

École Normale Supérieure

# Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 8 / 20 novembre 2008

dans le cours d'aujourd'hui, on se focalise sur la compilation des **langages fonctionnels**

on va notamment expliquer

- la compilation des fonctions comme valeurs de première classe
- l'optimisation des appels terminaux
- l'allocation mémoire (GC)
- le filtrage

---

## fonctions comme valeurs de première classe

considérons un mini-fragment d'Ocaml

```
e ::= c
    | x
    | fun x → e
    | e e
    | let x = e in e
    | let rec x = e in e
    | if e then e else e
```

# Fonctions imbriquées

comme dans mini-Pascal, les fonctions peuvent être imbriquées

```
let somme n =  
  let f x = x*x in  
  let rec boucle i =  
    if i = n then 0 else f i + boucle (i+1)  
  in  
  boucle 0
```

et la portée statique est la même

mais il est également possible de prendre des fonctions en argument

```
let carré f x = f (f x)
```

et d'en renvoyer

```
let f x = if x < 0 then fun y -> y - x else fun y -> y + x
```

notamment par application partielle

```
let f x =  
  let g y = x*y in g
```

dans ce dernier cas, la valeur renvoyée par `f` est une fonction qui utilise `x` mais le tableau d'activation de `f` vient précisément de disparaître !

on ne peut donc pas compiler les fonctions comme dans le cas de Pascal

la solution consiste à utiliser une **fermeture**, c'est-à-dire une structure de données allouée sur le tas (pour survivre aux appels de fonctions) contenant

- un **pointeur vers le code** de la fonction à appeler
- les valeurs des variables susceptibles d'être utilisées par ce code ; cette partie s'appelle l'**environnement**

P. J. Landin, *The Mechanical Evaluation of Expressions*,  
The Computer Journal, 1964

# Variables de l'environnement

quelles sont justement les variables qu'il faut mettre dans l'environnement de la fermeture représentant  $\text{fun } x \rightarrow e$  ?

ce sont les **variables libres** de  $\text{fun } x \rightarrow e$

l'ensemble des variables libres d'une expression se calcule ainsi :

$$\begin{aligned}fv(c) &= \emptyset \\fv(x) &= \{x\} \\fv(\text{fun } x \rightarrow e) &= fv(e) \setminus \{x\} \\fv(e_1 \ e_2) &= fv(e_1) \cup fv(e_2) \\fv(\text{let } x = e_1 \text{ in } e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\fv(\text{let rec } x = e_1 \text{ in } e_2) &= (fv(e_1) \cup fv(e_2)) \setminus \{x\} \\fv(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3)\end{aligned}$$



## Exemple

considérons le programme suivant qui approxime  $\int_0^1 x^n dx$

```
let rec pow i x = if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x =
    if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

(on ne considère pas de variables globales ici ; ce programme est donc vu comme `let rec pow = ... in let integrate_xn n = ... in ()`)

faisons apparaître la construction `fun` explicitement et examinons les différentes fermetures

```
let rec pow =  
  fun i ->  
    fun x -> if i = 0 then 1. else x *. pow (i-1) x
```

- dans la première fermeture, `fun i ->`, l'environnement est `{pow}`
- dans la seconde, `fun x ->`, il vaut `{i, pow}`

# Exemple

```
let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum =
    fun x -> if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

- pour `fun n ->`, l'environnement vaut `{pow}`
- pour `fun x ->`, l'environnement vaut `{eps, f, sum}`

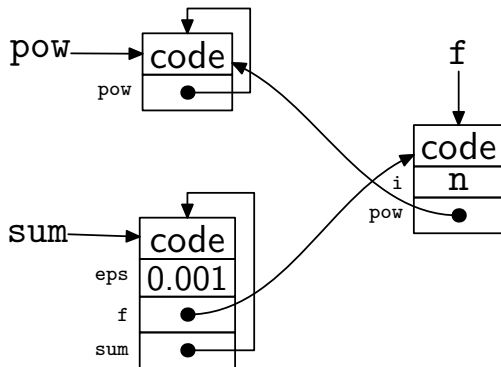
la fermeture peut être représentée de la manière suivante :

- un unique bloc sur le tas, dont
- le premier champ contient l'adresse du code
- les champs suivants contiennent les valeurs des variables libres, et uniquement celles-là

(d'autres solutions sont possibles : l'environnement dans un second bloc ; fermetures chaînées ; fermeture contenant toutes les variables liées au point de création)

# Exemple

```
let rec pow i x = if i = 0 then 1. else x *. pow (i-1) x
let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x = if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```



une façon relativement simple de compiler les fermetures consiste à procéder en deux temps

- 1 on recherche dans le code toutes les constructions  $\text{fun } x \rightarrow e$  et on les remplace par une opération explicite de construction de fermeture

$$\text{clos } f [y_1, \dots, y_n]$$

où les  $y_i$  sont les variables libres de  $\text{fun } x \rightarrow e$  et  $f$  le nom donné à une déclaration de fonction de la forme

$$\text{letfun } f [y_1, \dots, y_n] x = e'$$

où  $e'$  est obtenu à partir de  $e$  en y supprimant récursivement les constructions  $\text{fun}$  (*closure conversion*)

- 2 on compile le code obtenu, qui ne contient plus que des déclarations de fonctions de la forme  $\text{letfun}$

# Exemple

sur l'exemple, cela donne

