Introduction à CAML

Solange Coupet Grimal

Cours Logique, Déduction, Programmation 2002–2004*

1 Évaluation par nom et par valeur

1.1 Étude d'un exemple

Dans l'exemple du calcul de la puissance, on aurait pu définir la fonction `puiss` de la façon suivante :

```
# let sqr = fun x -> x*.x;;
val sqr : float -> float = <fun>
# let rec puiss' n x =
  if n=0 then 1.
  else
    if (n mod 2)=0 then
      sqr (puiss' (n/2) x)
    else
      x*.sqr(puiss' ((n-1)/2) x);;
            val puiss' : int -> float -> float = <fun>
# puiss' 20000000 1.;;
- : float = 1.
```

Le résultat est instantané sur la machine protis.

Comment se fait l'évaluation de

`sqr (puiss' (n/2) x)` ?

Lorsqu'on évalue `sqr(puiss' (n/2) x)`, c'est-à-dire

`(fun x -> x*.x) (puiss' (n/2) x)`

deux stratégies sont possibles :

*Texte révisé par Luigi Santocanale le 7 octobre 2005, et adapté au langage Objective Caml.

Évaluation par valeur :

1. Évaluation de la valeur `v` de `(puiss' (n/2) x)`.
2. Évaluation de la valeur de `v*.v`.

Dans cette stratégie on évalue l'argument avant d'appliquer la fonction.

Évaluation par nom :

1. Dans l'expression `x*.x`, on substitue à `x` l'expression `(puiss' (n/2) x)`.
2. On évalue ensuite l'expression obtenue, c'est-à-dire `(puiss' (n/2) x)*.(puiss' (n/2) x)`.

De façon générale, lors de l'évaluation par nom d'une application de la forme

```
( (fun x -> <expr(x)>) argument )
```

on substitue `x` dans `expr(x)` par l'argument *avant* évaluation de celui-ci. Puis on évalue l'expression obtenue.

Avec l'exemple de la fonction `puiss'`, on comprend que l'*évaluation par valeur soit bien plus efficace*. C'est celle qui est réellement implante dans Caml. Cependant les 2 stratégies ne donnent pas toujours les mêmes résultats. L'exemple suivant le montre.

```
# let c = fun x -> 0;;
val c : 'a -> int = <fun>
# let rec f n = n*(f n);;
val f : int -> int = <fun>
# c f;;
- : int = 0
# c (f 5);;
Stack overflow during evaluation (looping recursion?).
```

En effet `(f 5)` est un programme qui boucle. Autrement dit, l'évaluation de `(f 5)` ne termine jamais. L'évaluation par valeur de `(c (f 5))` ne termine donc jamais non plus. Dans une évaluation par nom de `(fun x -> 0) (f 5)`, on commence par remplacer toutes les occurrences de `x` dans l'expression `0` à droite de la flèche par `(f 5)`. Comme `x` n'apparait dans cette expression, `(f 5)` disparait et donc le calcul se termine et la fonction renvoie `0`.

1.2 Stratégie d'évaluation de Caml

Schématiquement une expression sans variable libre se réduit de la façon suivante :
– Constante ou abstraction : rien faire.[1]
– application `(e1 e2)` :
on évalue `e1 --->  e'1`,
on évalue `e2 --->  e'2`,

[1] Une constante est la valeur de elle même. Une expression introduite par `fun ... ->` est sa propre valeur.

si `e'1` est une abstraction de la forme `(fun x -> e''1)`,on substitue `e'2` dans `e''1` en renommant éventuellement des variables et on réitère le processus.

Exemple.

```
# let mult x y = x*y;;
val mult : int -> int -> int = <fun>
# let triple = mult 3;;
val triple : int -> int = <fun>
# let sqr x = x*x;;
val sqr : int -> int = <fun>
# let comp f g x = f (g x);;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Rappelons que l'application est associative gauche : l'espression `(comp sqr triple (2*3))` est donc implicitement parenthésé de cette façon : `(((comp sqr) triple) (2*3))`.

Réduction de `(comp sqr triple (2*3))`

1. Réduction de `(comp sqr triple)` :
   (a) Réduction de `comp sqr` :
   ```
   comp sqr = (fun f g x -> (f (g x)) (fun x -> x*x)  -->
              (fun g x ->(fun x -> x*x)(g x))
   ```
   (b) Réduction de `triple` :
   ```
   triple = mult 3 = (fun x y -> x*y) 3 ---> (fun y -> 3*y)
   ```
   (c) Réduction de `(fun g x ->(fun x -> x*x)(g x))(fun y -> 3*y)` :
   ```
         (fun g x ->(fun x -> x*x)(g x))(fun y -> 3*y)  --->
           (fun x ->(fun x -> x*x)((fun y -> 3*y)x)
   ```
2. Réduction : `2*3 --> 6`,
3. Réduction :
   ```
   (fun x -> (fun x -> x*x)((fun y -> 3*y)x)) 6
            --> (fun x -> x*x)((fun y -> 3*y) 6)
   ```
4. Réduction de `(fun x -> x*x)((fun y -> 3*y) 6)`
   (a) réduction de
   ```
   (fun y -> 3*y) 6 -->3*6 --> 18
   ```
   (b) Réduction de
   ```
   (fun x -> x*x) 18  --> 18 * 18 = 324
   ```

Remarque importante : on ne réduit *jamais* sous un "`fun ->`". Ainsi,

