École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 9 / 2 décembre 2016

objectif

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

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x \rightarrow e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

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** (en anglais *closure*)

c'est 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 $\texttt{fun}\ x \to e$?

ce sont les **variables libres** de $\texttt{fun}\ x \to e$

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

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

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

```
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
```

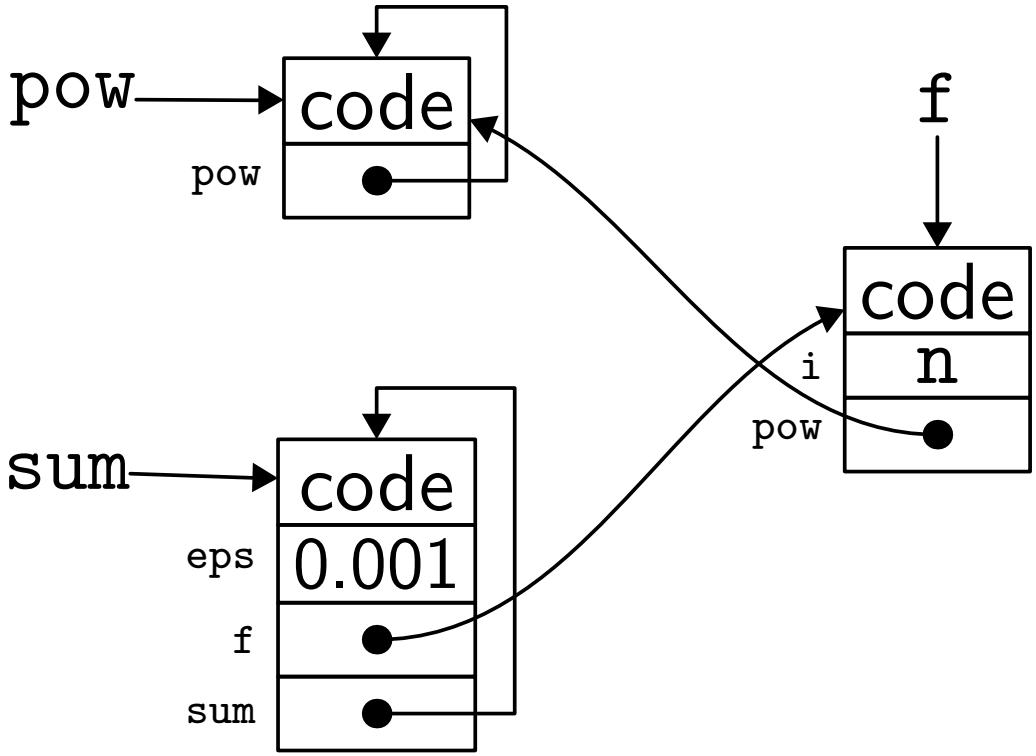
- pour `fun n ->`, l'environnement vaut $\{\texttt{pow}\}$
- pour `fun x ->`, l'environnement vaut $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$

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 $\texttt{fun}\ x \to e$ et on les remplace par une opération explicite de construction de fermeture

   $$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

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

   $$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

   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

- `Vglobal` *s* : une variable globale (introduite par `let`) de nom *s*
- `Vlocal` *n* : une variable locale (introduite par `let in`), à la position *n* dans le tableau d'activation
- `Vclos` *n* : une variable contenue dans la fermeture, à la position *n*
- `Varg` : l'unique argument de la fonction (le `x` de `fun x -> e`)

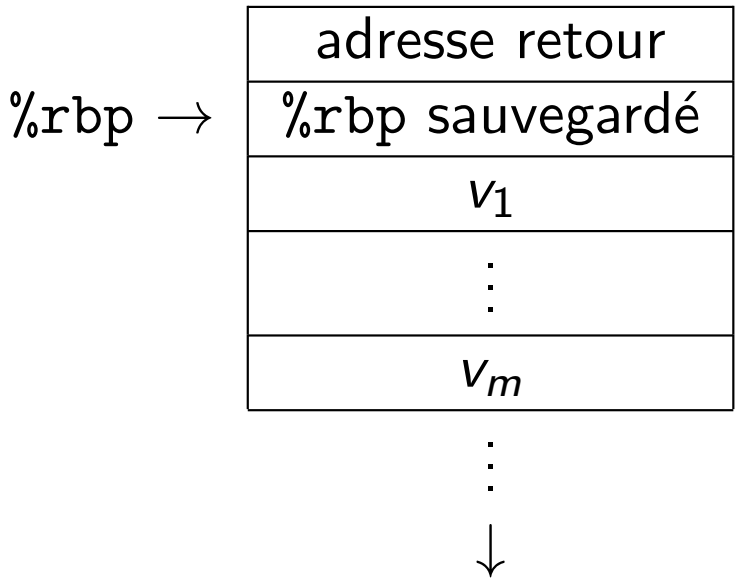
cette analyse de portée peut être réalisée conjointement avec l'explicitation des fermetures

