

École Normale Supérieure

Langages de programmation et compilation

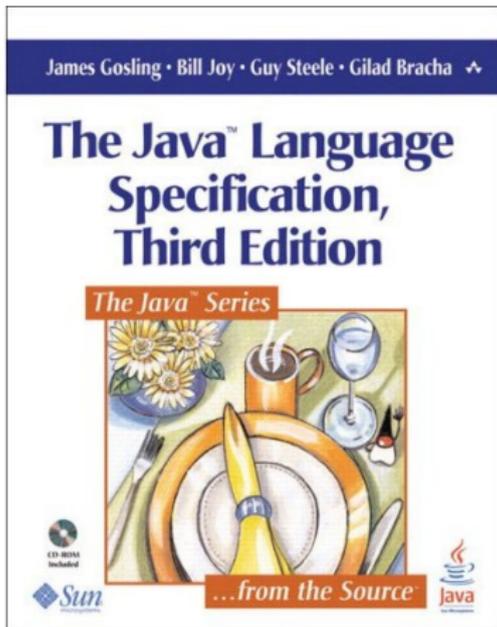
Jean-Christophe Filliâtre

Cours 3 / 7 octobre 2016

comment définir la signification des programmes écrits dans un langage ?

la plupart du temps, on se contente d'une description informelle, en langue naturelle (norme ISO, standard, ouvrage de référence, etc.)

s'avère peu satisfaisant, car souvent imprécis, voire ambigu



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

la **sémantique formelle** caractérise mathématiquement les calculs décrits par un programme

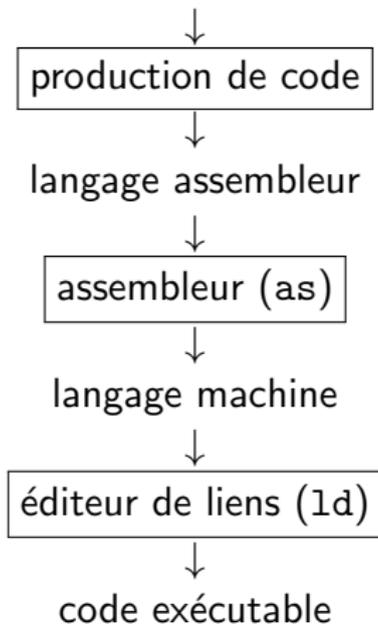
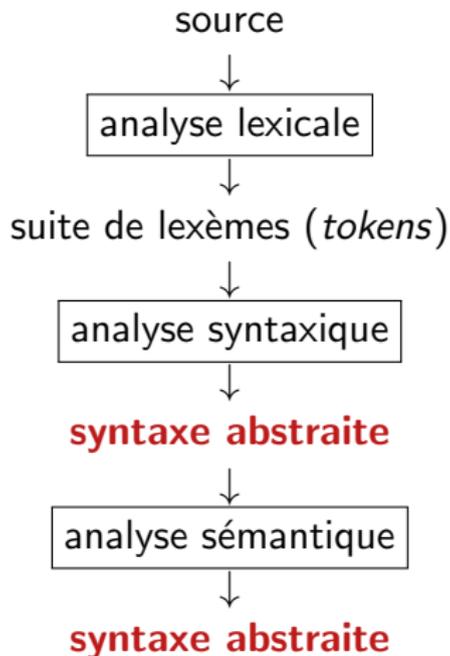
utile pour la réalisation d'outils (interprètes, compilateurs, etc.)

indispensable pour raisonner sur les programmes

mais qu'est-ce qu'un programme ?

en tant qu'objet syntaxique (suite de caractères),
il est trop difficile à manipuler

on préfère utiliser la **syntaxe abstraite**



les textes

```
2*(x+1)
```

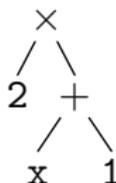
et

```
(2 * ((x) + 1))
```

et

```
2 * (* je multiplie par deux *) ( x + 1 )
```

représentent tous le même **arbre de syntaxe abstraite**



on définit une syntaxe abstraite par une grammaire, de la manière suivante

$e ::= c$	<i>constante</i>
x	<i>variable</i>
$e + e$	<i>addition</i>
$e \times e$	<i>multiplication</i>
\dots	

se lit « une expression e est

- soit une constante,
- soit une variable,
- soit l'addition de deux expressions,
- etc. »

la notation $e_1 + e_2$ de la syntaxe abstraite emprunte le symbole de la syntaxe concrète

mais on aurait pu tout aussi bien choisir $Add(e_1, e_2)$, $+(e_1, e_2)$, etc.

