

Introduction à CAML

Solange Coupet Grimal

Cours Logique, Dédution, 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;;

#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);;

#puiss' 20000000 1.;;
- : float = 1.0
```

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 :

Évaluation par valeur :

*Texte révisé par Luigi Santocanale le 10 octobre 2004.

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

$$((\text{fun } x \rightarrow \langle \text{expr}(x) \rangle) \text{ argument})$$

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 implantée dans Caml. Cependant les 2 stratégies ne donnent pas toujours les mêmes résultats. L'exemple suivant le montre.

```
#let c = fun x -> 0;;
c : 'a -> int = <fun>
```

```
#let rec f n = n*(f n);;
f : int -> int = <fun>
```

```
#c f;;
- : int = 0
```

```
#c (f 5);;
Uncaught exception: Out_of_memory
```

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'apparaît dans cette expression, `(f 5)` disparaît 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

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

- (e1 e2) :
 - on évalue e1 ----> e'1.
 - on évalue e2 ----> e'2.
 - 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;;
# let triple = mult 3;;
# let sqr x = x*x;;
# let comp f g x = f (g x);;
```

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

1. Réduction de (comp sqr triple) :
 - (a) Réduction de comp sqr :

$$\text{comp sqr} = (\text{fun } f \text{ } g \text{ } x \rightarrow (f \text{ } (g \text{ } x)) \text{ } (\text{fun } x \rightarrow x*x)) \rightarrow (\text{fun } g \text{ } x \rightarrow (\text{fun } x \rightarrow x*x)(g \text{ } x))$$
 - (b) Réduction de triple :

$$\text{triple} = \text{mult } 3 = (\text{fun } x \text{ } y \rightarrow x*y) \text{ } 3 \rightarrow (\text{fun } y \rightarrow 3*y)$$
 - (c) Réduction de (fun g x -> (fun x -> x*x)(g x))(fun y -> 3*y) :

$$(\text{fun } g \text{ } x \rightarrow (\text{fun } x \rightarrow x*x)(g \text{ } x))(\text{fun } y \rightarrow 3*y) \rightarrow (\text{fun } x \rightarrow (\text{fun } x \rightarrow x*x)((\text{fun } y \rightarrow 3*y)x))$$
2. Réduction : 2*3 --> 6,
3. Réduction :

$$(\text{fun } x \rightarrow (\text{fun } x \rightarrow x*x)((\text{fun } y \rightarrow 3*y)x)) \text{ } 6 \rightarrow (\text{fun } x \rightarrow x*x)((\text{fun } y \rightarrow 3*y) \text{ } 6)$$
4. Réduction de (fun x -> x*x)((fun y -> 3*y) 6)
 - (a) réduction de

$$(\text{fun } y \rightarrow 3*y) \text{ } 6 \rightarrow 3*6 \rightarrow 18$$
 - (b) Réduction de

$$(\text{fun } x \rightarrow x*x) \text{ } 18 \rightarrow 18 * 18 = 324$$

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

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

est sous forme réduite.

²rappelons que l'application est associative gauche. Donc une telle expression doit être parenthésé comme (((comp sqr) triple) (2*3)).

1.3 Application à la récursion

Considérons :

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

Soit F définie par :

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

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

De façon générale, une définition

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

exprime que `my_f` est point fixe de $F = \text{fun } f \rightarrow \text{<expr(f)>}$, i.e. que $F f = f$. On peut redéfinir le "let rec" en Caml par une fonction "fix" telle que pour toute fonction F d'ordre supérieur, $(\text{fix } F)$ soit point fixe de F c'est-à-dire que $F (\text{fix } F) = (\text{fix } F)$. D'où la définition :

```
#let rec fix F = F(fix F);;  
fix : ('a -> 'a) -> 'a = <fun>
```

```
#let F = fun f n -> if n=0 then 1 else n * f (n-1);;  
F : (int -> int) -> int -> int = <fun>
```

```
#let fact = fix F;;  
Uncaught exception: Out_of_memory
```

Pourquoi?

```
#let rec fix F = F(fun n -> fix F n);;  
fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

```
#let fact = fix F;;  
fact : int -> int = <fun>
```

Pourquoi?

```
#fact 4;;  
- : int = 24
```

2 Définition par cas, filtrage

2.1 Avec fun et function³

On peut écrire avec fun des définitions par cas :

```
#let neg = fun true -> false
           | false -> true;;
```

```
neg : bool -> bool = <fun>
```

Exemple avec une fonction à deux arguments :

```
#let xor = fun true true -> false
           | false true -> true
           | true false -> true
           | false false -> false;;
```

```
xor : bool -> bool -> bool = <fun>
```

On peut aussi utiliser des "motifs" ou "pattern"

```
#let xor = fun false x -> x
           | true x -> neg x ;;
```

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 = fun 0 -> 1
                 | n -> n*fact(n-1);;
```

```
fact : int -> int = <fun>
```

Exercice : Fibonacci.

