Introduction à la programmation fonctionnelle

Cours 2

kn@lri.fr
http://www.lri.fr/~kn

université PARIS-SACLAY

Plan

Rappel

```
let rec fact n =
  if n <= 1 then (* cas de base *)
     1
  else (* cas récursif *)
     n * fact (n-1)   (* on se rappelle sur n-1, donc on
                         arrivera à 1 ou 0 à un moment *)
```

Observons l'exécution « `fact 6` ».

```
fact 6
6 * fact 5
    5 * fact 4
        4 * fact 3
            3 * fact 2
                2 * fact 1
                    1 ← cas de base
                2
            6
        24
    120
720
```


En mémoire

Dans les architectures modernes, une zone de la mémoire est allouée pour le programme et utilisée de façon particulière : **la pile d'exécution**.
C'est dans cette zone que sont allouée les variables locales à la fonction. Lorsque l'on fait un appel de fonction, le code machine effectue les instructions suivantes

- Empile les arguments de la fonction sur la pile
- Place l'adresse de l'instruction sur la pile
- Saute à l'adresse de la fonction

La fonction appelée :

- S'exécute pour calculer son résultat
- Lit l'adresse sauvegardée sur la pile et y revient

Conventions x86_64

On va illustrer le comportement des fonctions récursives avec du code assembleur intel 64 bits

Le code est tout à fait similaire sur d'autres architectures (MIPS, ARM, ...)

Quelques points à savoir :

- La pile « grandit » vers les adresses décroissantes (i.e. le sommet de la pile est une grande adresse et plus on empile, plus on se rapproche de 0)
- L'adresse du sommet de pile est stockée dans le registre `rsp`
- Le registre `rax` est celui dans lequel les fonctions écrivent leur valeur de retour

En mémoire

- On suppose que le code de la fonction `fact` se trouve à l'adresse `0x012345`
- Le code principal (le « main ») appel depuis l'adresse `0xabcde`.


```
fact:
   cmpq 1, %rsp[+8]     #compare n à 1
   jle   Lthen          #si <=, then
   movq %rsp[+8], %rax  # sinon copier n dans rax
   subq 1, %rax         # rax := rax - 1
   push %rax            # place n-1 sur la pile
   call fact            # appelle fact, la valeur de
                        # retour est dans rax
                        # ici : adresse 0x12345
   addq 8, %rsp         # on nettoie la pile
   imulq %rsp[+8], %rax #
   ret                  # on revient
Lthen:
   movq 1, %rax
   ret
main:
       push 6
       call fact
▶       ...        #située à l'adresse 0xabcde
```


Stack overflow ?

La pile grandit à chaque appel récursif.

Si on fait trop d'appels (en particulier mauvais cas de base, on ne s'arrête jamais), la pile dépasse sa taille maximale autorisée ⇒ Erreur Stack Overflow

Par défaut sous linux, la pile fait 8192 octets.

Elle peut être agrandie par le système ou l'utilisateur (command `ulimit -s`)

Pour la factorielle, la solution n'est pas satisfaisante, on utilise 16000ko pour calculer `fact 1000` !

Récursion terminale

Considérons la fonction `fact_alt` :

```
let rec fact_alt n acc =
   if n <= 1 then
     acc
   else
     fact_alt (n - 1) (n * acc)
fact_alt 6 1
       fact_alt 5 6
           fact_alt 4 30
               fact_alt 3 120
                   fact_alt 2 360
                       fact_alt 1 720
                       720 ← cas de base
                   720
               720
           720
 720
```

Récursion terminale (2)

La fonction `fact_alt` calcule son résultat « en descendant » :

- Elle utilise un argument auxiliaire `acc` dans lequel elle accumule les résultats partiels
- Lorsqu'on arrive au cas de base, le calcul est terminé
- Les « retours » d'appels récursifs ne font que propager le résultat

`fact_alt` est une fonction récursive terminale.

Une fonction est récursive terminale si tous les appels récursifs sont terminaux.

