Logiques de programmes, deuxième cours

Variables et boucles : la logique de Hoare

Xavier Leroy

2021-03-11

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Les bases de la logique de Hoare

Les triplets de Hoare

Les triplets «faibles» :

Signification intuitive :

« Si la commande c, démarrée dans un état initial satisfaisant P, termine, alors l'état final satisfait Q. »

On verra aussi les triplets «forts» $[P] \ c \ [Q]$ qui garantissent la terminaison : « la commande c, démarrée dans un état initial satisfaisant P, termine toujours, et l'état final satisfait Q.»

IMP : un petit langage impératif à contrôle structuré

Expressions arithmétiques :

$a ::= \mathtt{x}$	variable du programme
$\mid 0 \mid 1 \mid \dots$	constantes
$\mid a_1 + a_2 \mid a_1 \times a_2 \mid \dots$	opérations

Expressions booléennes :

$b ::= a_1 \leq a_2 \mid \dots$	comparaisons
$\mid b_1 \ \mathtt{and}\ b_2 \mid \mathtt{not}\ b \mid \dots$	connecteurs

Commandes :

$c ::= \mathtt{skip}$	commande vide
$\mid \mathtt{x} := a$	affectation
$\mid c_1; c_2$	séquence
$\mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$	conditionnelle
$\mid \mathtt{while}\ b\ \mathtt{do}\ c$	boucle

Les règles de la logique de Hoare «faible» pour IMP

Une règle par construction du langage de commandes.

$$\{ P \}\ \texttt{skip}\ \{ P \} \qquad\qquad \{ Q[x \leftarrow a] \}\ x := a\ \{ Q \}$$

$$\frac{\{ P \}\ c_1\ \{ Q \} \qquad \{ Q \}\ c_2\ \{ R \}}{\{ P \}\ c_1; c_2\ \{ R \}}$$

$$\frac{\{ P \land b \}\ c_1\ \{ Q \} \qquad \{ P \land \neg b \}\ c_2\ \{ Q \}}{\{ P \}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{ Q \}}$$

$$\frac{\{ P \land b \}\ c\ \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ c\ \{ P \land \neg b \}}$$

Les règles génériques

La règle de conséquence :

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Peut aussi s'écrire avec deux règles : une qui renforce la précondition, l'autre qui affaiblit la postcondition.

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q \,\}}{\{\, P \,\}\ c\ \{\, Q \,\}} \qquad\qquad \frac{\{\, P \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Note : la règle du haut est dérivable des deux règles du bas, et réciproquement.

Un exemple de dérivation

$$\dfrac{\top \Rightarrow 0 = 0 \land 1 = 1 \qquad \dfrac{\{\, 0 = 0 \land 1 = 1 \,\}\ \mathrm{x} := 0\ \{\, \mathrm{x} = 0 \land 1 = 1 \,\} \quad \{\, \mathrm{x} = 0 \land 1 = 1 \,\}\ \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}}{\{\, 0 = 0 \land 1 = 1 \,\}\ \mathrm{x} := 0; \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}}}{\{\, \top \,\}\ \mathrm{x} := 0; \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}}$$

Un exemple de dérivation

Une notation plus compacte, sous forme de programme IMP annoté par des assertions :

$$
\begin{array}{ll}
 & \{\top\} \Rightarrow \\
 & \{0 = 0 \land 1 = 1\} \\
x := 0; & \\
 & \{x = 0 \land 1 = 1\} \\
y := 1 & \\
 & \{x = 0 \land y = 1\}
\end{array}
$$

Vérification d'un «vrai» programme : la division euclidienne

$$\{0 \leq a\} \Rightarrow \{a = b \cdot 0 + a \ \wedge \ 0 \leq a\}$$

r := a;

$$\{a = b \cdot 0 + r \ \wedge \ 0 \leq r\}$$

q := 0;

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

while $r \geq b$ do

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r \geq b\} \Rightarrow$$

$$\{a = b \cdot (q+1) + (r - b) \ \wedge \ 0 \leq r - b\}$$

    r := r − b;

$$\{a = b \cdot (q+1) + r \ \wedge \ 0 \leq r\}$$

    q := q + 1

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

done

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r < b\} \Rightarrow$$

$$\{q = a/b \ \wedge \ r = a \bmod b\}$$

Travaux dirigés : le calcul du maximum (Mike Gordon)

1. Écrire un programme IMP qui met dans $\mathtt{x}$ le maximum des valeurs de $\mathtt{x}$ et de $\mathtt{y}$.

