Programmation Fonctionnelle
Cours 10
Efficacité

Delia Kesner

Table de matières

Utilisation de ressources par OCaml

- Ressources limitées :
  1. La mémoire est une ressource nécessaire pour stocker (temporairement) des valeurs créées par les programmes.
  2. Le temps d'exécution est aussi une ressource.
  3. D'autres ressources, comme les communications avec l'extérieur.
- Comment OCaml gère ces ressources ?
- Comment écrire des programmes fonctionnelles qui sont plus efficaces (i.e. moins gourmands en ressources) ?

Allocation dynamique de mémoire

- On est souvent amené à construire des valeurs intermédiaires qui n'ont pas besoin d'exister pendant toute la vie du programme.
- Est-ce que la construction fréquente des valeurs intermédiaires consomme de plus en plus de mémoire ?
- Exemple : produit scalaire de deux vecteurs. On construit d'abord une liste des produits des composantes (valeur intermédiaire), puis on calcule sa somme.

Produit scalaire I

```
let scp v1 v2 =
  List.fold_left (+) 0 (List.map2 ( * ) v1 v2);;
# val scp : int list -> int list -> int = <fun>

scp [2;4;6] [1;3;5];;
# - : int = 44
```

Version alternative avec calcul intermédiaire I

```
let scp_bis v1 v2 =
  let liste_des_produits = List.map2 ( * ) v1 v2
  in List.fold_left (+) 0 liste_des_produits;;
# val scp_bis : int list -> int list -> int = <fun>

scp_bis [2;4;6] [1;3;5];;
# - : int = 44
```

Cas 1 : Eválua tion des expressions

- Exemple : produit scalaire de deux valeurs.
- Une valeur intermédiaire est nécessaire pendant l'exécution de la fonction, mais il n'y a d'accès direct à cette valeur dans le reste du programme.
- Ces valeurs intermédiaires sont sur une *pile* (angl. : stack). Leur durée de vie est limitée à l'évaluation d'une expression et peut être déterminée par le compilateur.

Cas 2 : Structures mutables

- Exemple : tableau dans lequel des structures sont ajoutées et/ou supprimées.
- Quand une structure n'est plus accessible alors son espace mémoire peut être réutilisé.
- C'est la tâche du Garbage Collector (GC, parfois *Ramasse-miettes*, ou *Glaneur de cellules*), qui est lancé quand le système OCaml a besoin de mémoire.
- Technique de *Lisp*, aujourd'hui aussi en *Java*, *Python*, ...

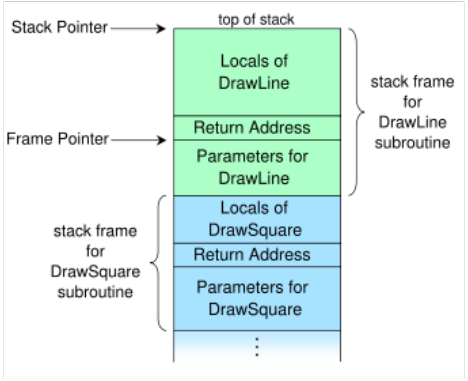
Conclusion sur l'utilisation de mémoire

- L'allocation de mémoire ne coûte (presque) rien.
- La mémoire utilisée exclusivement dans des calculs locaux (évaluation d'une expression) est tout de suite libérée quand elle n'est plus nécessaire.
- La mémoire utilisée dans des structures globales peut être libérée par le GC quand elle n'est plus nécessaire.
- Il ne faut pas hésiter à utiliser des définitions intermédiaires dans les programmes, car cela contribue à la clarté du code.

Récurrence et mémoire

- À chaque appel de fonction il faut stocker plusieurs informations d'environnement pour pouvoir ensuite les récupérer
- Ces informations sont stockées dans une zone de mémoire appelée la *pile* (angl.: call stack)
- À chaque appel de fonction, OCaml alloue donc de la mémoire sur le call stack
- Quand l'appel de la fonction termine, ces informations sont balayées du sommet de la pile.
- Conséquence : si trop d'appels de fonction imbriquées, on risque d'épuiser l'espace disponible pour la pile.

Exemple : `DrawSquare` appèle `DrawLine`



Merci, Wikipédia !

