Sémantique des langages de programmation première partie: sémantique naturelle

Yves Bertot

Février 2008

1 Pourquoi étudier la sémantique des langages de programmation?

L'objectif de la sémantique des langages de programmation est de fournir des méthodes pour raisonner sur les programmes et de permettre la programmation sans erreurs. La sémantique des langages de programmation doit permettre d'éviter l'introduction d'erreur, en se reposant sur deux techniques:

- Grâce à la sémantique du langage de programmation, on pourra décrire précisément le comportement du programme et le comparer à un comportement attendu, par exemple sous la forme de relations logiques entre valeurs d'entrée et valeurs de sortie.
- La sémantique permettra également de définir des disciplines de programmation et de montrer que ces disciplines permettent d'éviter certaines classes d'erreurs de programmation.

Cette discipline jouit d'un intérêt économique réel. L'informatique embarquée, utilisée de plus en plus fréquemment dans tous les appareils de la vie moderne, requiert un niveau de qualité bien plus important que les application traditionnelles de l'informatique. De nombreuse applications requièrent maintenant un niveau de qualité des programmes que seules des méthodes formelles de développement et de vérification peuvent assurer. Les domaines les plus demandeurs actuellement sont ceux du transport (Matra Transport, Alsthom, Dassault-Aviation), des télécommunication (France-Télécom) et le domaine des transactions commerciales (Bull CP8, GemPlus, Schlumberger).

Ce cours est conçu comme une introduction aux différentes techniques utilisées en sémantique des langages de programmation. Il s'inspire des premiers chapitres d'un livre écrit en 1993 par Glynn Winskel, *The formal semantics of programming languages, an introduction*, et publié par MIT Press dans la série *Foundations of Computing*. Nous tâcherons de présenter les différents aspects suivants:

1. La sémantique naturelle: présenter l'exécution des programmes comme un système logique,
2. La sémantique dénotationnelle: décrire les programmes comme des fonctions mathématiques,
3. La sémantique axiomatique: décrire les propriétés logiques assurées pour les valeurs produites par les programmes.
4. Preuves par récurrence: applications aux langages de programmation,
5. Exécution de spécifications sémantiques,
6. Preuves en sémantique (compilation).

Tout au long de notre travail, nous appuierons notre étude sur l'example d'un tout petit langage, que nous appellerons `little`.

2 Description des structures de données

Nous allons volontairement réduire au maximum le langage que nous étudions. L'objectif est de faciliter notre étude en ne voyant qu'une fois chacun des concepts qui seraient répétés pour un langage plus réaliste.

2.1 Expressions et instructions

La première étape de traitement d'un programme est l'analyse syntaxique. Le résultat de cette analyse est un *terme de syntaxe abstraite*. La définition du langage passe par la description de l'ensemble des termes qui seront admis. Voici la définition du langage `little`.

- Soit V un ensemble de variables, les éléments de V seront notés v, v', v_i, w, x, ...
- le langage des expressions arithmétiques, exp, dont les éléments seront notés e, e', etc, est l'ensemble des termes de la forme v, n (ou n est un nombre entier), ou $plus(e_1, e_2)$,
- le langage des expressions booléennes, $bexp$, dont les éléments seront notés b, b', etc, est l'ensemble des termes de la forme $greater(b, b')$,
- le langage des instructions, $little$, dont les éléments seront notés I, I', etc, est l'ensemble des termes de la forme $assign(v, e)$, $sequence(I_1, I_2)$, $while(b, I)$.

Dans ce langage on peut représenter le fragment de programme suivant:

```
v1 := 0;
v2 := 0;
while( v3 > v2) {
  v1 := v2;
  v2 := v2 + 1
}
```

Voici le terme utilisé pour représenter ce programme:

$$\begin{array}{l} sequence(assign(v_1, 0), \\ \quad sequence(assign(v_2, 0), \\ \quad\quad while(greater(v_3, v_2), \\ \quad\quad\quad sequence(assign(v_1, v_2), \\ \quad\quad\quad\quad assign(v_2, plus(v_2, 1)))))) \end{array}$$

Dans la suite du cours, nous utiliserons la représentation des termes sous leur forme "mathématique" ou dans leur syntaxe "java", suivant ce qui est le mieux adapté pour nos besoins.

