Sémantique des Langages de Programmation

Introduction

Stefano Guerrini
stefano.guerrini@univ-paris13.fr

LIPN - Institut Galilée, Université Paris Nord 13
Sup Galillée Informatique, 1ère année

2009–2010

Qu'est-ce que la sémantique ?

Sémantique

Étude du sens, de la signification dans le langage.

(Dictionnaire Robert)

Étude du sens des unités linguistiques et de leurs combinaisons.

Aspect de la logique qui traite de l'interprétation et de la signification des systèmes formels, par opposition à la syntaxe, entendue comme l'étude des relations formelles entre formules de tels systèmes.

(Dictionnaire Larousse)

Mots clés :

- sens et signification
- unités linguistiques et leurs combinaisons
- dichotomie sémantique/syntaxe

Et la sémantique des langages de programmation ?

Sémantique des Langages de Programmation

En informatique théorique, la sémantique formelle (des langages de programmation) est l'étude de la signification des programmes informatiques vus en tant qu'objets mathématiques.

Comme en linguistique, ici la sémantique désigne le lien entre un signifiant, le programme, et un signifié, objet mathématique qui dépendra des propriétés que l'on souhaite connaître du programme.

On appellera aussi sémantique le lien entre le langage signifiant (le langage de programmation) et le langage signifié (logique de Hoare, automates, ou autre).

(Wikipedia)

Mots clés :

- sémantique formelle
- programmes vus en tant qu'objets mathématiques
- dichotomie signifiant/signifié
- propriétés que l'on souhaite connaître des programme
- lien entre le langage de programmation (signifiant) et le langage signifié

Pourquoi la sémantique ?

Définir la sémantique d'un langage formel consiste à lui donner une signification mathématique, dans deux buts :

La description

Qui doit permettre de cerner très précisément ce que réalise un programme

- afin de permettre au programmeur de comprendre ce qu'il fait,
- afin de réaliser des traducteurs (compilateurs, interprètes) et des optimiseurs,
- afin de pouvoir appliquer des méthodes formelles de vérification.

La prescription (dans les sens du verbe *prescrire*)

Qui doit aider les concepteurs de langages à définir des langages cohérents, puissants et corrects.

A quoi il sert ?

- L'objectif final de la description formelle des langages c'est de fournir des méthodes pour raisonner sur les programmes et de permettre la programmation sans erreurs en se reposant sur deux techniques :
  1. 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 ;
  2. permettre également de définir des disciplines de programmation et de montrer que ces disciplines permettent d'éviter certaines classes d'erreurs de programmation.
- Les aspects de prescription deviennent de plus en plus important avec l'intérêt croissant à définition des *langages de domaines spécifiques*, comme par exemple dans le cas de
  - un langage pour aider les ingénieurs à calculer les trajectoires des satellites,
  - un langage pour commander un micro-onde lors de la cuisson d'un plat.

Est-ce qu'on l'utilise dans le monde réel ?

- La sémantique 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)
  - des transactions commerciales (Bull CP8, GemPlus, Schlumberger)
- Un exemple concret d'utilisation des méthodes formels : le développement et la validation du code du METEOR/Ligne 14 du métro.
  C'est évident que, dans des cas comme ça, on ne peut pas avoir du code qui contient des erreurs.

Et qu'est-ce qu'on fait dans le cours ?

- Ce cours est conçu comme une introduction aux méthodes et à les techniques utilisées en sémantique des langages de programmation.
- Tout au long de notre travail, nous appuierons notre étude sur l'exemple d'un tout petit langage impérative, que nous appellerons While.
- Référence principal :
  Hanne Riis Nielson, Flemming Nielson. Semantics with applications. John Wiley & Sons, 1992.
  `http://www.daimi.au.dk/hrn`

Nous tâcherons de présenter les différents aspects suivants :

1. La sémantique opérationnelle : qui présente l'exécution des programmes comme un système de déduction.
2. La sémantique dénotationnelle : qui décrit les programmes comme des fonctions mathématiques.
3. La sémantique axiomatique : qui décrit les propriétés logiques assurées pour les valeurs produites par les programmes.

La syntaxe

Syntaxe

Partie de la grammaire qui décrit les règles par lesquelles les unités linguistiques se combinent en phrases.

Ensemble des règles d'écriture d'un programme informatique permises dans un langage de programmation et formant la grammaire de ce langage.

(Dictionnaire Larousse)

- La syntaxe expose
  - l'alphabet des caractères utilisés et
  - la grammaire que définit le langage
    c'est-à-dire, les règles de construction des chaînes de caractères bien formées.
- Le langage définit par une grammaire ce sera l'ensemble des chaînes de caractères bien formées par rapport aux règles de la grammaire.