en OCaml, on réalise les arbres de syntaxe abstraite par des types récurifs

```
type binop = Add | Mul | ...
```

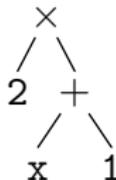
```
type expression =  
  | Cte of int  
  | Var of string  
  | Bin of binop * expression * expression  
  | ...
```

l'expression $2 * (x + 1)$ est représentée par

```
Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

il n'y a pas de constructeur dans la syntaxe abstraite pour les parenthèses

dans la syntaxe **concrète** $2 * (x + 1)$,
les parenthèses servent à reconnaître cet arbre



plutôt qu'un autre ; le cours 6 expliquera comment

on appelle **sucre syntaxique** une construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite

elle est donc traduite à l'aide d'autres constructions de la syntaxe abstraite (généralement à l'analyse syntaxique)

exemples :

- en OCaml, l'expression $[e_1; e_2; \dots; e_n]$ est du sucre pour
$$e_1 :: e_2 :: \dots :: e_n :: []$$
- en C, l'expression $a[i]$ est du sucre pour $*(a+i)$

c'est sur la syntaxe abstraite que l'on va définir la sémantique

il existe de nombreuses approches

- sémantique axiomatique
- sémantique dénotationnelle
- sémantique par traduction
- sémantique opérationnelle

encore appelée **logique de Hoare**

(*An axiomatic basis for computer programming*, 1969)

caractérise les programmes par l'intermédiaire des propriétés satisfaites par les variables ; on introduit le triplet

$$\{P\} i \{Q\}$$

signifiant « si la formule P est vraie avant l'exécution de l'instruction i , alors la formule Q sera vraie après »

exemple :

$$\{x \geq 0\} x := x + 1 \{x > 0\}$$

exemple de règle :

$$\{P[x \leftarrow E]\} x := E \{P(x)\}$$

la **sémantique dénotationnelle** associe à chaque expression e sa dénotation $\llbracket e \rrbracket$, qui est un objet mathématique représentant le calcul désigné par e

exemple : expressions arithmétiques avec une seule variable x

$$e ::= x \mid n \mid e + e \mid e * e \mid \dots$$

la dénotation peut être une fonction qui associe à la valeur de x la valeur de l'expression

$$\begin{aligned} \llbracket x \rrbracket &= x \mapsto x \\ \llbracket n \rrbracket &= x \mapsto n \\ \llbracket e_1 + e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) + \llbracket e_2 \rrbracket(x) \\ \llbracket e_1 * e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) \times \llbracket e_2 \rrbracket(x) \end{aligned}$$

(encore appelée sémantique dénotationnelle à la Strachey)

on peut définir la sémantique d'un langage en le traduisant vers un langage dont la sémantique est déjà connue

la **sémantique opérationnelle** décrit l'enchaînement des calculs élémentaires qui mènent de l'expression à son résultat (sa valeur)

elle opère directement sur des objets syntaxiques (la syntaxe abstraite)

deux formes de sémantique opérationnelle

- « sémantique naturelle » ou « à grands pas » (*big-steps semantics*)

$$e \xrightarrow{v}$$

- « sémantique à réductions » ou « à petits pas » (*small-steps semantics*)

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

illustrons la sémantique opérationnelle sur le langage **mini-ML**

$e ::=$	x	identificateur
	c	constante (1, 2, ..., <i>true</i> , ...)
	op	primitive (+, ×, <i>fst</i> , ...)
	$\text{fun } x \rightarrow e$	fonction
	$e e$	application
	(e, e)	paire
	$\text{let } x = e \text{ in } e$	liaison locale

```
let compose = fun f → fun g → fun x → f (g x) in  
let plus = fun x → fun y → + (x, y) in  
compose (plus 2) (plus 4) 36
```