Exercices

1. *Ecrire un terme pour un programme qui échange les valeurs de deux variables v_1 et v_2, en utilisant une variable intermédiaire v_3.*

2. *Ecrire un terme pour un programme qui met dans une variable v_2 la factorielle de la valeur d'une variable v_1.*

3. *(facultatif, cet exercice sera probablement utilisé lors d'un cours ultérieur) Définir en Java une classe statique* `little` *représentant la structure de données des termes du langage little. Chaque opérateur pourra être représenté par une méthode statique de cette classe. Les variables et les entiers seront représentés par des opérateurs supplémentaires* `var` *et* `num`*, chacun ayant un seul champ, respectivement de type* `String` *et* `int`*. On fournira également des méthodes de discrimination, notées,* `is_var`*,* `is_nat`*,* `is_plus`*, etc, qui retournent une valeur booléenne,* `true` *si l'objet considéré a été construit par la métode* `var`*,* `nat`*, etc. En plus on fournira deux méthodes* `p1` *et* `p2`*, qui retourne le premier ou le deuxième sous-terme de chaque terme respectivement, lorsque ce sous-terme existe.*

3 Etat des machines

Le comportement d'un programme impératif s'exprime par la modification d'un état. Il importe de déterminer comment nous allons le représenter. Pour le langage qui nous intéresse, il suffit de modéliser cet état par une correspondance entre les variables du programme et des valeurs. Par nature, il n'existe qu'un nombre fini de variables dans le programme et nous pourrons nous contenter de représenter l'état comme une liste de paires dont le premier élément est une

variable et le second est une valeur. Cette liste de paires sera également un terme, dans un langage défini de la façon suivante:

- le langage des paires est constitué des expressions de la forme $pair(v, n)$ où v est une variable et n est un nombre
- le langage des états machines, *state*, dont les éléments seront notés s, s', etc, est l'ensemble des termes *nil* ou $cons(pair(v, n), s)$.

Pour une écriture plus concise, nous écrirons souvent

$$(v, n) \cdot s$$

au lieu de

$$cons(pair(v, n), s).$$

3.1 Notations pour la description des structures

Dans la présentation des ensembles de termes donnée ci-dessus, nous avons pris la peine d'indiquer les différentes catégories de variables qui seront utilisées pour chaque catégorie de terme.

Notre présentation a été volontairement verbeuse, pour laisser le temps au lecteur de s'accoutumer avec ces notations. Dans la littérature on trouvera plus souvent une présentation de la forme suivante:

$$\begin{array}{rcl} exp & = & v \mid n \mid plus(exp, exp) \\ bexp & = & greater(exp, exp) \\ little & = & assign(v, exp) \mid sequence(little, little) \mid while(exp, little) \\ state & = & nil \mid cons(pair(v, n), state) \end{array}$$

4 Descriptions sémantiques

4.1 Jugements

Dans notre travail sur la description des langages, nous allons raisonner sur des énoncés de la forme *l'exécution de l'instruction i à partir de l'état s retourne le nouvel état s'* ou bien *l'évaluation de la variable v dans l'état s retourne la valeur numérique n*. Pour chacun de ces énoncés, nous utiliserons une notation abrégée, dont le style est emprunté à la théorie de la preuve.

Dans cette section, nous allons énumérer les différents types d'énoncés qui apparaitront dans notre description du langage de programmation, en tâchant de démystifier les notations utilisées.

Pour chacun de ces énoncés nous utilisons des notations avec différents types de flèches, comme $\rightarrow$, *mapsto* ou $\rightsquigarrow$. La signification attachée à chacune de ces flèche est complètement arbitraire: nous n'utilisons ces flèches que pour disposer d'une notation compacte.

