Sémantique des langages de programmation
Quatrième partie: sémantique axiomatique

Yves Bertot

Décembre 2000

1 Prouver des propriétés de programmes

En sémantique opérationnelle ou en sémantique dénotationnelle, les objets principaux de la description sont les états de la machine. Tout raisonnement sur les variables d'un programme à un endroit donné se fait par l'intermédiaire de cet état. A l'usage, cette omniprésence des états et la nécessité de distinguer entre les différents états successifs atteints par la machine rend le raisonnement sur les calculs effectués dans le programmes lourd et incompréhensible. La sémantique axiomatique est une réponse au besoin de parler directement des propriétés des valeurs manipulées dans le programme.

1.1 Un exemple

Pour le programme suivant, nous voulons démontrer que la valeur finale de `S` est 5050.

```
S := 0;
N := 1;
while 101 > N do
  S := S + N;
  N := N + 1;
done
```

Une façon simple mais fastidieuse de vérifier cette propriété est de calculer la valeur de `N` en appliquant répétitivement les règles de la sémantique opérationnelle (voir leçon numéro 1), ou bien en calculant la valeur de la fonction dénotée par ce programme sur la variable `N` (voir leçon numéro 2).

De toutes façon cette méthode n'est pas satisfaisante si nous voulons remplacer 101 par un nombre p arbitraire, ou par une variable `P` ayant une valeur initiale p, ce qui est équivalent.

Essayons de raisonner un peu sur notre programme. Après les deux premières affectations nous savons que la valeur en ce point pour la variable `S` est 0 et la valeur pour la variable `N` est 1. Nous pouvons représenter cela par une annotation que nous allons ajouter dans le programme:

```
S := 0;
N := 1;
{S = 0 ∧ N = 1}
while 101 > N do
  S := S + N
  N := N + 1
done
```

Nous pouvons également ajouter une information que nous connaissons déjà à la fin du programme. C'est que forcément, lorsque l'on sort de la boucle, la condition `101 > N` n'est plus vérifiée. De plus, si l'on rentre dans la boucle, même après plusieurs itérations, c'est que la condition est encore vérifiée. On peut donc écrire le programme annoté suivant:

```
S := 0;
N := 1;
{S = 0 ∧ N = 1}
while 101 > N do
{101 > N }
  S := S + N
  N := N + 1
done
{101 ≤ N}
```

Nous n'avons pas encore d'informations pour la valeur de `S` en fin de boucle. Pour réunir ces informations, analysons le comportement du programme après l'exécution des premières itérations. En fin de première itération, la valeur de `N` est 2 et la valeur de `S` est formée de la façon suivante:

$$\begin{array}{rcccc} & & \text{valeur initiale de S} & & \text{valeur initiale de N} \\ S & = & 0 & + & 1 \end{array}$$

Après la deuxième itération, la valeur de `N` est 3 et pour `S`:

$$\begin{array}{rcccc} S & = & 0+1 & + & 2 \end{array}$$

En répétant le procédé, nous arrivons à la connaissance, qu'après la $k^{\text{ième}}$ intération, on a les égalités (même après la $0^{\text{ième}}$, si l'on peut dire, c'est à dire avant la première itération).

$$\begin{array}{rcccc} N & = & k & + & 1 \\ S & = & 1+\cdots+(k-1) & + & \text{k} \end{array}$$

On peut exprimer la relation entre S et N indépendamment de k en utilisant la première égalité pour réécrire la seconde, on obtient une propriété qui est vraie au début et à la fin de chaque itération:

$$S = 1+\cdots+(N-1)$$

En utilisant un résultat connu d'arithmétique:

$$S = N(N-1)/2$$

Si l'on s'intéresse aux propriétés de la variable `N`, on peut également déduire la propriété suivante, qui est vraie au début et à la fin de chaque itération (même la dernière):

$$N \leq 101$$

Ces informations peuvent être ajoutées au programmes à l'aide d'une annotation `invariant` qui doit être distinguée de l'annotation de début d'itération, car elle est également vérifiée à la fin de la dernière boucle. Notez que l'invariant est également répété après la sortie de boucle, puisque sortir de la boucle ne change pas la valeur des variables:

```
S := 0;
N := 1;
{S = 0 ∧ N = 1}
while 101 > N do
invariant { S = N(N − 1)/2 ∧ N ≤ 101}
{101 > N }
  S := S + N
  N := N + 1
done
{101 ≤ N ∧ N ≤ 101 ∧ S = N(N − 1)/2}
```

