Introduction à la sémantique axiomatique

Yves Bertot

Janvier 2009

1 Prouver des propriétés de programmes

La sémantique naturelle, comme nous l'avons décrite dans un cours précédent, permet de décrire précisément comment fonctionne les programmes d'un langage. En principe, on peut utiliser cette description très précise pour démontrer que le résultat d'un calcul satisfait certaines propriétés. En pratique, cela ne marche pas bien, car le travail de preuve croule sous les détails.

Il est possible de décrire le comportement des programmes d'une manière qui facilite la démonstration de propriétés pour les résultats. Ceci repose sur un nouveau style de description sémantique, connu sous le nom de *logique de Hoare* ou *sémantique axiomatique*.

1.1 Un exemple

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

```
S := 0;
N := 1;
while N < 101 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.

De toutes façon cette méthode n'est pas satisfaisante si nous voulons remplacer 101 par un nombre p arbitraire, ou par une variable `P` dont la valeur n'est pas connue à l'avance.

Essayons de raisonner un peu sur notre programme. Après les deux premières affectations nous savons que la valeur avant d'entrer dans la boucle 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 N < 101 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 `N < 101` 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 N < 101 do
{N < 101 }
  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:

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

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

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

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).

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

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 N < 101 do
invariant { S = N(N − 1)/2 ∧ N ≤ 101}
{N < 101 }
  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. En fait, nous nous reposerons sur un système de preuve comme Coq pour prendre ces formules en charge.

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. Les formules logiques sont également conçues pour être un sur-ensemble des expressions booléennes suffisant pour exprimer toutes les opérations logiques effectuées dans les règles de sémantique axiomatique. Une porte ouverte pour exprimer des propriétés logiques arbitraires est fournie par la possibilité de faire référence au nom de prédicats arbitraires. Les assertions seront dénotées par des variables a, a', a_n et seront décrites par la syntaxe suivante:

$$assert \quad ::= \quad b|a_1 \wedge a_2|s(e_1, \ldots, e_n)|\neg a|true|false$$

Une assertion de la forme $s(e_1, \ldots, e_n)$ est appelée un *prédicat*. Le sens de cette formule établi par convention. En particulier, lors de l'utilisation de ces règles dans un système de preuve, on maintiendra une table de correspondance entre les noms de prédicats et des prédicats sur des listes d'entiers définis directement dans le système de preuve.

2.1 Substitution

La substitution sur les expressions arithmétiques est définie par une fonction récursive structurelle:

$$\begin{aligned} n[v\backslash e] &= n \\ v[v\backslash e] &= e \\ v'[v\backslash e] &= v' \quad \text{si } v' \neq v \\ (e_1 + e_2)[v\backslash e] &= e_1[v\backslash e] + e_2[v\backslash e] \end{aligned}$$

Sur les expressions booléennes c'est similaire:

$$(e_1 < e_2)[v\backslash e] \quad = \quad e_1[v\backslash e] < e_2[v\backslash e]$$

et encore pour les assertions:

$$\begin{array}{rcl} true[v\backslash E] & = & true \\ false[v\backslash E] & = & false \\ (a_1 \wedge a_2)[v\backslash e] & = & a_1[v\backslash e] \wedge a_2[v\backslash e] \\ (\neg a)[v\backslash e] & - & \neg(a[v\backslash e]) \end{array}$$

3 Validité de la sémantique axiomatique