chaque fonction à un unique argument (`Varg`), qu'on passera dans `%rdi`

la fermeture sera passée dans `%rsi`

le tableau d'activation ressemble à ceci,
où $v_1, \ldots, v_m$ sont les variables locales

il est intégralement construit par l'appelé



expliquons maintenant comment compiler

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

pour compiler la construction

$$\texttt{Eclos}(f,\ [y_1, \ldots, y_n])$$

on procède ainsi

1. on alloue un bloc de taille $n+1$ sur le tas (avec `malloc`)
2. on stocke l'adresse de f dans le champ 0
   (f est une étiquette dans le code et on obtient son adresse avec $\$f$)
3. on stocke les valeurs des variables $y_1, \ldots, y_n$ dans les champs 1 à n
4. on renvoie le pointeur sur le bloc

note : on se repose sur un GC pour libérer ce bloc lorsque ce sera possible (le fonctionnement d'un GC sera expliqué plus loin)

pour compiler un appel de la forme

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

on procède ainsi

1. on compile e_1 dans le registre `%rsi`
   (sa valeur est un pointeur p_1 vers une fermeture)
2. on compile e_2 dans le registre `%rdi`
3. on appelle la fonction dont l'adresse est contenue dans le premier champ de la fermeture, avec `call *(%rsi)`
   (saut à une adresse calculée, indirecte)

pour compiler un accès à une variable

$$\texttt{Evar}\ x$$

on distingue quatre cas selon la valeur de x

- `Vglobal` s : la valeur se trouve à l'adresse donnée par l'étiquette s
- `Vlocal` n : la valeur se trouve à l'adresse donnée par n(`%rbp`)
- `Vclos` n : la valeur se trouve à l'adresse donnée par n(`%rsi`)
- `Varg` : la valeur se trouve dans `%rdi`

enfin, pour compiler la déclaration

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

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 `%rbp` et on positionne `%rbp`
3. on évalue e
4. on restaure `%rbp` et on désalloue le tableau d'activation
5. on exécute `ret`

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

$$f\ e_1\ \ldots\ e_n$$

et où la fonction f est définie par

$$\texttt{let}\ f\ x_1 \ldots x_n = e$$

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*)

on trouve aujourd'hui des fermetures dans

- Java (depuis 2014 et Java 8)
- C++ (depuis 2011 et C++11)

dans ces langages, les fonctions anonymes sont appelées des **lambdas**

une fonction est un objet comme un autre, avec une méthode `apply`

```
LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
  ... f.apply(x) ...
}
```

une fonction anonyme est introduite avec `->`

```
map(l, x -> { System.out.print(x); return x+y; })
```

le compilateur construit un objet fermeture (qui capture ici `y`) avec une méthode `apply`

une fonction anonyme est introduite avec `[]`

```
for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

on spécifie les variables capturées dans la fermeture (ici `y`)

on peut spécifier une capture par référence (ici de `s`)

```
for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

optimisation des appels terminaux

Définition

On dit qu'un **appel** $(f\ e_1\ \ldots\ e_n)$ *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, l'adresse de retour sauvegardée y est la bonne)

dit autrement, on peut faire un saut (`jump`) plutôt qu'un appel (`call`)

considérons

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

une compilation classique donne

```
fact:
        movq    $1, %rdx
        cmpq    %rdx, %rsi
        jle     L0              # n <= 1 ?
        imulq   %rsi, %rdi      # acc <- acc * n
        decq    %rsi            # n <- n - 1
        call    fact
        ret
L0:     movq    %rdi, %rax
        ret
```

en optimisant l'appel terminal, on obtient

```
fact:   movq    $1, %rdx
        cmpq    %rdx, %rsi
        jle     L0              # n <= 1 ?
        imulq   %rsi, %rdi      # acc <- acc * n
        decq    %rsi            # n <- n - 1
        jmp     fact
L0:     movq    %rdi, %rax
        ret
```

le résultat est une **boucle**

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

```
while (n > 1) {
  acc = acc * n;
  n   = n - 1
}
```