```
letfun fun2 [i,pow] x =  
  if i = 0 then 1. else x *. pow (i-1) x  
letfun fun1 [pow] i =  
  clos fun2 [i,pow]  
let rec pow =  
  clos fun1 [pow]  
letfun fun3 [eps,f,sum] x =  
  if x >= 1. then 0. else f x +. sum (x +. eps)  
letfun fun4 [pow] n =  
  let f = pow n in  
  let eps = 0.001 in  
  let rec sum = clos fun3 [eps,f,sum] in  
  sum 0. *. eps  
let integrate_xn =  
  clos fun4 [pow]
```

pour compiler la construction

$$\text{clos } f [y_1, \dots, y_n]$$

on procède ainsi

- 1 on alloue un bloc de taille  $n + 1$  sur le tas
- 2 on stocke l'adresse de  $f$  dans le champ 0  
( $f$  est une étiquette dans le code et on obtient son adresse avec l'instruction MIPS [1a](#))
- 3 on stocke les valeurs des variables  $y_1, \dots, y_n$  dans les champs 1 à  $n$
- 4 on renvoie le pointeur sur le bloc

note : on se repose sur le GC pour libérer ce bloc lorsque ce sera possible (le fonctionnement du GC est décrit plus loin)



pour compiler un appel de la forme

$$e_1 \ e_2$$

on procède ainsi

- 1 on compile  $e_1$ , dont la valeur doit être un pointeur  $p_1$  vers une fermeture
- 2 on compile  $e_2$ , soit  $v_2$  sa valeur
- 3 on passe les deux arguments  $p_1$  et  $v_2$  (dans des registres ou sur la pile) à la fonction dont l'adresse est contenue dans le premier champ de  $p_1$

enfin, pour compiler la déclaration

$$\text{letfun } f [y_1, \dots, y_n] x = e$$

on procède comme pour une déclaration usuelle de fonction ayant deux arguments

- 1 on alloue le tableau d'activation, contenant notamment la place pour les variables locales de  $e$
- 2 on y sauvegarde  $\$ra$  et  $\$fp$
- 3 on évalue  $e$ 
  - en particulier, on va chercher la valeur de  $y_i$  dans l'environnement, dont l'adresse est donnée par le premier argument
  - la valeur de  $x$  est directement donnée par le second argument
- 4 on restaure  $\$ra$  et  $\$fp$  et on désalloue le tableau d'activation
- 5 on revient à  $\$ra$

il est inutilement coûteux de créer des fermetures intermédiaires dans un appel où  $n$  arguments sont fournis

$$f \ e_1 \ \dots \ e_n$$

et où la fonction  $f$  est définie par

$$\text{let } f \ x_1 \ \dots \ x_n = e$$

un appel « traditionnel » pourrait être fait, où tous les arguments sont passés d'un coup

en revanche, une application partielle de  $f$  produirait une fermeture

Ocaml fait cette optimisation ; sur du code « premier ordre » on obtient donc la même efficacité qu'avec un langage non fonctionnel

une autre optimisation est possible : lorsque l'on est sûr qu'une fermeture ne survivra pas à la fonction dans laquelle elle est créée, elle peut être alors allouée sur la pile plutôt que sur le tas

c'est le cas de la fermeture pour `f` dans

```
let integrate_xn n =  
  let f x = ... in  
  let eps = 0.001 in  
  let rec sum x = if x >= 1. then 0. else f x +. sum (x +. eps) in  
  sum 0. *. eps
```

pour s'assurer que cette optimisation est possible, il faut effectuer une analyse statique non triviale (*escape analysis*)

---

## optimisation des appels terminaux

## Définition

On dit qu'un **appel** ( $f e_1 \dots e_n$ ) qui apparaît dans le corps d'une fonction  $g$  est **terminal** (*tail call*) si c'est la dernière chose que  $g$  calcule avant de renvoyer son résultat.

par extension, on peut dire qu'une fonction est **récursive terminale** (*tail recursive function*) s'il s'agit d'une fonction récursive dont tous les appels récursifs sont des appels terminaux

l'appel terminal n'est pas nécessairement un appel récursif

```
let g x = let y = x*x in f y
```

dans une fonction récursive, on peut avoir des appels récursifs terminaux et d'autres qui ne le sont pas

```
let rec f91 n =  
  if n <= 100 then f91 (f91 (n + 11)) else n - 10
```

quel intérêt du point de vue de la compilation ?

on peut détruire le tableau d'activation de la fonction où se trouve l'appel **avant** de faire l'appel, puisqu'il ne servira plus ensuite

mieux, on peut le réutiliser pour l'appel terminal que l'on doit faire en particulier, la valeur sauvegardée de  $\$ra$  y est la bonne

dit autrement, on peut faire un saut (`j`) plutôt qu'un appel (`jal`)



# Exemple

considérons

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

une compilation classique donne

```
fact:  sub    $sp, $sp, 4      # sauvegarde $ra,$fp,acc,n  
      sw     $ra, 0($sp)     # (ici seulement $ra)  
      ble   $a1, 1, L1      # n <= 1?  
      mul   $a0, $a0, $a1  
      sub   $a1, $a1, 1  
      jal   fact  
      b     L2  
L1:    move  $v0, $a0        # renvoie acc  
L2:    lw    $ra, 0($sp)     # restaure $ra  
      add   $sp, $sp, 4  
      jr    $ra
```

# Exemple

en optimisant l'appel terminal, on obtient

```
loop:  sub    $sp, $sp, 4
       sw    $ra, 0($sp)
L0:    ble   $a1, 1, L1
       mul   $a0, $a0, $a1
       sub   $a1, $a1, 1
       j     L0                ### j au lieu de jal! ###
L1:    move  $v0, $a0
       lw    $ra, 0($sp)
       add   $sp, $sp, 4
       jr   $ra
```

(dans ce cas précis, on peut même déterminer qu'il est inutile de sauvegarder \$ra, ce qui réduit encore le code)

# Exemple

on a notamment économisé la sauvegarde / restauration de  $\$ra$  à chaque appel

le résultat est une **boucle !**

le code est en effet identique à ce qu'aurait donné la compilation d'un programme tel que

```
while n > 1 do begin
    acc := acc * n ;
    n := n - 1
end
```

et ce, bien qu'on n'ait pas de traits impératifs dans le langage considéré

# Conclusion

une fonction récursive terminale est donc aussi efficace qu'une boucle (en Caml, elle est même plus efficace qu'une boucle écrite avec des références, car les références ont un coût relativement élevé)

en particulier, l'espace de pile utilisé devient constant ; on évite ainsi  
Stack overflow during evaluation (looping recursion?).