Factorielle avec récurrence terminale I

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

fact 3;;
# - : int = 6
```

Exemple d'exécution

```
fact 3 -> 3 * (fact 2)
       -> 3 * (2 * fact 1)
       -> 3 * (2 * (1 * fact 0))
       -> 3 * (2 * (1 * 1))
       -> 3 * (2 * 1)
       -> 3 * 2
       -> 6
```

Factorielle avec récurrence non terminale I

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

fact 1000000;;
# Stack overflow during evaluation (looping recursion?).
```

Récurrence non terminale versus récurrence terminale

- Appel terminal: c'est un appel final.
- Récurrence terminale (angl. *tail recursion*) : fonction recursive, avec tous les appels récursifs à des positions terminales.
- Dans de nombreux cas, il est possible de transformer une récurrence non terminale en une récurrence terminale, il faut réécrire le programme.
- Conséquence: plus besoin de pile intermédiaire.
- Ceci permet d'éviter les dépassements de pile (*stack overflow*).
- Avantage : peu importe la profondeur de la récurrence, l'utilisation de l'espace mémoire reste constante.
- Une fonction recursive avec récurrence terminale est exécutée comme une boucle !
- Inconvénient: pour certains algorithmes, cette transformation peut s'avérer très complexe, surtout si la logique de l'algorithme repose fortement sur la récurrence non terminale.

Factorielle avec récurrence terminale I

```
(*Un accumulateur pour calculer le resultat partielle de la fonction *)

let fact n =
    let rec aux acc l  = match l with
         0 -> acc
       | n -> aux (n * acc) (n - 1)
     in aux 1 n;;
# val fact : int -> int = <fun>


fact 3;;
# - : int = 6
```

Exemple d'exécution

```
aux 1 3 -> aux (3*1) (3-1)
        -> aux 3 2
        -> aux (2*3) (2-1)
        -> aux 6 1
        -> aux (1*6) (1-0)
        -> aux 6 0
        -> 6
```

La fonction reverse avec récurrence non terminale I

```
let rec ajout_fin l e = match l with
  | h::r -> h::(ajout_fin r e)
  | [] -> [e];;
# val ajout_fin : 'a list -> 'a -> 'a list = <fun>

let rec reverse l = match l with
  | h::r -> ajout_fin (reverse r) h
  | [] -> [];;
# val reverse : 'a list -> 'a list = <fun>

reverse [1;2;3];;
# - : int list = [3; 2; 1]
```

Exemple d'exécution

```
reverse [1;2;3] -> ajout_fin (reverse [2;3]) 1
                -> ajout_fin (ajout_fin (reverse [3]) 2) 1
                -> ajout_fin (ajout_fin (ajout_fin ((reverse []) 3) 2) 1
                -> ajout_fin (ajout_fin (ajout_fin ([] 3) 2) 1
                -> ajout_fin (ajout_fin [3] 2) 1
                -> ajout_fin (3:: (ajout_fin [] 2)) 1
                -> ajout_fin (3:: [2]) 1
                -> ajout_fin [3;2] 1
                -> 3:: (ajout_fin [2] 1)
                -> 3:: 2:: (ajout_fin [] 1)
                -> 3:: 2:: [1]
                -> 3:: [2;1]
                -> [3;2;1]
```

La fonction reverse avec récurrence terminale I

```
let reverse l =
  let rec rev_aux acc l = match l with
    | [] -> acc
    | h::r -> rev_aux (h::acc) r
  in rev_aux [] l;;
# val reverse : 'a list -> 'a list = <fun>

reverse [1;2;3];;
# - : int list = [3; 2; 1]
```

Exemple d'exécution

```
reverse [1;2;3] -> rev_aux [] [1;2;3]
                -> rev_aux [1] [2;3]
                -> rev_aux [2;1] [3]
                -> rev_aux [3;2;1] []
                -> [3;2;1]
```

Qu'est-ce qu'on a gagné ?