De l'ensemble des inégalités et égalités réunies en fin de programme nous pouvons déduire que la valeur finale de `N` est 101 et la valeur finale de `S` est

$$\frac{101 \times 100}{2} = 5050$$

1.2 La logique de Hoare

La logique de Hoare est un cadre logique pour raisonner sur les programmes annotés comme celui que nous avons étudié dans la section précédente. On y manipule des énoncés de la forme suivante:

$$\{P\}I\{Q\}$$

Dans cette notation I est une instruction, P est une formule logique appelée une *pré-condition* et Q est une formule logique appelée *post-condition*. Cette formule doit être lue de la façon suivante: *si la propriété P est vraie pour les valeurs des variables du programme avant l'exécution de I et si l'exécution termine, alors la propriété Q est assurée après l'exécution.*

La logique de Hoare est donnée par une collection de règles d'inférence du même style que les règles que nous avons utilisées pour la sémantique naturelle (première leçon). Il y a une règle pour chaque forme d'instruction du langage, plus une règle de conséquence.

1.2.1 Sémantique axiomatique pour l'addition

La règle pour l'affectation a la forme suivante:

$$\overline{\{P[x\backslash E]\}x := E\{P\}}$$

Tâchons d'expliquer cette règle. Si la variable x apparait dans P, alors on sait que cette variable aura dans cette formule logique la valeur que E pouvait prendre avant exécution. Il est donc naturel qu'il soit nécessaire que P, où toutes les occurrences de x ont été remplacées par E soit vraie avant exécution. Cette règle fait intervenir une opération que nous avons notée $\cdot[\cdot\backslash\cdot]$ que nous décrirons plus tard, après que nous aurons également décrit plus précisément le langage que nous utiliserons pour les assertions.

Exercices

1. *Quelle est la précondition P pour que le jugement* $\{P\}$`x := 0`$\{x = 0\}$ *soit prouvable,*
2. *Même question pour* $\{P\}$`x := 0`$\{x = 1\}$,
3. *Même question pour* $\{P\}$`x := x + 1`$\{x = 3\}$.

1.2.2 Sémantique axiomatique pour la séquence

La règle pour la séquence est la suivante:

$$\frac{\{P\}I_1\{P'\} \qquad \{P'\}I_2\{P''\}}{\{P\}I1;I_2\{P''\}}$$

Cette règle va de soi.

Exercices

4. *Quelle est la précondition P pour que le jugement*

$$\{P\}\texttt{x := 0;y:=1}\{x = 0 \wedge y = 1\}$$

soit prouvable, écrire la dérivation qui permet de le prouver.

1.2.3 Sémantique axiomatique pour la boucle

La règle pour la boucle est la suivante:

$$\frac{\{P \wedge b\}I\{P\}}{\{P\}\texttt{while } b \texttt{ do } I \texttt{ end}\{\neg b \wedge P\}}$$

Dans cette règle `P` joue le rôle que nous avons décrit comme *invariant* dans notre exemple précédent. Il faut noter que l'expression booléenne b est utilisée

comme formule logique, vraie ou fausse suivant l'endroit. Il est remarquable que l'expression P est utilisée partout, avant, après, et au milieu.

La règle de conséquence n'est attachée à aucune instruction particulière, mais permet de faire des raisonnements supplémentaires.

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}I\{P_3\} \quad P_3 \Rightarrow P_4}{\{P_1\}I\{P_4\}}$$

La règle de conséquence introduit un nouveau type de jugements simplement constitués de formules logiques, dont il faut vérifier la validité. Nous ne nous préoccuperons pas des règles qui doivent être ajoutées au système pour effectuer ces raisonnements là.

Exercices

5. *Ecrire la dérivation complète correspondant à l'exemple donné en première section de cette leçon.*

2 Le langage d'assertions

Dans cette section, nous revenons sur le langage utilisé dans les assertions et sur la définition de l'opération de substitution.

Nous reprenons le même langage des expressions arithmétiques, mais nous y ajoutons des opérations: la soustraction et le produit. Les formules logiques sont également conçues pour être un sur-ensemble des expressions booléennes, mais nous y ajoutons beaucoup plus d'éléments. La description formelle des langages pourrait être reprise sous forme longue comme dans la leçon 1, mais nous nous contenterons de la forme courte, où exp désigne les expressions arithmétiques et $bexp$ désigne les expression logiques:

$$\begin{array}{rcl} exp & = & n \mid v \mid exp + exp \mid exp - exp \mid exp \times exp \\ bexp & = & true \mid false \mid exp = exp \mid exp > exp \mid bexp \wedge bexp \mid bexp \vee bexp \mid \\ & & \neg exp \mid bexp \Rightarrow bexp \mid \forall v.bexp \mid \exists v.bexp \end{array}$$