```
Fatal error : exception Stack_overflow
```

il est important de noter que la notion d'appel terminal n'a rien à voir avec les langages fonctionnels, et que sa compilation peut être optimisée dans tous les langages

---

GC

les langages fonctionnels (mais d'autres encore, tels Java) reposent sur un mécanisme **automatique** de la gestion du tas (allocation dynamique et libération de blocs mémoire), appelé **GC** pour *Garbage Collector*

en français, GC est traduit par « ramasse-miettes » ou encore « glâneur de cellules »

en l'absence de GC, on se repose sur une libération explicite par l'utilisateur (opération `free`), avec tous les risques qu'elle comporte

principe : l'espace alloué sur le tas à une donnée (fermeture, enregistrement, tableau, constructeur, etc.) qui n'est plus **atteignable** à partir des variables du programme peut être **recupéré** afin d'être réutilisé pour d'autres données

difficulté : on ne peut généralement pas déterminer statiquement (à la compilation) le moment où une donnée n'est plus atteignable  $\Rightarrow$  le GC fait donc partie de l'exécutable, typiquement sous forme d'une bibliothèque invoquée par le code produit par le compilateur (*runtime*)

dans la suite, on appelle **bloc** toute portion élémentaire du tas allouée par le programme ; un bloc peut contenir un ou plusieurs pointeurs vers d'autres blocs (ses **fil**s) mais aussi des données autres (caractères, entiers, pointeurs en dehors du tas, etc.)

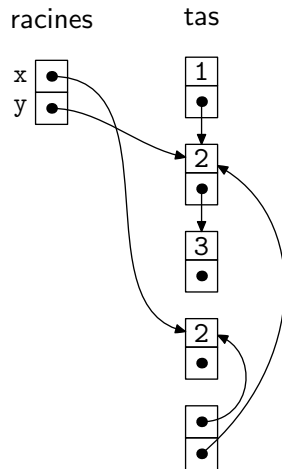
étant donné un instant de l'exécution du programme, on appelle **racine** toute variable active à ce moment-là (variable globale ou variable locale contenue dans un tableau d'activation ou dans un registre)

on dit qu'un bloc est **vivant** s'il est accessible à partir d'une racine *i.e.* s'il existe un chemin de pointeurs menant d'une racine à ce bloc



# Exemple

```
let x,y =  
  let l = [1; 2; 3] in  
  (List.filter even l, List.tl l)  
...
```



l'allocation mémoire peut être réalisée en **chaînant** les blocs mémoire libres (*freelist*)

l'allocation est alors réalisée en cherchant dans cette liste un bloc assez grand ; on le retire alors de la liste, et on rajoute à la liste le morceau restant, le cas échéant

deux stratégies possibles

- on prend le premier bloc assez grand (*first fit*)
- on choisit un bloc assez grand de taille minimale (*best fit*)

problèmes :

- la mémoire se fragmente : de plus en plus de petits blocs
  - ⇒ de la mémoire est gâchée et la recherche devient coûteuse
  - ⇒ il faut **compacter**
- on n'a toujours pas déterminé sous quelle condition libérer un bloc

on considère une première solution, appelée **comptage des références** (*reference counting*)

l'idée est d'associer à chaque bloc le nombre de pointeurs qui pointent sur ce bloc (depuis des racines ou depuis d'autres blocs)

quand ce nombre tombe à zéro, on sait qu'on peut libérer le bloc, et on décrémente les compteurs de tous ses fils

la mise à jour du compteur a lieu lors d'une **affectation** (explicite ou implite comme dans `1::x`) de la forme  $b.f \leftarrow p$ ; il faut

- décrémente le compteur du bloc correspondant à l'ancien pointeur  $b.f$ ; s'il tombe à 0, désallouer ce bloc
- incrémenter le compteur du bloc  $p$

problèmes :

- la mise à jour des compteurs est très coûteuse
- les **cycles** dans les structures de données rendent les blocs correspondant irrécupérables

le comptage des références est rarement utilisé dans les GC (une exception est le langage Perl) mais parfois explicitement par certains programmeurs

on considère une autre solution, plus efficace, appelée **marquer et balayer** (*mark and sweep*)

elle procède en deux temps

- ① on marque tous les blocs atteignables à partir des racines (en utilisant un parcours en profondeur et un bit dans chaque bloc)
- ② on examine tous les blocs et on récupère ceux qui ne sont pas marqués (et on supprime les marques sur les autres); ils sont remis dans la *freelist*

quand on veut allouer un bloc, on examine la *freelist*; si elle est vide, c'est un bon moment pour effectuer un GC

le marquage utilise un parcours en profondeur, comme ceci