```
let distr_pair = fun f → fun p → (f (fst p), f (snd p)) in  
let p = distr_pair (fun x → x) (40, 2) in  
+ (fst p, snd p)
```

la conditionnelle peut être définie comme

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \stackrel{\text{def}}{=} \text{opif } (e_1, ((\text{fun } _ \rightarrow e_2), (\text{fun } _ \rightarrow e_3)))$$

où *opif* est une primitive

les branches sont **gelées** à l'aide de fonctions

de même, la récursivité peut être définie comme

$$\text{rec } f \ x = e \stackrel{\text{def}}{=} \text{opfix } (\text{fun } f \rightarrow \text{fun } x \rightarrow e)$$

où *opfix* est un opérateur de point fixe, satisfaisant

$$\text{opfix } f = f (\text{opfix } f)$$

exemple

$$\text{opfix } (\text{fun } \text{fact} \rightarrow \text{fun } n \rightarrow \text{if } n = 0 \text{ then } 1 \text{ else } \times (n, \text{fact } (-(n, 1))))$$

sémantique opérationnelle à grands pas de mini-ML

on cherche à définir une relation entre une expression e et une **valeur** v

$$e \xrightarrow{v} v$$

les valeurs sont ainsi définies

$v ::=$	c	constante
	op	primitive non appliquée
	$\text{fun } x \rightarrow e$	fonction
	(v, v)	paire

pour définir $e \xrightarrow{v} v$, on a besoin des notions de **règles d'inférence** et de **substitution**

une relation peut être définie comme la **plus petite relation** satisfaisant un ensemble d'axiomes de la forme

$$\overline{P}$$

et un ensemble d'implications de la forme

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

exemple : on peut définir la relation $\text{Pair}(n)$ par les deux règles

$$\overline{\text{Pair}(0)} \quad \text{et} \quad \frac{\text{Pair}(n)}{\text{Pair}(n+2)}$$

qui doivent se lire comme

$$\begin{array}{l} \text{d'une part } \text{Pair}(0) \\ \text{et d'autre part } \forall n. \text{Pair}(n) \Rightarrow \text{Pair}(n+2) \end{array}$$

la plus petite relation satisfaisant ces deux propriétés coïncide avec la propriété « n est un entier pair » :

- les entiers pairs sont clairement dedans, par récurrence
- s'il y avait un entier impair, on pourrait enlever le plus petit

une **dérivation** est un arbre dont les nœuds correspondent aux règles et les feuilles aux axiomes ; exemple

$$\frac{\frac{\frac{}{\text{Pair}(0)}}{\text{Pair}(2)}}{\text{Pair}(4)}}$$

l'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence

Définition (variables libres)

L'ensemble des **variables libres** d'une expression e , noté $fv(e)$, est défini par récurrence sur e de la manière suivante :

$$\begin{aligned}
 fv(x) &= \{x\} \\
 fv(c) &= \emptyset \\
 fv(op) &= \emptyset \\
 fv(\text{fun } x \rightarrow e) &= fv(e) \setminus \{x\} \\
 fv(e_1 \ e_2) &= fv(e_1) \cup fv(e_2) \\
 fv((e_1, e_2)) &= fv(e_1) \cup fv(e_2) \\
 fv(\text{let } x = e_1 \text{ in } e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\})
 \end{aligned}$$

Une expression sans variable libre est dite **close**.

$$fv(\text{let } x = +(20, 1) \text{ in } (\text{fun } y \rightarrow +(y, y)) x) = \emptyset$$

$$fv(\text{let } x = \mathbf{z} \text{ in } (\text{fun } y \rightarrow (x y) \mathbf{t})) = \{z, t\}$$

Définition (substitution)

Si e est une expression, x une variable et v une valeur, on note $e[x \leftarrow v]$ la **substitution** de toute occurrence libre de x dans e par v , définie par

$$\begin{aligned}
 x[x \leftarrow v] &= v \\
 y[x \leftarrow v] &= y \quad \text{si } y \neq x \\
 c[x \leftarrow v] &= c \\
 op[x \leftarrow v] &= op \\
 (\text{fun } x \rightarrow e)[x \leftarrow v] &= \text{fun } x \rightarrow e \\
 (\text{fun } y \rightarrow e)[x \leftarrow v] &= \text{fun } y \rightarrow e[x \leftarrow v] \quad \text{si } y \neq x \\
 (e_1 \ e_2)[x \leftarrow v] &= (e_1[x \leftarrow v] \ e_2[x \leftarrow v]) \\
 (e_1, e_2)[x \leftarrow v] &= (e_1[x \leftarrow v], e_2[x \leftarrow v]) \\
 (\text{let } x = e_1 \text{ in } e_2)[x \leftarrow v] &= \text{let } x = e_1[x \leftarrow v] \text{ in } e_2 \\
 (\text{let } y = e_1 \text{ in } e_2)[x \leftarrow v] &= \text{let } y = e_1[x \leftarrow v] \text{ in } e_2[x \leftarrow v] \\
 &\quad \text{si } y \neq x
 \end{aligned}$$

`((fun x → +(x, x)) x) [x ← 21] = (fun x → +(x, x)) 21`

`+(x, let x = 17 in x) [x ← 3] = +(3, let x = 17 in x)`

`(fun y → y y) [y ← 17] = fun y → y y`

$$\overline{c \xrightarrow{v} c} \quad \overline{op \xrightarrow{v} op} \quad \overline{(\text{fun } x \rightarrow e) \xrightarrow{v} (\text{fun } x \rightarrow e)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)} \quad \frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\text{let } x = e_1 \text{ in } e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} (\text{fun } x \rightarrow e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

note : on a fait le choix d'une stratégie d'**appel par valeur**

il faut ajouter des règles pour les primitives ; par exemple

$$\frac{e_1 \xrightarrow{v} + \quad e_2 \xrightarrow{v} (n_1, n_2) \quad n = n_1 + n_2}{e_1 \ e_2 \xrightarrow{v} n}$$

$$\frac{e_1 \xrightarrow{v} fst \quad e_2 \xrightarrow{v} (v_1, v_2)}{e_1 \ e_2 \xrightarrow{v} v_1}$$

$$\frac{e_1 \xrightarrow{v} opif \quad e_2 \xrightarrow{v} (true, ((fun _ \rightarrow e_3), (fun _ \rightarrow e_4))) \quad e_3 \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} opfix \quad e_2 \xrightarrow{v} (fun \ f \rightarrow e) \quad e[f \leftarrow opfix (fun \ f \rightarrow e)] \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

$$\begin{array}{c}
 + \xrightarrow{v} + \quad \frac{20 \xrightarrow{v} 20 \quad 1 \xrightarrow{v} 1}{(20, 1) \xrightarrow{v} (20, 1)} \quad \text{fun } \dots \xrightarrow{v} \quad 21 \xrightarrow{v} 21 \quad \frac{\vdots}{+(21, 21) \xrightarrow{v} 42} \\
 \hline
 \frac{+(20, 1) \xrightarrow{v} 21 \quad (\text{fun } y \rightarrow +(y, y)) \quad 21 \xrightarrow{v} 42}{\text{let } x = +(20, 1) \text{ in } (\text{fun } y \rightarrow +(y, y)) \quad x \xrightarrow{v} 42}
 \end{array}$$

donner la dérivation de

$(\text{opfix } F) 2$

avec F défini comme