```
fun x -> 2*3*x;;
```

est sous forme réduite.

1.3 Application à la récursion

Considérons :

```
fact = fun n -> if n=0 then 1 else n * fact (n-1)
```

Soit `g` définie par :

```
g = fun f n -> if n=0 then 1 else n * f (n-1)
```

On remarque que `fact = g( fact )`. On dit que `fact` est point fixe de `g`. Lorsque l'on donne la définition `fact` à l'aide de `let rec`, on exprime que `fact` est point fixe de la fonction d'ordre supérieur `g`.

De façon générale, une définition

```
let rec monf = <expr (monf)>;;
```

exprime que `monf` est point fixe de `g = fun f -> <expr(f)>`, i.e. que `g f = f`. On peut redéfinir le `let rec` du langage Caml par une fonction `fix` telle que, pour toute fonction `g` d'ordre supérieur, `fix g` soit point fixe de `g`, c'est-à-dire que `g (fix g) = (fix g)`. D'où la définition :

```
# let rec fix g = g(fix g);;
val fix : ('a -> 'a) -> 'a = <fun>
# let g = fun f n -> if n=0 then 1 else n * f (n-1);;
val g : (int -> int) -> int -> int = <fun>
# let fact = fix g;;
Stack overflow during evaluation (looping recursion?).
```

Pourquoi ?

```
# let rec fix g = g(fun n -> fix g n);;
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# let fact = fix g;;
val fact : int -> int = <fun>
# fact 4;;
- : int = 24
```

Pourquoi ?

2 Définition par cas, filtrage

2.1 Avec `function`

On peut écrire avec `function`[2] des définitions par cas :

[2]Le langage CamlLight permet le filtrage avec `fun`. Ce n'est pas le cas de OCaml.

```
# let neg = function
    true -> false
  | false -> true;;
    val neg : bool -> bool = <fun>
```

Rappelons que `function` accepte un seul argument :

```
# let xor = function
    true true -> false
  | false true -> true
  | true false -> true
  | false false ->false;;
        Characters 24-33:
      true true -> false
      ^^^^^^^^^
The constructor true expects 0 argument(s),
but is here applied to 1 argument(s)
# let xor = function
    (true,true) -> false
  | (false, true) -> true
  | (true,false) -> true
  | (false,false) ->false;;
        val xor : bool * bool -> bool = <fun>
```

On peut aussi utiliser des *motifs* ou *patterns* (en anglais) :

```
# let xor = function
    (false, x)  -> x
  | (true, x) -> neg x;;
    val xor : bool * bool -> bool = <fun>
```

Filtrage (ou pattern matching) : mise en correspondance entre un motif et une valeur effective.

Les cas peuvent ne pas être disjoints. Dans ce cas, à l'appel de la fonction, c'est le premier motif dans l'ordre écrit par le programmeur, filtrant l'argument, qui est utilisé.

Exemple.

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

Exercice : Fibonacci.

```
# let rec fib = function
    0 -> 1
  | 1 -> 1
  | n -> fib(n-1)+fib(n-2);;
      val fib : int -> int = <fun>
```

```
# fib 20;;
- : int = 10946
```

Si on ne prévoit pas tous les cas on obtient un « warning » :

```
# let f = function
    0 -> 1
  |  1 -> 1;;
    Characters 8-39:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
  ........function
      0 -> 1
    |  1 -> 1..
val f : int -> int = <fun>
# f(0);;
- : int = 1
# f(1);;
- : int = 1
# f(3);;
Exception: Match_failure ("", 24, -38).
```

De même, si on ne passe jamais dans un cas :

```
# let f = function
    n -> 2*n
  | 0 -> 1;;
    Characters 34-35:
Warning: this match case is unused.
    | 0 -> 1;;
      ^
val f : int -> int = <fun>
```

Les variables employées dans un même motif doivent être *toutes différentes*, par exemple on ne peut pas utiliser un motif de la forme `(x,y,y)` comme dans l'exemple suivant :

```
# function (x,y,y) -> x + y
  | (x,y,z) -> x + y + z;;
  Characters 14-15:
  function (x,y,y) -> x + y
                ^
This variable is bound several times in this matching
```

Mais on peut employer `_` qui filtre tout, en particulier des valeurs différentes dans un même motif.