```
parcours(x) =  
  si x est un pointeur sur le tas non encore marqué  
    marquer x  
  pour chaque champ f de x  
    parcours(x.f)
```





problème : le marquage est un algorithme **récur­sif**, qui va donc utiliser une taille de pile proportionnelle à la profondeur du tas ; celle-ci peut être aussi grande que le tas lui-même

solution : on utilise la structure traversée elle-même pour encoder la pile d'appels récur­sifs (*pointer reversal*)

# Marquer

le parcours est un peu plus compliqué mais n'utilise plus que deux variables et un champ entier `done[x]` dans chaque bloc

```
parcours(x) =  
  si x est un pointeur sur le tas non encore marqué  
    t <- null ; marquer x ; done[x] <- 0  
    tant que vrai  
      i <- done[x]  
      si i < nombre de champs de x  
        y <- x.fi  
        si y est un pointeur sur le tas non encore marqué  
          x.fi <- t ; t <- x ; x <- y  
          marquer x ; done[x] <- 0  
        sinon  
          done[x] <- i+1  
      sinon  
        y <- x ; x <- t  
        si x = null alors c'est terminé  
        i <- done[x] ; t <- x.fi ; x.fi <- y ; done[x] <- i+1
```

c'est une bonne solution pour déterminer les blocs à récupérer  
(en particulier, on récupère bien les cycles inatteignables)

pas encore une vraie solution au problème de la fragmentation

considérons encore une autre solution, appelée **s'arrêter et copier** (*stop and copy*)

l'idée est de découper le tas en deux moitiés

- 1 on n'en utilise qu'une seule, dans laquelle on alloue linéairement
- 2 lorsqu'elle est pleine, on copie tout ce qui est atteignable dans l'autre moitié, et on échange le rôle des deux moitiés

bénéfices immédiats :

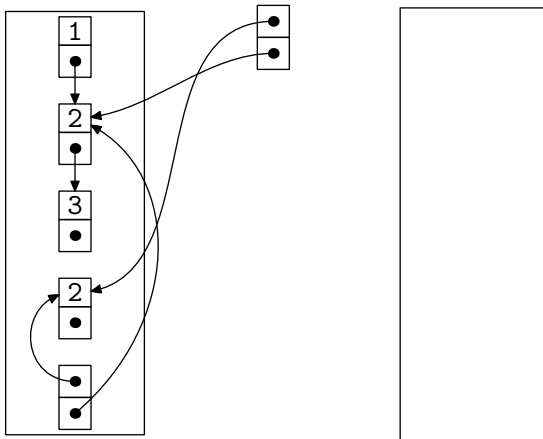
- l'allocation est très peu coûteuse (une addition et une comparaison)
- plus de problème de fragmentation

# Exemple

from-space

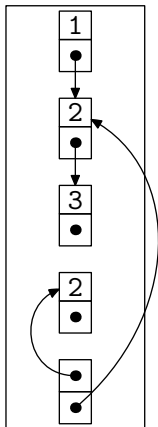
racines

to-space



# Exemple

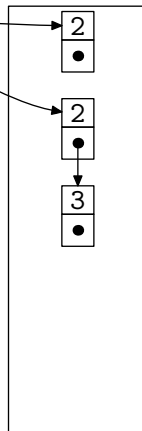
from-space



racines



to-space



problème : il faut effectuer la copie en utilisant un espace constant

solution : on va effectuer un parcours en largeur et utiliser l'espace d'arrivée comme zone de stockage des pointeurs à copier

lorsqu'un bloc a été déplacé de la première zone (`from-space`) vers la seconde (`to-space`) alors on utilise son premier champ pour indiquer où il a été copié

# Algorithme de Cheney

on commence par écrire une fonction qui copie le bloc à l'adresse  $p$ , si cela n'a pas encore été fait

next désigne le premier emplacement libre dans to-space

```
forward(p) =  
  si p pointe dans from-space  
    si p.f1 pointe dans to-space  
      renvoyer p.f1  
    sinon  
      pour chaque champ fi de p  
        next.fi ← p.fi  
      p.f1 ← next  
      next ← next + taille du bloc p  
      renvoyer p.f1  
  sinon  
    renvoyer p
```



# Algorithme de Cheney

on peut alors réaliser la copie, en commençant par les racines

