Programmation Fonctionnelle Avancée
4 : Évaluation paresseuse

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

treinen@irif.fr

10 octobre 2018

© Roberto Di Cosmo et Ralf Treinen

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.
- Parfois utile de savoir pour des questions d'efficacité
- 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

```
#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 8.2.0) et clang (version 6.0.1).
- 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).
- Pas confondre avec des *optimisations* de code faites par le compilateur, comme 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 ... arg_n ....*
  > *The order in which the expressions expr, arg_1, ..., 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 specifie *)
(print_string "gauche\n"; fun x -> x)
(print_string "droite\n"; 42)
;;

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


Conséquence d'un mauvais ordre d'order 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 ... *)
let r = (fun x -> print_string "corps\n"; x + x)
        (print_string "argument\n";35+24);;

(* l'argument est toujours evalue *)
let r = (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* : $f(\perp) = \perp$, où $\perp$ 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 pas 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 = fun 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", avec les fermetures

- En profitant du fait qu'en OCaml on 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.
- Une liste infinie paresseuse a la forme

  `fun () -> Cons(`e_1`, fun () -> Cons(`e_2`, ...))`

  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 lltip = Nil | LCons of 'a * 'a lazylist
and 'a lazylist = unit -> 'a lltip

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

let rec take n s = match n with
 | 0 -> []
 | n -> match s() with Nil -> []
        | LCons(v,r) -> v::(take (n-1) r);;

let fact_nat = fact_from 0;;

take 5 fact_nat;;
```

Limites de notre évaluation paresseuse “du pauvre”

- La petite astuce avec les 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.

Exemples (lazy3.ml)

```
type 'a lltip = LCons of 'a * (unit -> 'a lltip)
and 'a cleverlist_cell =
  | Nil
  | Cons of 'a * 'a cleverlist
  | L of (unit -> 'a lltip)
and 'a cleverlist = 'a cleverlist_cell ref

let fact_nat =
  let rec aux n  =
    (fun () ->  LCons(fact n, aux (n+1)))
  in ref (L (aux 0))
;;
```



```
let rec take_l (n:int) (s:'a cleverlist) =
  let rec take_and_update n ll s =
    (* au debut, !s = L(ll) *)
    if n=0 then []
    else match ll() with
      LCons(v,r) ->
        let sr = ref (L r)
        in
        s := (Cons(v,sr));
        v::(take_and_update (n-1) r sr)
  in
  if n=0 then []
  else match !s with
  | Nil -> []
  | Cons(v,r) -> v::(take_l (n-1) r)
  | L(ll) -> take_and_update n ll s
;;

take_l 5 fact_nat;;

take_l 7 fact_nat;;
```

Une solution efficace *et* fonctionnelle ?

- C'était un exercice un peu pénible ...
- Peut-on simplement écrire un module qui fournit 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.
- Il nous faut un support par le système OCaml : le module `Lazy`, et un mot-clé spécifique.

Le module Lazy de la librairie OCaml

```
type 'a t
exception Undefined
val force : 'a t -> 'a
val lazy_is_val : 'a t -> bool
```

- 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 de l'évaluer*.
- Lazy.force s force l'évaluation de la suspension s, renvoie le résultat, et remplace la valeur dans la structure de donnée : *si appelé sur une partie déjà dégelée, il ne refait pas le calcul*.
- Lazy.lazy_is_val s teste si la suspension s est déjà dégelée.

Exemples (stream1.ml)

```
type 'a streamtip = Nil | Cons of 'a * 'a stream
and  'a stream = 'a streamtip Lazy.t;;

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

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

let fact_nat = fact_from 0;;

take 5 fact_nat;;
```


Streams : la liste des factoriels ré-visitée avec Lazy

- On a placé les `lazy` *exactement* là où on avait mis des fermetures `fun () ->`, et on a placé des `Lazy.force` *exactement* là où on avait forcé l'évaluation en appliquant à l'argument ().
- Remarquez qu'un type s est distingué 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*.
- On peut maintenant modifier le code de telle sorte que le calcul de chaque nouvel élément de `fact_nat` utilise *une seule* multiplication.

Exemples (stream2.ml)

```
let rec fact_from_fast n prev : int stream =
  lazy (let next = n * prev in
        Cons(next, fact_from_fast (n + 1) next));;

let fact_nat_fast =
  lazy (Cons(1,fact_from_fast 1 1)) ;;

take 50 fact_nat_fast;;
```


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 (_, tl) -> ...
```

peut s'écrire de façon équivalente comme :

```
match s with
| lazy Nil -> ...
| lazy (Cons (_, tl)) -> ...
```


Attention à l'évaluation stricte !

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