2. Spécifier ce programme en logique de Hoare.

3. Vérifier le programme vis-à-vis de cette spécification.

1. Écrire un programme IMP qui met dans $\mathtt{x}$ le maximum des valeurs de $\mathtt{x}$ et de $\mathtt{y}$.

$$\mathtt{if}\ \mathtt{x} < \mathtt{y}\ \mathtt{then}\ \mathtt{x} := \mathtt{y}\ \mathtt{else}\ \mathtt{skip}$$

2. Spécifier ce programme en logique de Hoare.

3. Vérifier le programme vis-à-vis de cette spécification.

1. Écrire un programme IMP qui met dans $\mathtt{x}$ le maximum des valeurs de $\mathtt{x}$ et de $\mathtt{y}$.

$$\mathtt{if}\ \mathtt{x} < \mathtt{y}\ \mathtt{then}\ \mathtt{x} := \mathtt{y}\ \mathtt{else}\ \mathtt{skip}$$

2. Spécifier ce programme en logique de Hoare.

$$\{\,\top\,\}\ \mathtt{if}\ \mathtt{x} < \mathtt{y}\ \mathtt{then}\ \mathtt{x} := \mathtt{y}\ \mathtt{else}\ \mathtt{skip}\ \{\,\mathtt{x} = \max(\mathtt{x}, \mathtt{y})\,\}$$

3. Vérifier le programme vis-à-vis de cette spécification.

1. Écrire un programme IMP qui met dans $\mathtt{x}$ le maximum des valeurs de $\mathtt{x}$ et de $\mathtt{y}$.

$$\mathtt{if}\ \mathrm{x} < \mathrm{y}\ \mathtt{then}\ \mathrm{x} := \mathrm{y}\ \mathtt{else}\ \mathtt{skip}$$

2. Spécifier ce programme en logique de Hoare.

$$\{\, \top \,\}\ \mathtt{if}\ \mathrm{x} < \mathrm{y}\ \mathtt{then}\ \mathrm{x} := \mathrm{y}\ \mathtt{else}\ \mathtt{skip}\ \{\, \mathrm{x} = \mathsf{max}(\mathrm{x}, \mathrm{y}) \,\}$$

3. Vérifier le programme vis-à-vis de cette spécification.

$$\{\, \mathrm{x} < \mathrm{y} \wedge \top \,\} \Rightarrow \{\, \mathrm{y} = \mathsf{max}(\mathrm{y}, \mathrm{y}) \,\}\ \mathrm{x} := \mathrm{y}\ \{\, \mathrm{x} = \mathsf{max}(\mathrm{x}, \mathrm{y}) \,\}$$

$$\{\, \mathrm{x} \geq \mathrm{y} \wedge \top \,\} \Rightarrow \{\, \mathrm{x} = \mathsf{max}(\mathrm{x}, \mathrm{y}) \,\}\ \mathtt{skip}\ \{\, \mathrm{x} = \mathsf{max}(\mathrm{x}, \mathrm{y}) \,\}$$

On conclut par la règle pour les conditionnelles.

Est-ce la bonne spécification ?

Elle est satisfaite par de nombreux programmes !

$$
\begin{array}{lll}
\{\top\} & x := y & \{x = \max(x, y)\} \\
\{\top\} & y := x & \{x = \max(x, y)\} \\
\{\top\} & x := 1; y := 0 & \{x = \max(x, y)\}
\end{array}
$$

Est-ce la bonne spécification ?

Elle est satisfaite par de nombreux programmes !

$$\begin{array}{lll} \{\,\top\,\} & x := y & \{\,x = \max(x, y)\,\} \\ \{\,\top\,\} & y := x & \{\,x = \max(x, y)\,\} \\ \{\,\top\,\} & x := 1; y := 0 & \{\,x = \max(x, y)\,\} \end{array}$$

La réponse est non ! On voulait dire

> *La valeur de* x *à la fin du programme est le maximum des valeurs de* x *et de* y *au début du programme.*

Variables auxiliaires

Une solution est de spécifier en utilisant des variables mathématiques $\alpha, \beta, \ldots$, distinctes des variables du programme $\mathtt{x}, \mathtt{y}, \ldots$:

$$\{\, \mathtt{x} = \alpha \land \mathtt{y} = \beta \,\}\ \mathsf{c}\ \{\, \mathtt{x} = \mathsf{max}(\alpha, \beta) \,\}$$

Ces variables auxiliaires sont implicitement quantifiées universellement en tête du triplet :

$$\forall \alpha, \beta, \quad \{\, \mathtt{x} = \alpha \land \mathtt{y} = \beta \,\}\ \mathsf{c}\ \{\, \mathtt{x} = \mathsf{max}(\alpha, \beta) \,\}$$

Variables fantômes

Autre possibilité : spécifier en utilisant des variables du langage de programmation qui n'apparaissent pas dans le programme à spécifier :

$$\{\, \mathtt{x} = \mathtt{z} \,\}\; c\; \{\, \mathtt{x} = \mathsf{max}(\mathtt{z}, \mathtt{y}) \,\} \qquad \text{avec } \mathtt{z} \text{ non libre dans } c$$

Ces variables fantômes $\mathtt{z}$ gardent leur valeur pendant l'exécution de c et permettent donc de parler de l'état ≪avant≫ dans la postcondition.