```
scan <- next <- début de to-space
pour chaque racine r
  r <- forward(r)
tant que scan < next
  pour chaque champ fi de scan
    scan.fi <- forward(scan.fi)
  scan <- scan + taille du bloc scan
```

la zone de to-space située entre scan et next représente les blocs dont les champs n'ont pas encore été mis à jour

noter que scan avance, mais que next aussi !

bien que très élégant, cet algorithme a au moins un défaut : il modifie la localité des données *i.e.* des blocs qui étaient proches avant la copie ne le sont plus nécessairement après

dans un système avec virtualisation de la mémoire ou caches mémoire, la localité est importante

il est possible de modifier l'algorithme de Cheney pour effectuer un mélange de parcours en largeur et de parcours en profondeur (cf. Appel, chapitre 13)

dans de nombreux programmes, la plupart des valeurs ont une durée de vie courte, et celles qui survivent à plusieurs collections sont susceptibles de survivre à beaucoup d'autres collections

d'où l'idée d'organiser le tas en plusieurs **générations**

- $G_0$  contient les valeurs les plus récentes, et on y fait des collections fréquemment
- $G_1$  contient des valeurs toutes plus anciennes que celles de  $G_0$ , et on y fait des collections moins fréquemment
- etc.

en pratique, il y a quelques difficultés pour identifier les racines de chaque génération, en particulier parce qu'une affectation peut introduire un pointeur de  $G_1$  vers  $G_0$  par exemple (cf. Appel, chapitre 13)

enfin, il n'est pas souhaitable que le programme soit interrompu trop longtemps le temps d'une collection (gênant pour les programmes interactifs, critique pour les programmes temps-réel)

pour y remédier, on utilise un **GC incrémental**, qui marque les blocs petit à petit, au fur et à mesure des appels au GC

une simple marque ne suffit plus, il faut au moins trois couleurs (cf. Appel, chapitre 13)

le GC d'Ocaml est un GC à deux générations

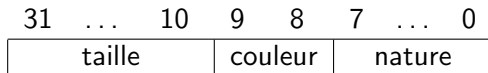
- un GC mineur (valeurs jeunes) : *Stop & Copy*
- un GC majeur (valeurs vieilles) : *Mark & Sweep* incrémental

la zone to-space du GC mineur est la zone du GC majeur

maintenant qu'on connaît les besoins du GC, on peut expliquer la **représentation des données** sur le tas (ici dans le cas d'Ocaml)

chaque bloc sur le tas est précédé d'un **entête** dont la taille est un **mot** (4 octets sur une architecture 32 bits, 8 sur une architecture 64 bits)

l'entête contient la taille du bloc, sa nature et 2 bits utilisés par le GC ainsi, sur une architecture 32 bits, l'entête est de la forme



la taille est ici un nombre de mots ; si la taille est  $n$ , le bloc tout entier, avec son entête, occupe donc  $n + 1$  mots

la nature du bloc est un entier codé sur 8 bits (0..255) ; elle permet de distinguer

- flottant
- chaîne de caractères
- tableau de flottants
- objet
- fermeture
- le cas général d'un bloc structuré : enregistrement, tableau,  $n$ -uplet, constructeur ; dans ce dernier cas, l'entier indique de quel constructeur il s'agit (pour le filtrage)



la taille du bloc étant codée sur 22 bits, on a sur une architecture 32 bits

```
# Sys.max_array_length ;;  
- : int = 4194303
```

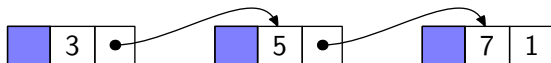
les chaînes de caractères sont en revanche représentées de manière compacte (4 caractères par mot), ce qui donne

```
# Sys.max_string_length ;;  
- : int = 16777211
```

une valeur Ocaml tient sur un mot ; c'est

- soit un entier impair  $2n + 1$ , représentant alors la valeur  $n$  de type `int` ou un constructeur constant encodé par  $n$  (`true`, `false`, `[]`, etc.)
- soit un pointeur (nécessairement pair pour des raisons d'alignement), qui peut pointer dans ou en dehors du tas

exemple : la valeur `1 :: 2 :: 3 :: []` est ainsi représentée



le GC teste donc le bit de poids faible pour déterminer si un champ est un pointeur ou non, car en présence de polymorphisme, le compilateur ne peut pas indiquer au GC quels sont les champs qui sont des pointeurs

```
let f x = (x, x)
```

enfin, il est important de rappeler que le mode de passage d'Ocaml est le passage **par valeur**, même si de nombreuses valeurs se trouvent être des pointeurs

---

# filtrage

dans les langages fonctionnels, on trouve généralement une construction appelée **filtrage** (*pattern-matching*), utilisée dans

- les définitions de fonctions

$$\text{function } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

- les conditionnelles généralisées

$$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

- les gestionnaires d'exceptions

$$\text{try } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

objectif du compilateur : transformer ces constructions de haut niveau en séquences de **tests élémentaires** (tests de constructeurs et comparaisons de valeurs constantes) et d'accès à des champs de valeurs structurées

dans ce qui suit, on considère la construction

$$\text{match } x \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

(à laquelle il est aisé de se ramener avec un `let`)

un **motif** (*pattern*) est défini par la syntaxe abstraite

$$p ::= x \mid C(p, \dots, p)$$

où  $C$  est un **constructeur**, qui peut être

- une constante telle que `false`, `true`, `0`, `1`, `"hello"`, etc.
- un constructeur constant de type somme, tel que `[]` ou par exemple `Empty` déclaré par `type t = Empty | ...`
- un constructeur d'arité  $n \geq 1$  tel que `::` ou par exemple `Node` déclaré par `type t = Node of t * t | ...`
- un constructeur de  $n$ -uplet, avec  $n \geq 2$

## Définition (motif linéaire)

On dit qu'un motif  $p$  est **linéaire** si toute variable apparaît au plus une fois dans  $p$ .

exemple : le motif  $(x, y)$  est linéaire, mais  $(x, x)$  ne l'est pas

note : Ocaml n'admet les motifs non linéaires que dans les motifs OU

```
# let (x,x) = (1,2) ;;
```

Variable x is bound several times in this matching