```
fun fact → fun n → if n = 0 then 1 else × (n, fact (-(n, 1)))
```

il existe des expressions e pour lesquelles il n'y a pas de valeur v telle que $e \xrightarrow{v}$

exemple : $e = 1\ 2$

exemple : $e = (\text{fun } x \rightarrow x\ x)\ (\text{fun } x \rightarrow x\ x)$

pour établir une propriété d'une relation définie par un ensemble de règles d'inférence, on peut raisonner par **récurrence** sur la dérivation

cela signifie par récurrence structurelle *i.e.* on peut appliquer l'hypothèse de récurrence à toute sous-dérivation
(de manière équivalente, on peut dire que l'on raisonne par récurrence sur la hauteur de la dérivation)

en pratique, on raisonne par récurrence sur la dérivation et par cas sur la dernière règle utilisée

Proposition (l'évaluation produit des valeurs closes)

Si $e \xrightarrow{v} v$ alors v est une valeur.

De plus, si e est close, alors v l'est également.

preuve par récurrence sur la dérivation $e \xrightarrow{v} v$
cas d'une application

$$\begin{array}{ccc}
 (D_1) & (D_2) & (D_3) \\
 \vdots & \vdots & \vdots \\
 e_1 \xrightarrow{v} (\text{fun } x \rightarrow e) & e_2 \xrightarrow{v} v_2 & e[x \leftarrow v_2] \xrightarrow{v} v \\
 \hline
 e_1 e_2 \xrightarrow{v} v
 \end{array}$$

par HR v est une valeur

si e est close alors e_1 et e_2 aussi, et par HR $\text{fun } x \rightarrow e$ et v_2 sont closes, donc $e[x \leftarrow v_2]$ est close, et par HR v aussi

(exercice : traiter les autres cas)

Proposition (déterminisme de l'évaluation)

Si $e \xrightarrow{v} v$ et $e \xrightarrow{v} v'$ alors $v = v'$.

par récurrence sur les dérivations de $e \xrightarrow{v} v$ et de $e \xrightarrow{v} v'$
cas d'une paire $e = (e_1, e_2)$

$$\begin{array}{cc}
 \begin{array}{c} (D_1) \\ \vdots \\ e_1 \xrightarrow{v} v_1 \end{array} & \begin{array}{c} (D_2) \\ \vdots \\ e_2 \xrightarrow{v} v_2 \end{array} \\
 \hline
 (e_1, e_2) \xrightarrow{v} (v_1, v_2)
 \end{array}
 \qquad
 \begin{array}{cc}
 \begin{array}{c} (D'_1) \\ \vdots \\ e_1 \xrightarrow{v} v'_1 \end{array} & \begin{array}{c} (D'_2) \\ \vdots \\ e_2 \xrightarrow{v} v'_2 \end{array} \\
 \hline
 (e_1, e_2) \xrightarrow{v} (v'_1, v'_2)
 \end{array}$$

par HR on a $v_1 = v'_1$ et $v_2 = v'_2$ donc $v = (v_1, v_2) = (v'_1, v'_2) = v'$

(exercice : traiter les autres cas)

remarque : la relation d'évaluation n'est pas nécessairement déterministe

exemple : on ajoute une primitive *random* et la règle

$$\frac{e_1 \xrightarrow{v} \text{random} \quad e_2 \xrightarrow{v} n_1 \quad 0 \leq n < n_1}{e_1 \ e_2 \xrightarrow{v} n}$$

on a alors $\text{random } 2 \xrightarrow{v} 0$ aussi bien que $\text{random } 2 \xrightarrow{v} 1$

on peut programmer un **interprète** en suivant les règles de la sémantique naturelle

on se donne un type pour la syntaxe abstraite des expressions