La logique ≪forte≫ (correction totale)

Les triplets ≪forts≫ :

Signification intuitive :

≪ La commande c, démarrée dans un état initial satisfaisant P, termine dans un état final satisfaisant Q.≫

Les règles de la logique de Hoare ≪forte≫ pour IMP

Seules les boucles peuvent causer la non-terminaison
⇒ pour les autres constructions d'IMP, les règles ≪fortes≫ sont semblables aux règles ≪faibles≫.

$$[\,P\,]\ \texttt{skip}\ [\,P\,] \qquad\qquad [\,Q[x \leftarrow a]\,]\ x := a\ [\,Q\,]$$

$$\frac{[\,P\,]\ c_1\ [\,Q\,] \qquad [\,Q\,]\ c_2\ [\,R\,]}{[\,P\,]\ c_1; c_2\ [\,R\,]} \qquad \frac{[\,P \wedge b\,]\ c_1\ [\,Q\,] \qquad [\,P \wedge \neg b\,]\ c_2\ [\,Q\,]}{[\,P\,]\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ [\,Q\,]}$$

$$\frac{P \Rightarrow P' \qquad [\,P'\,]\ c\ [\,Q'\,] \qquad Q' \Rightarrow Q}{[\,P\,]\ c\ [\,Q\,]}$$

Vérifier la terminaison d'une boucle

La technique du variant : une expression a, à valeurs positives, qui décroît strictement à chaque tour de boucle.

$$\frac{\forall \alpha,\ [\,P \land b \land a = \alpha\,]\ c\ [\,P \land 0 \le a < \alpha\,]}{[\,P\,]\ \texttt{while}\ b\ \texttt{do}\ c\ [\,P \land \neg b\,]}$$

La boucle termine forcément, au bout d'au plus N itérations, où N est la valeur initiale du variant a.

Note : en cas de boucles imbriquées, on vérifie la terminaison de chaque boucle indépendamment des autres.
(Contrairement à Turing 1949 et Floyd 1967.)

Vérifier la terminaison de la division euclidienne

Le variant est la variable r.

$$\{0 \le a \wedge 0 < b\} \Rightarrow \{a = b \cdot 0 + a \wedge 0 \le a \wedge 0 < b\}$$

r := a;

$$\{a = b \cdot 0 + r \wedge 0 \le r \wedge 0 < b\}$$

q := 0;

$$\{a = b \cdot q + r \wedge 0 \le r \wedge 0 < b\}$$

while r $\geq$ b do

$$\{a = b \cdot q + r \wedge 0 \le r \wedge 0 < b \wedge r \ge b \wedge r = \alpha\} \Rightarrow$$

$$\{a = b \cdot (q + 1) + (r - b) \wedge 0 \le r - b \wedge 0 < b \wedge 0 \le r - b < \alpha\}$$

    r := r − b;

$$\{a = b \cdot (q + 1) + r \wedge 0 \le r \wedge 0 < b \wedge 0 \le r < \alpha\}$$

    q := q + 1

$$\{a = b \cdot q + r \wedge 0 \le r \wedge 0 < b \wedge 0 \le r < \alpha\}$$

done

$$\{a = b \cdot q + r \wedge 0 \le r \wedge r < b\} \Rightarrow$$

$$\{q = a/b \wedge r = a \bmod b\}$$

Généralisation à plusieurs variants

On peut avoir besoin de plusieurs variants $a_1, \ldots, a_n$ et d'un ordre bien fondé $\prec$ entre les n-uplets d'entiers.

(Par exemple, un ordre lexicographique.)

$$\frac{\forall \alpha_1, \ldots, \alpha_n, \quad \begin{array}{c} [\, P \wedge b \wedge (a_1, \ldots, a_n) = (\alpha_1, \ldots \alpha_n) \,] \\ c \\ [\, P \wedge (a_1, \ldots, a_n) \prec (\alpha_1, \ldots \alpha_n) \,] \end{array}}{[\, P \,] \; \texttt{while} \; b \; \texttt{do} \; c \; [\, P \wedge \neg b \,]}$$

Ajouter des règles à la logique

Une règle dérivée : la conditionnelle sans `else`

Notation : $\texttt{if}\ b\ \texttt{then}\ c \stackrel{def}{=} \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ \texttt{skip}$

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, Q \,\} \qquad P \wedge \neg b \Rightarrow Q}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c\ \{\, Q \,\}}$$

Démonstration.

Par la dérivation suivante :

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, Q \,\} \qquad \dfrac{P \wedge \neg b \Rightarrow Q \qquad \{\, Q \,\}\ \texttt{skip}\ \{\, Q \,\}}{\{\, P \wedge \neg b \,\}\ \texttt{skip}\ \{\, Q \,\}}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ \texttt{skip}\ \{\, Q \,\}}$$

□

Une règle dérivée : la boucle `do...while`

Notation : $\texttt{do}\ c\ \texttt{while}\ b \stackrel{def}{=} c; \texttt{while}\ b\ \texttt{do}\ c$

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad Q \wedge b \Rightarrow P}{\{\,P\,\}\ \texttt{do}\ c\ \texttt{while}\ b\ \{\,Q \wedge \neg b\,\}}$$

Démonstration.

