Programmation Fonctionnelle
Cours 10
Efficacité et Algorithmes

December 6, 2012

Table de matières

Utilisation de ressources par OCaml

- Ressources limitées :
  1. Mémoire utilisée par l'exécution du programme, nécessaire pour stocker (temporairement) des valeurs créées.
  2. Temps d'exécution pris pour l'exécution du programme.
  3. Éventuellement autres ressources, comme des communications avec l'extérieur.
- Comment est-ce que OCaml en général gère ces ressources ?
- Comment écrire des programmes fonctionnelles qui sont plus efficaces (= 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 ne va pas consommer 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.

Exemples (scalarproduct.ml)

```
(* produit scalaire *)

let scp v1 v2 =
  List.fold_left (+) 0 (List.map2 ( * ) v1 v2);;

scp [2;4;6] [1;3;5];;
```


Exemples (scalarproduct1.ml)

```
(* produit scalaire *)
(* code plus lisible grace a une definition locale *

let scp v1 v2 =
  let liste_des_produits = List.map2 ( * ) v1 v2
  in List.fold_left (+) 0 liste_des_produits;;

scp [2;4;6] [1;3;5];;
```


Cas 1 : évaluation des expressions

- Exemple : produit scalaire de deux valeurs.
- La valeur intermédiaire est nécessaire seulement pendant l'exécution de la fonction, il n'y a même pas un identificateur qui permet d'accéder directement à cette valeur.
- Ces valeurs sont mises sur une pile (angl. : stack). Leur durée de vie est limitée à l'évaluation d'une expression et peut être déterminée par le compilateur.
- Pour en savoir plus : voir un cours de compilation.

Cas 2 : structure mutables

- Exemple : table de hachage dans laquelle des structures sont ajoutées, et aussi supprimées.
- Quand une structure (par exemple une liste) 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.
- Existe aujourd'hui également en Java.
- Pour en savoir plus : voir un cours de compilation.

Conclusion sur l'utilisation de mémoire

- L'allocation de mémoire ne coûte (presque) rien.
- Mémoire qui est utilisée seulement dans des calculs locaux (évaluation d'une expression) est tout de suite libérée quand elle n'est plus nécessaire.
- Mémoire utilisée dans des structure globales (tas, angl. heap) peut être libérée par le GC quand plus nécessaire.
- N'hésitez pas de faire des définitions intermédiaires dans votre programme si ça contribue à la clarté du code.

Appels de fonctions

- L'appel de fonctions n'est pas cher en OCaml.
- En plus, les compilateurs peuvent éventuellement remplacer dans le code engendré l'appel de fonctions directement par leur code (inlining).
- Dans le cas d'inlining, une fonction est traduite comme une macro (mais avec la bonne sémantique – celle des fonctions).
- Peut produire du code plus grand. Le degré d'inlining peut être contrôlé par une option (dans le cas `oc mlopt`)
- C'est évidemment seulement possible pour des fonctions non recursives.

Récurrence et boucles

- Qu'est-ce qu'il se passe lors d'un appel d'une fonction ?
- Il y a une zone de mémoire appelée la pile (angl.: call stack) où sont stockées (parmi d'autres) les valeurs locales et les paramètres des fonctions.
- À chaque appel de fonction, OCaml alloue de la mémoire sur la pile.
- Quand cet appel de fonction est terminée, sa mémoire locale est balayée du sommet de la pile.
- Conséquence : quand on fait trop d'appels de fonctions on risque d'épuiser l'espace disponible pour la pile.

Exemple : `DrawSquare` appèle `DrawLine`

Exemples (tail1.ml)

```
(* n'est PAS un appel terminal *)

let rec f x = 2 * (f x) ;;
f 0;;
```


Appels terminaux

- Appel terminal à une fonction : une fonction f appèle une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f.
- Dans ce cas, on n'a plus besoin des valeurs locales de la fonction f quand on lance g — l'espace sur la pile peut être libérée avant de lancer g.
- Récurrence terminale (tail recursion) : fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage : peut importe la profondeur de récurrence, l'utilisation d'espace reste constante : la fonction recursive est exécutée comme une boucle !

Exemples (tail2.ml)

```
(* appel terminal *)

let rec f x = f (2*x);;
f 0;;
```


Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux.
- Permet d'éviter des dépassements de pile (stack overflow).
- Exemple : `List.fold_left` est recursive terminal, `List.fold_right` ne l'est pas.