```
type expression = ...
```

et on définit une fonction

```
val valeur : expression -> expression
```

correspondant à la relation \xrightarrow{v} (puisque'il s'agit d'une fonction)

```
type expression =  
  | Var    of string  
  | Const of int  
  | Op     of string  
  | Fun    of string * expression  
  | App    of expression * expression  
  | Paire  of expression * expression  
  | Let    of string * expression * expression
```

il faut coder l'opération de substitution $e[x \leftarrow v]$

```
val subst : expression -> string -> expression -> expression
```

on suppose v close (donc pas de problème de capture de variable)

```
let rec subst e x v = match e with
| Var y ->
  if y = x then v else e
| Const _ | Op _ ->
  e
| Fun (y, e1) ->
  if y = x then e else Fun (y, subst e1 x v)
| App (e1, e2) ->
  App (subst e1 x v, subst e2 x v)
| Paire (e1, e2) ->
  Paire (subst e1 x v, subst e2 x v)
| Let (y, e1, e2) ->
  Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
```

la sémantique naturelle est réalisée par la fonction

```
val valeur : expression -> expression
```

```
let rec valeur = function  
  | Const _ | Op _ | Fun _ as v ->  
    v  
  | Paire (e1, e2) ->  
    Paire (valeur e1, valeur e2)  
  | Let(x, e1, e2) ->  
    valeur (subst e2 x (valeur e1))  
  ...
```

```
...
| App (e1, e2) ->
  begin match valeur e1 with
  | Fun (x, e) ->
    valeur (subst e x (valeur e2))
  | Op "+" ->
    let (Paire (Const n1, Const n2)) = valeur e2 in
    Const (n1 + n2)
  | Op "fst" ->
    let (Paire(v1, v2)) = valeur e2 in v1
  | Op "snd" ->
    let (Paire(v1, v2)) = valeur e2 in v2
  end
```

```
# valeur
(Let
  ("x",
    App (Op "+", Paire (Const 1, Const 20)),
    App (Fun ("y", App (Op "+", Paire (Var "y", Var "y")))
        Var "x"))));;
```

- : expression = Const 42

le filtrage est volontairement non-exhaustif

```
# valeur (Var "x");;
```

```
# valeur (App (Const 1, Const 2));;
```

```
Exception: Match_failure ("", 87, 6).
```

(on pourrait préférer un type option, une exception explicite, etc.)

l'évaluation peut ne pas terminer

par exemple sur

```
(fun x → x x) (fun x → x x)
```

```
# let b = Fun ("x", App (Var "x", Var "x")) in  
  valeur (App (b, b));;
```

Interrupted.

ajouter les opérateurs *opif* et *opfix* à cet interprète

peut-on éviter l'opération de substitution ?

idée : on interprète l'expression e à l'aide d'un environnement donnant la valeur courante de chaque variable (un dictionnaire)

```
val valeur : environnement -> expression -> valeur
```

problème : le résultat de

```
let x = 1 in fun y -> +(x, y)
```

est une fonction qui doit « mémoriser » que $x = 1$

réponse : il faut utiliser une **clôture**

on utilise le module Map pour les environnements

```
module Smap = Map.Make(String)
```

on définit un nouveau type pour les valeurs

```
type valeur =  
  | Vconst of int  
  | Vop     of string  
  | Vpaire  of valeur * valeur  
  | Vfun    of string * environnement * expression  
  
and environnement = valeur Smap.t
```

```
val valeur : environnement -> expression -> valeur
```

```
let rec valeur env = function
| Const n ->
    Vconst n
| Op op ->
    Vop op
| Paire (e1, e2) ->
    Vpaire (valeur env e1, valeur env e2)
| Var x ->
    Smap.find x env
| Let (x, e1, e2) ->
    valeur (Smap.add x (valeur env e1) env) e2
| ...
```