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad \dfrac{\dfrac{Q \wedge b \Rightarrow P \qquad \{\,P\,\}\ c\ \{\,Q\,\}}{\{\,Q \wedge b\,\}\ c\ \{\,Q\,\}}}{\{\,Q\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\,Q \wedge \neg b\,\}}}{\{\,P\,\}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \{\,Q \wedge \neg b\,\}}$$

□

Une règle dérivée : la boucle comptée `for`

Notation : si h, i sont deux variables distinctes,

$$\texttt{for}\ i = \ell\ \texttt{to}\ h\ \texttt{do}\ c \quad \stackrel{def}{=} \quad i := \ell; \texttt{while}\ i \leq h\ \texttt{do}\ (c; i := i + 1)$$

On peut dériver un triplet fort qui garantit la terminaison de la boucle, pourvu que le corps c de la boucle ne contienne pas d'affectations ni à i ni à h.

$$\frac{[\,P \land i \leq h\,]\ c\ [\,P[i \leftarrow i + 1]\,] \qquad i, h \text{ non affectées dans } c}{[\,P[i \leftarrow \ell]\,]\ \texttt{for}\ i = \ell\ \texttt{to}\ h\ \texttt{do}\ c\ [\,P \land i > h\,]}$$

Le variant est l'expression $h - i + 1$, qui décroît de 1 à chaque tour.

Une règle dérivée : l'affectation à la manière de Floyd

$$\{ P \} \; \mathrm{x} := a \; \{ \exists x_0, \mathrm{x} = a[\mathrm{x} \leftarrow x_0] \wedge P[\mathrm{x} \leftarrow x_0] \}$$

Démonstration.

Notons $Q \overset{def}{=} \exists x_0, \mathrm{x} = a[\mathrm{x} \leftarrow x_0] \wedge P[\mathrm{x} \leftarrow x_0]$.

$$\frac{P \Rightarrow Q[\mathrm{x} \leftarrow a] \qquad \{ Q[\mathrm{x} \leftarrow a] \} \; \mathrm{x} := a \; \{ Q \}}{\{ P \} \; \mathrm{x} := a \; \{ Q \}}$$

En effet, $Q[\mathrm{x} \leftarrow a] = \exists x_0, a = a[\mathrm{x} \leftarrow x_0] \wedge P[\mathrm{x} \leftarrow x_0][\mathrm{x} \leftarrow a]$

$= \exists x_0, a = a[\mathrm{x} \leftarrow x_0] \wedge P[\mathrm{x} \leftarrow x_0]$

et il suffit de prendre $x_0 = \mathrm{x}$. □

Quelques règles admissibles

Conjonction, disjonction, quantification :

$$\frac{\{\, P_1 \,\}\; c\; \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\; c\; \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\; c\; \{\, Q_1 \wedge Q_2 \,\}} \qquad \frac{\{\, P_1 \,\}\; c\; \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\; c\; \{\, Q_2 \,\}}{\{\, P_1 \vee P_2 \,\}\; c\; \{\, Q_1 \vee Q_2 \,\}}$$

$$\frac{\forall x \in X,\ \{\, P(x) \,\}\; c\; \{\, Q(x) \,\} \quad X \neq \emptyset}{\{\, \forall x \in X.\ P(x) \,\}\; c\; \{\, \forall x \in X.\ Q(x) \,\}} \qquad \frac{\forall x \in X,\ \{\, P(x) \,\}\; c\; \{\, Q(x) \,\}}{\{\, \exists x \in X.\ P(x) \,\}\; c\; \{\, \exists x \in X.\ Q(x) \,\}}$$

Démonstration.

Récurrence sur c et inversion sur les dérivations de $\{\, P_1 \,\}\; c\; \{\, Q_1 \,\}$, $\{\, P_2 \,\}\; c\; \{\, Q_2 \,\}$, etc. □

Extensions du langage de programmation

Le contrôle non structuré

Goto considered harmful ... or not ?

Commandes : $c ::= \dots \mid \texttt{goto}\ \ell \mid \ell : c$

Associer un invariant $L(\ell)$ à chaque étiquette ℓ.

Les triplets deviennent $L \vdash \{ P \}\ c\ \{ Q \}$.

$$L \vdash \{ L(\ell) \}\ \texttt{goto}\ \ell\ \{ \bot \} \qquad \frac{L \vdash \{ L(\ell) \}\ c\ \{ Q \}}{L \vdash \{ L(\ell) \}\ \ell : c\ \{ Q \}}$$

Le non-déterminisme

Permettre aux programmes d'avoir plusieurs comportements différents.

$c ::= \ldots$	
$\mid c_1 \mathbin{[\!]} c_2$	exécute c_1 ou c_2
$\mid x := \mathtt{choose}(N)$	met un nombre entre 0 et $N - 1$ dans x
$\mid \mathtt{havoc}\ x$	met un nombre quelconque dans x

Les autres constructions se déduisent de `havoc` :

$$x := \mathtt{choose}(N) \quad \approx \quad \mathtt{havoc}\ x;\ x := x \bmod N$$

$$c_1 \mathbin{[\!]} c_2 \quad \approx \quad x := \mathtt{choose}(2); \mathtt{if}\ x = 0\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$$

$$x := \mathtt{choose}(N) \quad \approx \quad x := 0 \mathbin{[\!]} x = 1 \mathbin{[\!]} \cdots \mathbin{[\!]} x := N - 1$$

Les règles pour le non-déterminisme

La règle pour le choix :

$$\frac{\{\, P \,\}\ c_1\ \{\, Q \,\} \qquad \{\, P \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ c_1 \mathbin{[\!]} c_2\ \{\, Q \,\}}$$