Un appel récursif est terminal si c'est la dernière instruction à s'exécuter.

Récursion terminale (3)

```
let rec fact_alt n acc =
   if n <= 1 then
     acc
   else
     fact_alt (n - 1) (n * acc)
```

```
let rec fact n =
   if n <= 1 then
     1
   else
     n * fact (n-1)
```

- Dans `fact_alt` l'appel récursif est terminal, c'est la dernière chose que l'on fait dans la fonction
- Dans `fact` l'appel récursif est non-terminal : il faut prendre la valeur qu'il renvoie et la multiplier par `n`. La multiplication s'effectue après l'appel récursif.

Le compilateur OCaml optimise les fonctions recursives terminales en les compilant comme des boucles, ce qui donne du code très efficace et qui consomme une quantité de mémoire bornée (et non pas une pile arbitrairement grande)

Récursion terminale (4)

On peut appliquer une technique générale pour transformer une boucle while en fonction récursive terminale. Soit le pseudo code (Java) :

```
int i = 0;
int res = 0;
while (i < 1000) {
  res = f (i, res);  //on calcule un résultat
                     //en fonction des valeurs
                     //précédentes et de l'indice de boucle
  i = i + 1;
}
return res;
```

Le code OCaml correspondant :

```
let rec loop i limit res =
   if i >= limit then res     (* return final *)
   else
      loop (i+1) limit (f i res)

let r = loop 0 1000 0
```


Récursion terminale (fin)

- La programmation récursive a souvent la réputation d'être inefficace
- C'est lié à une utilisation abusive de récursion non terminale
- Il faut écrire des fonctions récursives terminales dès que l'on essaye de programmer des boucles `for` ou `while` sur des entiers.

Attention, certains problèmes nécessitent forcément d'utiliser de la mémoire. Une fonction récursive non-terminale pourra alors être élégante, alors qu'un code impératif devra utiliser une boucle et une pile explicite.

Plan

Revenons sur `fact_alt`

- La fonction `fact_alt` est récursive terminale, c'est super.
- Par contre, ce n'est pas exactement ce qu'on a demandé !
- On veut une fonction `fact n` qui prend un seul argument en entrée!

Première solution :

```
let rec fact_alt n acc =
   if n <= 1 then
     acc
   else
     fact_alt (n - 1) (n * acc)

let fact n = fact_alt n 1
```

C'est bien, mais pas très satisfaisant (pourquoi ?)

Fonction imbriquée

En OCaml, une fonction est une valeur comme une autre. On peut donc définir des fonctions locales à des expressions :

```
let x = 42

let x2 =
   let carre n = n * n
   in
      carre x
```

La fonction `carre` est une fonction locale à l'expression dans laquelle elle se trouve. La notation :

```
let f x1 … xn =
        ef
in
  e
```

permet de définir la fonction et de ne l'utiliser que pour l'expression `e`. La fonction `f` n'est pas visible à l'extérieur.

Fonction imbriquée

Une utilisation courante des fonctions locales est la définition de fonctions auxiliaires à l'intérieur d'une fonction principale :

```
let fact m =
  let rec fact_alt n acc =
    if n <= 1 then
     acc
    else
     fact_alt (n - 1) (n * acc)
  in
    fact_alt m 1
```

Ici, il est impossible d'appeler `fact_alt` depuis l'extérieur (et en particulier avec de mauvais paramètres). C'est une forme d'encapsulation.

Remarque : `fact` n'a pas besoin d'être récursive, il n'y a que `fact_alt` qui doit être déclarée avec « `let rec` ».

Plan

Récursion mutuelle

Les appels récursifs ne sont pas limités à une seule fonction. Il est courant de vouloir définir deux fonctions qui s'appellent l'une l'autre :

```
let rec pair n =
   if n == 0 then
      true
   else
     impair (n-1)
and impair n =
   if n == 0 then
      false
   else
      pair (n-1)
```

```
pair 6
    impair 5
        pair 4
            impair 3
                pair 2
                    impair 1
                        pair 0
                        true  ← cas de base.
```

Remarque : ces deux fonctions sont récursives terminales !