```
let rec times1 n = function
  | lazy Nil -> lazy Nil
  | lazy (Cons(h,t)) -> lazy (Cons(n*h,times1 n t))

let times2 n s =
  let rec times_aux = function
    | lazy Nil -> Nil
    | lazy (Cons(h,t)) -> (Cons(n*h,lazy (times_aux t)))
  in lazy (times_aux s);;

let _ = times1 42 (fact_from 1);;

let _ = times2 42 (fact_from 1);;
```


Évaluation stricte et Lazy

- La fonction `times1` n'est pas "complètement paresseuse" : un argument passé à cette fonction est forcé par le filtrage par motif, le `lazy` s'applique seulement à la valeur envoyée.
- La fonction `times2` est complètement paresseuse, car l'appel à `times_aux` est protégé par un `lazy`.
- La fonction `times1` met bien une barrière à la recursion, mais elle regarde "un cran trop loin" dans son argument.
- La différence entre les deux peut être importante, comme on verra sur l'exemple suivant.

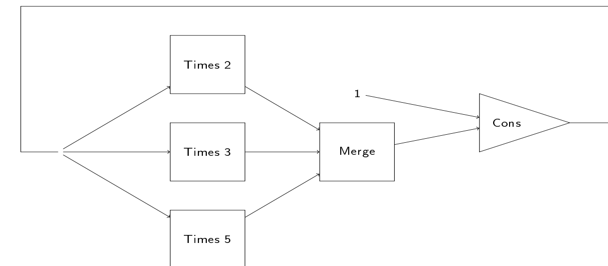
Exemples (times3.ml)

```
(* does not work *)
let rec powers = lazy (Cons(1,times1 2 powers));;

  take 5 powers;;

(* works *)
let rec powers = lazy (Cons(1,times2 2 powers));;

  take 5 powers;;
```

Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse
└Le module Lazy en OCaml

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 :



Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse
└Le module Lazy en OCaml

Exemples (hamming1.ml)

```
(* two-way merge *)
let merge s1 s2 =
  let rec merge_aux s1 s2 =
    match s1 with
    | lazy Nil -> Lazy.force s2
    | lazy (Cons(h1,t1)) ->
       match s2 with
       | lazy Nil -> Cons(h1,t1)
       | lazy (Cons(h2,t2)) ->
          if h1<h2 then
            (Cons(h1,lazy (merge_aux t1 s2)))
          else if h1=h2
          then (Cons(h1,lazy (merge_aux t1 t2)))
          else (Cons(h2,lazy (merge_aux s1 t2)))
  in lazy (merge_aux s1 s2);;
```

Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse
└Le module Lazy en OCaml

Exemples (hamming2.ml)

```
(* three-way merge *)
let merge3 s1 s2 s3 = merge s1 (merge s2 s3);;

let rec hamming = (* does not work *)
  lazy (Cons(1,merge3 (times1 2 hamming)
                      (times1 3 hamming)
                      (times1 5 hamming)));;

let rec hamming = (* correct *)
  lazy (Cons(1,merge3 (times2 2 hamming)
                      (times2 3 hamming)
                      (times2 5 hamming)));;

take 100 hamming;;
```

Les 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 streamhead = Nil | Cons of 'a * 'a stream
  and 'a stream = 'a streamhead Lazy.t

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

  (* un stream contenant les premiers n elements *)
  val take : int -> 'a stream -> 'a stream

  (* le stream sans les premiers n 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 streamhead = Nil | Cons of 'a * 'a stream
and 'a stream = 'a streamhead Lazy.t

(* concatenation *)
let (++) s1 s2 =
  let rec aux s = match s with
    | lazy Nil -> Lazy.force s2
    | lazy (Cons (hd, tl)) -> Cons (hd, lazy (aux tl))
in lazy (aux s1)
```


Un module pour les streams III

```
(* copie paresseuse des premiers n elements *)
let take n s =
  let rec take' n s = match n, s with
  | 0, _ -> Nil
  | n, s -> match Lazy.force s with
            | Nil -> Nil
            | Cons (hd, tl) ->
              Cons (hd, lazy (take' (n - 1) tl))
in lazy(take' n s)

(* le stream apres n elements, monolithique! *)
let drop n s =
  let rec drop' n s =
    match n with 0 -> Lazy.force s
```

Un module pour les streams IV

```
      | n -> match s with
             | lazy Nil -> Nil
             | lazy (Cons (_, tl)) -> (drop' (n - 1) tl) in
  match n with 0 -> s | n -> lazy (drop' n s)

(* retourne un stream, monolithique! *)
let reverse s =
  let rec reverse' acc s = match s with
    | lazy Nil -> acc
    | lazy (Cons (hd, tl)) ->
        reverse' (Cons (hd, lazy acc)) tl in
  lazy (reverse' Nil s)
end;;
```

Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse
└Le module Lazy en OCaml

Exercices

Réalisez les fonctions supplémentaires suivantes :

```
module type STREAMEXT = sig
  include module type of Stream
  val of_list : 'a list -> 'a stream
  val to_list : 'a stream -> 'a list
  val push : 'a -> 'a stream -> 'a stream
  val pop : 'a stream -> ('a * 'a stream) option
  val map : ('a -> 'b) -> 'a stream -> 'b stream
  val filter : ('a -> bool) -> 'a stream -> 'a stream
  val split : 'a stream -> 'a stream * 'a stream
  val join: 'a stream * 'a stream -> 'a stream
end;;
```

Assurez vous que votre réalisation est bien paresseuse.

Programmation Fonctionnelle Avancée 4 : Évaluation paresseuse
└Le module Lazy en OCaml

Remarques

Les opérations `drop` et `reverse` sont *monolithiques*, dans le sens que, dès qu'on essaye de prendre le premier élément du stream résultant :

- 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.