L'axiome pour `choose` :

$$\{\, Q[\mathtt{x} \leftarrow 0] \land \cdots \land Q[\mathtt{x} \leftarrow N-1] \,\}\ \mathtt{x := choose}(N)\ \{\, Q \,\}$$

ou $$\{\, \forall \alpha,\ 0 \leq \alpha < N \Rightarrow Q[\mathtt{x} \leftarrow \alpha] \,\}\ \mathtt{x := choose}(N)\ \{\, Q \,\}$$

L'axiome pour `havoc` :

$$\{\, \forall \alpha, Q[\mathtt{x} \leftarrow \alpha] \,\}\ \mathtt{havoc\ x}\ \{\, Q \,\}$$

ou $$\{\, Q[\mathtt{x} \leftarrow \mathtt{y}] \,\}\ \mathtt{havoc\ x}\ \{\, Q \,\}$$ si $\mathtt{y}$ n'apparait pas dans Q

Les assertions dynamiques

Introduisent dans le langage la possibilité d'échouer pendant l'exécution.

$c ::= \dots$

$\quad \mid \texttt{assert}\ b \quad$ (b est une expression booléenne ; adapté à la vérification dynamique)

$\quad \mid \texttt{assert}\ A \quad$ (A est une assertion logique ; adapté à la vérification statique)

La vérification doit garantir l'absence d'erreurs à l'exécution.

D'où la règle :

$$\{ P \wedge A \}\ \texttt{assert}\ A\ \{ P \wedge A \}$$

Erreurs dans les calculs arithmétiques

L'évaluation d'une expression arithmétique a ou booléenne b peut aussi provoquer une erreur à l'exécution : division entière par zéro, débordement arithmétique, etc.

On peut caractériser l'absence d'erreurs par un prédicat `Def` :

$$\begin{aligned}
\mathtt{Def}(cst) &= \mathtt{Def}(x) = \top \\
\mathtt{Def}(a_1 + a_2) &= \mathtt{Def}(a_1) \land \mathtt{Def}(a_2) \land \mathtt{MIN} \leq a_1 + a_2 \leq \mathtt{MAX} \\
\mathtt{Def}(a_1 / a_2) &= \mathtt{Def}(a_1) \land \mathtt{Def}(a_2) \land a_2 \neq 0 \land \mathtt{MIN} \leq a_1 / a_2 \leq \mathtt{MAX} \\
\mathtt{Def}(a_1 \leq a_2) &= \mathtt{Def}(a_1) \land \mathtt{Def}(a_2)
\end{aligned}$$

(etc.)

Erreurs dans les calculs arithmétiques

Dans les règles de la logique, on ajoute des préconditions pour garantir que toutes les expressions s'évaluent sans erreurs.

$$\{\, Q[x \leftarrow a] \wedge \mathtt{Def}(a) \,\}\ x := a\ \{\, Q \,\}$$

$$\frac{\{\, P \wedge b \,\}\ c_1\ \{\, Q \,\} \qquad \{\, P \wedge \neg b \,\}\ c_2\ \{\, Q \,\}}{\{\, P \wedge \mathtt{Def}(b) \,\}\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \{\, Q \,\}}$$

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, P \wedge \mathtt{Def}(b) \,\}}{\{\, P \wedge \mathtt{Def}(b) \,\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{\, P \wedge \neg b \,\}}$$

Liens avec la sémantique : correction de la logique

Quelle logique pour énoncer et traiter les assertions ?

La vision de Hoare :

- Une logique «sur mesure»,
- qui «parle» directement des variables du programme (`x`, ...) et des opérateurs du langage de programmation (+, `and`, ...)
- Une assertion = une proposition de cette logique.

Une vision plus pratique :

- Une logique «standard», p.ex. logique du 1er ordre + arithmétique.
- «Parle» des variables du programme et des opérateurs du langage via une traduction.
- Une assertion = un prédicat sur l'état mémoire.

La signification des assertions

Un état mémoire s associe une valeur à chaque variable du programme.

État mémoire (*store*) $s ::= \text{variable} \rightarrow \text{valeur}$

Une assertion P (portant sur les variables $\mathtt{x}$, $\mathtt{y}$, du programme) est interprétée comme un prédicat sur l'état mémoire s :

$$[\![P]\!]\; s = P[\mathtt{x} \leftarrow s(\mathtt{x}), \mathtt{y} \leftarrow s(\mathtt{y}), \ldots]$$

Exemple

L'assertion $0 \leq \mathtt{x} < y$ est le prédicat $\lambda s.\ 0 \leq s(\mathtt{x}) < s(\mathtt{y})$.

Sémantique des expressions

On se donne une sémantique dénotationnelle des expressions du langage : chaque expression a est interprétée comme une fonction $[\![a]\!]$: état mémoire $\rightarrow$ valeur. Typiquement :

$$[\![\mathtt{x}]\!]\ s = s(\mathtt{x}) \qquad [\![a_1 + a_2]\!] = [\![a_1]\!] \oplus [\![a_2]\!]$$

$$[\![cst]\!]\ s = cst \qquad [\![a_1 * a_2]\!] = [\![a_1]\!] \otimes [\![a_2]\!]$$

Les opérateurs $\oplus$, $\otimes$ dénotent l'addition et la multiplication du langage. Par exemple pour une arithmétique modulo 2^{32} :

$$n_1 \oplus n_2 = \mathtt{norm}(n_1 + n_2) \qquad n_1 \otimes n_2 = \mathtt{norm}(n_1 \times n_2)$$

$$\mathtt{norm}(n) = n \bmod 2^{32} \qquad \text{(arithmétique non signée)}$$

$$\mathtt{norm}(n) = (n + 2^{31}) \bmod 2^{32} - 2^{31} \qquad \text{(arithmétique signée)}$$

Substitution dans les assertions

$$\{\, Q[x \leftarrow a] \,\}\; x := a \;\{\, Q \,\}$$

Dans la règle pour l'affectation, que signifie $Q[x \leftarrow a]$?