```

| Fun (x, e) ->
  Vfun (x, env, e)
| App (e1, e2) ->
  begin match valeur env e1 with
  | Vfun (x, clos, e) ->
    valeur (Smap.add x (valeur env e2) clos) e
  | Vop "+" ->
    let Vpaire(Vconst n1,Vconst n2) = valeur env e2 in
    Vconst (n1 + n2)
  | Vop "fst" ->
    let Vpaire (v1, _) = valeur env e2 in v1
  | Vop "snd" ->
    let Vpaire (_, v2) = valeur env e2 in v2
  end

```

note : c'est ce que l'on fait quand on compile ML (cf cours 9)

ajouter l'opérateur *opif* à cet interprète

note : ajouter l'opérateur *opfix* est plus complexe
(on en reparlera dans le cours 9)

la sémantique naturelle ne permet pas de distinguer les expressions dont le calcul « plante », comme

1 2

des expressions dont l'évaluation ne termine pas, comme

`(fun x → x x) (fun x → x x)`

la sémantique opérationnelle **à petits pas** y remédie en introduisant une notion d'étape élémentaire de calcul $e_1 \rightarrow e_2$, que l'on va itérer

on peut alors distinguer trois situations

1. l'itération aboutit à une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

2. l'itération bloque sur e_n irréductible qui n'est pas une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n$$

3. l'itération ne termine pas

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots$$

on commence par définir une relation $\xrightarrow{\epsilon}$ correspondant à une réduction « en tête », c'est-à-dire au sommet de l'expression

deux règles :

$$(\text{fun } x \rightarrow e) v \xrightarrow{\epsilon} e[x \leftarrow v]$$

$$\text{let } x = v \text{ in } e \xrightarrow{\epsilon} e[x \leftarrow v]$$

note : là encore, on a fait le choix d'une stratégie d'**appel par valeur**

on se donne également des règles pour les primitives

$$+ (n_1, n_2) \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2$$

$$fst (v_1, v_2) \xrightarrow{\epsilon} v_1$$

$$snd (v_1, v_2) \xrightarrow{\epsilon} v_2$$

$$opfix (\text{fun } f \rightarrow e) \xrightarrow{\epsilon} e[f \leftarrow opfix (\text{fun } f \rightarrow e)]$$

$$opif (true, ((\text{fun } _ \rightarrow e_1), (\text{fun } _ \rightarrow e_2))) \xrightarrow{\epsilon} e_1$$

$$opif (false, ((\text{fun } _ \rightarrow e_1), (\text{fun } _ \rightarrow e_2))) \xrightarrow{\epsilon} e_2$$

pour réduire en profondeur, on introduit la règle d'inférence

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

où E est un **contexte**, défini par la grammaire suivante :

$$\begin{array}{l} E ::= \square \\ \quad | E e \\ \quad | v E \\ \quad | \text{let } x = E \text{ in } e \\ \quad | (E, e) \\ \quad | (v, E) \end{array}$$

un contexte est donc un « terme à trou », où \square représente le trou
par exemple

$$E \stackrel{\text{def}}{=} \text{let } x = +(2, \square) \text{ in let } y = +(x, x) \text{ in } y$$

$E(e)$ dénote le contexte E dans lequel \square a été remplacé par e
exemple :

$$E(+ (10, 9)) = \text{let } x = +(2, + (10, 9)) \text{ in let } y = +(x, x) \text{ in } y$$

la règle

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

permet donc d'évaluer une sous-expression

ainsi on a la réduction

$$\frac{+(1, 2) \xrightarrow{\epsilon} 3}{\text{let } x = +(1, 2) \text{ in } +(x, x) \rightarrow \text{let } x = 3 \text{ in } +(x, x)}$$

grâce au contexte $E \stackrel{\text{def}}{=} \text{let } x = \square \text{ in } +(x, x)$

tels qu'on les a choisis, les contextes impliquent ici une évaluation en appel par valeur et « de gauche à droite »

ainsi, $(+(1, 2), \square)$ n'est pas un contexte d'évaluation

on aurait pu également choisir une évaluation de droite à gauche

on note $\xrightarrow{*}$ la clôture réflexive et transitive de \rightarrow
(i.e. $e_1 \xrightarrow{*} e_2$ ssi e_1 se réduit en e_2 en zéro, une ou plusieurs étapes)

on appelle **forme normale** toute expression e telle qu'il n'existe pas d'expression e' telle que $e \rightarrow e'$

les valeurs sont des formes normales; les formes normales qui ne sont pas des valeurs sont les expressions erronées (comme 1 2)

on va écrire les fonctions suivantes :