³Ce qui suit s'applique au langage CAMLlight seulement. En OCAML on peut faire du filtrage à l'aide de `function` et `match ... with` seulement, en tenant compte que le filtrage est sur le premier argument seulement.

```
# let rec fib = fun 0 -> 1
                  | 1 -> 1
                  | n -> fib(n-1)+fib(n-2);;
```

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

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

```
#let f = fun 0 -> 1
          | 1 -> 1;;
```

```
Toplevel input:
>.....fun 0 -> 1
>          | 1 -> 1..
Warning: this matching is not exhaustive.
f : int -> int = <fun>
```

```
#f(0);;
- : int = 1
```

```
#f(1);;
- : int = 1
```

```
#f(3);;
Uncaught exception: Match_failure ("", 7363, 7394)
```

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

```
#let f = fun n -> 2*n
          | 0 -> 1;;
```

```
Toplevel input:
>          | 0 -> 1;;
>          ^
Warning: this matching case is unused.
f : int -> int = <fun>
```

Les variables employées dans un même motif doivent être *toutes différentes*. Mais on peut employer `_` qui filtre tout, en particulier des valeurs différentes dans un même motif.

Exemple.

```
#let xor= fun true true -> false
```

```

        | false false -> false
        | _         _   -> true  ;;
xor : bool -> bool -> bool = <fun>

```

Attention, La définition ci-dessous est incorrecte :

```

#let xor= fun   true  true    -> false
               | false false -> false
               | _         -> true;;

```

Toplevel input:

```

>.....fun      true  true    -> false
>               | false false -> false
>               | _         -> true..

```

This curried matching contains cases of different lengths.

```

#let xor = fun   (true, true) -> false
                 |(false,false)-> false
                 | _         -> true  ;;
xor : bool * bool -> bool = <fun>

```

On peut aussi filtrer avec `function` au lieu de `fun`, mais `function` n'accepte le filtrage que sur un seul motif par cas :

```

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

xor : bool * bool -> bool = <fun>

```

```

#let xor  = function   true  true  -> false
                      | false false -> false
                      | _         _   -> true  ;;

```

Toplevel input:

```

>               | _         _   -> true  ;;
>

```

Syntax error.

Donc il faut retenir que le filtrage avec `function` est déconseillé quand il y a plusieurs arguments.

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.

Exemple.

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

est évalué en 1.

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 qu'elles peuvent avoir des types différents selon les cas.

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

Considérons la fonction identité :

```
#let id x = x;;
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));;
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));;

comp : (bool -> bool) -> (int -> bool) -> int -> bool = <fun>

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

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;;
- : string -> 'a = <fun>

#failwith "essai";;
Uncaught 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 apparaît.

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'];;
```

Toplevel input:

```
>[1; 'a'];;  
>~~~~~
```

This expression has type `char list`,
but is used with type `int list`.

Observons qu'il s'agit plutôt d'un constructeur de types ou un type polymorphe. 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 `a::as`.

```
match l with  
  []      -> e1  
| (a::as) -> e2  ;;
```

Nous pouvons écrire les fonctions suivantes :

hd : renvoie le premier élément d'une liste non vide.

```
#let hd = function []-> failwith "hd"
              | (a::l) -> a;;
```

```
hd : 'a list -> 'a = <fun>
```

```
#hd [];;
```

```
Uncaught exception: Failure "hd"
```

```
#hd [1;2;3];;
```

```
- : int = 1
```

tl : renvoie la queue d'une liste non vide.

```
#let tl = function []-> failwith "tl"
              | (a::l) -> l;;
```

```
tl : 'a list -> 'a list = <fun>
```

```
#tl [1;2;3];;
```

```
- : int list = [2; 3]
```

```
#tl [1];;
```

```
- : int list = []
```

```
#tl [];;
```

```
Uncaught exception: Failure "tl"
```

append : calcule la concaténation de 2 listes.

```
#let rec append l1 l2 =
```

```
  match l1 with
```

```
    [] -> l2
```

```
  | (x::xs) -> (x::append xs l2) ;;
```

```
append : 'a list -> 'a list -> 'a list = <fun>
```

```
#let (@@) = append;;
```

```
#[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]
```

En OCAML, l'opérateur @ est prédéfini :

```
# (@);;
- : '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];;
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 ;;

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 [];;
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 [] ;;
```

En CAMLlight les itérateurs sur les listes

```
it_list f a [b1; b2; ... ;bn] = (f ... (f (f a b1) b2) ... bn)  
                                a si la liste est vide
```

```
list_it f [a1; a2; ... ; an] b = (f a1 (f a2 ... (f an b)) ... )  
                                b si la liste est vide
```

sont prédéfinis. En OCAML, ces itérateurs sont appelés `fold_left` et `fold_right` :

```
# open List;;  
# fold_left;;  
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>  
# fold_right;;  
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>  
# let list_it = fold_left ;;  
list_it : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>  
# let it_list = fold_right ;;  
it_list : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

On peut alors définir :

```
# let length = list_it (fun cpt a -> 1+ cpt) 0;;  
# let rev     = list_it (fun reverse a -> a:: reverse) [];;  
# let concat = it_list (fun a l -> a::l);;
```