```
# let x,0 | 0,x = ... ;;
```

dans ce qui suit, on ne considère que des motifs linéaires (et on ne considère pas les motifs OU)



les valeurs sont ici

$$v ::= C(v, \dots, v)$$

où  $C$  désigne le même ensemble de constantes et de constructeurs que dans la définition des motifs

## Définition (filtrage)

*On dit qu'une valeur  $v$  **filtre** un motif  $p$  s'il existe une substitution  $\sigma$  de variables par des valeurs telle que  $v = \sigma(p)$ .*

note : on peut supposer de plus que le domaine de  $\sigma$ , c'est-à-dire l'ensemble des variables  $x$  telles que  $\sigma(x) \neq x$ , est inclus dans l'ensemble des variables de  $p$

il est clair que toute valeur filtre  $p = x$ ; d'autre part

## Proposition

*Une valeur  $v$  filtre  $p = C(p_1, \dots, p_n)$  si et seulement si  $v$  est de la forme  $v = C(v_1, \dots, v_n)$  avec  $v_i$  qui filtre  $p_i$  pour tout  $i = 1, \dots, n$ .*

preuve :

- soit  $v$  qui filtre  $p$ ; on a donc  $v = \sigma(p)$  pour un certain  $\sigma$ , soit  $v = C(\sigma(p_1), \dots, \sigma(p_n))$  et on pose donc  $v_i = \sigma(p_i)$
- réciproquement, si  $v_i$  filtre  $p_i$  pour tout  $i$ , alors il existe des  $\sigma_i$  telles que  $v_i = \sigma_i(p_i)$ ; comme  $p$  est linéaire, les domaines des  $\sigma_i$  sont deux à deux disjoints et on a donc  $\sigma_i(p_j) = p_j$  si  $i \neq j$   
en posant  $\sigma = \sigma_1 \circ \dots \circ \sigma_n$ , on a 
$$\begin{aligned}\sigma(p_i) &= \sigma_1(\sigma_2(\dots \sigma_n(p_i))\dots) \\ &= \sigma_1(\sigma_2(\dots \sigma_i(p_i))\dots) \\ &= \sigma_1(\sigma_2(\dots v_i)\dots) \\ &= v_i\end{aligned}$$

donc  $\sigma(p) = C(\sigma(p_1), \dots, \sigma(p_n)) = C(v_1, \dots, v_n) = v$  □

## Définition

*Dans le filtrage*

$$\text{match } x \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

*si  $v$  est la valeur de  $x$ , on dit que  $v$  filtre le cas  $p_i$  si  $v$  filtre  $p_i$  et si  $v$  ne filtre pas  $p_j$  pour tout  $j < i$*

*Le résultat du filtrage est alors  $\sigma(e_i)$ , où  $\sigma$  est la substitution telle que  $\sigma(p_i) = v$ .*

si  $v$  ne filtre aucun  $p_i$ , le filtrage conduit à une erreur d'exécution (exception `Match_failure` en Ocaml)

considérons un premier algorithme de compilation du filtrage

on suppose disposer de

- $constr(e)$ , qui renvoie le constructeur de la valeur  $e$ ,
- $\#_i(e)$ , qui renvoie sa  $i$ -ième composante

autrement dit, si  $e = C(v_1, \dots, v_n)$  alors  $constr(e) = C$  et  $\#_i(e) = v_i$

note : on a vu comment les valeurs d'OCaml étaient représentées, et on en déduit dans ce cas comment réaliser ces deux fonctions

# Compilation du filtrage

on commence par la compilation d'une ligne de filtrage

$$\text{code}(\text{match } e \text{ with } p \rightarrow \text{action}) = F(p, e, \text{action})$$

où la fonction de compilation  $F$  est définie ainsi :

$$F(x, e, \text{action}) =$$

let  $x = e$  in  $\text{action}$

$$F(C, e, \text{action}) =$$

if  $\text{constr}(e) = C$  then  $\text{action}$  else  $\text{error}$

$$F(C(p), e, \text{action}) =$$

if  $\text{constr}(e) = C$  then  $F(p, \#_1(e), \text{action})$  else  $\text{error}$

$$F(C(p_1, \dots, p_n), e, \text{action}) =$$

if  $\text{constr}(e) = C$  then

$F(p_1, \#_1(e), F(p_2, \#_2(e), \dots F(p_n, \#_n(e), \text{action}) \dots)$

else  $\text{error}$

# Exemple

considérons par exemple

```
match x with 1 :: y :: z -> y + length z
```

sa compilation donne le (pseudo-)code suivant :

```
if constr(x) = :: then
  if constr(#1(x)) = 1 then
    if constr(#2(x)) = :: then
      let y = #1(#2(x)) in
      let z = #2(#2(x)) in
      y + length(z)
    else error
  else error
else error
```

note :  $\#_2(x)$  est calculée plusieurs fois  $\Rightarrow$  on pourrait introduire des let dans la définition de  $F$  pour y remédier

# Preuve de correction

montrons que si  $e \xrightarrow{*} v$  alors

$$\begin{array}{ll} F(p, e, action) \xrightarrow{*} \sigma(action) & \text{s'il existe } \sigma \text{ telle que } v = \sigma(p) \\ F(p, e, action) \xrightarrow{*} error & \text{sinon} \end{array}$$

preuve : par récurrence sur  $p$

- $p = x$  ou  $p = C$  : c'est immédiat
- $p = C(p_1, \dots, p_n)$  :
  - si  $constr(v) \neq C$ , il n'existe pas de  $\sigma$  telle que  $v = \sigma(p)$  et  $F(C(p_1, \dots, p_n), e, action) = error$
  - si  $constr(v) = C$ , on a  $v = C(v_1, \dots, v_n)$ ;  $\sigma$  telle que  $v = \sigma(p)$  existe si et seulement s'il existe des  $\sigma_i$  telles que  $v_i = \sigma_i(p_i)$   
si l'une des  $\sigma_i$  n'existe pas, alors l'appel  $F(p_i, \#_i(e), \dots)$  se réduit en  $error$  et  $F(p, e, action)$  également  
si toutes les  $\sigma_i$  existent, alors par hypothèse de récurrence

$$\begin{aligned} F(p, e, action) &= F(p_1, \#_1(e), F(p_2, \#_2(e), \dots F(p_n, \#_n(e), action) \dots)) \\ &= F(p_1, \#_1(e), F(p_2, \#_2(e), \dots \sigma_n(action) \dots)) \\ &= \sigma_1(\sigma_2(\dots \sigma_n(action) \dots)) = \sigma(action) \end{aligned} \quad \square$$

## Filtrer plusieurs lignes

pour filtrer plusieurs lignes, on remplace *error* par le passage à la ligne suivante

$$\text{code}(\text{match } x \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n) = \\ F(p_1, x, e_1, F(p_2, x, e_2, \dots F(p_n, x, e_n, \text{error}) \dots))$$

où la fonction de compilation  $F$  est maintenant définie par :

$$F(x, e, \text{succès}, \text{échec}) =$$

let  $x = e$  in succès

$$F(C, e, \text{succès}, \text{échec}) =$$

if  $\text{constr}(e) = C$  then succès else échec

$$F(C(p_1, \dots, p_n), e, \text{succès}, \text{échec}) =$$

if  $\text{constr}(e) = C$  then

$F(p_1, \#_1(e), F(p_2, \#_2(e), \dots F(p_n, \#_n(e), \text{succès}, \text{échec}) \dots, \text{échec})$

else échec



# Exemple

la compilation de

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

donne le code suivant