4.1.1 Evaluation des expressions

Pour déterminer la valeur d'une variable dans un état machine donné, nous utiliserons un énoncé de la forme *dans l'état s l'expression e a la valeur n*. Nous écrirons cet énoncé sous la forme suivante:

$$s \vdash e \rightarrow n$$

Il ne faut pas chercher à attribuer une signification précise aux symboles $\vdash$ et $\rightarrow$. Nous sommes simplement en présence d'une relation entre trois objets s, v, et n, et les symboles $\vdash$ et $\rightarrow$ sont utilisés pour séparer les différents objets mis en relation. Le fait que cette relation soit ternaire et que ces symboles soient utilisés, dans cet ordre, pour représenter cette relation est purement conventionnel.

Nous utiliserons la même notation pour représenter l'évaluation d'expressions booléennes.

4.1.2 Mise à jour de l'état

L'affectation provoque le changement de la valeur associée à une variable. Dans notre représentation logique cela se traduit par la création d'un nouvel état qui ne diffère de l'ancien que par la valeur associée à la variable. Nous utiliserons donc un énoncé de la forme *dans l'état initial s, la mise à jour de la variable v avec la valeur n retourne un nouvel état s'*. Nous écrirons cet énoncé de la façon suivante:

$$s \vdash v, n \mapsto s'$$

Ici encore, l'utilisation des symboles $\vdash$ et $\mapsto$ est purement conventionnelle.

4.1.3 Exécution d'une instruction

Lorsque l'on exécute une instruction, on modifie l'état de la machine. Nous utiliserons donc des énoncés de la forme *dans l'état s l'exécution d'une instruction I retourne le nouvel état s'*. Nous écrirons cet énoncé de la façon suivante:

$$s \vdash I \rightsquigarrow s'.$$

Lorsque l'on écrit ce jugement, on exprime également le fait que l'exécution de I termine.

4.1.4 Termination d'une instruction

Il peut être intéressant de se munir d'une notation distincte pour décrire le fait que l'exécution d'un programme donné termine. Nous utiliserons des énoncés de la forme *dans l'état s, l'exécution d'un programme i termine*. Nous écrirons cet énoncé de la façon suivante:

$$s \vdash i \downarrow$$

4.2 Formules logiques auxiliaire

Nous serons également amenés à traiter des énoncés simples sur les entiers naturels, de la forme $n_1 > n_2$, $n_1 \geq n_2$ ou bien $n_1 + n_2 = n_3$. La signification de ces énoncés sera toujours directe, étant donné que n_1, n_2, et n_3 sont censées représentées des valeurs entières connues.

4.3 Schémas de raisonnements: les règles d'inférence

Suivant l'approche décrite dans ce chapitre, décrire la sémantique du langage de programmation, c'est tout simplement donner les principes qui permettent d'affirmer qu'un énoncé est vrai. Ces principes prennent la forme d'étapes de raisonnement mettant en jeu plusieurs énoncés.

La notation est encore inspirée de la théorie de la preuve. On représente les étapes élémentaires de raisonnement logique par des règles d'inférence, qui ont toutes la forme *si les propositions P_1, ..., P_k sont prouvées, alors on peut prouver la proposition Q*. La tradition est de représenter ces règles d'inférence avec une barre horizontale de la manière suivante:

$$\frac{P_1 \dots P_k}{Q}$$

Les proposition P_i sont généralement appelées les *prémisses* de la règle, tandis que Q est appelée la conclusion.

Les raisonnements représentés sont schématiques dans la mesure où les propositions P_1, ..., P_k, Q peuvent contenir des variables, éventuellement partagées entre plusieurs propositions. Un nombre fini de règles permet donc de représenter l'ensemble (infini) des exécutions possibles de programmes.

Nous allons maintenant énumérer l'ensemble des règles d'inférences qui permettent de décrire la sémantique de notre langage d'exemple.

4.4 Evaluation des variables

Pour connaitre la valeur d'une variable v dans un état donné s, il faut que la valeur de cette variable soit spécifiée dans l'état. Si l'état est effectivement représenté par une liste de paires, deux cas peuvent se produire, suivant que la première paire fait référence à la variable v ou non.

