

Université Paris-Sud

Compilation et langages

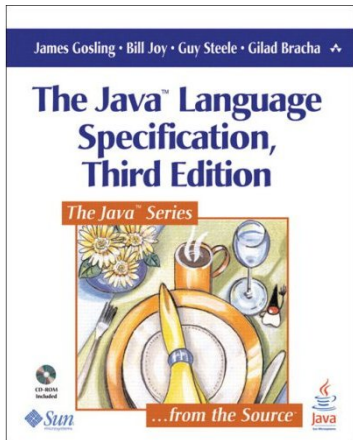
Thibaut Balabonski
d'après Jean-Christophe Filliâtre @ ENS

Cours 4 / 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 va à nouveau se concentrer sur la **syntaxe abstraite**

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 représenté 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 la valeur de x à 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*)

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

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 |
| | $ \quad op$ | primitive non appliquée |
| | $ \quad \text{fun } x \rightarrow e$ | fonction |
| | $ \quad (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

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

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**.

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

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

Définition (substitution)

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

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

$$((\text{fun } x \rightarrow +(x, x)) \text{ } \color{red}{x}) [x \leftarrow 21] = (\text{fun } x \rightarrow +(x, x)) 21$$

$$+(\color{red}{x}, \text{let } x = 17 \text{ in } x) [x \leftarrow 3] = +(3, \text{let } x = 17 \text{ in } x)$$

$$(\text{fun } y \rightarrow y \ y) [y \leftarrow 17] = \text{fun } y \rightarrow y \ y$$

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

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

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

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

Il faut ajouter des règles pour les primitives ; par exemple

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

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

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

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

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

Donner la dérivation de

$(\text{opfix } F) 2$

avec F défini comme

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

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 **récurrance** sur la dérivation

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

En pratique, on raisonne par récurrance 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

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

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 $e = (e_1, e_2)$

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

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

$$(\text{fun } x \rightarrow x \ x) (\text{fun } x \rightarrow x \ x)$$

```
# 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 ?

Idee : 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

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

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
```

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

Quid des langages impératifs ?

On peut définir une sémantique opérationnelle à grands pas pour un langage avec des traits impératifs

On associe typiquement un **état** S à l'expression évaluée

$$S, e \xrightarrow{v} S', v$$

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...

On en reparlera avec le cours sur le typage.