Exemples (foldright.ml)

```
(* defined in library as List.fold_right *)
(* List.fold_right f [a1; ...; an] b is  *)
(* f a1 (f a2 (... (f an b) ...))        *)

let rec fold_right f l b = match l with
  | [] -> b
  | h::r -> f h (fold_right f r b);;

(* n'est PAS recursive terminale ! *)

fold_right (fun x y -> x::y) [1;2;3;4] [];;
```


Exemples (foldleft.ml)

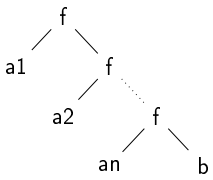
```
(* defined in library as List.fold_left *)
(* List.fold_left f b [a1; ...; an] is  *)
(* (f (... (f (f b a1) a2) ...) an)     *)
let rec fold_left f b = function
  | [] -> b
  | h::r -> fold_left f (f b h) r;;

(* cette fonction est recursive terminale ! *)

fold_left (+) 0 [1;2;3;4];;
```

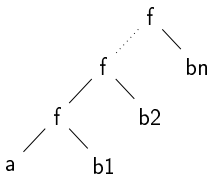

La fonction `fold.right`

Évaluation de `List.fold_right f [ 1; 2; ... n] b` :



La fonction `fold.left`

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



Récurrence terminale et exceptions

```
type 'a fifo = {
  mutable stack:  'a list;
};;
let empty = {stack = []};;
let put x queue = queue.stack <- x::queue.stack;;
exception Queue_empty;;
(* donne la liste privee de son dernier element, et le d
let rec last = function
  | h1::h2::r -> let p,x = last (h2::r) in (h1::p),x
  | [h]       -> [],h
  | []        -> raise Queue_empty
;;
let get queue =
  let p,x = last queue.stack
  in queue.stack <- p; x
;;

let q = empty;;
put 1 q;;
put 2 q;;
put 3 q;;
get q;;
get q;;
get q;;
get q;;
```


Est-ce efficace ?

- Question de la complexité (ici, dans le temps)
- 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 ...
- Mieux : faire une analyse de l'algorithme.

Comment faire une analyse ?

- Pour faire une analyse réaliste et précise il faudrait avoir un modèle de coût qui pour toute opération primitive dit combien son exécution coûte.
- On développe de tels modèles de coût pour des mécanismes de calcul théoriques (voir le cours de Complexité du M1), mais OCaml est trop complexe pour le faire.

Solution pragmatique

- Dans les cas considérés ici, le corps de chaque fonction fait un petit nombre d'opérations primitives, et des appels récursifs.
- On va donc simplement compter le nombre d'appel 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 justifié 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 serait si cher que son coût majore le coût total.
- L'argument est : dans tous les corps des fonctions (dans notre cas) les autres opérations primitifs prennent un temps qui est à peu près le même, le coût total est alors

  nombre d'appels de fonction∗cout des opérations dans une fonction

Sur l'exemple de la pile

- Toutes les fonctions sont non récursifs, chaque fonction consiste en un filtrage par motif, ou l'application d'un constructeur.
- On dit que les 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 dépend de la taille de la pile : quand la pile a taille n, alors le coût de cette fonction est n.

Quel est le coût cumulé ?

- On ne demande pas quel est le coût d'une seule opération, mais on regarde toute une série d'actions put/get, et on demande combien il coûte d'exécuter toute cette série.
- C'est une question plus pertinente pour les structures de container car les décisions prises comment stocker un élément 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 put ensemble.
  - premier get : coût $n-1$.
  - deuxième get : coût $n-2$.
  - On total : $n+(n-1)+(n-2)+\ldots+1=\frac{n*(n-1)}{2}$

  Complexité quadratique dans le cas le pire. ☹

Comment faire mieux ?

- Il y a une meilleure solution : elle utilise une solution efficace pour la fonction qui inverse une liste (on verra un peu plus tard pourquoi c'est utile).
- Regardons des solutions pour la fonction qui inverse une liste.

Exemples (reverse1.ml)

```
let rec append l1 l2 = match l1 with
  | h1::r1 -> h1::(append r1 l2)
  | [] -> l2
;;

let rec reverse = function
  | h::r -> append (reverse r) [h]
  | [] -> []
;;

reverse [1;2;3];;
```


Coût de la fonction reverse naïve ?

- Pour une liste de longueur n, il y a n appels récursifs de reverse.
- Dans le corps de `reverse`, il y a un appel de `ppend`.
- Le coût de `ppend` est $m1$ = longueur du premier argument.
- Coût total : $n+(n-1)+(n-2)+$

Exemples (reverse2.ml)

```
let reverse l =
  let rec rev_aux acc = function
    | [] -> acc
    | h::r -> rev_aux (h::acc) r
  in rev_aux [] l
;;

reverse [1;2;3];;
```


Comment est-ce que ça fonctionne ?

- Il faut d'abord comprendre la fonction `rev_ ux`.
- `rev_ ux` *acc* *l*2 = *reverse*(*l*2) @ *acc*.
- Démonstration : par induction sur *l*2.

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. ☺

Meilleure implémentation du container FIFO

Idée : On maintient deux piles :

- Une pile d'entrée où on met des éléments à stocker.
- Une pile de sortie dont on prend les éléments sortant.
- Quand la pile de sortie est vide, on transfère la pile d'entrée vers la pile de sortie, mais en l'inversant.

Exemples (fifo21.ml)

```
type 'a fifo = {
  mutable instack:  'a list;
  mutable outstack: 'a list
};;
let empty = {instack = []; outstack = []};;
let reverse l =
  let rec reverse_aux acc = function
    | [] -> acc
    | h::r -> reverse_aux (h::acc) r
  in reverse_aux [] l
;;
let put x queue = queue.instack <- x::queue.instack
```


Exemples (fifo22.ml)

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


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 ! ☺