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

le programme obtenu est plus efficace

en particulier car on accède moins à la mémoire
(on n'utilise plus `call` et `ret`)

l'espace de pile utilisé devient constant

en particulier, on évite ainsi tout débordement de pile qui serait dû à un trop grand nombre d'appels imbriqués

```
Stack overflow during evaluation (looping recursion?).
```

```
Fatal error: exception Stack_overflow
```

```
Exception in thread "main" java.lang.StackOverflowError
```

```
Segmentation fault
```

etc.

il est important de noter que la notion d'appel terminal

- n'a rien à voir avec les langages fonctionnels
  $\Rightarrow$ sa compilation peut être optimisée dans tous les langages
  (par exemple, `gcc -O2` le fait)

- n'est pas liée à la récursivité
  (même si c'est le plus souvent une fonction récursive qui fera déborder la pile)

exercice : étant donné le type d'arbres binaires

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

écrire une fonction qui calcule la hauteur d'un arbre

```
val height: 'a tree -> int
```

le code naturel

```
let rec height = function
| Empty           -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

va provoquer un débordement de pile sur un arbre de grande hauteur

au lieu de calculer la hauteur h de l'arbre, calculons $k(h)$ pour une fonction k quelconque, appelée **continuation**

```
val height: 'a tree -> (int -> 'b) -> 'b
```

on appelle cela la **programmation par continuations**
(en anglais *continuation-passing style*, ou CPS)

le programme voulu s'en déduira avec la continuation identité,
c'est-à-dire `height t (fun h -> h)`

le code prend alors la forme suivante

```
let rec height t k = match t with
 | Empty ->
     k 0
 | Node (l, _, r) ->
     height l (fun hl ->
     height r (fun hr ->
     k (1 + max hl hr)))
```

on constate que tous les appels à `height` et `k` sont **terminaux**

le calcul de `height` se fait donc en espace de pile constant

on a remplacé l'espace sur la pile par de l'espace **sur le tas**

il est occupé par les fermetures

la première fermeture capture `r` et `k`, la seconde `hl` et `k`

bien sûr, il y a d'autres solutions, ad hoc, pour calculer la hauteur d'un arbre sans faire déborder la pile (par exemple un parcours en largeur)

de même qu'il y a d'autres solutions si le type d'arbres est plus complexe (arbres mutables, hauteur stockée dans le nœud, pointeurs parents, etc.)

mais la solution à base de CPS a le mérite d'être **mécanique**

et si le langage optimise l'appel terminal
mais ne propose pas de fonctions anonymes (par exemple C) ?

il suffit de construire des fermetures soi-même, à la main

on peut même le faire avec un type ad hoc

```
enum kind { Kid, Kleft, Kright };

struct Kont {
  enum kind kind;
  union { struct Node *r; int hl; };
  struct Kont *kont;
};
```

et une fonction pour l'appliquer