```
val réduction_tête : expression -> expression
```

correspond à $\xrightarrow{\epsilon}$

```
val décompose : expression -> contexte * expression
```

décompose une expression sous la forme $E(e)$
avec e réductible en tête

```
val réduction1 : expression -> expression option
```

correspond à \rightarrow

```
val réduction : expression -> expression
```

correspond à $\xrightarrow{*}$

on commence par caractériser les valeurs

```
let rec est_une_valeur = function
  | Const _ | Op _ | Fun _ ->
      true
  | Var _ | App _ | Let _ ->
      false
  | Paire (e1, e2) ->
      est_une_valeur e1 && est_une_valeur e2
```

on écrit ensuite la réduction en tête

```
let réduction_tête = function
| App (Fun(x, e1), e2) when est_une_valeur e2 ->
    subst e1 x e2
| Let (x, e1, e2) when est_une_valeur e1 ->
    subst e2 x e1
| App (Op "+", Paire (Const n1, Const n2)) ->
    Const(n1 + n2)
| App (Op "fst", Paire (e1, e2))
    when est_une_valeur e1 && est_une_valeur e2 ->
    e1
| App (Op "snd", Paire (e1, e2))
    when est_une_valeur e1 && est_une_valeur e2 ->
    e2
| _ ->
    raise PasDeReduction (* on ne peut réduire *)
```

un contexte E peut être directement représenté par une fonction
(la fonction qui à e associe $E(e)$)

```
type contexte = expression -> expression
```

```
let trou = fun e -> e
let app_gauche ctx e2 = fun e -> App (ctx e, e2)
let app_droite v1 ctx = fun e -> App (v1, ctx e)
let paire_gauche ctx e2 = fun e -> Paire (ctx e, e2)
let paire_droite v1 ctx = fun e -> Paire (v1, ctx e)
let let_gauche x ctx e2 = fun e -> Let (x, ctx e, e2)
```

```
let rec décompose e = match e with
  (* on ne peut décomposer *)
  | Var _ | Const _ | Op _ | Fun _ ->
    raise PasDeReduction
  (* cas d'une réduction en tête *)
  | App (Fun (x, e1), e2) when est_une_valeur e2 ->
    (trou, e)
  | Let (x, e1, e2) when est_une_valeur e1 ->
    (trou, e)
  | App (Op "+", Paire (Const n1, Const n2)) ->
    (trou, e)
  | App (Op ("fst" | "snd"), Paire (e1, e2))
    when est_une_valeur e1 && est_une_valeur e2 ->
    (trou, e)
  ...
```

```
...
(* cas d'une réduction en profondeur *)
| App (e1, e2) ->
    if est_une_valeur e1 then begin
        let (ctx, rd) = décompose e2 in
            (app_droite e1 ctx, rd)
    end else begin
        let (ctx, rd) = décompose e1 in
            (app_gauche ctx e2, rd)
        end
| Let (x, e1, e2) ->
    let (ctx, rd) = décompose e1 in
        (let_gauche x ctx e2, rd)
| Paire (e1, e2) ->
    ...
```

```
let réduction1 e =  
  try  
    let ctx, e' = décompose e in  
    Some (ctx (réduction_tête e'))  
  with PasDeReduction ->  
    None
```

enfin

