Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse

Pierre Letouzey

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

20 février 2023

L'ordre d'évaluation

- Quand on programme, on a souvent envie de, ou besoin de, savoir dans quel ordre nos commandes ou expressions sont évaluées :
  - C'est parfois utile de savoir pour des questions d'efficacité
  - C'est souvent nécessaire de savoir quand il y a des effets de bord
- Il n'est pas toujours simple de répondre, avec les langages actuels.
- Par exemple :
  - Ordre d'évaluation des arguments dans l'appel d'une fonction (procédure, méthode) ?
  - Ordre d'évaluation des sous-expressions combinées par un opérateur ?

Exemple : C

```c
#include <stdio.h>

int foo (int x, int y)
{
  return x+y;
}

int main ()
{
  int i = 1;

  /* order of evaluation of arguments */
  printf("%d\n",foo (i++,i--));

  /* order of evaluation of summands  */
  printf("%d\n",foo(i++,1)+foo(i--,1));
}
```

Sur l'exemple en C

- On obtient des résultats différents avec les compilateurs gcc (version 10.2.1) et clang (version 11.0.1).
- C Standard, subclause 6.5 [ISO/IEC 9899 :2011] :

  Except as specified later, side effects and value computations of subexpressions are unsequenced.

- C.-à-d. : L'ordre d'évaluation des arguments dans un appel de fonction n'est pas spécifié.
- C'est pour traiter ces questions qu'on étudie la *sémantique* des langages de programmation !

La stratégie d'évaluation dans les langages fonctionnels

Pour les langages fonctionnels, une question majeure est de savoir ce qu'il se passe quand on applique une fonction :

1. Est-ce que le compilateur essaye d'évaluer le corps d'une fonction avant qu'elle soit appliquée ?
   Si on écrit `fun x -> x+fact(100)`, est-ce que le factoriel est calculé à chaque appel ou une seul fois ?
2. Si on écrit une expression `(e a)` (e appliquée à a) est-ce qu'on commence par évaluer e ou a ? Attention, dans les langages fonctionnels les deux sont des expressions.
3. Si on écrit `(fun x -> e) a`, est-ce qu'on évalue l'argument `a` d'abord, ou fait-on le passage de paramètres d'abord ?
4. On ne peut pas trouver la réponse à ces questions par des tests car le comportement peut être non spécifié !

Question 1 : Pas d'évaluation "sous le λ"

- OCaml n'essaye pas d'évaluer le corps d'une fonction avant qu'elle ne soit appliquée à un argument.
- Tous les langages fonctionnels font ce choix.
- Il y a plein de bonnes raisons d'arrêter l'évaluation quand on rencontre une abstraction (aussi appelée "lambda" du λ-calcul utilisé dans le cours de *Sémantique* pour traiter ces questions de façon générale).
- Ne pas confondre avec des *optimisations* de code faites par le compilateur, comme la propagation de constantes.

Exemples (lambda.ml)

```
let f = function x -> 1/0 * x;;
(* pas d'erreur *)

f 42;;
(* exception division par zero *)
```

Question 2 : ordre non spécifié

- OCaml manual, section 6.7 (Expressions) :

  The expression expr $arg_1 \ldots arg_n$ *....*
  The order in which the expressions expr, $arg_1, \ldots, arg_n$ *are evaluated is not specified.*

- Si on a vraiment besoin d'un ordre spécifique il faut le forcer avec la construction `let ... in ...`

Exemples (order.ml)

```
(* l'ordre n'est pas spécifié *)
(print_string "gauche\n"; fun x -> x)
(print_string "droite\n"; 42)
;;

(* forcer un ordre d'évaluation *)
let f = print_string "gauche\n"; fun x -> x
in f (print_string "droite\n"; 42)
```

Conséquence d'un mauvais ordre d'évaluation

- Un ordre d'évaluation non attendu peut avoir des conséquences néfastes dans le cas d'effets de bord.
- Problème : on n'est pas toujours conscient des effets de bord qui peuvent se produire.
- Exemple : une fonction qui lit les lignes d'un fichier et qui renvoie leur concaténation.
- Les opérations d'expressions régulières peuvent aussi avoir des effets de bord !