C'est le prédicat $[\![Q]\!]\, s$ où $s(x)$ est remplacé par $[\![a]\!]\, s$.

Par exemple :

$$\begin{aligned}[\![\, (x < 10)\, [x \leftarrow x + 1] \,]\!]\, s &= (s(x) < 10)\, [s(x) \leftarrow [\![x + 1]\!]\, s] \\ &= [\![x + 1]\!]\, s < 10 = (s(x) \oplus 1) < 10\end{aligned}$$

On a, par construction

$$[\![\, Q[x \leftarrow a] \,]\!]\, s = [\![Q]\!]\, (s[x \leftarrow [\![a]\!]\, s])$$

Ceci valide sémantiquement la règle de Hoare pour l'affectation.

Erreurs dans les expressions arithmétiques

Pour modéliser les erreurs dans les expressions arithmétiques (p.ex. division par zéro), on peut ajouter une dénotation `err` :

$$[\![a]\!]\ s \in \mathbb{Z} + \{\texttt{err}\}$$

L'assertion substituée $Q[x \leftarrow a]$ exige que $[\![a]\!]\ s \neq \texttt{err}$:

$$[\![Q[x \leftarrow a]]\!]\ s = [\![a]\!]\ s \neq \texttt{err} \wedge [\![Q]\!]\ (s[x \leftarrow [\![a]\!]\ s])$$

L'assertion $\texttt{Def}(a)$ doit garantir que $[\![a]\!]\ s \neq \texttt{err}$.

Ceci valide sémantiquement la règle

$$\{\, Q[x \leftarrow a] \wedge \texttt{Def}(a) \,\}\ x := a\ \{\, Q \,\}$$

Sémantique des commandes

La sémantique des commandes doit pouvoir traiter

- la divergence (non-terminaison) (boucle `while`, ...)
- les erreurs à l'exécution ($1/0$, assertions dynamiques)
- le non-déterminisme (`choose`, `havoc`, $c_1 \mathbin{[\!]} c_2$)

On choisit une sémantique opérationnelle à réductions :

$$c/s \quad \to \quad c'/s'$$
$$c/s \quad \to \quad \texttt{err}$$

c : commande s : état mémoire «avant»	une étape de calcul	c' : commande résiduelle s' : état mémoire «après» `err` : erreur à l'exécution

Règles de réduction pour IMP (cf. cours du 2019-11-28)

$$
\begin{array}{rll}
(x := a)/s & \rightarrow \texttt{skip}/s[x \leftarrow [\![a]\!]\ s] & \\
(\texttt{skip}; c_2)/s & \rightarrow c_2/s & \\
(c_1; c_2)/s & \rightarrow (c_1'; c_2)/s' & \text{si } c_1/s \rightarrow c_1'/s' \\
(c_1; c_2)/s & \rightarrow \texttt{err} & \text{si } c_1/s \rightarrow \texttt{err} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s & \rightarrow c_1/s & \text{si } [\![b]\!]\ s \text{ est vrai} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s & \rightarrow c_2/s & \text{si } [\![b]\!]\ s \text{ est faux} \\
(\texttt{while}\ b\ \texttt{do}\ c)/s & \rightarrow \texttt{skip}/s & \text{si } [\![b]\!]\ s \text{ est faux} \\
(\texttt{while}\ b\ \texttt{do}\ c)/s & \rightarrow (c; \texttt{while}\ b\ \texttt{do}\ c)/s & \text{si } [\![s]\!]\ b \text{ est vrai} \\
(\texttt{havoc}\ x)/s & \rightarrow \texttt{skip}/s[x \leftarrow n] & \text{pour tout } n \\
(\texttt{assert}\ A)/s & \rightarrow \texttt{skip}/s & \text{si } [\![A]\!]\ s \text{ est vrai} \\
(\texttt{assert}\ A)/s & \rightarrow \texttt{err} & \text{si } [\![A]\!]\ s \text{ est faux}
\end{array}
$$

Suites de réductions

Les comportements possibles d'une commande c correspondent à des suites de réductions pour c/s.