```
int apply(struct Kont *k, int v) { ... }
```

cela s'appelle la **défonctionnalisation** (Reynolds 1972)

filtrage

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

- les définitions de fonctions

$$\texttt{function}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

- les conditionnelles généralisées

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

- les gestionnaires d'exceptions

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

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

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

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

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

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

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

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

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

preuve :

- soit v qui filtre p ; on a donc $v = \sigma(p)$ pour un certain σ, soit $v = C(\sigma(p_1), \ldots, \sigma(p_n))$ 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 $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$, on a
  $$\begin{aligned}\sigma(p_i) &= \sigma_1(\sigma_2(\ldots \sigma_n(p_i)) \ldots) \\ &= \sigma_1(\sigma_2(\ldots \sigma_i(p_i)) \ldots) \\ &= \sigma_1(\sigma_2(\ldots v_i) \ldots) \\ &= v_i\end{aligned}$$
  donc $\sigma(p) = C(\sigma(p_1), \ldots, \sigma(p_n)) = C(v_1, \ldots, v_n) = v$ □

Définition

Dans le filtrage

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

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

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

autrement dit, si $e = C(v_1, \ldots, v_n)$ alors $\mathit{constr}(e) = C$ 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

on commence par la compilation d'une ligne de filtrage

$$\mathit{code}(\texttt{match } e \texttt{ with } p \to \mathit{action}) = F(p, e, \mathit{action})$$

où la fonction de compilation F est définie ainsi :

$$\begin{array}{l} F(x, e, \mathit{action}) = \\ \quad \texttt{let } x = e \texttt{ in } \mathit{action} \\ F(C, e, \mathit{action}) = \\ \quad \texttt{if } \mathit{constr}(e) = C \texttt{ then } \mathit{action} \texttt{ else } \mathit{error} \\ F(C(p), e, \mathit{action}) = \\ \quad \texttt{if } \mathit{constr}(e) = C \texttt{ then } F(p, \#_1(e), \mathit{action}) \texttt{ else } \mathit{error} \\ F(C(p_1, \ldots, p_n), e, \mathit{action}) = \\ \quad \texttt{if } \mathit{constr}(e) = C \texttt{ then} \\ \qquad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), \mathit{action}) \ldots) \\ \quad \texttt{else } \mathit{error} \end{array}$$

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 $e \xrightarrow{\star} v$ alors

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

preuve : par récurrence sur p

- $p = x$ ou $p = C$: c'est immédiat
- $p = C(p_1, \ldots, p_n)$:
  - si $constr(v) \neq C$, il n'existe pas de σ telle que $v = \sigma(p)$ et $F(C(p_1, ..., p_n), e, action) = error$
  - si $constr(v) = C$, on a $v = C(v_1, \ldots, v_n)$; 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 $F(p_i, \#_i(e), \ldots)$ se réduit en $error$ et $F(p, e, action)$ également

    si toutes les σ_i existent, alors par hypothèse de récurrence

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

□

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

$$\begin{aligned} &\mathit{code}(\texttt{match } x \texttt{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n) = \\ &\quad F(p_1, x, e_1, F(p_2, x, e_2, \ldots F(p_n, x, e_n, \mathit{error}) \ldots)) \end{aligned}$$

où la fonction de compilation F est maintenant définie par :

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

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 $\mathit{constr}(e) \neq []$ alors nécessairement $\mathit{constr}(e) = ::$)

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**

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

dont la signification est

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

l'algorithme F procède récursivement sur la matrice

- $n = 0$

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

- $m = 0$

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

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

$$M = \left| \begin{array}{llll} e_1 & e_2 & \dots & e_m \\ x_{1,1} & p_{1,2} & \dots & p_{1,m} \quad \to \quad action_1 \\ \vdots & & & \\ x_{n,1} & p_{n,2} & \dots & p_{n,m} \quad \to \quad action_n \end{array} \right|$$

on élimine cette colonne en introduisant des `let`

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

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

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

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

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

d'où

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

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

d'où

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

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

d'où

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

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

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

on pose alors

$$
\begin{aligned}
F(M) = \quad & \texttt{case } \mathit{constr}(e_1) \texttt{ in} \\
& \quad C \Rightarrow F(M_C) \\
& \quad D \Rightarrow F(M_D) \\
& \quad E \Rightarrow F(M_E) \\
& \quad \texttt{otherwise} \Rightarrow F(M_R)
\end{aligned}
$$

cet algorithme termine

en effet, la grandeur

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

diminue strictement à chaque appel récursif à F, si on pose

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

la correction de cet algorithme est laissée en exercice

indication : utiliser l'interprétation de la matrice comme

$$
\begin{array}{l}
\texttt{match}\ (e_1, e_2, \ldots, e_m)\ \texttt{with} \\
\mid\ (p_{1,1}, p_{1,2}, \ldots, p_{1,m}) \rightarrow \mathit{action}_1 \\
\mid\ \ldots \\
\mid\ (p_{n,1}, p_{n,2}, \ldots, p_{n,m}) \rightarrow \mathit{action}_n
\end{array}
$$

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

$$\begin{array}{l}\texttt{case } \mathit{constr}(e_1) \texttt{ in} \\ \quad C \Rightarrow F(M_C) \\ \quad D \Rightarrow F(M_D) \\ \quad E \Rightarrow F(M_E) \\ \quad \texttt{otherwise} \Rightarrow F(M_R)\end{array}$$

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 lorqu'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

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

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
```

la suite

- TD 9
  - aide au projet
- prochain cours
  - compilation des langages objets