La sémantique naturelle et la sémantique axiomatique sont deux approches pour décrire précisément le comportement des programmes. Il est important de vérifier si ces deux approches sont cohérentes. Pour exprimer cette cohérence, nous devons relier les environnements tels que nous les manipulons dans la sémantique naturelle et les assertions telles que nous les manipulons en sémantique axiomatique. Pour ceci, nous introduisons cinq nouvelles notions: les *valuations*, l'*interprétation* d'une formule logique dans une valuation, la *validité d'une formule logique*, la *validité d'une dérivation de sémantique axiomatique* et l'*adaptation* d'une valuation à un environnement.

Définition 1 *Une* valuation *est une fonction de l'ensemble des variables vers l'ensemble des valeurs entières.*

Les valuations jouent à peu près le même rôle que les environnements, sauf que les environnements n'associent une valeur entière qu'à un nombre fini de variables. Avec des environnements, il peut arriver qu'une expression arithmétique n'ait pas de valeur associée. En revanche, une valuation permet toujours d'associer une valeur à une expression arithmétique, ce qui est décrit précisément par la notion d'interprétation.

Définition 2 *L'interprétation $\mathcal{I}^a_g(e)$ d'une expression arithmétique e dans une valuation g correspond au calcul de la valeur de cette expression arithmétique en utilisant la valuation pour chacune des variables apparaissant dans l'expression. Ce calcul est donné par les équations suivantes:*

$$\begin{array}{rcl} \mathcal{I}^a_g(x) & = & g(x) \\ \mathcal{I}^a_g(n) & = & n \\ \mathcal{I}^a_g(e_1 + e_2) & = & \mathcal{I}^a_g(e_1) + \mathcal{I}^a_g(e_2) \end{array}$$

On définit de la même manière l'interprétation des expressions booléennes et des assertions notées $\mathcal{I}^b$ et $\mathcal{I}$.

$$\mathcal{I}^b_g(e_1 < e_2) \quad = \quad \texttt{true} \textit{ si } \mathcal{I}^a_g(e_1) < \mathcal{I}^a_g(e_2)$$

$$
\begin{aligned}
\mathcal{I}^b_g(e_1 < e_2) &= \texttt{false}\ si\ \mathcal{I}^a_g(e_2) \leq \mathcal{I}^a_g(e_1) \\
\mathcal{I}(\neg a) &= \neg\mathcal{I}(a) \\
\mathcal{I}(a_1 \wedge a_2) &= \neg\mathcal{I}(a_1) \wedge \mathcal{I}(a_2) \\
\mathcal{I}(\texttt{true}) &= true \\
\mathcal{I}(\texttt{false}) &= false \\
\mathcal{I}(a_1 \Rightarrow a_2) &= \mathcal{I}(a_1) \Rightarrow \mathcal{I}(a_2)
\end{aligned}
$$

Ainsi, une assertion peut être vraie pour un certain choix des valeurs des variables et fausse pour un autre choix. Pour les implications qui apparaissent dans les dérivations de sémantique axiomatique, il est nécessaire de vérifier si ces implications sont universellement vraies. C'est ce que nous décrivons avec la notion de validité.

Définition 3 *Une assertion est* valide *si et seulement si son interprétation dans toute valuation est valide.*

Nous pouvons maintenant généraliser cette notion de validité aux dérivations de sémantique axiomatique.

Définition 4 *Une dérivation de sémantique axiomatique est* valide *si toutes les implications qu'elle contient sont valides.*

Les valuations et les environnements sont des objets de nature différentes, toutefois il est possible d'adapter une valuation g à un environnement ρ pour obtenir une nouvelle valuation qui associe à tout variable x la valeur qui lui est associée par l'environnement lorsqu'elle existe ou la valeur $g(x)$ par défaut. Ceci est exprimé dans la définition suivante:

Définition 5 *Nous appellerons* adaptation de g à ρ *la fonction notée* ρ^g *définie par les équations suivantes:*

$$
\begin{aligned}
((x, v) \cdot \rho')^g(x) &= v \\
((y, v) \cdot \rho')^g(x) &= \rho'^g(x)\ si\ x \neq y \\
\emptyset^g(x) &= g(x)
\end{aligned}
$$

En ayant précisé toutes ces notions, nous pouvons énoncer un résultat sur la correction de notre sémantique axiomatique: *Pour toute paire d'assertions P et Q, pour toute instruction i, et pour toute paire d'environnements ρ_1 et ρ_2, s'il existe une dérivation valide pour le triplet $\{P\}\ i\ \{Q\}$ et s'il existe une dérivation pour le jugement $\rho_1 \vdash i \rightsquigarrow \rho_2$, alors pour toute valuation g, si l'interprétation de P dans l'adaptation de g à ρ_1 est satisfaite alors l'interprétation Q dans l'adaptation de g à ρ_2 l'est également.*