Syntaxe concrète et syntaxe abstraite

- La syntaxe concrète représente les suites de caractères des mot du langage.
- Pour analyser un langage de programmation c'est fondamental que la syntaxe concrète soit non-ambiguë (unicité de l'arbre de parsage) ou fournisse les règles pour gérer les cas d'ambiguïté.

  Exemple. Dans le cas des expressions arithmétiques. Pour l'expression $3 + 2 * 4$, la syntaxe concrète doit choisir si cette expression correspond à $(3 + 2) * 4$ où à $3 + (2 * 4)$.
  Ces deux possibilités en correspondant à deux arbres de parsage distingués.
- La syntaxe abstraite donne une structure aux données entrées.
- Dans la syntaxe abstraite les objets syntaxiques sont directement représentés par des arbres (ou des termes) qui sont en relation directe (ce sont isomorphes) avec les correspondants arbres de parsage.
- Dans ce cours on n'est pas intéressés aux détailles de la syntaxe concrète et on considérera seulement la syntaxe abstraites des langages qu'on analysera.

Définitions compositionnelle ou inductives

- Une définition inductive d'un ensemble X est caractérisée par la donnée d'un ensemble de règles de construction des éléments de X.
- Parmi ces règles on distingue :
  - Les règles de base. Qui affirment que des éléments appartiennent à X.
  - Les règles inductives. Qui donnent un moyen de construire d'autres éléments de X à partir de ceux déjà construits.
- L'ensemble X est l'ensemble des objets constructibles
  - en partant des règles (des objets) de base
    par exemple : $0 \in \mathbb{N}$
  - en appliquant un nombre fini des règles inductives
    par exemple : $x + 1 \in \mathbb{N}$ si $x \in \mathbb{N}$
- C'est-à-dire, l'ensemble X est le plus petit ensemble d'objets t.q.
  - les objets de bases donnés par le règles de bases sont contenus en X
  - l'ensemble X est clos par rapport aux opérations de construction d'objets donnée par les règles inductives.

Règles inductives

- Les règles de construction d'un ensemble inductif A ont la forme suivante (avec $n = 0$ dans les règles de base) :

$$\text{si } a^1 \in A^1, ..., a^n \in A^n, a_1 \in A, ..., a_m \in A$$
$$\text{alors } f(a^1, ...a^n, a_1, ...a_m) \in A$$

- où les A^i sont des ensembles qui peuvent être définis à leur fois par induction
  - les règles d'un ou plusieurs ensemble A^i peuvent utiliser des éléments de A ou d'autres ensembles A^j aussi, dans ce cas là on dit que ces ensembles sont définis par induction simultanée ou mutuelle
- et f est la description formelle de la règle (une fonction) qui donne un élément de A a partir des élément $a^1, \ldots, a^n, a_1, \ldots, a_m$.
- Pour écrire la définition d'un séries d'ensembles inductives $A^1, \ldots, A^k$
  - on utilisera des méta-variables $a^i, a^i_1, a^i_k, \ldots$ pour dénoter des éléments de A^i
  - et si $R_1, \ldots, R_k$ ce sont les règles qui définissent l'ensemble A^i
  - on écriera

$$a^i ::= R_1 \mid \ldots \mid R_k$$

Arbres et définitions inductives

La construction d'un élément x d'un ensemble inductif X corresponds à une arbre.

1. Si x est défini par une règle de base, alors l'arbre contiens seulement une noeud a
2. si x est défini par une règle inductive

$$a^i \in A^i, x_j \in X \implies f(a^1, ...a^n, x_1, ...x_m) \in X$$

alors l'arbre correspondant a

- une noeud f comme racine
- des noeuds a^i (des feuilles) comme fils de f
- les racines des arbres des éléments x_i comme fils de f

Le langage While

- Catégories syntaxiques
  - NUM : l'ensemble des numéraux ;
  - VAR : l'ensemble des variables (un ensemble de noms ou identificateurs) ;
  - AEXP : l'ensemble des expressions arithmétiques ;
  - BEXP : l'ensemble des expressions booléennes ;
  - STM : l'ensemble des assertions ou "statements".
- Méta-variables : $n \in \text{NUM}$ $x \in \text{VAR}$ $a \in \text{AEXP}$ $b \in \text{BEXP}$ $s \in \text{STM}$.
- La syntaxe abstraite (la définition des catégories syntaxiques) :

$$n ::= \mathtt{0} \mid \mathtt{1} \mid n\,\mathtt{0} \mid n\,\mathtt{1}$$
$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$
$$b ::= \mathtt{true} \mid \mathtt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$$
$$S ::= x := a \mid \mathtt{skip} \mid S_1; S_2 \mid \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \mid \mathtt{while}\ b\ \mathtt{do}\ S$$

On n'est pas intéressé à la définition concrète de l'ensemble des identificateurs pour le variables (la catégorie VAR).