Si la variable v apparait au début de la liste de paires, l'état s a alors la forme $(v, n) \cdot s'$ et la valeur de l'expression est alors n. Nous exprimons ceci par la règle d'inférence suivante:

$$\frac{}{(v, n) \cdot s' \vdash v \rightarrow n} \quad (1)$$

Si la variable v a une valeur n dans un état s, elle a encore la même valeur dans un environnement où l'on ajoute une autre paire associant une variable à une valeur. Nous exprimons ceci par la règle d'inférence suivante:

$$\frac{s \vdash v \rightarrow n}{(v', n') \cdot s \vdash v \rightarrow n}$$

Ici il faut faire attention! Que se passe-t-il si les variables v' et v sont en fait la même variable? Dans ce cas l'état de la machine est constitué de telle sorte que la variables v peut prendre deux valeurs. Pour éviter cette situation, nous utilisons une règle plus précise, qui indique que la valeur n est celle associée à v seulement si elle n'est pas masquée par la valeur v', ce qui serait le cas si v et v' étaient la même variable. On trouve alors la règle suivante:

$$\frac{s \vdash v \to n \quad v \neq v'}{(v', n') \cdot s \vdash v \to n}$$

La condition $v \neq v'$ est souvent considérée comme une condition de nature différente de l'autre prémisse, et l'usage veut que l'on mette ce genre de condition annexe sur le coté de la règles, entre parenthèses:

$$\frac{s \vdash v \to n}{(v', n') \cdot s \vdash v \to n} \qquad (v \neq v') \tag{2}$$

4.5 Evaluation des expressions

4.5.1 Valeurs numériques immédiates

En dehors des variables, on est aussi amené à évaluer des valeurs numériques et des expressions d'additions. Pour les valeurs numériques, c'est simple: elles sont égales à leur valeur, dans n'importe quel état:

$$\frac{}{s \vdash n \to n} \tag{3}$$

4.5.2 Addition

Si l'évaluation de deux expressions e_1 et e_2 dans un même état s retourne deux valeurs numériques n_1 et n_2, alors l'évaluation de l'expression $plus(e_1, e_2)$ dans le même état s retourne la somme de n_1 et n_2. Ceci s'exprime de la façon suivante:

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2}{s \vdash plus(e_1, e_2) \to n_1 + n_2} \tag{4}$$

Bien qu'il n'y ait que deux prémisses dans cette règle, il y a en fait trois opérations qui sont effectuées: l'évaluation des deux sous-expression et l'addition. Lorsque l'on utilise les règles d'inférence dans un outil de manipulation automatique on préfère parfois représenter la règle dans une forme qui montre mieux ces trois aspects:

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2 \quad n_1 + n_2 = n}{s \vdash plus(e_1, e_2) \to n}$$

4.5.3 Comparaisons

On peut être amené à avoir plusieurs règles pour le même opérateur. Ainsi, la comparaison se traite de façon similaire à l'addition, mais une règle de raisonnement représente le cas ou la valeur de la comparaison est le booléen `true` et une règle de raisonnement représente le cas où la valeur de la comparaison est le booléen `false`:

$$\frac{s \vdash e_1 \rightarrow n_1 \quad s \vdash e_2 \rightarrow n_2 \quad n_1 > n_2}{s \vdash greater(e_1, e_2) \rightarrow true} \tag{5}$$

$$\frac{s \vdash e_1 \rightarrow n_1 \quad s \vdash e_2 \rightarrow n_2 \quad n_2 \geq n_1}{s \vdash greater(e_1, e_2) \rightarrow false} \tag{6}$$

4.6 Modifier la valeur associée à une variable

Si l'on veut associer la variable v à la valeur n dans l'état de la machine, on va produire un nouvel état. La modification est simple à décrire si la variable v apparait dans la première paire de l'état: la valeur dans la première paire est modifiée, mais le reste de l'état mémoire reste inchangé. Ceci s'exprime par la règle d'inférence suivante.