On aura l'occasion de revoir les fonctions mutuellement récursives lorsqu'on travaillera sur des structures de données plus complexes que des entiers.

Plan

Inférence de types

Le compilateur OCaml effectue une inférence de types :

- Il devine le type des variables
- Il en déduit le type des expressions
- Il en déduit le type des fonctions

```
# let x = 2 ;;
val x : int = 2
# let y = 3 ;;
val y : int = 3
# let f n = n + 1;;
val f : int -> int = <fun>
# let g x = sqrt (float x);;
val g : int -> float = <fun>
```

Inférence de types (2)

Le compilateur affecte à chaque variable de programme une variable de type.

Il pose ensuite des équations entre ces variables de types

Si des équations sont contradictoires, OCaml indique une erreur de type

```
let f n  =
   if n <= 1 then
     42
   else
     n + 45
αn : type de n
αf : type de retour de f
αn = int (car n <= 1)
αn = int (car n + 45)
αf = int (car on renvoie 42)
αf = int (car on n + 45)
n : αn= int
f : αn -> αf = int -> int
```

```
let g n =
   if n <= 1 then
     42
   else
     "Boo!"
αn : type de n
αg : type de retour de g
αn = int (car n <= 1)
αg = int (car on renvoie 42)
αg = string (car on renvoie "Boo!")
n : int ≠ string ⇒ erreur de typage
```


Type des fonctions

Soit une fonction :

```
let f x1 … xn =
   ef
```

Le type de `f` se note : t_1 `->` … `->` t_n `->` t_f où :

- t_i est le type du paramètre x_i (pour $1 \leq i \leq n$)
- t_f est le type de retour de la fonction

Types de fonctions (exemples)

```
let mult_add a b c =  a * b + c;; (* int -> int -> int -> int *)

let carte n =
  if n = 1 then "As"
  else if n = 11 then "Valet"
  else if n = 12 then "Reine"
  else if n = 13 then "Roi"
  else if n > 13 then "invalide"
  else string_of_int n
   (* int -> string *)

let us_time h m =
  let s = if h > 12 then "pm" else "am" in
  let h = if h > 12 then h - 12 else h in
  Printf.printf "%d:%d %s" h m s
   (* int -> int -> unit *)

let to_seq j h m s =
  float (j * 3600 * 24 + h * 3600 + m * 60) +. s
   (* int -> int -> int -> float -> float *)
```


Utilité du typage

Well-typed expressions do not go wrong.

Robin Milner 1978.

Dans les langages statiquement typé (comme OCaml), les opérations sont toujours appliquées à des arguments du bon type.

Certaines classes d'erreurs sont donc impossible. Cela donne du code robuste et permet d'éviter toute un catégorie de tests.

Certaines erreurs *dynamiques* sont toujours possible : division par 0, stack overflow, ...

L'inféference de type évite au programmeur le travail fastidieux d'écrire les types pour toutes les variables de son programme.

Utilité du typage (2)

Comme les tests, le typage est un **outil précieux** pour le développement et la maintenance des programmes.

Histoire d'horreur (adaptée)

Un programmeur Javascript définit une fonction `f(s)` qui prend en argument une chaîne de caractère représentant une date `s`. Cette fonction est une fonction utilitaire, utilisée par plein d'autres collaborateurs dans leur code.
Le programmeur décide changer cette fonction en `f(d,m,y)` qui prend trois entiers et il oublie de prévenir ses collègues.
Le code de l'application continue de s'exécuter. La fonction `f` est appelée avec une chaîne de caractère. Utilisée comme un nombre, elle est convertie en `NaN` (nombre invalide) et ne provoque pas d'erreur, elle renvoie juste des résultats incorrects.

En OCaml (mais aussi en Java ou en C++) : les collègues sont avertis automatiquement, le code ne compile plus ! Ils modifient leur code pour appeler `f` avec les bons arguments.