (p : 'a -> bool) (xs : 'a list) : ('a list) =
  match xs with
  | [] -> []
  | x::xs -> if (p x) then x::(filter p xs)
     else (filter p xs)              ;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Exemples d'itérateurs (2)

- `fold_right` :

```
# let rec fold_right (f : 'a -> 'b -> 'b) (xs : 'a list) (b : 'b) : 'b =
    match xs with
  | [] -> b
  | x::xs -> (f x (fold_right f xs b))
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

`right f [e1; e2; e3] r ==> f e1 (f e2 (f e3 r)))`

- `fold_left` :

```
# let rec fold_left (f : 'a -> 'b -> 'a) (a : 'a) (xs : 'b list) : 'a =
  match xs with
  | [] -> a
  | x::xs -> (fold_left f (f a x) xs)
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

`left f r [e1; e2; e3]  ==> f (f (f r e1) e2) e3`

Utilisation des itérateur `fold_*` (1)

```
# let longueur1 (l:'a list) : int =
    List.fold_left (fun x y -> x+1) 0 l
val longueur1 : 'a list -> int = <fun>

# let longueur2 (l:'a list) : int =
    List.fold_right (fun x y -> y + 1) l 0
val longueur2 : 'a list -> int = <fun>

# let somme1  (l:int list) : int =
    List.fold_left (fun x y -> x + y) 0 l
val somme1 : int list -> int = <fun>

# let somme2 (l:int list) : int =
    List.fold_right (fun x y -> x + y) l 0
val somme2 : int list -> int = <fun>
```

L'utilisation de `List.fold_right` ou `List.fold_left` est indiférent dès lors que la fonction itérée est associative et commutative. On peut alors préfèrer `List.fold_left` qui est récursive terminale.

Utilisation des itérateur `fold_*` (2)

- La fonction itérée par `List.map` est simplement le constructeur cons (`::`) :

```
# let map (f : 'a -> 'b) (xs : 'a list) : 'b list =
    List.fold_right (fun x r -> (f x)::r) xs []
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- La fonction itérée par `List.filter` sélectionne les éléments à retenir pour le résultat final :

```
# let filter (p : 'a -> bool) (xs : 'a list) : 'a list =
    List.fold_right (fun x r -> if (p x) then x::r else r) xs []
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Utilisation des itérateur `fold_*` (3)

- version `fold_right` :

```
# let find_max (xs : 'a list) : 'a =
   match xs with
   | [] -> raise (Invalid_argument "find_max")
   | x::xs -> List.fold_right max xs x
val find_max : 'a list -> 'a = <fun>
```

```
 find_max [e1;e2;e3] -> right max [2;3] 1
-> max 2 (right [3] 1) -> max 2 (max 3 (right max [] 1))
-> max 2 (max 3 1) -> max 2 3 -> 3
```

- version `fold_left` :

```
# let find_max (xs : 'a list) : 'a =
  match xs with
  | [] -> raise (Invalid_argument "find_max")
  | x::xs -> List.fold_left max x xs
val find_max : 'a list -> 'a = <fun>
```

```
find max [e1;e2;e3] -> left max 1  [2;3]
-> left max (max 2 1) [3]
-> left max 2 [3] -> left max (max 3 2) [] -> left max 3 [] -> 3
```

`fold` : Récursivité terminale

`List.fold_left` accumule «à l'envers» : on peut définir la fonction rev directement avec `fold_left`, version récursive terminale :

```
# let rev (xs:'a list) : ('a list) =
    List.fold_left (fun r x -> x::r) [] xs
val rev : 'a list -> 'a list = <fun>

# let map (f:'a -> 'b) (xs:'a list) : ('b list) =
    List.fold_left (fun r x -> x::r) []
      (List.fold_left (fun r x -> (f x)::r) [] xs)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

`find` : recherche d'un élément dans une liste

- version utilisant `filter` :

```
# let find (p : 'a -> bool) (xs : 'a list) : 'a =
  match (List.filter p xs) with
  | [] -> raise Not_found
  | x::_ -> x
val find : ('a -> bool) -> 'a list -> 'a = <fun>
```

- Comment écrire un `find` en utilisant un `fold` et en s'arrêtant au premier élément trouvé qui vérifie le prédicat ?
  Idée : déclencher une exception quand un élément trouvé
- version retournant au premier élément trouvé :

