

Université Paris-Sud

Compilation et langages

Thibaut Balabonski
d'après Jean-Christophe Filliâtre @ ENS

Cours 9 / 25 novembre 2016

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
- le filtrage

Fonctions comme valeurs de première classe

Considérons un mini-fragment d'OCaml

$$\begin{aligned}
 e &::= c \\
 &\quad | x \\
 &\quad | \text{fun } x \rightarrow e \\
 &\quad | e \ e \\
 &\quad | \text{let } [\text{rec}] \ x = e \text{ in } e \\
 &\quad | \text{if } e \text{ then } e \text{ else } e \\
 \\
 d &::= \text{let } [\text{rec}] \ x = e \\
 \\
 p &::= d \ \dots \ d
 \end{aligned}$$

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 on a une notion de portée statique des variables

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 !

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

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 $\text{fun } x \rightarrow e$

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

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

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
```

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 $\{\text{pow}\}$
- dans la seconde, `fun x ->`, il vaut $\{i, \text{pow}\}$

```
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}`

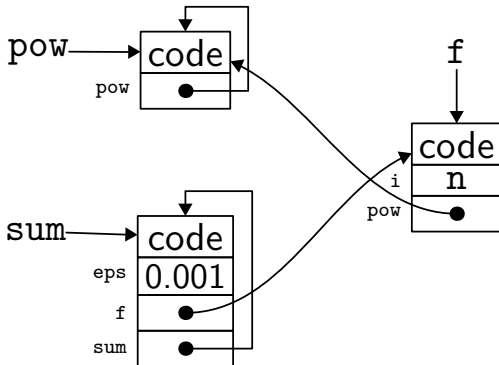
La fermeture peut être représentée de la manière suivante :

- 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)

```

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 `fun x → 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 `fun x → e` et f le nom donné à une déclaration globale 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 `fun` (*closure conversion*)

2. On compile le code obtenu, qui ne contient plus que des déclarations de fonctions de la forme `letfun`

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]
```

Avant

```

type var = string

type expr =
  | Evar of var
  | Efun of var * expr
  | Eapp of expr * expr
  | Elet of var * expr * expr
  | Eif of expr * expr * expr
  | ...

type decl = var * expr

type prog = decl list

```

Après

```

type var =
  | Vglobal of string
  | Vlocal of int
  | Vclos of int
  | Varg

type expr =
  | Evar of var
  | Eclos of string * var list
  | Eapp of expr * expr
  | Elet of int * expr * expr
  | Eif of expr * expr * expr
  | ...

type decl =
  | Let of string * expr
  | Letfun of string * expr

type prog = decl list

```


En particulier, un identificateur peut représenter

- $V_{\text{global}} s$: une variable globale (introduite par `let`) de nom s
- $V_{\text{local}} n$: une variable locale (introduite par `let in`),
à la position n dans le tableau d'activation
- $V_{\text{clos}} n$: une variable contenue dans la fermeture, à la position n
- V_{arg} : l'unique argument de la fonction (le x de `fun x -> e`)

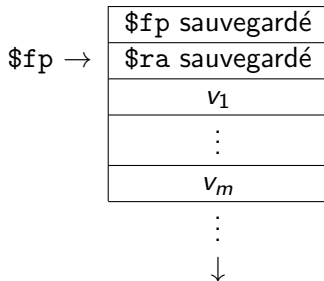
Cette analyse de portée peut être réalisée conjointement avec l'explicitation des fermetures

Chaque fonction a un unique argument (Varg), qu'on passera dans \$a0

La fermeture sera passée dans \$a1

Le tableau d'activation ressemble à ceci,
où v_1, \dots, v_m sont les variables locales

Il est intégralement construit par l'appelé



Expliquons maintenant comment compiler

- la construction d'une fermeture $\text{Eclos}(f, l)$
- un appel de fonction $\text{Eapp}(e_1, e_2)$
- l'accès à une variable $\text{Evar } x$
- une déclaration de fonction $\text{Letfun}(f, e)$

Pour compiler la construction

$$\text{Eclos}(f, [y_1, \dots, y_n])$$

on procède ainsi

1. on alloue un bloc de taille $n + 1$ sur le tas (`syscall 9`)
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 `la`)
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

Pour compiler un appel de la forme

$$\text{Eapp}(e_1, e_2)$$

on procède ainsi

1. on compile e_1 dans le registre $\$a1$
(sa valeur est un pointeur p_1 vers une fermeture)
2. on compile e_2 dans le registre $\$a0$
3. on appelle la fonction dont l'adresse est contenue dans le premier champ de la fermeture, avec l'instruction MIPS `jalc` (similaire à `jal`, mais saute vers une adresse calculée plutôt que directement donnée par une étiquette)

Pour compiler un accès à une variable

$\text{Evar } x$

on distingue quatre cas selon la valeur de x

- $\text{Vglobal } s$: la valeur se trouve à l'adresse donnée par l'étiquette s
- $\text{Vlocal } n$: la valeur se trouve à l'adresse donnée par $n(\$fp)$
- $\text{Vclos } n$: la valeur se trouve à l'adresse donnée par $n(\$a1)$
- Varg : la valeur se trouve dans $\$a0$

Enfin, pour compiler la déclaration

$$\text{Letfun}(f, e)$$

on procède comme pour une déclaration usuelle de fonction

1. on alloue le tableau d'activation, contenant notamment la place pour les variables locales de e
2. on y sauvegarde $\$fp$ et $\$ra$, et on positionne $\$fp$
3. on évalue e
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 = ... 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
```

Mais 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 rékursifs 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)

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
        sw     $ra, 0($sp)     #
        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
```