Exemples (read1.ml)

```
(* Wrong: relies on evaluation order *)

let rec read ic =
  try
    (input_line ic) ^ (read ic)
  with
    End_of_file -> ""
;;

print_string (read (open_in "myfile"))
```

Exemples (read2.ml)

```
(* Function read corrected *)

let rec read ic =
  try
    let thisline = input_line ic
    in thisline ^ (read ic)
  with
    End_of_file -> ""
;;

print_string (read (open_in "myfile"))
```

Question 3 : On évalue l'argument avant de le passer en paramètre

- Le choix de OCaml est d'évaluer l'expression fonctionnelle et les arguments d'abord.
- Ce choix n'est pas le seul possible : d'autres langages fonctionnels, comme Haskell, passent d'abord l'argument, non évalué, en paramètre à la fonction, et ne lancent le calcul qu'au moment où on aura besoin de son résultat.
- Cela permet à Haskell, par exemple, de ne jamais calculer `fact 100` dans une expression `(fun x -> 3) (fact 100)`

Exemples (strict.ml)

```
(* l'argument d'abord ... *)
(fun x -> print_string "corps\n"; x + x)
  (print_string "argument\n";35+24);;

(* l'argument est toujours évalué *)
(fun x -> print_string "corps\n"; 0)
  (print_string "argument\n";35+24);;
```

OCaml fait de l'évaluation stricte

- L'ensemble de ces choix s'appelle une *stratégie d'évaluation*.
- Celle utilisée par OCaml est appelée *évaluation stricte*.
- Toutes les fonctions f qu'on peut écrire en OCaml sont *strictes* : $f(\bot) = \bot$, où $\bot$ est un calcul infini.
- (voir plus en profondeur dans le cours de Sémantique).

Évaluation stricte ou paresseuse ?

- Avantages des l'évaluation stricte :
  - plus facile à mettre en œuvre.
  - la complexité est plus prévisible.
- Avantages de l'évaluation paresseuse :
  - un argument non utilisé n'est pas évalué.
  - permet de travailler avec des structures infinies !
- Inconvénient de l'évaluation paresseuse : il nous faut un mécanisme pour mémoriser un argument évalué (pour éviter qu'il soit évalué plusieurs fois).

Structures infinies

- Exemple : Pour écrire un algorithme combinatoire, on a besoin de calculer souvent le factoriel d'un entier naturel.
- Nous ne voulons pas le recalculer à chaque fois qu'on en a besoin ; on préfère garder la liste de tous les factoriels.
- Pour cela, on écrit le code suivant, mais on s'aperçoit qu'il ne fait pas ce que l'on veut (pourquoi ?) :

```
let rec fact = function 0->1 | n -> n*(fact (n-1));;
let rec fact_from n = (fact n)::(fact_from (n+1));;
let fact_nat = fact_from 0;;
```

Évaluation paresseuse

Notre défi :

- on veut que la liste (infinie) ne soit pas calculée tout de suite
- mais seulement au fur et à mesure, quand on a besoin d'en prendre des éléments

Évaluation paresseuse “du pauvre”, via des fermetures

- En profitant du fait qu'OCaml n'évalue pas le corps d'une fonction, il est possible de simuler un calcul paresseux en protégeant le calcul par une fonction.
- On change la définition du type des listes pour prévoir une “queue” de liste dont l'évaluation est bloquée sous une abstraction.
- Ici, une liste infinie paresseuse a donc la forme