```
# let xor = function
    (true, true)  -> false
  | (false,false)   -> false
```

```
    |   _       -> true;;
        val xor : bool * bool -> bool = <fun>
```

2.2 Filtrage avec `match ... with`

```
match e with
    p1  -> e1
  | p2  -> e2
...

  | pn -> en ;;
```

renvoie la valeur de `ei` si `pi` est le premier motif auquel `e` correspond.

```
# match (5, 3) with
    (_ ,0) -> 0
  |(x,y)  -> x/y;;
    - : int = 1
```

est évalué en 1.

On peut utiliser `match ... with ...` avec `fun` ou `function` pour écrire des fonctions qui « filtrent » deux arguments :

```
# let xor = fun x y -> match (x,y) with
    (true, true)  -> false
  | (false,false)   -> false
  |   _       -> true;;
      val xor : bool -> bool -> bool = <fun>
```

3 Polymorphisme

On a déjà vu les projections `fst` et `snd` :

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
```

`'a` et `'b` sont des variables de types. On dit que `fst` et `snd` sont polymorphes puisqu'elles peuvent avoir des types différents selon les cas.

Dans `fst(true,0)`, `fst` a pour type `bool * int -> bool`.

Considérons la fonction identité :

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

```
# id (4,(id(5),id(true)));;
- : int * (int * bool) = (4, (5, true))
```

Ici, la variable de type `'a` est remplacé par le type `int * (int * bool)` dans la première occurrence, par `int` dans la seconde et par `bool` dans la troisième.

La fonction qui calcule la composition de 2 fonctions `f` et `g` est définie par :

```
# let comp f g x = f(g(x));;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Le type synthétisé par Caml est le plus général possible. On peut le *particulariser* en rajoutant une contrainte qui n'est acceptée que si elle est compatible avec les autres.

```
# let comp (f:bool->bool) (g:int->bool) =
  fun x -> f(g(x));;
  val comp : (bool -> bool) -> (int -> bool) -> int -> bool = <fun>
# let (comp : (bool -> bool) -> (int -> bool) -> int -> bool) =
  fun f g x -> f(g(x));;
  val comp : (bool -> bool) -> (int -> bool) -> int -> bool = <fun>
```

La synthèse de type s'appuie sur un algorithme dit *d'unification*. Autre exemple :

```
# let
  id = fun x->x
in
  (id 1, id true);;
      - : int * bool = (1, true)
```

Ici, let `id = fun x -> x` est une définition *locale* à l'expression retournée qui est `(id 1, id true)`. Dans la définition locale `id` est de type `'a->'a` et `'a` est remplacé respectivement par `int` puis par `bool`.

4 Les listes

Remarque préliminaire : il existe une fonction prédéfinie `failwith` :

```
# failwith "essai";;
Exception: Failure "essai".
```

Le type d'une expression `failwith <chaine>` est arbitraire et évalue de sorte à être compatible avec le type du contexte dans lequel elle apparait.

Le type `list` est prédéfini. Voici quelques exemples :

```
# [1;2;3];;
- : int list = [1; 2; 3]
# [1];;
- : int list = [1]
# ['a';'b';'c'];;
```

```
- : char list = ['a'; 'b'; 'c']
# [];;
- : 'a list = []
# [1;'a'];;
Characters 3-6:
  [1;'a'];;
     ^^^
This expression has type char but is here used with type int
```

Observons qu'il s'agit plutôt d'un constructeur de types ou un type polymorphe : `'a list`.
On introduit une liste par les constructeurs :
`[]` – la liste vide,
`::`– opérateur infixe, rajoute un élément en tête de liste.

```
# fun x y -> x::y;;
- : 'a -> 'a list -> 'a list = <fun>
# 6::[3;1];;
- : int list = [6; 3; 1]
```

On peut alors filtrer une liste `l` en considérant 2 cas :
– soit `l` est de la forme `[]`,
– soit `l` est de la forme `t::q`.