- Le coût de `reverse` est n où n est la longueur de la liste : on dit que `reverse` a une complexité linéaire. ☺
- La nouvelle fonction `reverse` est recursive terminale : l'exécution est aussi efficace en ce qui concerne la mémoire. ☺

La fonction `fold.right` I

```
(* defined in library as List.fold_right *)
(* List.fold_right f [a1; ...; an] b is *)
(* f a1 (f a2 (... (f an b) ...)) *)
(* Ceci n'est PAS une fonction recursive terminale ! *)

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

La fonction `fold.right` II

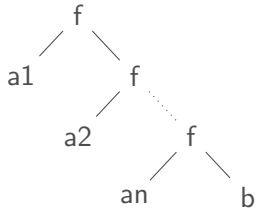
```
let sum l = fold_right (+) l 0 ;;
# val sum : int list -> int = <fun>

sum [1;2;3];;
# - : int = 6
```

Exemple d'exécution

```
sum [1;2;3] -> fold_right (+) [1;2;3] 0
            -> (+) 1 (fold_right (+) [2;3] 0)
            -> (+) 1 ((+) 2 (fold_right (+) [3] 0))
            -> (+) 1 ((+) 2 ((+) 3 (fold_right (+) [] 0)))
            -> (+) 1 ((+) 2 ((+) 3 0))
            -> (+) 1 ((+) 2 3)
            -> (+) 1 5
            -> 6
```

La fonction `fold.right`

Évaluation de `List.fold_right f [a1; a2; ... an] b` :



La fonction `fold.left` I

```
(* defined in library as List.fold_left *)
(* List.fold_left f b [a1; ...; an] is *)
(* (f (... (f (f b a1) a2) ...) an) *)
(* Ceci est une fonction recursive terminale ! *)

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

La fonction `fold.left` II

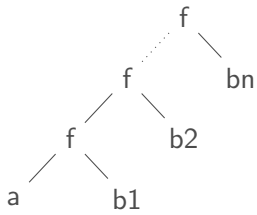
```
let sum l = fold_left (+) 0 l;;
# val sum : int list -> int = <fun>

sum [1;2;3;4];;
# - : int = 10
```

Exemple d'exécution

```
sum [1;2;3] -> fold_left (+) 0 [1;2;3]
            -> fold_left (+) ((+) 0 1) [2;3]
            -> fold_left (+) 1 [2;3]
            -> fold_left (+) ((+) 1 2) [3]
            -> fold_left (+) 3 [3]
            -> fold_left (+) ((+) 3 +3) []
            -> fold_left (+) 6 []
            -> 6
```

La fonction `fold.left`

Évaluation de `List.fold_left f a [b1; b2; ... bn]` :



Récurrence terminale et exceptions

La fonction suivante n'est pas recursive terminale :

```
let rec f arg1... argn =
  try (* some code *) (f arg'1... arg'n)
  with Not_found -> ()
```

Version avec récurrence terminale:

```
let f  arg1... argn =
  let rec aux  a1... an = (* some code *) (aux a'1... a'n) in
  try (aux arg1... argn)
  with Not_found -> ()
```

Factorielle avec récurrence terminale I

```
(*Un accumulateur pour calculer le resultat partielle de la fonction *)

let fact n =
    let rec aux acc l  = match l with
         0 -> acc
       | n -> aux (n * acc) (n - 1)
     in aux 1 n;;
# val fact : int -> int = <fun>


fact 3;;
# - : int = 6
```

Factorielle avec récurrence terminale desactivée I

```
exception NegativeNumber;;
# exception NegativeNumber

let rec factorial_tail_safe n acc =
  try
    if n < 0 then raise NegativeNumber
    else if n = 0 then acc
    else factorial_tail_safe (n - 1) (n * acc)
with NegativeNumber -> 0;;
# val factorial_tail_safe : int -> int -> int = <fun>
```

Explications

L'ajout d'un bloc `try` peut désactiver l'optimisation d'une function récursive terminale. Cela se produit parce que le bloc try doit potentiellement capturer des exceptions qui remontent, et OCaml ne peut pas toujours garantir que la pile des appels sera éliminée si une exception est levée.