En optimisant l'appel terminal, on obtient

```
fact:   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

```
fact:   ble      $a1, 1, L1
        mul      $a0, $a0, $a1
        sub      $a1, $a1, 1
        j        fact
L1:     move     $v0, $a0
        jr      $ra
```


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 !

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

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

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 σ de variables par des valeurs telle que $v = \sigma(p)$.*

Note : on peut supposer de plus que le domaine de σ , 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 σ , 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 σ_i telles que $v_i = \sigma_i(p_i)$; comme p est linéaire, les domaines des σ_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 σ 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 $\#_i(e) = v_i$

Rappel : d'après la représentation des données en Ocaml

- si e est une valeur immédiate, alors elle est elle-même le constructeur et il n'y a pas de composantes
- si e est un bloc, alors l'entête donne le constructeur, et les champs suivants donnent les composantes

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 :

```

$$\begin{aligned} F(x, e, \text{action}) &= \\ &\quad \text{let } x = e \text{ in } \text{action} \\ F(C, e, \text{action}) &= \\ &\quad \text{if } \text{constr}(e) = C \text{ then } \text{action} \text{ else } \text{error} \\ F(C(p), e, \text{action}) &= \\ &\quad \text{if } \text{constr}(e) = C \text{ then } F(p, \#_1(e), \text{action}) \text{ else } \text{error} \\ F(C(p_1, \dots, p_n), e, \text{action}) &= \\ &\quad \text{if } \text{constr}(e) = C \text{ then} \\ &\quad \quad F(p_1, \#_1(e), F(p_2, \#_2(e), \dots F(p_n, \#_n(e), \text{action}) \dots)) \\ &\quad \text{else } \text{error} \end{aligned}$$

```

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

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

$F(p, e, action) \xrightarrow{*} \sigma(action)$	s'il existe σ telle que $v = \sigma(p)$
$F(p, e, action) \xrightarrow{*} error$	sinon

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 σ telle que $v = \sigma(p)$ et $F(C(p_1, \dots, p_n), e, action) = error$
 - si $constr(v) = C$, on a $v = C(v_1, \dots, v_n)$; σ telle que $v = \sigma(p)$ existe si et seulement s'il existe des σ_i telles que $v_i = \sigma_i(p_i)$
 si l'une des σ_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 σ_i existent, alors par hypothèse de récurrence

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

□

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 :

$$\begin{aligned} F(x, e, \text{succès}, \text{échec}) &= \\ &\quad \text{let } x = e \text{ in succès} \\ F(C, e, \text{succès}, \text{échec}) &= \\ &\quad \text{if } \text{constr}(e) = C \text{ then succès else échec} \\ F(C(p_1, \dots, p_n), e, \text{succès}, \text{échec}) &= \\ &\quad \text{if } \text{constr}(e) = C \text{ then} \\ &\quad \quad F(p_1, \#_1(e), F(p_2, \#_2(e), \dots F(p_n, \#_n(e), \text{succès}, \text{échec}) \dots, \text{échec}) \\ &\quad \text{else échec} \end{aligned}$$

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 $constr(e) = ::$)

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

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

L'algorithme F procède récursivement sur la matrice

- $n = 0$

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

- $m = 0$

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

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

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m \\ \textcolor{red}{x}_{1,1} & p_{1,2} & \dots & p_{1,m} \rightarrow action_1 \\ \vdots & & & \\ \textcolor{red}{x}_{n,1} & p_{n,2} & \dots & p_{n,m} \rightarrow action_n \end{array} \right|$$

on élimine cette colonne en introduisant des let

$$F(M) = F \left| \begin{array}{cccc} e_2 & \dots & e_m & \\ p_{1,2} & \dots & p_{1,m} & \rightarrow \text{let } x_{1,1} = e_1 \text{ in } action_1 \\ \vdots & & & \\ p_{n,2} & \dots & p_{n,m} & \rightarrow \text{let } x_{n,1} = e_1 \text{ in } action_n \end{array} \right|$$

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

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

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

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

d'où

$$M_C = \left| \begin{array}{cccc} \#_1(e_1) & e_2 & \dots & e_m \\ q & p_{1,2} & \dots & p_{1,m} \rightarrow \textit{action}_1 \\ - & p_{3,2} & & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } \textit{action}_3 \\ - & p_{5,2} & & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } \textit{action}_5 \\ t & p_{6,2} & \dots & p_{6,m} \rightarrow \textit{action}_6 \end{array} \right|$$

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

d'où

$$M_D = \left| \begin{array}{ccc} e_2 & \dots & e_m \\ p_{2,2} & p_{2,m} \rightarrow action_2 \\ p_{3,2} & p_{3,m} \rightarrow \text{let } x = e_1 \text{ in } action_3 \\ p_{5,2} & \dots & p_{5,m} \rightarrow \text{let } y = e_1 \text{ in } action_5 \end{array} \right|$$

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

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

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

diminue 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

$$\begin{array}{l} \text{match } (e_1, e_2, \dots, e_m) \text{ with} \\ | (p_{1,1}, p_{1,2}, \dots, p_{1,m}) \rightarrow \textit{action}_1 \\ | \dots \\ | (p_{n,1}, p_{n,2}, \dots, p_{n,m}) \rightarrow \textit{action}_n \end{array}$$

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)

Considérons

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

c'est-à-dire la matrice

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

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
```

- 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
```