```
let rec réduction e =  
  match réduction1 e with None -> e | Some e' -> réduction e'
```

un tel interprète n'est pas très efficace

il passe son temps à recalculer le contexte puis à l'« oublier »

on peut faire mieux, par exemple en utilisant un *zipper*

équivalence des deux sémantiques opérationnelles

nous allons montrer que les deux sémantiques opérationnelles sont équivalentes pour les expressions dont l'évaluation termine sur une valeur, *i.e.*

$$e \xrightarrow{v} v \quad \text{si et seulement si} \quad e \xrightarrow{*} v$$

Lemme (passage au contexte des réductions)

Supposons $e \rightarrow e'$. Alors pour toute expression e_2 et toute valeur v

1. $e e_2 \rightarrow e' e_2$
2. $v e \rightarrow v e'$
3. $\text{let } x = e \text{ in } e_2 \rightarrow \text{let } x = e' \text{ in } e_2$

preuve : de $e \rightarrow e'$ on sait qu'il existe un contexte E tel que

$$e = E(r) \quad e' = E(r') \quad r \xrightarrow{\epsilon} r'$$

considérons le contexte $E_1 \stackrel{\text{def}}{=} E e_2$; alors

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \rightarrow E_1(r')} \quad \text{i.e.} \quad \frac{r \xrightarrow{\epsilon} r'}{e e_2 \rightarrow e' e_2}$$

(de même pour les cas 2 et 3)

équivalence des deux sémantiques opérationnelles

Proposition (“grands pas” implique “petits pas”)

Si $e \xrightarrow{v} v$, alors $e \xrightarrow{*} v$.

preuve : par récurrence sur la dérivation de $e \xrightarrow{v} v$
supposons que la dernière règle soit

$$\frac{e_1 \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

équivalence des deux sémantiques opérationnelles

par HR on a

$$e_1 \rightarrow \dots \rightarrow v_1 = (\text{fun } x \rightarrow e_3)$$

$$e_2 \rightarrow \dots \rightarrow v_2$$

$$e_3[x \leftarrow v_2] \rightarrow \dots \rightarrow v$$

par passage au contexte (lemme précédent) on a également

$$e_1 e_2 \rightarrow \dots \rightarrow v_1 e_2$$

$$v_1 e_2 \rightarrow \dots \rightarrow v_1 v_2$$

$$e_3[x \leftarrow v_2] \rightarrow \dots \rightarrow v$$

en insérant la réduction

$$(\text{fun } x \rightarrow e_3) v_2 \xrightarrow{\epsilon} e_3[x \leftarrow v_2]$$

on obtient la réduction complète

$$e_1 e_2 \rightarrow \dots \rightarrow v$$



équivalence des deux sémantiques opérationnelles

pour le sens inverse (“petits pas” implique “grands pas”) on a besoin de deux lemmes

Lemme (les valeurs sont déjà évaluées)

$v \xrightarrow{v} v$ pour toute valeur v .

preuve : immédiat

Lemme (réduction et évaluation)

Si $e \rightarrow e'$ et $e' \xrightarrow{v} v$, alors $e \xrightarrow{v} v$.

preuve : on commence par les réductions de tête i.e. $e \xrightarrow{\epsilon} e'$

supposons par exemple $e = (\text{fun } x \rightarrow e_1) v_2$ et $e' = e_1[x \leftarrow v_2]$

on construit la dérivation

$$\frac{(\text{fun } x \rightarrow e_1) \xrightarrow{v} (\text{fun } x \rightarrow e_1) \quad v_2 \xrightarrow{v} v_2 \quad e_1[x \leftarrow v_2] \xrightarrow{v} v}{(\text{fun } x \rightarrow e_1) v_2 \xrightarrow{v} v}$$

en utilisant le lemme précédent ($v_2 \xrightarrow{v} v_2$) et l'hypothèse $e' \xrightarrow{v} v$

équivalence des deux sémantiques opérationnelles

montrons maintenant que si $e \xrightarrow{\epsilon} e'$ et $E(e') \xrightarrow{v} v$ alors $E(e) \xrightarrow{v} v$

par récurrence structurelle sur E ; on vient de faire le cas $E = \square$

considérons par exemple $E = E' e_2$

on a $E(e') \xrightarrow{v} v$ c'est-à-dire $E'(e') e_2 \xrightarrow{v} v$, qui a la forme

$$\frac{E'(e') \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{E'(e') e_2 \xrightarrow{v} v}$$

par HR on a $E'(e) \xrightarrow{v} (\text{fun } x \rightarrow e_3)$, et donc

$$\frac{E'(e) \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{E'(e) e_2 \xrightarrow{v} v}$$

c'est-à-dire $E(e) \xrightarrow{v} v$

□

Proposition (“petits pas” implique “grands pas”)

Si $e \xrightarrow{} v$, alors $e \xrightarrow{v} v$.*

preuve : supposons $e \rightarrow e_1 \rightarrow \dots \rightarrow e_n \rightarrow v$

on a $v \xrightarrow{v} v$ et donc par le lemme précédent $e_n \xrightarrow{v} v$

de même $e_{n-1} \xrightarrow{v} v$

et ainsi de suite jusqu'à $e \xrightarrow{v} v$

quid des langages impératifs ?

on peut définir une sémantique opérationnelle, à grands pas ou à petits pas, pour un langage avec des traits impératifs

on associe typiquement un **état** S à l'expression évaluée / réduite

$$S, e \xrightarrow{v} S', v \quad \text{ou encore} \quad S, e \rightarrow S', e'$$

exemple de règle :

$$\frac{S, e \xrightarrow{v} S', v}{S, x := e \xrightarrow{v} S' \oplus \{x \mapsto v\}, \text{void}}$$

l'état S peut être décomposé en plusieurs éléments, pour modéliser une pile (des variables locales), un tas, et plus encore...

ce cours reprend très largement des notes de cours de Xavier Leroy
(cours de remise à niveau « Programmation, sémantique, typage »)