Induction structurelle

- Pour démontrer qu'une propriété P est vraie de tous les éléments d'un ensemble défini inductivement, il suffit de calquer le raisonnement sur la structure de l'ensemble.
- C'est-à-dire
  1. pour chaque règle de base de type $a \in X$, il faut montrer $P(a)$
  2. pour les règles inductives de la forme :

$$a^i \in A^i, x_j \in X \implies f(a^1, ...a^n, x^1, ...x^m) \in X$$

Il faut montrer que

$$a^i \in A^i, x_j \in X \text{ et } P(x_j) \implies P(f(a^1, ...a^n, x_1, ...x_m))$$

- Le principe d'induction structurelle peut être démontrer en utilisant le principe d'induction de l'arithmétique.
  - Par induction sur les nombres de règles qui servent à construire une élément a.
  - C'est-à-dire, par induction sur la taille de l'arbre de a.

Domaines et fonctions sémantiques

- Au fin d'interpréter un langage il faut trouver une fonction qui associe une signifié à chaque signifiant.

  signifiants : les élément syntaxique du langage

  signifiés : des valeurs sémantiques

- Le but de la sémantiques est de trouver
  - les domaines sémantiques, c'est-à-dire, les domaines des signifiés, à associer à chaque catégorie syntaxique
  - une fonction sémantique qui associe à chaque élément d'une catégorie syntaxique sont signifié, c'est-à-dire, un élément du correspondant domaine sémantique.

Sémantique des numéraux

- Domaine sémantique : l'ensemble des entiers $\mathbb{Z}$
- Fonction sémantique :

$$\mathcal{N} : \textsc{Num} \to \mathbb{Z}$$

t.q., par exemple $\mathcal{N}[\![101]\!] = 5$ $\mathcal{N}[\![1]\!] = 1$ $\mathcal{N}[\![0]\!] = 0$

Attention. Noter les $[\![]\!]$ à la place des () dans les fonctions sémantiques. C'est une convention !

- Définition inductive de la fonction sémantique :

$$\begin{aligned}
\mathcal{N}[\![0]\!] &= 0 \\
\mathcal{N}[\![1]\!] &= 1 \\
\mathcal{N}[\![n\ 0]\!] &= 2 * \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ 1]\!] &= 2 * \mathcal{N}[\![n]\!] + 1
\end{aligned}$$

L'état

- L'interprétation d'une expression arithmétique dépend des valeurs associées aux variables qui ce trouvent dans l'expression.
- L'état c'est l'association d'une valeur (du bon type) à chaque variable.
- Si les variables sont toutes de type entier, un état s c'est une fonction $s : \textsc{Var} \to \mathbb{Z}$.
- L'état c'est donc une élément de l'ensemble :

$$\textsc{State} = \textsc{Var} \to \mathbb{Z}$$

- La fonction sémantique qu'interprète les expression arithmétiques
  - prend comme arguments une expression
  - et un état
  - et renvoie la valeur de l'expression.
- Mais si on définit

$$\mathcal{A} : (\textsc{Aexp} \times \textsc{State}) \to \mathbb{Z}$$

on ne peut pas donner une interprétation directe d'une expression, car on a toujours besoin d'un état s pour interpréter une expression a (l'argument de la fonction c'est une pair (a, s)).

Sémantique des expressions arithmétiques

- On définit la fonction qu'interprète les expressions arithmétique comme une fonction que prends comme argument une expression et renvoie une fonction que prend comme argument une état et renvoie la valeur de l'expression.
- Donc, les argument de cette fonction sont dans $\textsc{Aexp}$
- et les valeurs sont dans $\textsc{State} \to \mathbb{Z}$.

$$\mathcal{A} : \textsc{Aexp} \to (\textsc{State} \to \mathbb{Z})$$

$$\begin{aligned}
\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s &= s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 * a_2]\!]s &= \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{aligned}$$

Sémantique des expressions booléennes

$$\mathcal{B} : \textsc{Bexp} \to (\textsc{State} \to \mathbb{T}) \qquad \mathbb{T} = \{tt, ff\}$$

$$\mathcal{B}[\![\texttt{true}]\!]s = tt$$

$$\mathcal{B}[\![\texttt{false}]\!]s = ff$$

$$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} tt & \text{si } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ ff & \text{sinon} \end{cases}$$

$$\mathcal{B}[\![a_1 \leq a_2]\!]s = \begin{cases} tt & \text{si } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ ff & \text{sinon} \end{cases}$$

$$\mathcal{B}[\![\neg\, b]\!]s = \begin{cases} tt & \text{si } \mathcal{B}[\![b]\!]s = ff \\ ff & \text{sinon} \end{cases}$$

$$\mathcal{B}[\![b_1 \wedge b_2]\!]s = \begin{cases} tt & \text{si } \mathcal{B}[\![b_1]\!]s = tt \text{ et } \mathcal{B}[\![b_2]\!]s = tt \\ ff & \text{sinon} \end{cases}$$

Dépendance de l'état

- L'état est une fonction de l'ensemble de toutes les variables, mais en effet seulement un nombre fini de variables apparaissent dans une expression.
- La valeur d'une expression dépend exclusivement des valeurs des variables qui contient, et donc

> si on évalue une expression dans deux états qui associent les mêmes valeurs aux variables de l'expression on obtient le même résultat

- Pour formaliser la précédente propriété on a besoin de la notion de variable libre.