```
match l with
    []   -> expr1
  | (t::q)  -> expr2
```

Nous pouvons écrire les fonctions suivantes :

```
# let hd = function
    []-> failwith "hd"
  | (a::l) -> a;;
    val hd : 'a list -> 'a = <fun>
# hd [];;
Exception: Failure "hd".
# hd [1;2;3];;
- : int = 1
```

```
# let tl = function
    []-> failwith "tl"
  | (a::l) -> l;;
    val tl : 'a list -> 'a list = <fun>
# tl [1;2;3];;
- : int list = [2; 3]
```

```
# tl [1];;
- : int list = []
# tl [];;
Exception: Failure "tl".
```

```
# let rec append l1 l2 =
  match l1 with
      []   -> l2
    |  (x::xs ) -> (x::append xs l2);;
      val append : 'a list -> 'a list -> 'a list = <fun>
# let (@@) = append;;
val ( @@ ) : 'a list -> 'a list -> 'a list = <fun>
# [1;2;3] @@ [4;5;6;7;8];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
# [] @@ [1;2];;
- : int list = [1; 2]
# [1;2] @@ [];;
- : int list = [1; 2]
```

Observons que, à l'aide de la définition `let (@@) = append`, on a défini un opérateur binaire infixe.

En OCAML, un opérateur binaire analogue à `append` est prédéfini. Il est noté `@` :

```
# (@);;
- : 'a list -> 'a list -> 'a list = <fun>
# [1;2;3]@[4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

`rev` : calcule l'inverse d'une liste donnée. Une première version, « naive » :

```
# let rec naive_rev = function
    []  ->  []
  |  (a::l)  -> naive_rev l @@ [a];;
    val naive_rev : 'a list -> 'a list = <fun>
# naive_rev [1;2;3];;
- : int list = [3; 2; 1]
# naive_rev [];;
- : 'a list = []
```

Pourquoi cette première version est-elle naive ? Quelle est la complexité de l'algorithme mis en place ?

```
# let rec revacc acc liste  =
```

```
  match liste with
      []        -> acc
    |  (x::xs)   -> revacc (x::acc) xs;;
      val revacc : 'a list -> 'a list -> 'a list = <fun>
# revacc [1;2;3;4;5] [6;7;8;9];;
- : int list = [9; 8; 7; 6; 1; 2; 3; 4; 5]
# let rev = revacc [];;
val rev : '_a list -> '_a list = <fun>
# rev [1;2;3;4;5;6];;
- : int list = [6; 5; 4; 3; 2; 1]
# rev [];;
- : int list = []
```

Version plus élégante :

```
# let rev liste =
  let rec revacc acc liste =
    match liste with
        []        -> acc
      |  (x::xs)   -> revacc (x::acc) xs
  in
    revacc liste [];;
            val rev : 'a list -> 'a list = <fun>
```

Remarquons que ces opérations sur les listes – `hd`,`tl`,`append` `rev` – sont déjà définies dans le module `List` :

```
# List.hd;;
- : 'a list -> 'a = <fun>
# List.tl;;
- : 'a list -> 'a list = <fun>
# List.append;;
- : 'a list -> 'a list -> 'a list = <fun>
# List.rev;;
- : 'a list -> 'a list = <fun>
```

Dans ce module on peut aussi trouver les itérateurs sur les listes `fold_left` et `fold_right` :

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Si `f` est une fonction à 2 arguments, alors la valeur de

```
List.fold_left f x0 [x1;x2; ...; xn]
```

est la même que

```
f( ... (f (f x0 x1) x2) xn)
```

La valeur de

```
List.fold_right f [xn; ...; x2; x1] x0
```

est par contre la même que

```
f(xn ... (f x2 (f x1 x0) x2))
```

On peut s'apercevoir de la différence à l'aide des exemples suivants :

```
# let f = (fun x y -> x + (y *2) );;
val f : int -> int -> int = <fun>
# List.fold_left f 0 [1;2;3];;
- : int = 12
# List.fold_right f [1;2;3] 0;;
- : int = 17
# List.fold_right f [3;2;1] 0;;
- : int = 11
```

On peut alors définir :

```
# let length = List.fold_left (fun cpt a -> 1+ cpt) 0;;
val length : '_a list -> int = <fun>
# let rev    = List.fold_left (fun reverse a -> a:: reverse) [];;
val rev : '_a list -> '_a list = <fun>
# let concat = List.fold_right (fun  a l -> a::l);;
val concat : '_a list -> '_a list -> '_a list = <fun>
```

Pour finir, remarquons qu'on peut acceder aux noms definis dans un module sans les prefixer avec le nom du module. On fait cela à l'aide de `open`, ce qui revient, pour le module `List`, à :

```
# open List;;
# hd [1;2;3];;
- : int = 1
# fold_left (+) 0 [1;2;3];;
- : int = 6
#
```