```
# let rec find (p : 'a -> bool) (xs : 'a list) : 'a =
  match xs with
  | [] -> raise Not_found
  | x::xs -> if (p x) then x else (find p xs)
val find : ('a -> bool) -> 'a list -> 'a = <fun>
```

Retour sur le typage

- un paramètre de type dans le type des paramètres d'une fonction indique que celle-ci n'explore pas la structure de l'argument passé :

```
# let rec length (l : 'a list) : int = match l with
  | [] -> 0
  | (_ :: t) -> 1 + length t
val length : 'a list -> int = <fun>
```

la fonction `length` ne regarde pas la structure des éléments de la liste.

- un paramètre de type dans le type du résultat qui n'apparait pas dans le type de ses paramètres indique un calcul qui ne dépend pas des paramètres. Cela peut être un calcul qui boucle :

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

Le `'b` du résultat n'apparait dans les types des paramètres.

Constructeurs et enregistrements

```
1   type 'a lin = {tete : 'a ; queue : 'a l1}
2   and  'a l1 = V1 | P1 of 'a lin
3
4   let ml1 = P1 {tete = 1; queue = P1 {tete = 33; queue = P1 {tete = 12; queue↩
        = V1}}}
5
6   type 'a l2 = V2 | P2 of {tete : 'a ; queue : 'a l2}
7
8   let ml2 = P2 {tete = 1; queue = P2 {tete = 33; queue = P2 {tete = 12; queue↩
        = V2}}}
9
10  let calcule_taille o = Obj.reachable_words (Obj.repr o)
11
12  calcule_taille ml1
13
14  calcule_taille ml2
```

Piles (1)

- un constructeur `create`
- 3 opérateurs : `push`, `pop`, `top`
- représentée par une liste

```
# let push e p = e :: p
val push : 'a -> 'a list -> 'a list = <fun>

# let pop p = match p with
  | [] -> failwith "pop"
  | _::q -> q
val pop : 'a list -> 'a list = <fun>

# let top p = match p with
  | [] -> failwith "top"
  | t::_ -> t
val top : 'a list -> 'a = <fun>
```

Piles (2)

évaluateur suffixe d'expressions arithmétiques:

```
1  # let decode p jeton = match jeton with
2    |"+" -> let t1 = top p in let p1 = pop p in let t2 = top p1 in let p2 = pop↩
           p1 in push (t1+t2) p2
3    | "*" -> let t1 = top p in let p1 = pop p in let t2 = top p1 in let p2 = ↩
          pop p1 in push (t1*t2) p2
4      | x -> push (int_of_string x) p
5  val decode : int list -> string -> int list = <fun>
6
7  # let rec eval p l = match l,p with
8    | [],[] -> failwith "eval"
9    | [], s::_  -> s
10   | t::q,_ -> eval (decode p t) q
11 val eval : int list -> string list -> int = <fun>
12 # eval (create()) ["33"; "44"; "+"; "11"; "2"; "+"; "*"];;
13 - : int = 1001
```

Listes d'association (1)

Listes donc chaque élément comprend une clé et une valeur. Il y a une association clé-valeur.

La recherche de la valeur associée à une clé s'effectue de manière séquentielle, de la tête de la liste jusqu'à ce que la clé soit trouvée.

Dans le module `List` :

- `assoc` : 'a -> ('a * 'b) list -> 'b
  `assoc a [...; (a,b); ...]` retourne `b` si `(a,b)` est l'association la plus à gauche
  déclenche l'exception `Not_found` si la clé n'existe pas
- `mem_assoc : 'a -> ('a * 'b) list -> bool` : comme `assoc` mais retourne `true` si la clé existe et `false` sinon

Listes d'association (2)

- recherche :

```
1  let rec recherche (cle:'a)  (l: ('a * 'b) list) : 'b =
2    match l with
3    | [] -> raise Not_found
4    | (c,v)::q ->  if cle = c then v else recherche cle q
```

- enlève une association :