```
if constr(x) = [] then
  1
else
  if constr(x) = :: then
    if constr(#1(x)) = 1 then
      let y = #2(x) in 2
    else
      if constr(x) = :: then
        let z = #1(x) in let y = #2(x) in z
      else error
  else
    if constr(x) = :: then
      let z = #1(x) in let y = #2(x) in z
    else error
```

cet algorithme est peu efficace car

- on effectue plusieurs fois les mêmes tests (d'une ligne sur l'autre)
- on effectue des tests redondants (si  $constr(e) \neq []$  alors nécessairement  $constr(e) = ::$ )

# Un autre algorithme

on considère un autre algorithme, qui considère le problème du filtrage des  $n$  lignes dans sa globalité

on représente le problème sous forme d'une **matrice**

$$\left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ p_{1,1} & p_{1,2} & \dots & p_{1,m} & \rightarrow & action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} & \rightarrow & action_n \end{array} \right|$$

dont la signification est

$$\begin{array}{l} \text{match } (e_1, e_2, \dots, e_m) \text{ with} \\ | (p_{1,1}, p_{1,2}, \dots, p_{1,m}) \rightarrow action_1 \\ | \dots \\ | (p_{n,1}, p_{n,2}, \dots, p_{n,m}) \rightarrow action_n \end{array}$$

# Un autre algorithme

l'algorithme  $F$  procède récursivement sur la matrice

- $n = 0$

$$F \left| \begin{array}{ccc} e_1 & \dots & e_m \end{array} \right| = error$$

- $m = 0$

$$F \left| \begin{array}{c} \rightarrow action_1 \\ \vdots \\ \rightarrow action_n \end{array} \right| = action_1$$

# Un autre algorithme

si toute la colonne de gauche se compose de **variables**

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ \color{red}{x_{1,1}} & p_{1,2} & \dots & p_{1,m} \rightarrow \text{action}_1 \\ \vdots & & & \\ \color{red}{x_{n,1}} & p_{n,2} & \dots & p_{n,m} \rightarrow \text{action}_n \end{array} \right|$$

on élimine cette colonne en introduisant des let

$$F(M) = F \left| \begin{array}{ccc} e_2 & \dots & e_m \\ p_{1,2} & \dots & p_{1,m} \rightarrow \text{let } x_{1,1} = e_1 \text{ in } \text{action}_1 \\ \vdots & & \\ p_{n,2} & \dots & p_{n,m} \rightarrow \text{let } x_{n,1} = e_1 \text{ in } \text{action}_n \end{array} \right|$$

# Un autre algorithme

sinon, c'est que la colonne de gauche contient au moins un motif construit  
supposons par exemple qu'il y ait dans cette colonne trois constructeurs  
différents,  $C$  d'arité 1,  $D$  d'arité 0 et  $E$  d'arité 2

$$M = \begin{array}{cccc|ccc} e_1 & e_2 & \dots & e_m & & & \\ C(q) & p_{1,2} & \dots & p_{1,m} & \rightarrow & & action_1 \\ D & p_{2,2} & & p_{2,m} & \rightarrow & & action_2 \\ x & p_{3,2} & & p_{3,m} & \rightarrow & & action_3 \\ E(r, s) & p_{4,2} & & p_{4,m} & \rightarrow & & action_4 \\ y & p_{5,2} & & p_{5,m} & \rightarrow & & action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \rightarrow & & action_6 \\ E(u, v) & p_{7,2} & \dots & p_{7,m} & \rightarrow & & action_7 \end{array}$$

pour chaque constructeur  $C$ ,  $D$  et  $E$ , on construit la sous-matrice  
correspondant au filtrage d'une valeur pour ce constructeur

# Un autre algorithme

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ C(q) & p_{1,2} & \dots & p_{1,m} \rightarrow action_1 \\ D & p_{2,2} & & p_{2,m} \rightarrow action_2 \\ x & p_{3,2} & & p_{3,m} \rightarrow action_3 \\ E(r, s) & p_{4,2} & & p_{4,m} \rightarrow action_4 \\ y & p_{5,2} & & p_{5,m} \rightarrow action_5 \\ C(t) & p_{6,2} & & p_{6,m} \rightarrow action_6 \\ E(u, v) & p_{7,2} & \dots & p_{7,m} \rightarrow action_7 \end{array} \right|$$

d'où

$$M_C = \left| \begin{array}{cccc} \#_1(e_1) & e_2 & \dots & e_m \\ q & p_{1,2} & \dots & p_{1,m} \rightarrow action_1 \\ - & p_{3,2} & & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } action_3 \\ - & p_{5,2} & & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } action_5 \\ t & p_{6,2} & \dots & p_{6,m} \rightarrow action_6 \end{array} \right|$$

# Un autre algorithme

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ C(q) & p_{1,2} & \dots & p_{1,m} \rightarrow \text{action}_1 \\ D & p_{2,2} & & p_{2,m} \rightarrow \text{action}_2 \\ x & p_{3,2} & & p_{3,m} \rightarrow \text{action}_3 \\ E(r, s) & p_{4,2} & & p_{4,m} \rightarrow \text{action}_4 \\ y & p_{5,2} & & p_{5,m} \rightarrow \text{action}_5 \\ C(t) & p_{6,2} & & p_{6,m} \rightarrow \text{action}_6 \\ E(u, v) & p_{7,2} & \dots & p_{7,m} \rightarrow \text{action}_7 \end{array} \right|$$

d'où

$$M_D = \left| \begin{array}{ccc} e_2 & \dots & e_m \\ p_{2,2} & & p_{2,m} \rightarrow \text{action}_2 \\ p_{3,2} & & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } \text{action}_3 \\ p_{5,2} & \dots & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } \text{action}_5 \end{array} \right|$$



# Un autre algorithme

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ C(q) & p_{1,2} & \dots & p_{1,m} \rightarrow action_1 \\ D & p_{2,2} & & p_{2,m} \rightarrow action_2 \\ x & p_{3,2} & & p_{3,m} \rightarrow action_3 \\ E(r, s) & p_{4,2} & & p_{4,m} \rightarrow action_4 \\ y & p_{5,2} & & p_{5,m} \rightarrow action_5 \\ C(t) & p_{6,2} & & p_{6,m} \rightarrow action_6 \\ E(u, v) & p_{7,2} & \dots & p_{7,m} \rightarrow action_7 \end{array} \right|$$

d'où

$$M_E = \left| \begin{array}{cccc} \#_1(e_1) & \#_2(e_1) & e_2 & \dots & e_m \\ - & - & p_{3,2} & & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } action_3 \\ r & s & p_{4,2} & & p_{4,m} \rightarrow action_4 \\ - & - & p_{5,2} & & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } action_5 \\ u & v & p_{7,2} & \dots & p_{7,m} \rightarrow action_7 \end{array} \right|$$

enfin on définit une sous-matrice pour les autres valeurs (de constructeurs différents de  $C$ ,  $D$  et  $E$ ), c'est-à-dire pour les variables

$$M_R = \left| \begin{array}{ccc} e_2 & \dots & e_m \\ p_{3,2} & & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } action_3 \\ p_{5,2} & \dots & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } action_5 \end{array} \right|$$

on pose alors

$$F(M) = \text{case } \textit{constr}(e_1) \text{ in}$$
$$C \Rightarrow F(M_C)$$
$$D \Rightarrow F(M_D)$$
$$E \Rightarrow F(M_E)$$
$$\text{otherwise} \Rightarrow F(M_R)$$

cet algorithme termine

en effet, la grandeur

$$\sum_{i,j} \text{taille}(p_{i,j})$$

diminuit strictement à chaque appel récursif à  $F$ , si on pose

$$\begin{aligned} \text{taille}(x) &= 1 \\ \text{taille}(C(p_1, \dots, p_n)) &= 1 + \sum_{i=1}^n \text{taille}(p_i) \end{aligned}$$

la correction de cet algorithme est laissée en exercice

indication : utiliser l'interprétation de la matrice comme