$$\texttt{fun () -> Cons(}e_1\texttt{, fun () -> Cons(}e_2\texttt{, ...))}$$

où e_i est l'expression (non évaluée car protégée par l'abstraction) qui donne le i-ème élément de la liste.

```
let fact n =
  let rec calc = function 0 -> 1 | n -> n*(calc (n-1))
  in Printf.printf "calcule␣fact(%d)\n" n; calc n

type 'a lazylist = unit -> 'a lazycell
and 'a lazycell = Nil | Cons of 'a * 'a lazylist

let rec fact_from n : int lazylist =
  fun () -> Cons (fact n, fact_from (n+1))

let rec take n s =
  if n = 0 then []
  else match s () with
       | Nil -> []
       | Cons(v,r) -> v::(take (n-1) r)

let fact_nat = fact_from 0
let _ = take 5 fact_nat
let _ = take 5 fact_nat
```

```
(* Pour une liste infinie, pas besoin de cas de base *)
type 'a lazylist = unit -> 'a lazycell
and 'a lazycell = Cons of 'a * 'a lazylist

let rec fact_from n =
  fun () -> Cons (fact n, fact_from (n+1))

let rec take n s =
  if n = 0 then [] else
    match s () with Cons(v,r) -> v::(take (n-1) r)

let fact_nat = fact_from 0
let _ = take 5 fact_nat

(* Et pour calculer les fact de proche en proche : *)
let rec fact_from2 n p =
  fun () -> Cons (p, fact_from2 (n+1) ((n+1)*p));;

take 10 (fact_from2 0 1);;
```

Limites de notre évaluation paresseuse “du pauvre”

- Cette utilisation des fermetures permet d’écrire des structures de données paresseuses, mais ce n’est pas encore ce que nous voulons !
- Notre code permet de décrire des listes infinies, *mais* chaque fois qu’on visite la même liste, on relance le calcul de ses éléments. C’est très inefficace !

La bonne solution : Paresse + *partage* !

- Chaque fois qu'une partie d'une structure de données paresseuse est *dégelée* et calculée, on veut qu'elle soit remplacée, silencieusement, par le résultat de ce calcul dans la structure de donnée.
- La prochaine fois qu'on y accède, on veut retrouver la partie de la liste infinie déjà évaluée.
- Essayons une solution "moins pauvre" (mais impérative)

```
(* Une valeur paresseuse sans calculs redondants *)

type 'a work =
  | Later of (unit -> 'a)
  | Done of 'a
type 'a mylazy = 'a work ref

let mklazy (f : unit -> 'a) : 'a mylazy = ref (Later f)

let force (r : 'a mylazy) : 'a =
  match !r with
  | Done x -> x
  | Later f -> let x = f () in r := Done x; x

let ex = mklazy (fun () -> Printf.printf "calcul\n"; 42)
let _ = force ex
let _ = force ex
```

```
(* Listes paresseuses utilisant mylazy *)

type 'a lazylist = 'a lazycell mylazy
and 'a lazycell = Cons of 'a * 'a lazylist

let rec fact_from n =
  mklazy (fun () -> Cons (fact n, fact_from (n+1)))

let fact_nat = fact_from 0

let rec take (n:int) (s:'a lazylist) =
  if n=0 then []
  else match force s with Cons(v,r) -> v::(take (n-1) r)

let _ = take 5 fact_nat
let _ = take 7 fact_nat
```

Une solution efficace *et* fonctionnelle ?

- C'était un exercice un peu pénible ... et impératif
- Peut-on simplement écrire une fonction *lazy*, telle que *(lazy e)* donne l'expression *e* en forme paresseuse ?
- Non, *lazy* ne peut pas être une fonction, car OCaml évalue toujours l'argument avant d'appliquer une fonction.
- Solution jusqu'ici : *(mklazy (fun () -> e))*
- Sinon, il nous faut un support par le système OCaml :
  - un mot-clé spécifique *lazy* créant de la paresse
  - un module `Lazy` fournissant `Lazy.force` et autre

Le module Lazy de la librairie OCaml

```
type 'a t
val force : 'a t -> 'a
val is_val : 'a t -> bool
val from_fun : (unit -> 'a) -> 'a t
```