2.1 Variables liées

Définition 1 *Une occurrence d'une variable v est* liée *si elle apparait à l'intérieur d'une expression quantifiée universellement par un $\forall v \ldots$ ou à l'intérieur d'une expression quantifiée existentiellement par un $\exists v \ldots$ Une variable qui n'est pas liée est dite* libre. *Si E est une expression arbitraire on note $FV(E)$ l'ensemble des variables libres qu'elle contient.*

Dans une expression de la forme $\exists i.k = i \times l$ on comprend que k et l représentent des entiers particuliers, tandis que i est un nom que l'on peut changer à volonté, si l'on ne choisit pas une variable présente à l'intérieur de l'expression quantifiée et si l'on change toutes les occurrences liées en même temps (comme l'argument

d'une fonction ou d'une procédure en programmation). Ainsi, l'expression $\forall x.x = x$ est vraie l'est encore si l'on change x en y pour obtenir $\forall y.y = y$, mais ne l'est plus si l'on oublie de remplacer l'une des instances de x pour obtenir $\forall y.y = x$. De même si l'on observe l'expression

$$\forall y.\exists x.x = y + 1,$$

elle est vraie si l'on remplace x par z pour obtenir

$$\forall y.\exists z.z = y + 1,$$

Mais elle ne l'est plus si l'on remplace x par y pour obtenir

$$\forall y.\exists y.y = y + 1$$

Exercices

6. *Quelles sont les variables libres (resp. liées) dans l'expression*

$$i = y + 3 \wedge (\exists i.y = 4 \times i)$$

7. *Renommer la variable liée correspondant à la quantification universelle de x en t dans l'expression suivante:*

$$\forall x.\forall z.\exists y.\ (y > x \wedge \exists x.y = z \times x)$$

2.2 Substitution

La substitution sur les expressions arithmétiques est définie par une fonction récursive structurelle[1] de la façon suivante

$$\begin{array}{rcl} n[v\backslash E] & = & n \\ v[v\backslash E] & = & E \\ v'[v\backslash E] & = & v' \quad \text{si } v' \neq v \\ (e_1 + e_2)[v\backslash E] & = & e_1[v\backslash E] + e_2[v\backslash E] \\ (e_1 - e_2)[v\backslash E] & = & e_1[v\backslash E] - e_2[v\backslash E] \\ (e_1 \times e_2)[v\backslash E] & = & e_1[v\backslash E] \times e_2[v\backslash E] \end{array}$$

La substitution sur les formules logiques est définie de façon similaire, mais il faut faire attention aux variables liées dans les quantificateurs $\forall$ et $\exists$.

$$true[v\backslash E] \quad = \quad true$$

[1] Je rappelle que j'appelle ainsi les fonctions récursives qui ne se rappellent que sur des sous-termes de l'argument initial. La terminaison de ces fonctions est assurée parce que la taille de l'argument décroit à chaque appel récursif.

$$
\begin{array}{rcl}
false[v\backslash E] & = & false \\
(e_1 = e_2)[v\backslash E] & = & e_1[v\backslash E] = e_2[v\backslash E] \\
\ldots & & \\
(\forall v.bexp)[v\backslash E] & = & \forall v.bexp \\
(\forall v'.bexp)[v\backslash E] & = & \forall v'.(bexp[v\backslash E]) \quad \text{si } v' \neq v \text{ et } v' \notin FV(E) \\
(\forall v'.bexp)[v\backslash E] & = & \forall v''.(bexp[v'\backslash v''][v\backslash E]) \quad \text{si } v' \neq v \text{ et } v' \in FV(E) \\
 & & \qquad\qquad \text{et } v'' \notin FV(E) \cup FV(bexp) \\
(\exists v.bexp)[v\backslash E] & = & \exists v.bexp \\
(\exists v'.bexp)[v\backslash E] & = & \exists v'.(bexp[v\backslash E]) \quad \text{si } v' \neq v \text{ et } v' \notin FV(E) \\
(\exists v'.bexp)[v\backslash E] & = & \exists v''.(bexp[v'\backslash v''][v\backslash E]) \quad \text{si } v' \neq v \text{ et } v' \in FV(E) \\
 & & \qquad\qquad \text{et } v'' \notin FV(E) \cup FV(bexp)
\end{array}
$$

3 Etats mémoire et assertions

Nous rappelons que l'état mémoire $\perp$ correspond à la valeur retournée par la dénotation des instructions lorsque l'exécution de ces instructions ne termine pas. Ici nous nous intéressons à la correction partielle des programmes, c'est à dire au conditions vérifiées par les états mémoire retournés lorsque les programmes terminent. On admet alors que l'état mémoire indéfini satisfait toutes les assertions.