```
1  let rec enleve (cle : 'a) (l : ('a * 'b) list) : ('a * 'b) list =
2    match l with
3    | [] -> []
4    | (c,v)::q -> if cle = c then (enleve cle q) else (c,v)::(enleve cle ↩
         q)
```

Files (1)

Files d'atente (ou queues) :

- 1 constructeur : create
- 2 opérateurs enq (ou add ou push) et deq (ou take ou pop)
- version naïve de enq :

```
# type 'a t = 'a list
# let create() = []
val create : unit -> 'a list = <fun>
# let enq x q = q @ [x]
val enq : 'a -> 'a list -> 'a list = <fun>
# let deq q = match q with
  | [] -> failwith "Empty"
  | h::r -> (h,r)
val deq : 'a list -> 'a * 'a list = <fun>
# let length q = List.length q
val length : 'a list -> int = <fun>
```

Files (2)

- optimisante : utilise 2 listes

```
1  type 'a t = { debut : 'a list ; fin : 'a list }
2
3  exception Empty_queue
4
5  let create () = { debut = []; fin = []  }
6
7  let rec pop q =
8    match q.debut with
9    | [] -> (
10       match List.rev q.fin with
11       | [] -> raise Empty_queue
12       | x::xs -> x, { debut = xs ; fin = [] }
13     )
14   | [x] -> x, { debut = List.rev q.fin; fin = [] }
15   | x::xs -> x, { q with debut = xs } (* {debut = xs; fin = q.fin} *)
16
17 let push elem q =
18   { debut = q.debut ; fin = elem::q.fin }
```

Programmation modulaire

- découpage en *unités logiques* plus petites ;

But: réalisation d'un module séparément des autres modules

Mise en œuvre: un module possède une *interface*, la vérification des interface est effectuée à l'assemblage des différents modules.

Intérêts:

- découpage logique ;
- abstraction des données (spécification et réalisation) ;
- indépendance de l'implantation ;
- réutilisation.

Compilation séparée

- découpage en *unités de compilation*, compilables séparément

programmation modulaire $\neq$ compilation séparée

les 2 approches sont nécessaires:

- Pour cela la spécification d'un module doit être vérifiable par un compilateur :
  - on se limite à la vérification de types
  - l'interface sera spécification de modules
  - et contiendra l'information de typage et de compilation pour les autres modules

Langage de modules d'OCaml

2 parties:

- *structure* : pour la partie réalisation/implantation
- *signature* : pour la partie spécification/interface

Le langage de modules est indépendant du langage de base.

Parallèle entre: le langage de base (*valeur : type*)
et le langage de module (*structure : signature*) ! ! !

Modules simples

Implantation: d'un module est une suite de définitions

- de valeurs y compris fonctionnelles
- de types
- d'exceptions
- de sous-modules

Spécification: d'un module est une suite de déclarations et de spécifications de types.

Notation: une signature sera écrite en MAJUSCULE et une structure en Minuscule dont l'initiale est en majuscule.

Implantation d'un module Nqueue

```
1  module Nqueue =
2
3  struct
4
5   type 'a t = 'a list              (* type unit = () *)
6
7   let create() =  []
8
9   let enq x q = q @ [x]
10
11  let deq q =
12   match q with
13      [] -> failwith "Empty"
14      | h::r -> (h,r)
15
16  let length q = List.length q
17
18 end ;;
```

Synthèse d'une signature

L'exemple précédent donne la signature suivante :

```
1  module Nqueue :
2  sig
3    type 'a t = 'a list
4    val create : unit -> 'a list
5    val enq : 'a -> 'a list -> 'a list
6    val deq : 'a list  -> 'a * 'a list
7    val length : 'a list  -> int
8  end
```

Modules : déclarations encapsulées

modules simples (structures)
⇓
ensemble de définitions

leurs types (signatures)
⇓
ensemble de spécifications de types

```
1  module Example =                (* signature inferee *)
2  struct                           sig
3    type t = int                     type t = int
4    module M =                       module M :
5     struct                           sig
6      let succ x = x+1                 val succ : int -> int
7     end                              end
8    let two = M.succ(1)              val two : int
9  end                              end
```

Accès aux éléments d'un module (1)

L'accès à un élément d'un module se fait par la notation "point".