- une valeur de type `'a Lazy.t` est appelée une *suspension* et contient un calcul paresseux de type `'a`.
- on construit des valeurs de type `'a Lazy.t` avec le mot clé réservé **lazy** : l'expression **lazy** (expr) crée une suspension contenant le calcul expr *sans l'évaluer*.
- `Lazy.force s` force l'évaluation de la suspension s, renvoie le résultat, et remplace la valeur dans la structure de donnée : *sur une suspension déjà dégelée, on ne refait pas le calcul*.
- `Lazy.is_val s` teste si la suspension s est déjà dégelée.

Exemples (stream1.ml)

```
type 'a stream = 'a cell Lazy.t
and 'a cell = Cons of 'a * 'a stream

let rec fact_from n =
  lazy (Cons (fact n, fact_from (n+1)))

let fact_nat = fact_from 0

let rec take n (s:'a stream) =
  if n = 0 then []
  else match Lazy.force s with
       | Cons(v,r) -> v :: (take (n-1) r)

let _ = take 5 fact_nat
let _ = take 7 fact_nat
```

Streams : la liste des factoriels re-visitée avec `Lazy`

- On a placé les `lazy` *exactement* là où on avait mis des fermetures `fun () ->`
- On a placé des `Lazy.force` *exactement* là où on avait forcé l'évaluation en appliquant à l'argument `()` (ou ensuite utilisé notre fonction `force` maison).
- Remarquez qu'un type `s` est distinct du type `(s Lazy.t)`. La présence ou absence de `Lazy.force` est donc vérifiée par le système de typage.
- La promesse est tenue : le calcul des premiers 5 éléments est fait *seulement la première fois*.

Exemples (stream2.ml)

```
(* Version calculant les fact de proche en proche.
   Invariant : p = n! *)

let rec fast_fact_from n p : int stream =
  lazy (Cons (p, fast_fact_from (n+1) ((n+1)*p)))

let fast_fact_nat = fast_fact_from 0 1

let _ = take 50 fast_fact_nat
```

Syntaxe plus moderne

En OCaml il est possible d'utiliser le mot clé `lazy` même dans les motifs utilisés dans les définitions par cas ; cela permet souvent de se passer de l'usage de `Lazy.force`. Par exemple, un morceau de code tel que

```
match Lazy.force s with
| Nil -> ...
| Cons (h, t) -> ...
```

peut s'écrire de façon équivalente comme :

```
match s with
| lazy Nil -> ...
| lazy (Cons (h, t)) -> ...
```

Attention à l'évaluation stricte !

Est-ce que ces deux fonctions ont le même comportement ?

```
let times n s =
  let rec timesrec = function
    | lazy(Cons(h,t)) -> Cons(n*h,lazy(timesrec t))
  in lazy (timesrec s);;

let rec times_bis n = function
  | lazy(Cons(h,t)) -> lazy(Cons(n*h,times_bis n t))

let _ = times 42 (fact_from 0)
let _ = times_bis 42 (fact_from 0)
```

Évaluation stricte et Lazy

- La fonction `times` est complètement paresseuse, car l'appel à `timesrec` est protégé par un `lazy`.
- La fonction `times_bis` n'est pas complètement paresseuse : un argument passé à cette fonction est forcé par le `function`, le `lazy` s'applique seulement à la valeur renvoyée.
- Bref, la fonction `times_bis` met bien une barrière à la récursion, mais elle regarde "un cran trop loin" dans son argument.
- La différence entre les deux peut être importante, comme on verra sur les exemples à venir.
- Note : `times` peut aussi être écrite par récursion mutuelle.