$$\frac{}{(v, n') \cdot s \vdash v, n \mapsto (v, n) \cdot s} \tag{7}$$

Si l'on sait mettre à jour l'état s pour la variable v et la valeur n en obtenant un nouvel état s', alors la mise à jour de l'état $(v', n') \cdot s$ pour cette variable et cette valeur fournit l'état $(v', n') \cdot s'$. Notez que la valeur associée à la variable v' n'est alors pas changée. Ce schéma de raisonnement est valable si v et v' sont des variables distinctes. Ceci s'exprime par la règle d'inférence suivante:

$$\frac{s \vdash v, n \mapsto s'}{(v', n') \cdot s \vdash v, n \mapsto (v', n') \cdot s'} \quad (v \neq v') \tag{8}$$

Exercices

4. *On ajoute au langage des expressions booléennes $true$ et $false$, Ecrivez les règles d'inférence qui décrivent l'évaluation de ces expressions (s'inspirer des valeurs numériques).*

5. *On ajoute au langage une expression $minus$ pour représenter la soustraction. Ecrivez les variantes possibles de règles d'inférence qui décrivent l'évaluation des expression contenant des soustractions.*

6. *Même question avec un opérateur d'égalité.*

4.7 Exécution des instructions

4.7.1 Affectation

L'évaluation d'une expression e dans un état s retourne la valeur n, et que la mise à jour de l'état pour la variable v et la valeur n à partir de l'état s retourne le nouvel état s', alors l'exécution de l'affectation $assign(v, e)$ dans l'état s termine et retourne le nouvel état s'.

$$\frac{s \vdash e \rightarrow n \quad s \vdash v, n \mapsto s'}{s \vdash assign(v, e) \rightsquigarrow s'} \tag{9}$$

4.7.2 Séquence

Si l'exécution d'une instruction I_1 à partir d'un état s retourne un état s' et l'exécution d'une instruction I_2 à partir de s' retourne s'', alors l'exécution de l'instruction $sequence(I_1, I_2)$ à partir de l'état s termine et retourne l'état s''.

$$\frac{s \vdash I_1 \rightsquigarrow s' \quad s' \vdash I_2 \rightsquigarrow s''}{s \vdash sequence(I_1, I_2) \rightsquigarrow s''} \tag{10}$$

4.7.3 Boucle

Si dans l'état s l'expression e s'évalue à $true$ et l'exécution de l'instruction I termine en retournant le nouvel état s' et si dans l'état s' l'exécution de l'instruction $while(e, I)$ termine en retournant un nouvel état s'' alors dans l'état s l'exécution de l'instruction $while(e, I)$ termine en retournant l'état s''.

$$\frac{s \vdash e \rightarrow true \quad s \vdash i \rightsquigarrow s' \quad s' \vdash while(e, i) \rightsquigarrow s''}{s \vdash while(e, i) \rightsquigarrow s''} \tag{11}$$

Une autre règle d'inférence permet de décrire le comportement de l'instruction de boucle lorsque l'expression s'évalue à $false$. Dans ce cas, l'exécution s'arrête sans changement d'état:

$$\frac{s \vdash e \rightarrow false}{s \vdash while(e, I) \rightsquigarrow s} \tag{12}$$

Exercices

7. *Ajoutez dans le langage une instruction $if(e, i_1, i_2)$ et la ou les règles d'inférences qui sont adaptées.*

8. *En supposant que le comportement de l'instruction if est déjà spécifié, écrire une règle pour la boucle qui se contente de réutiliser la sémantique de if.*

5 Utiliser la spécification sémantique sur des programmes

Les règles d'inférences sont faites pour s'emboiter, en utilisant d'autres règles pour prouver les prémisses d'une règle.

Considérons la séquence de commandes suivantes, qui échange la valeur de deux variables (en supposant que la somme ne donne pas lieu à un débordement).

```
x := x + y;
y := x - y;
x := x - y;
```

Nous voulons décrire l'exécution de cette séquence d'instructions. Prenons les notations suivantes:

$$\begin{array}{lcllcl} P_2 & = & \text{y := x - y; x := x - y} & P_1 & = & \text{x := x + y; } P_2 \\ s_0 & = & (\text{“x”}, n_1) \cdot (\text{“y”}, n_2) \cdot s & s_1 & = & (\text{“x”}, n_1 + n_2) \cdot (\text{“y”}, n_1) \cdot s \\ s_2 & = & (\text{“x”}, n_1 + n_2) \cdot (\text{“y”}, n_1) \cdot s & s_3 & = & (\text{“x”}, n_2) \cdot (\text{“y”}, n_1) \cdot s \\ s_4 & = & (\text{“y”}, n_2) \cdot s & & & \end{array}$$