```
match  $(e_1, e_2, \dots, e_m)$  with  
|  $(p_{1,1}, p_{1,2}, \dots, p_{1,m}) \rightarrow action_1$   
|  $\dots$   
|  $(p_{n,1}, p_{n,2}, \dots, p_{n,m}) \rightarrow action_n$ 
```

le type de l'expression  $e_1$  permet d'optimiser la construction

```
case constr( $e_1$ ) in
   $C \Rightarrow F(M_C)$ 
   $D \Rightarrow F(M_D)$ 
   $E \Rightarrow F(M_E)$ 
  otherwise  $\Rightarrow F(M_R)$ 
```

dans de nombreux cas :

- pas de test si un seul constructeur (par ex.  $n$ -uplet) :  $F(M) = F(M_C)$
- pas de cas otherwise lorsque  $C$ ,  $D$  et  $E$  sont les seuls constructeurs
- un simple if then else lorsqu'il n'y a que deux constructeurs
- une table de saut lorsqu'il y a un nombre fini de constructeurs
- un arbre binaire ou une table de hachage lorsqu'il y a une infinité de constructeurs (par ex. chaînes)

# Exemple

considérons

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

c'est-à-dire la matrice

$$M = \left| \begin{array}{l} x \\ [] \rightarrow 1 \\ 1::y \rightarrow 2 \\ z::y \rightarrow z \end{array} \right|$$

on obtient

```
case constr(x) in
  [] -> 1
  :: -> case constr(#1(x)) in
    1 -> let y = #2(x) in 2
    otherwise -> let z = #1(x) in let y = #2(x) in z
```

# Avantages supplémentaires

- détection des **cas redondants**

lorsqu'une action n'apparaît pas dans le code produit

exemple

```
match x with false -> 1 | true -> 2 | false -> 3
```

donne

```
case constr(x) in false -> 1 | true -> 2
```

- détection des **filtrages non exhaustifs**

lorsque *error* apparaît dans le code produit

exemple

```
match x with 0 -> 0 | 1 -> 1
```

donne

```
case constr(x) in 0 -> 0 | 1 -> 1 | otherwise -> error
```



## la semaine prochaine

- TD 8 le mercredi 26 novembre
  - programmation d'un GC *stop & copy*
- Cours 9 le jeudi 27 novembre
  - compilation des langages objets