Copier un fichier avec récurrence terminale desactivée I

```
let rec copy_lines ci co =
  try
    let x = input_line ci  in
      output_string co x;
      output_string co "\n" ;
      copy_lines ci co
  with
      End_of_file -> ();;
# val copy_lines : in_channel -> out_channel -> unit = <fun>
```

Copier un fichier avec récurrence terminale desactivée II

```
let copy infile outfile =
  let ci = open_in infile
  and co = open_out outfile
  in
    copy_lines ci co;
    close_in ci;
    close_out co;;
# val copy : string -> string -> unit = <fun>
```

Copier un fichier avec récurrence terminale I

```
type 'a option = None | Some of 'a;;
# type 'a option = None | Some of 'a

let rec copy_lines ci co =
  let x =
    try  Some (input_line ci)
    with End_of_file -> None
  in match x with
     | Some l -> output_string co l; output_string co "\n";
                 copy_lines ci co
     | None -> () ;;
# val copy_lines : in_channel -> out_channel -> unit = <fun>
```

Copier un fichier avec récurrence terminale II

```
let copy infile outfile =
  let ci = open_in infile
  and co = open_out outfile
  in
    copy_lines ci co;
    close_in ci;
    close_out co;;
# val copy : string -> string -> unit = <fun>
```

Structures de données

- La structure de donnée la plus simple en OCaml pour stocker un nombre illimité de données : la liste.
- L'opération la moins coûteuse pour ajouter un nouvel élément à une liste : ajout au début de la liste.
- L'opération la plus simple pour extraire un élément d'une liste : décomposition de la liste en deux parties tête et reste.

Les piles I

```
type 'a container = {mutable contents : 'a list};;
# type 'a container = { mutable contents : 'a list; }

let empty = {contents = []};;
# val empty : '_weak1 container = {contents = []}

let put x c = c.contents <- x::c.contents;;
# val put : 'a -> 'a container -> unit = <fun>

exception Container_empty;;
# exception Container_empty
```

Les piles II

```
let get c = match c.contents with
  | h::r -> c.contents <- r; h
  | [] -> raise Container_empty;;
# val get : 'a container -> 'a = <fun>

let c = empty;;
# val c : '_weak1 container = {contents = []}
put 1 c;;
# - : unit = ()
put 2 c;;
# - : unit = ()
put 3 c;;
# - : unit = ()
```

Les piles III

```
get c;;
# - : int = 3
get c;;
# - : int = 2
get c;;
# - : int = 1
get c;;
# Exception: Container_empty.
```

Listes et piles

- On obtient donc de cette façon une pile (angl.: stack) :
  les éléments sortent dans un ordre inverse à l'ordre d'entrée.
- On parle aussi de LIFO : *Last In, First Out*.
- Comment réaliser une structure de données qui est *FIFO* (*First In, First Out*) ?

FIFO I

```
type 'a fifo = {
  mutable stack:  'a list;
};;
# type 'a fifo = { mutable stack : 'a list; }
let empty = {stack = []};;
# val empty : '_weak1 fifo = {stack = []}
let put x queue = queue.stack <- x::queue.stack;;
# val put : 'a -> 'a fifo -> unit = <fun>
exception Queue_empty;;
# exception Queue_empty
```

FIFO II

```
(* donne la liste privee de son dernier element, et le dernier element *)
let rec last l = match l with
  | h1::h2::r -> let p,x = last (h2::r) in (h1::p),x
  | [h]       -> [],h
  | []        -> raise Queue_empty;;
# val last : 'a list -> 'a list * 'a = <fun>
let get queue =
  let p,x = last queue.stack in
      queue.stack <- p;
  x;;
# val get : 'a fifo -> 'a = <fun>


let q = empty;;
# val q : '_weak1 fifo = {stack = []}
```

FIFO III

```
put 1 q;;
# - : unit = ()
put 2 q;;
# - : unit = ()
put 3 q;;
# - : unit = ()
get q;;
# - : int = 1
get q;;
# - : int = 2
get q;;
# - : int = 3
get q;;
# Exception: Queue_empty.
```

Est-ce efficace ?

- Comment évaluer le temps d'exécution ?
- Des expériences avec un chronomètre (*benchmarks*) ? Trop aléatoire, les résultats risquent de varier avec le type de compilation (code octet ou native), du processeur, de mémoire, de la charge de la machine par des autres processus ...
- Encore mieux : analyse de l'algorithme.

Comment analyser un algorithme ?

- Pour faire une analyse réaliste et précise il faudrait avoir un modèle de coût qui nous indique combien coûte l'exécution de chaque opération primitive.
- On développe de tels modèles de coût pour des mécanismes de calculs théoriques (voir le cours de *Complexité* du M1), car OCaml en entier est trop complexe pour le faire.