Nous allons construire une dérivation pour l'énoncé suivant:

$$s_0 \vdash P_1 \rightsquigarrow s_3$$

Le squelette de la dérivation a la forme suivante:

$$\dfrac{\dfrac{D_1}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1} \qquad \dfrac{\dfrac{D_2}{s_1 \vdash \texttt{y:=x-y} \rightsquigarrow s_2} \qquad \dfrac{D_3}{s_2 \vdash \texttt{x:=x-y} \rightsquigarrow s_3}}{s_1 \vdash P_2 \rightsquigarrow s_3}}{s_0 \vdash P_1 \rightsquigarrow s_3}$$

Dans cette figure, D_1, D_2 et D_3 représentent des sous-dérivations, chacune pour une affectation. La dérivation D_1 est donnée dans la figure suivante:

$$\dfrac{D_4 \qquad \overline{(\text{“x”}, n_1) \cdot s_4 \vdash (\text{“x”}, n_1 + n_2) \mapsto (\text{“x”}, n_1 + n_2) \cdot s_4}}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1}$$

La dérivation D_4 est donnée dans la figure suivante:

$$\dfrac{\dfrac{}{(\text{“x”}, n_1) \cdot (\text{“y”}, n_2) \cdot s \vdash \text{“x”} \to n_1} \qquad \dfrac{\overline{(\text{“y”}, n_2) \cdot s \vdash \text{“y”} \to n_2}}{(\text{“x”}, n_1) \cdot (\text{“y”}, n_2) \cdot s \vdash \text{“y”} \to n_2}}{s_0 \vdash \texttt{x+y} \to n_1 + n_2}$$

Exercices

9. *Construire les dérivations pour D_2 et D_3, en supposant que la sémantique pour la soustraction est donnée par la règle suivante:*

$$\dfrac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2}{s \vdash \mathit{minus}(e_1, e_2) \to n_1 - n_2}$$

10. *Construire la dérivation pour l'énoncé suivant:*

$$(\text{“x”}, 1) \cdot s \vdash \texttt{while(2>x)\{x:= x+x\}} \rightsquigarrow (\text{“x”}, 2) \cdot s$$

11. *On considère que la taille d'une expression arithmétique est le nombre d'opérateurs qui entre dans sa fabrication, en assimilant les variables et les valeurs numériques à des expressions de taille 1. l'expression $plus(x, 1)$ est de taille 3. Pour les dérivations, on considère que la taille est le nombre de règles qui interviennent dans leur construction (c'est la même chose, n'est-ce pas?). Quel est la taille minimale de la dérivation pour une expression arithmétique de taille n. Si l'on considère que l'évaluation a lieu dans un état de taille p, existe-t-il une taille maximale pour les dérivations sur les expressions de taille n?*

12. *Quelle est la taille de la dérivation décrivant l'exécution de l'expression $while(x > y, assign(y, y + 1))$ dans l'état $(x, n) \cdot (y, 0) \cdot s$?*

13. *Quelle est la taille de la dérivation décrivant l'exécution de l'expression*

    $$while(x > y, sequence(while(x > z, assign(z, z + 1)), assign(y, y + 1)))$$

    dans l'état $(x, n) \cdot (y, 0) \cdot (z, 0) \cdot s$?

14. *Ecrire un programme qui prend en entrée des représentations d'un environnement et d'une instruction et retourne en sortie la taille de la dérivation calculant l'exécution de cet instruction dans cet environnement lorsque cette taille existe (le programme pourra ne peut pas terminer s'il n'existe pas une telle dérivation). Tant qu'à faire, il ne sera pas plus difficile que ce programme retourne aussi l'environnement final. Le langage de programmation est laissé au choix de l'étudiant, mais Ocaml est conseillé.*