```
# Nqueue.enq;;
- : 'a -> 'a list -> 'a list = <fun>
```

Y compris pour les champs d'enregistrements :

```
#  module Toto =   struct type t = {x:int; y:int}  end;;
module Toto : sig type t = { x: int; y: int } end
# let u = {Toto.x=3; Toto.y=18};;
val u : Toto.t = {Toto.x=3; Toto.y=18}
```

Ce qui peut être simplifié par l'**ouverture** du module :

```
# open Nqueue;
  # let q = Nqueue.create() in let q1 =  enq "Bob" q in
    enq "Alice" q1 ;;
- : string list = ["Bob"; "Alice"]
```

Accès aux éléments d'un module (2)

Exemple: :

```
# Example.two;;
- : int = 2

# Example.M.succ;;
- : int -> int = <fun>

# Example.M.succ (Example.two);;
- : int = 3
```

Ouverture locale: : Module.(...)

```
# Example.(M.succ two + two) ;;
- : int = 5
# M.succ ;;
Error: Unbound module M
```

Compilation séparée (1)

Unité de compilation: 2 *fichiers*

- 1 fichier d'interface (`.mli`) + 1 fichier d'implantation (`.ml`)

Sans précision:

```
1  module Nom = (
2     struct
3       contenu du fichier nom.ml
4     end :
5     sig
6       contenu du fichier nom.mli
7     end)
```

Correspondance: nom de module et nom de fichier

- module `Nom` correspond aux fichiers : `nom.ml` et `nom.mli`
- environnement de typage : répertoires d'accès aux fichiers

Compilation séparée (2)

fichier interface: : `fqueue.mli`

```
1  type 'a t
2  exception Empty_queue
3  val create : unit -> 'a t
4  val pop : 'a t -> 'a * 'a t
5  val push : 'a -> 'a t -> 'a t
6  val to_list : 'a t -> 'a list
7  val from_list : 'a list -> 'a t
```

Compilation séparée (3)

fichier implantation: : `fqueue.ml`

```
1  type 'a t = { debut : 'a list ; fin : 'a list }
2
3  exception Empty_queue
4
5  let create () = { debut = []; fin = []  }
6
7  let rec pop q =
8    match q.debut with
9    | [] -> begin
10       match List.rev q.fin with
11       | [] -> raise Empty_queue
12       | x::xs -> x, { debut = xs ; fin = [] }
13     end
14   | [x] -> x, { debut = List.rev q.fin; fin = [] }
15   | x::xs -> x, { q with debut = xs }
16
17 let push elem q =
18   { debut = q.debut ; fin = elem::q.fin }
19
20 let to_list q = q.debut @ List.rev q.fin
21
22 let from_list l = {debut = l ; fin = [] }
```

Compilation séparée (4)

Compilation:

```
$ ocamlc -c fqueue.mli
$ ocamlc -c fqueue.ml
```

Fichiers objet:

```
$ ls fqueue.cm*
fqueue.cmi fqueue.cmo
```

en natif:

```
$ ocamlopt -c fqueue.mli
$ ocamlopt -c fqueue.ml
$ ls fqueue.cm*
fqueue.cmi fqueue.cmx
```

Compilation séparée (5)

utilisation: `fmain.ml`

```
1  let q = Fqueue.create() ;;
2
3  let main () =
4    let q1 = Fqueue.push 3 q in
5    let q2 = Fqueue.push 4 q1 in
6    let t1,nq1 = Fqueue.pop q2 in
7    let t2,nq2 = Fqueue.pop nq1 in
8    let _ = print_int t1 in
9    let _ =  print_string " " in
10   let _ =  print_int t2 in
11   print_newline();;
12
13 main() ;;
```

compilation:

```
$ ocamlc fqueue.cmo fmain.ml -o main.exe
```

Exécution:

```
$ ./main.exe
3 4
```

Ouverture d'un module

- global

  Syntaxe: **open** mod-name ; ;

  Racourci: de la notation "point"

  Exemple:

```
1  # open Fqueue;;
2  # let q = create();;
3  val q : _'a  Fqueue.t = <abstr>
```

- local

  Syntaxe: **let open** mod-name **in** expr