Solution pragmatique

- Dans les cas considérés ici, le corps de chaque fonction exécute un petit nombre d'opérations primitives, et des appels récursifs.
- On va donc simplement compter le nombre d'appels récursifs, car c'est le facteur déterminant.
- Ce modèle permet d'estimer comment le coût d'exécution évolue avec la taille des données (linéaire, quadratique, exponentiel, ...).
- Ce modèle n'est plus praticable quand les opérations deviennent complexes (par exemples, des boucles for).

Justification de ce modèle

- Ce n'est pas l'invocation d'une fonction en soit qui est si cher.
- Dans tous les corps des fonctions (dans notre cas) les autres opérations primitives prennent un temps qui est à peu près le même. Le coût total est alors

  nombre d'appels de fonction * coût des opérations dans une fonction

Sur l'exemple de la pile

- Les fonctions utilisées sont non récursives, et chaque fonction consiste en un filtrage par motif, ou l'application d'un constructeur.
- Ces fonctions ont un coût constant (car le coût ne dépend pas de la taille actuelle de la pile).

Sur l'exemple du container FIFO

- `put` : coût 1 (pas d'appel récursif)
- `get` : le coût est n, où n est la taille de la pile (donc le coût dépend de la taille de la pile).

Quel est le coût *cumulé* ?

- C'est le coût d'exécution d'une série d'actions put/get.
- C'est une question plus pertinente pour les structures de container: les décisions prises pour stocker un élément dans le container peuvent avoir des conséquences sur des actions ultérieures.

Coût cumulé pour notre container FIFO ?

- Seulement n put : coût n. ☺
- Une séquence alternante de put - get - put - get de longueur n : coût n (car la pile a taille 1 quand on exécute get). ☺
- Le cas le pire : n actions put, suivi de n action get :
  - coût n pour les n premiers put.
  - premier get : coût $n-1$.
  - deuxième get : coût $n-2$.
  - Au total : $n+(n-1)+(n-2)+\ldots+1=\frac{n*(n-1)}{2}$

  Complexité quadratique dans le cas le pire. ☹

Meilleure implémentation du container FIFO

L'idée consiste à utiliser deux piles :

- Une pile d'entrée où on stocke des éléments.
- Une pile de sortie d'où on retire des éléments.
- Quand la pile de sortie est vide, on transfère la pile d'entrée vers la pile de sortie, mais en l'*inversant*.
- Inversion d'une liste : on utilise la version avec récurrence terminale !

FIFO revisitée I

```
type 'a fifo = {
  mutable instack:  'a list;
  mutable outstack: 'a list
};;
# type 'a fifo = {
  mutable instack : 'a list;
  mutable outstack : 'a list;
}
let empty = {instack = []; outstack = []};;
# val empty : '_weak1 fifo = {instack = []; outstack = []}
```

FIFO revisitée II

```
let reverse l =
  let rec reverse_aux acc l = match l with
    | [] -> acc
    | h::r -> reverse_aux (h::acc) r
in reverse_aux [] l;;
# val reverse : 'a list -> 'a list = <fun>

let put x queue = queue.instack <- x::queue.instack;;
# val put : 'a -> 'a fifo -> unit = <fun>

exception Queue_empty;;
# exception Queue_empty
```

FIFO revisitée III

```
let get queue = match queue.outstack with
  | h::r -> queue.outstack <- r; h
  | [] ->
    match queue.instack with
      | [] -> raise Queue_empty
      | _::_ ->
        match reverse queue.instack with
          | h::r ->
            begin
              queue.instack <- [];
              queue.outstack <- r;
              h
            end
;;
```

FIFO revisitée IV

```
# Lines 7-13, characters 1-8:
 7 | .match reverse queue.instack with
 8 |       | h::r ->
 9 |         begin
10 |           queue.instack <- [];
11 |           queue.outstack <- r;
12 |           h
13 |         end
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val get : 'a fifo -> 'a = <fun>
```

Coût cumulé ?

- n actions put et get mélangés.
- Les actions put coûtent 1, et les actions get aussi *quand la pile de sortie est non vide*.
- Coût des get quand la pile de sortie est vide ?
- C'est le coût cumulé des actions `reverse` qui est ...
- ... le nombre des actions get effectuées, qui est $\leq n$.
- Conséquence : complexité linéaire ! ☺