- Terminaison dans l'état final s' : réductions vers $\texttt{skip}/s'$

$$c/s \to c_1/s_1 \to \cdots \to \texttt{skip}/s'$$

- Terminaison en erreur : réductions vers $\texttt{err}$

$$c/s \to c_1/s_1 \to \cdots \to \texttt{err}$$

- Divergence : suite infinie de réductions

$$c/s \to \cdots \to c_n/s_n \to \cdots$$

- Blocage : (ne se produit pas si la relation $\to$ est complète)

$$c/s \to c_1/s_1 \to \cdots \to c'/s' \not\to \quad \text{avec } c' \neq \texttt{skip}$$

Correction de la logique vis-à-vis de la sémantique opérationnelle

L'interprétation intuitive des triplets :

$\{ P \} \; c \; \{ Q \}$	≪la commande c, démarrée dans un état initial s satisfaisant P, ne fait pas d'erreurs, et si elle termine, l'état final satisfait Q ≫
$[P] \; c \; [Q]$	≪la commande c, démarrée dans un état initial s satisfaisant P, termine toujours sans erreurs, et l'état final satisfait Q ≫

Est-ce vrai de toutes les exécutions de c/s possibles d'après la sémantique opérationnelle ?

Correction sémantique de la logique faible

Théorème (Correction sémantique de la logique faible)

Supposons $\{\, P \,\}\ c\ \{\, Q \,\}$. *Soit* s *un état mémoire tel que* $[\![P]\!]\ s$.

1. *Sûreté : il est impossible que* $c/s \xrightarrow{*} \mathtt{err}$
2. *Correction partielle : si* $c/s \xrightarrow{*} \mathtt{skip}/s'$, *alors* $[\![Q]\!]\ s'$.

Nous allons esquisser plusieurs démonstrations. La première approche est inspirée par les démonstrations de sûreté de systèmes de types.

Correction sémantique de la logique faible

Lemme (Sûreté et préservation)

Supposons $\{ P \}\ c\ \{ Q \}$ *et* $[\![P]\!]\ s$.

1. *Sûreté immédiate :* $c/s \not\rightarrow \texttt{err}$
2. *Préservation : si* $c/s \rightarrow c'/s'$, *alors il existe une précondition* P' *telle que* $\{ P' \}\ c'\ \{ Q \}$ *et* $[\![P']\!]\ s'$.

Démonstration.

Par cas sur les règles de réduction $c/s \rightarrow \ldots$ et par inversion sur la dérivation de $\{ P \}\ c\ \{ Q \}$. □

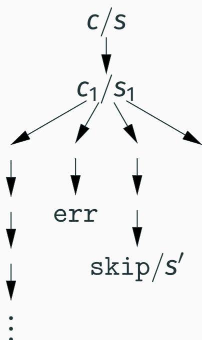
Correction sémantique de la logique faible

Le théorème de correction sémantique s'ensuit facilement :

1. Sûreté : supposons $c/s \xrightarrow{*} c'/s' \rightarrow \texttt{err}$.
   Par préservation, il existe P' tel que $\{\,P'\,\}\ c'\ \{\,Q\,\}$ et $[\![P']\!]\ s'$.
   Par sûreté immédiate, $c'/s' \not\rightarrow \texttt{err}$. Contradiction.

2. Correction partielle : supposons $c/s \xrightarrow{*} \texttt{skip}/s'$.
   Par préservation, il existe P' tel que $\{\,P'\,\}\ \texttt{skip}\ \{\,Q\,\}$ et $[\![P']\!]\ s'$.
   Par inversion sur $\{\,P'\,\}\ \texttt{skip}\ \{\,Q\,\}$, on a $P' \Rightarrow Q$.
   Donc, $[\![Q]\!]\ s'$ comme attendu.

L'arbre des réductions



Une exécution du programme = une branche de l'arbre.

Le programme termine toujours
= toutes les branches sont finies
= l'arbre peut être décrit par un prédicat inductif.

La terminaison comme prédicat inductif

$\texttt{Term}\ c\ s\ Q$: ≪la commande c démarrée dans l'état s termine toujours, et l'état final satisfait Q≫.

$$\frac{[\![Q]\!]\ s}{\texttt{Term}\ \texttt{skip}\ s\ Q}$$

$$\frac{c \neq \texttt{skip} \qquad c/s \not\to \texttt{err} \qquad (\forall c', s',\ c/s \to c'/s' \Rightarrow \texttt{Term}\ c'\ s'\ Q)}{\texttt{Term}\ c\ s\ Q}$$

Ce prédicat étant inductif, il est faux si une séquence infinie de réductions existe.

Correction sémantique de la logique forte

Le triplet sémantique : ≪si l'état initial satisfait P, la commande c termine dans un état satisfaisant Q≫

$$[[\,P\,]]\ c\ [[\,Q\,]] \quad \stackrel{def}{=} \quad \forall s,\ [\![P]\!]\ s \Rightarrow \texttt{Term}\ c\ s\ Q$$

On montre que cette définition satisfait les axiomes et les règles d'inférences de la logique de Hoare :

- $[[\,P\,]]\ \texttt{skip}\ [[\,P\,]]$
- Si $[[\,P\,]]\ c_1\ [[\,Q\,]]$ et $[[\,Q\,]]\ c_2\ [[\,R\,]]$ alors $[[\,P\,]]\ c_1; c_2\ [[\,R\,]]$
- etc.

Théorème (Correction sémantique de la logique forte)

Si $[\,P\,]\ c\ [\,Q\,]$ est dérivable, alors $[[\,P\,]]\ c\ [[\,Q\,]]$ est vrai.