Exemples (times2.ml)

```
(* times via recursion interne *)

let times n s =
  let rec timesrec = function
    | lazy(Cons(h,t)) -> Cons(n*h,lazy(timesrec t))
  in lazy (timesrec s)

(* Equivalent : times via recursion mutuelle *)

let rec times n s =
  lazy (timescell n (Lazy.force s))
and timescell n = function
  | Cons (h,t) -> Cons (n*h,times n t)

let _ = times 42 (fact_from 0)
```

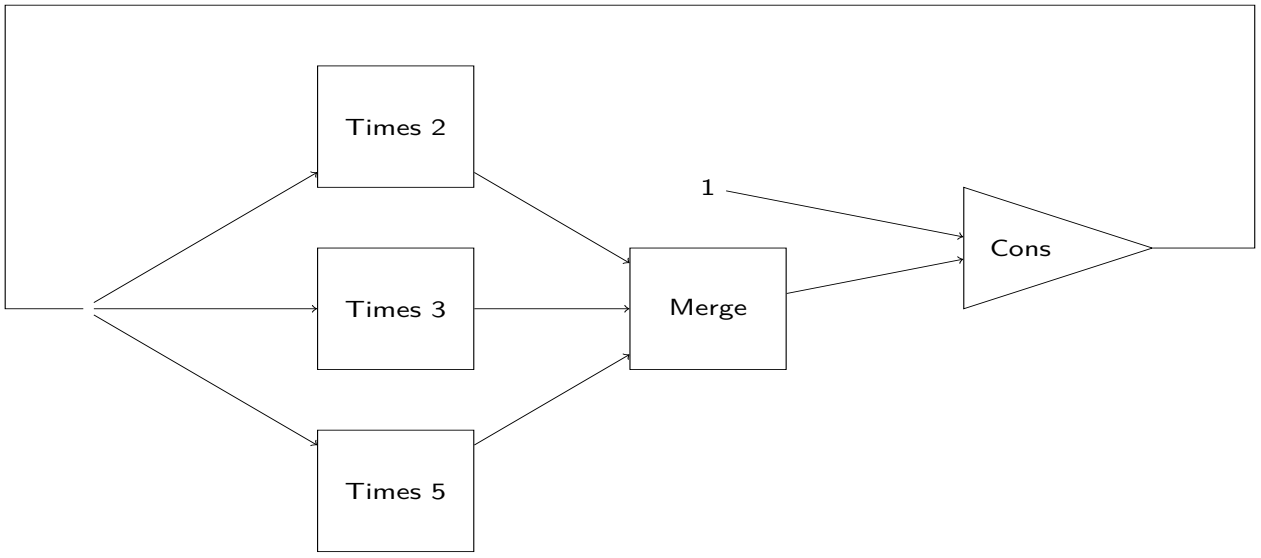
Exemples (times3.ml)

```
(* does not work *)
let rec powers = lazy (Cons(1,times_bis 2 powers))

let _ = take 5 powers

(* works *)
let rec powers = lazy (Cons(1,times 2 powers))

let _ = take 5 powers
```

```
(* Les streams peuvent finir, même sans cas de base *)
exception Empty

let rec countdown n =
  lazy (if n < 0 then raise Empty
        else Cons (n, countdown (n-1)))

let _ = take 11 (countdown 10)
let _ = take 12 (countdown 10)
let _ = take 11 (times 2 (countdown 10))
let _ = take 11 (times_bis 2 (countdown 10))

let rec safe_take n s =
  if n = 0 then []
  else match Lazy.force s with
       | Cons (h,t) -> h::(safe_take (n-1) t)
       | exception Empty -> []

let _ = safe_take 20 (countdown 10)
```

Application : les nombres de Hamming

- La *séquence de Hamming* est le flot de tous les entiers de la forme $2^i * 3^j * 5^k$ pour $i, j, k \geq 0$, *dans l'ordre strictement ascendant*. Le début de ce flot est
  1 2 3 4 5 6 8 9 10 12 15 16 18 . . .
- Schéma :