En notation mathématique ceci s'écrit de la façon suivante:

$$\forall P, Q, i, \rho_1, \rho_2 g, \{P\}\ i\ \{Q\} \Rightarrow \rho_1 \vdash i \rightsquigarrow \rho_2 \Rightarrow \mathcal{I}_{\rho_1{}^g}(P) \Rightarrow \mathcal{I}_{\rho_2{}^g}(Q)$$

Ce n'est pas l'objet de ce cours, mais on peut démontrer formellement cet énoncé, par exemple à l'aide des systèmes Coq et Isabelle.

4 Calcul de conditions de vérifications

Fabriquer les dérivations de sémantique axiomatique est un travail long et fastidieux. Nous allons maintenant décrire une fonction qui simule la fabrication d'une telle dérivation et collecte les implications qui apparaissent dans la dérivation. En pratique, il est au moins nécessaire de connaître la post condition (l'assertion Q dans un triplet de Hoare de la forme $\{P\}\ i\ \{Q\}$ et les assertions qui seront utilisées comme invariant pour chaque boucle `while`. Nous allons modifier le langage de programmation pour ajouter ces annotations, et quelques autres.

4.1 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, une annotation est nécessairement fournie pour décrire l'invariant de la boucle. Nous construisons deux fonctions. La première retourne la pré-condition minimale pour un programme et une post-condition.

$$\begin{aligned}
\texttt{pre}(\{A\}I, B) &= A \\
\texttt{pre}(v := a, B) &= B[v\backslash a] \\
\texttt{pre}(I_0; I_1, B &= \texttt{pre}(I_0, \texttt{pre}(I_1, B)) \\
\texttt{pre}(\texttt{while}\ b\ \texttt{do}\ \{D\}I, B) &= \{D\}
\end{aligned}$$

La valeur retournée est la formule logique la moins contraignante pour assurer que les annotations dans le programme et la post-condition fournie en argument soient satisfaites. Cette formule est appelée la *pré-condition la plus faible*. 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 vis-à-vis de la post-condition. Pour assurer cette cohérence, il faut vérifier que les relations entre les différentes assertions, les instructionset la post-condition. La deuxième fonction calcule une collection de formules logique dont il faudra vérifier la validité.

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:

$$\begin{aligned}
\texttt{vc}(\{A\}I, B) &= \{A \Rightarrow \texttt{pre}(I, B)\} \cup \texttt{vc}(I, B) \\
\texttt{vc}(v := a, B) &= \emptyset
\end{aligned}$$

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

On peut démontrer que l'ensemble des conditions calculées par la fonction `vc` suffit à construire une dérivation de sémantique axiomatique.

Théorème 1 *Pour toute instruction annotée i et toute post-condition $\{Q\}$, si l'ensemble de conditions $\texttt{vc}(I, Q)$ est prouvable, alors l'assertion $\{\texttt{pre}(i, Q)\}\ i\ \{Q\}$ 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.

4.2 Ajouter des instructions conditionnelles

Si l'on ajoute au langage des instructions 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:

$$\begin{aligned} \texttt{pre}(\{A\}\texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\{B\}) &= A \\ \texttt{vc}(\{A\}\texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\{B\}) &= \texttt{vc}(\{A \wedge b\}i_1\{B\}) \cup \texttt{vc}(\{A \wedge \neg b\}i_2\{B\}) \end{aligned}$$

Une approche alternative est de ne pas imposer l'utilisation d'une annotation pour les instructions conditionnelles et de modifier la fonction de pré-condition pour que la précondition soit:

$$\begin{aligned} \texttt{pre}(\texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\{B\}) &= (b \Rightarrow \texttt{pre}(i_1\{B\})) \wedge (\neg b \Rightarrow \texttt{pre}(i_2\{B\})) \\ \texttt{vc}(\texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\{B\}) &= \texttt{vc}(i_1\{B\}) \cup \texttt{vc}(i_2\{B\}) \end{aligned}$$

Exercices

6. *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}
```