Définition 2 *Nous dirons qu'un état mémoire σ satisfait une assertion A et nous noterons $\sigma \models A$ si l'assertion est vraie lorsque l'état mémoire est défini pour les variables libres apparaissant dans l'assertion et que les variables libres de l'assertion sont remplacées par leurs valeurs suivant l'état mémoire. Par induction structurelle sur les équation cette définition est également donnée par les égalités suivantes:*

$$
\begin{array}{rcll}
\sigma & \models & true & \\
\sigma & \models & (a_0 = a_1) & si\ \mathcal{A}[\![a_0]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma) \\
\sigma & \models & (a_0 > a_1) & si\ \mathcal{A}[\![a_0]\!](\sigma) > \mathcal{A}[\![a_1]\!](\sigma) \\
\sigma & \models & A \wedge B & si\ \sigma \models A\ et\ \sigma \models B \\
\sigma & \models & A \vee B & si\ \sigma \models A\ ou\ \sigma \models B \\
\sigma & \models & \neg A & s'il\ n'est\ pas\ vrai\ que\ \sigma \models A \\
\sigma & \models & A \Rightarrow B & si\ l'on\ a\ pas\ \sigma \models A\ ou\ si\ \sigma \models B \\
\sigma & \models & \forall v.A & si\ \sigma[v \leftarrow n] \models A\ pour\ tout\ n \\
\sigma & \models & \exists v.A & si\ \sigma[v \leftarrow n] \models A\ pour\ une\ valeur\ de\ n \\
\perp & \models & A &
\end{array}
$$

4 Validité de la sémantique axiomatique

Pour les règles de sémantique axiomatique, on considère deux propriétés générales:

1. Validité Si les hypothèses d'une règle sont valides alors la conclusion devrait l'être. Si cette propriété est assurée pour toutes les règles, alors tout énoncé obtenu par composition de ces règles devrait être une assertion valide.

2. Complétude On voudrait également que le système de preuve décrit par cette collection de règles soit également assez fort pour obtenir toutes les assertions vraies.

Le système de règles de la logique de Hoare est effectivement valide, ce qui s'exprime de la façon suivante:

Théorème 1 *si* $\{A\}I\{B\}$ *est prouvable par les règles de la logique de Hoare, alors pour tout état mémoire* σ*, si* $\sigma \models A$*, alors* $\mathcal{C}[\![I]\!](\sigma) \models B$*.*

Nous ne démontrerons pas ce théorème. Les étudiants intéressés pourront se reporter, par exemple à Winskel, *The Formal Semantics of Programming Languages, MIT Press*, pp.91–92.

5 Calcul de conditions de vérifications

Il n'y a pas de théorème montrant la complétude de la logique de Hoare, car ce théorème irait à l'encontre du théorème de complétude de Gödel. Parmi les conséquences de ce résultat négatif, il est impossible de fabriquer un outil qui prendrait des programmes annotés, terminerait toujours, et retournerait une réponse booléenne indiquant si les annotations sur ce programme sont valides. C'est dommage car cela permettrait d'écrire des programmes annotés avec des formules logiques indiquant ce que l'on attend de ces programmes et faire tourner l'outil pour vérifier que le programme est correct: plus de "déboggage" nécessaire.

Dans cette section, nous allons quand même décrire un outil de vérification de programmes annotés qui ramène la vérification à la vérification de la validité d'une collection de formule logiques.

5.1 Langage de programmes annotés

Nous considérons des programmes où l'on peut ajouter des annotations arbitrairement devant les instructions et où l'on dispose d'une annotation de fin, qui indique les propriétés attendues pour les variables à la fin de l'exécution (lorsque celle-ci est atteignable). Pour les boucles, nous supposons qu'une annotation est nécessairement fournie pour décrire l'invariant de la boucle. Nous construisons deux générateurs. Le premier retourne la pré-condition minimale pour un programme et une post-condition.

$$\begin{aligned} pre(\{A\}I, B) &= A \\ pre(v := a, B) &= B[v\backslash a] \end{aligned}$$

$$
\begin{aligned}
pre(I_0; I_1, B &= pre(I_0, pre(I_1, B)) \\
pre(\texttt{while } b \texttt{ do } \{D\}I, B) &= \{D\}
\end{aligned}
$$

La valeur retournée est la pré-condition la plus faible qui correspond aux indications données par le programmeur. S'il existe une assertion au début du programme, alors c'est cette assertion la pré-condition la plus faible. S'il n'existe pas d'annotations, il faut descendre jusqu'à la première assertion et remonter l'information jusqu'à début du programme, ce qui est simple car on n'a que des affectations à traverser ou des séquences d'affectations à traverser (les autres instructions portent une pré-condition).