Variables libres

$$\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(a_1 + a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 * a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 - a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)
\end{aligned}$$

$$\begin{aligned}
\mathrm{FV}(\texttt{true}) &= \emptyset \\
\mathrm{FV}(\texttt{false}) &= \emptyset \\
\mathrm{FV}(a_1 = a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \leq a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(\neg\, b) &= \mathrm{FV}(b) \\
\mathrm{FV}(b_1 \wedge b_2) &= \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)
\end{aligned}$$

Lemma

Soient s et s' deux états t.q.

$$\forall x \in \mathrm{FV}(a) : s\,x = s'\,x$$

Alors,

$$\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$$

Lemma

Soient s et s' deux états t.q.

$$\forall x \in \mathrm{FV}(b) : s\,x = s'\,x$$

Alors,

$$\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$$

Compositionalité

- Les expressions sont obtenu par composition des autres sous-expressions.
- La sémantique d'une expression est défini par induction structurelle et donc la valeur d'une expression est obtenue par composition des valeurs de ses sous-expressions.
- De plus, la valeur d'une expression ne dépend que des valeurs des sous-expressions et pas de l'effective structure des sous-expressions :
  - Exemple : dans un état s t.q. $s\mathtt{x} = 1$ et $s\mathtt{y} = 2$

    $$\mathcal{A}[\![(\mathtt{x}+\mathtt{y})*\mathtt{y}]\!]s = 6 \qquad \mathcal{A}[\![\mathtt{x}+\mathtt{y}]\!]s = 3 \qquad \mathcal{A}[\![\mathtt{11}]\!]s = 3 \qquad \mathcal{A}[\![\mathtt{11}*\mathtt{y}]\!]s = 6$$

    en replacent la sous-expression $\mathtt{x}+\mathtt{y}$ avec une expression $\mathtt{11}$ de la même valeur, la valeur de l'expression ne change pas.
- Pour formaliser la précédente propriété il faut formaliser d'abord la notion de substitution.

Substitutions

- On peut replacer une variable avec une expression dans une autre expression

$$n[a_0/y] = n \qquad \texttt{true}[a_0/y] = \texttt{true}$$

$$x[a_0/y] = \begin{cases} a_0 & \text{si } x = y \\ x & \text{sinon} \end{cases} \qquad \texttt{false}[a_0/y] = \texttt{true}$$

$$(a_1 + a_2)[a_0/y] = (a_1[a_0/y]) + (a_2[a_0/y]) \qquad (a_1 = a_2)[a_0/y] = (a_1[a_0/y]) + (a_2[a_0/y])$$

$$(a_1 * a_2)[a_0/y] = (a_1[a_0/y]) * (a_2[a_0/y]) \qquad (a_1 \leq a_2)[a_0/y] = (a_1[a_0/y]) * (a_2[a_0/y])$$

$$(a_1 - a_2)[a_0/y] = (a_1[a_0/y]) - (a_2[a_0/y]) \qquad (\neg\, b)[a_0/y] = \neg\, (b[a_0/y])$$

$$(b_1 \wedge b_2)[a_0/y] = (b_1[b_0/y]) \wedge (b_2[b_0/y])$$

- Il y à aussi une sorte de substitution, ou de mise à jour, pour les états.

$$(s[y \mapsto v])\, x = \begin{cases} v & \text{si } x = y \\ s\, x & \text{sinon} \end{cases}$$

Lemma

$$\mathcal{A}[\![a[a_0/y]]\!]s = \mathcal{A}[\![a]\!](s[\mathcal{A}[\![a_0]\!]s])$$

$$\mathcal{B}[\![b[a_0/y]]\!]s = \mathcal{B}[\![b]\!](s[\mathcal{A}[\![a_0]\!]s])$$

Corollary

$$\text{Si } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s$$
$$\mathcal{A}[\![a[a_1/y]]\!]s = \mathcal{A}[\![a[a_2/y]]\!]s$$
$$\mathcal{B}[\![b[a_1/y]]\!]s = \mathcal{B}[\![b[a_2/y]]\!]s$$