5 Assurer la correction totale

Le programme donné dans l'exercice 6 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.

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

6 Démontrer la correction du générateur de conditions

On peut démontrer que la fonction `vc` est correcte, en exprimant que si toutes les conditions engendrées par cette fonction sont logiquement valides, alors il existe une dérivation de logique de Hoare pour le triplet dont la précondition est celle calculée par `pre`, l'instruction celle donnée en argument à `vc` où l'on a oublié les annotations, et la post-condition est celle donnée en argument à `vc`.

Pour exprimer formellement le théorème de correction, nous utilisons la fonction $\langle\cdot\rangle$ définie de la façon suivante:

$$
\begin{array}{rcl}
\langle\{A\}I\rangle & = & \langle I\rangle \\
\langle v := a\rangle & = & v := a \\
\langle I_0; I_1\rangle & = & \langle I_0\rangle; \langle I_1\rangle \\
\langle\texttt{while}\ b\ \texttt{do}\ \{D\}I\ \texttt{done}\rangle & = & \texttt{while}\ b\ \texttt{do}\ \langle I\rangle\ \texttt{done}
\end{array}
$$

L'énoncé formel de correction a donc la forme suivante:

Théorème 2 *Pour toute instruction annotée I et toute post-condition Q, sous réserve de la validation des conditions* $\mathtt{vc}(I,Q)$*, il existe une dérivation de logique de Hoare pour le triplet* $\{\mathtt{pre}(I,Q)\}\langle I\rangle\{Q\}$.

Démonstration. Cette démonstration se fait par récurrence sur la taille de l'instruction I.

- Si `I` est une affectation $x := e$, on a $\mathtt{pre}(I,Q) = Q[x\backslash e]$, donc le triplet a prouver est:
$$\{Q[x\backslash e]\}x := e\{Q\}$$
Ce triplet est prouvé par la règle d'affectation de la sémantique axiomatique.

- Si `I` est une séquence $i1; i2$, alors $\mathtt{pre}(I,Q) = \mathtt{pre}(i1, pre(i2,Q))$ et
$$\mathtt{vc}(i_1; i_2, Q) = \mathtt{vc}(i_1, \mathtt{pre}(i_2, Q)) \cup \mathtt{vc}(i_2, Q).$$
Par hypothèse de récurrence sur les instructions plus petites i_1 et i_2, il existe des dérivations δ_1 et δ_2 pour les triplets $\{\mathtt{pre}(i_1, \mathtt{pre}(i_2,Q))\}\langle i_1\rangle\{\mathtt{pre}(i_2,Q)\}$ et $\{\mathtt{pre}(i_2,Q)\}\langle i_2\rangle\{Q\}$, nous pouvons alors utiliser la règle de sémantique axiomatique pour la séquence pour construire la dérivation suivante:
$$\frac{\overset{\delta_1}{\{\mathtt{pre}(i_1,\mathtt{pre}(i_2,Q))\}\langle i_1\rangle\{\mathtt{pre}(i_2,Q)\}} \quad \overset{\delta_2}{\{\mathtt{pre}(i_2,Q)\}\langle i_2\rangle\{Q\}}}{\{\mathtt{pre}(i_1;i_2,Q)\}\langle i_1;i_2\rangle\{Q\}}$$

- si `I` est une boucle `while` b `do` $\{\mathrm{A}\}i$ `done`, alors
$$\mathtt{vc}(I,Q) = \{\neg b \wedge A \Rightarrow Q, b \wedge A \Rightarrow \mathtt{pre}(i,A)\} \cup \mathtt{vc}(I,A).$$
alors l'hypothèse de récurrence permet d'assurer que le triplet $\{b\wedge A\}\langle i\{A\}$ a une dérivation δ, nous pouvons utiliser deux fois la règle de conséquence et une fois la règle du `while` pour construire la dérivation suivante:
$$\frac{A \Rightarrow A \qquad \dfrac{\dfrac{b \wedge A \Rightarrow pre(i,A) \qquad \overset{\delta}{\{pre(i,A)\}\langle i\rangle\{A\}} \qquad A \Rightarrow A}{\{b\wedge A\}\langle i\rangle\{A\}}}{\{A\}\langle I\rangle\{\neg b \wedge A\}} \qquad \neg B \Rightarrow Q}{\{A\}\langle I\rangle\{Q\}}$$

- Si l'instruction I est une instruction de la forme $\{A\}i$, alors on peut également construire une dérivation à l'aide de la règle de conséquence.