Avoir calculé la précondition la plus faible n'apporte qu'une information très faible sur la cohérence du programme observé. Pour étudier cette cohérence, il faut vérifier que les différentes assertions et les instructions forment un tout cohérent. Pour

Le deuxième générateur construit l'ensemble des formules logiques qui doivent être vérifiées pour qu'un programme annoté soit valide vis-à-vis d'une condition sur les valeurs des variables en sortie. Il est donné par les équations suivantes:

$$
\begin{aligned}
vc(\{A\}I, B) &= \{A \Rightarrow pre(I, B)\} \cup vc(I, B) \\
vc(v := a, B) &= \emptyset \\
vc(\{I_0; I_1, B) &= vc(I_0, pre(I_1, B)) \cup vc(I_1, B) \\
vc(\texttt{while } b \texttt{ do } \{D\}I\{B\}) &= vc(\{D \wedge b\}I, D) \cup \{(D \wedge \neg b) \Rightarrow B\}
\end{aligned}
$$

Il existe un résultat de complétude partielle qui montre l'importance des conditions de vérifications calculées par la fonction vc, exprimé par le théorème suivant:

Théorème 2 *Pour toute instruction annotée I et toute post-condition $\{B\}$, si l'ensemble de conditions $vc(I, B)$ est prouvable, alors l'assertion $\{pre(I, B)\}I\{B\}$ est prouvable.*

Il faut bien garder en mémoire qu'un programme cohérent avec post-condition termine dans un état qui satisfait la post-condition lorsque l'on donne en entrée un programme qui satisfait sa pré-condition: pour vérifier que ce programme est vraiment satisfaisant il faut aussi vérifier que la pré-condition est acceptable.

Prenons par exemple l'instruction annotée $x := 1$ et la post-condition $x = 3$. L'ensemble des conditions de vérifications est l'ensemble vide: toutes les conditions sont donc satisfaites! En fait l'incohérence ne se trouve pas dans les conditions de verifications générées mais dans la pré-condition: $1 = 3$. Cette pré-condition est évidemment fausse et l'on ne saura jamais fournir un état qui la satisfait et l'expression $\{1 = 3\}x := 1\{x = 3\}$ est donc bien prouvable.

5.2 Ajouter des instructions conditionnelles

Si l'on ajoute au langage des instruction conditionnelles $if - then - else$, elle doivent également porter une annotation décrivant les conditions attendues sur

les valeurs de variables en entrées, on étend alors les fonctions *pre* et *vc* de la façon suivante:

$$\begin{aligned} pre(\{A\}\texttt{if } b \texttt{ then } I_1 \texttt{ else } I_2\{B\} &= A \\ vc(\{A\}\texttt{if } b \texttt{ then } I_1 \texttt{ else } I_2\{B\} &= \{vc(\{A \wedge b\}I_1\{B\}, vc(\{A \wedge \neg b\}I_2\{B\})\} \end{aligned}$$

Exercices

8. *Le programme suivant décrit un algorithme de division par soustractions successives. Quelles sont les conditions de vérification calculée par vc pour ce programme, vérifier que les formules logiques engendrées sont cohérentes.*

```
{x+1 > 0 /\ x=a}
q:= 0;
while x+1 > y do
{a=q * y + r /\ x+1 > 0}
  x := x - y;
  q := q + 1;
done
{a= q * y + r /\ x + > 0 /\ y > x}
```

6 Assurer la correction totale

Le programme donné dans l'exercice 8 est cohérent. Pourtant, si y est négatif ce programme ne termine pas. En ce sens, le programme est faux. La technique que nous avons développé jusqu'ici ne décrit que les propriétés assurées lorsque l'exécution termine. Si l'exécution ne termine pas, alors rien n'est assuré!

La seule nécessité est d'exprimer que les boucles s'arrêteront, pour cela, il faut également annoter ces boucles avec une expression arithmétique pour laquelle nous exprimerons que cette expression doit décroitre strictement, tout en restant un nombre entier positif.

$$\begin{aligned} vc(\texttt{while } b \texttt{ do}\{D\} \texttt{ variant } e \quad I \quad \texttt{done}, B) & \\ = \quad & vc(\{D \wedge b\}I, D) \cup \\ & \{D \wedge \neg b \Rightarrow B, D \wedge b \Rightarrow e + 1 > 0\} \cup \\ & vc(\{D \wedge b \wedge e = a\}I, a > e) \\ & \text{où } a \text{ est une nouvelle variable} \end{aligned}$$