Exemples (hamming1.ml)

```
let rec merge s1 s2 = lazy (merge_cell s1 s2)
and merge_cell s1 s2 =
  match s1, s2 with
  | lazy (Cons(h1,t1)), lazy (Cons(h2,t2)) ->
     if h1 < h2 then Cons(h1, merge t1 s2)
     else if h1 = h2 then Cons(h1, merge t1 t2)
     else Cons(h2, merge s1 t2)

let merge3 s1 s2 s3 = merge s1 (merge s2 s3)

let rec hamming =
  lazy (Cons (1, merge3 (times 2 hamming)
                        (times 3 hamming)
                        (times 5 hamming)))
let _ = take 100 hamming
```

Exemples (filter1.ml)

```
(* Attention aux situations de famines *)

let rec filter f s =
  lazy (filter_cell f s)
and filter_cell f = function
  | lazy (Cons (h,t)) ->
      if f h then Cons (h, filter f t)
      else filter_cell f t

let rec nums n = lazy (Cons (n, nums (n+1)))
let pairs = filter (fun n -> n mod 2 = 0) (nums 0)
let _ = take 10 pairs (* ok *)
let unicorns = filter (fun n -> n mod 2 = 1) pairs
let _ = take 10 unicorns (* calcul infini *)
```

Quelques opérations de base sur les streams ou flots

- On peut maintenant, à l'aide de `Lazy`, écrire un module pour les flots (séquences potentiellement infinies). En Anglais : `streams`.
- Ne pas confondre avec le module `Stream` qui existe déjà dans la librairie standard OCaml et qui sert à construire des analyseurs récursifs descendants.

Un module pour les streams I

```
module type STREAM = sig
  type 'a stream = 'a cell Lazy.t
  and 'a cell = Nil | Cons of 'a * 'a stream

  (* concatenation de streams *)
  val (++) : 'a stream -> 'a stream -> 'a stream

  (* stream avec juste les n premiers elements *)
  val prefix : int -> 'a stream -> 'a stream

  (* stream sans les n premiers elements *)
  val drop : int -> 'a stream -> 'a stream
  val reverse : 'a stream -> 'a stream
end
```

Un module pour les streams II

```
module Stream : STREAM = struct
type 'a stream = 'a cell Lazy.t
and 'a cell = Nil | Cons of 'a * 'a stream

(* concatenation *)
let rec (++) s1 s2 = lazy (app s1 s2)
and app s1 s2 = match Lazy.force s1 with
   | Nil -> Lazy.force s2
   | Cons (h,t) -> Cons (h, t ++ s2)

(* recopie paresseuse des premiers n elements *)
```

Un module pour les streams III

```
let rec prefix n s = lazy (prefix' n s)
and prefix' n s = match n,s with
  | 0, _ -> Nil
  | _, lazy Nil -> Nil
  | _, lazy (Cons (h,t)) -> Cons (h, prefix (n-1) t

(* suppression de n elements, monolithique! *)
let drop n s =
  let rec drop' n s = match n, s with
    | 0, _ -> Lazy.force s
    | _, lazy Nil -> Nil
    | _, lazy (Cons (_,t)) -> drop' (n-1) t
  in if n = 0 then s else lazy (drop' n s)
```

Un module pour les streams IV

```
(* retourne un stream, monolithique! *)
let reverse s =
  let rec rev acc s = match Lazy.force s with
    | Nil -> acc
    | Cons (h,t) -> rev (Cons (h, lazy acc)) t
  in lazy (rev Nil s)
end
```

Remarques

- Si `s1` est une stream infinie, alors `s1 ++ s2` va se comporter comme `s1`, et `reverse s1` va boucler.
- Les opérations `drop` et `reverse` sont *monolithiques*, i.e. prendre le premier élément du stream résultat va déclencher de nombreux `Lazy.force` :
  - pour `drop n s`, les premiers n éléments de s sont forcés
  - pour `reverse s`, tout le stream s est forcé.

  On ne suspend donc que le début de l'opération, mais une fois qu'on essaye d'en voir le résultat, toute l'opération est exécutée d'un coup.