Des triplets sémantiques au lieu de triplets axiomatiques

De plus en plus d'auteurs font l'économie d'axiomatiser les triplets $[\,P\,]\;c\;[\,Q\,]$ et prennent directement la définition sémantique :

$$[\,P\,]\;c\;[\,Q\,] \;\stackrel{def}{=}\; [[\,P\,]]\;c\;[[\,Q\,]] \;\stackrel{def}{=}\; \forall s,\; [\![P]\!]\;s \Rightarrow \texttt{Term}\;c\;s\;Q$$

Ils montrent alors les axiomes et les règles d'inférences de la logique de Hoare comme autant de lemmes sur cette définition.

Cela permet alors de raisonner sur les programmes comme en logique de Hoare, mais de manière sémantiquement correcte par construction.

Ce n'est plus dans l'esprit ≪axiomatique≫ de Hoare (1969), mais simplifie le formalisme et l'ajout de règles *a posteriori*.

Des triplets sémantiques pour la logique faible

Peut-on suivre la même approche pour la logique faible ?

Oui, en remplaçant le prédicat $\texttt{Term}\ c\ s\ Q$ par un prédicat $\texttt{Safe}\ c\ s\ Q$ qui dit que les exécutions de c/s ne terminent pas en erreur, et que si elles terminent alors l'état final satisfait Q.

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \quad \stackrel{def}{=} \quad \forall s,\ [\![P]\!]\ s \Rightarrow \texttt{Safe}\ c\ s\ Q$$

Une définition coinductive de `Safe`

De même que le prédicat `Term` est naturellement inductif, le prédicat `Safe` est naturellement coinductif :

$$\frac{[\![Q]\!]\ s}{\texttt{Safe}\ \texttt{skip}\ s\ Q}$$

$$\frac{c \neq \texttt{skip} \qquad c/s \not\to \texttt{err} \qquad (\forall c', s',\ c/s \to c'/s' \Rightarrow \texttt{Safe}\ c'\ s'\ Q)}{\texttt{Safe}\ c\ s\ Q}$$

Avec un prédicat coinductif, on peut avoir des dérivations infinies en profondeur. Donc `Safe` $c\ s\ Q$ est vrai si c/s diverge sans erreurs. (par application infinie de la 2e règle)

Une définition *step-indexed* de Safe

Au lieu de coinduction, on peut utiliser la technique du «comptage de pas» (*step indexing*) :

$$\mathtt{Safe}\ c\ s\ Q \overset{def}{=} \forall n, \mathtt{Safe}^n\ c\ s\ Q$$

Le prédicat inductif $\mathtt{Safe}^n\ c\ s\ Q$ signifie que les exécutions de c/s ne font pas d'erreur dans les n premières étapes d'exécution, et satisfont Q si elles terminent en au plus n étapes.

$$\mathtt{Safe}^0\ c\ s\ Q \qquad \frac{[\![Q]\!]\ s}{\mathtt{Safe}^{n+1}\ \mathtt{skip}\ s\ Q}$$

$$\frac{c \neq \mathtt{skip} \qquad c/s \not\to \mathtt{err} \qquad (\forall c', s',\ c/s \to c'/s' \Rightarrow \mathtt{Safe}^n\ c'\ s'\ Q)}{\mathtt{Safe}^{n+1}\ c\ s\ Q}$$

Triplets sémantiques pour la logique faible

Tant avec la définition coinductive de `Safe` qu'avec la définition *step-indexed*, le triplet faible sémantique

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \quad \stackrel{def}{=} \quad \forall s,\ [\![P]\!]\ s \Rightarrow \mathtt{Safe}\ c\ s\ Q$$

satisfait les axiomes et les règles de la logique de Hoare faible :

- $\{\{\, P \,\}\}\ \mathtt{skip}\ \{\{\, P \,\}\}$
- Si $\{\{\, P \,\}\}\ c_1\ \{\{\, Q \,\}\}$ et $\{\{\, Q \,\}\}\ c_2\ \{\{\, R \,\}\}$ alors $\{\{\, P \,\}\}\ c_1; c_2\ \{\{\, R \,\}\}$
- Si $\{\{\, P \land b \,\}\}\ c\ \{\{\, P \,\}\}$ alors $\{\{\, P \,\}\}\ \mathtt{while}\ b\ \mathtt{do}\ \ c\ \{\{\, P \land \neg b \,\}\}$
- etc.

Correction sémantique de la logique faible

Il en découle une autre démonstration de la correction sémantique de la logique faible :

Théorème (Correction sémantique de la logique faible)

Si $\{ P \}\ c\ \{ Q \}$ *est dérivable, alors* $\{\{ P \}\}\ c\ \{\{ Q \}\}$ *est vrai.*

Démonstration.

Par récurrence sur la dérivation de $\{ P \}\ c\ \{ Q \}$. □

Liens avec la sémantique : complétude de la logique

La complétude d'une logique de Hoare

La réciproque de la correction sémantique :

> Toute propriété vraie des exécutions d'un programme c peut-elle être exprimée comme un triplet $\{ P \}\ c\ \{ Q \}$ et dérivée en logique de Hoare ?

À l'aide des triplets sémantiques, on peut poser la question plus précisément :

> Si $\{\{ P \}\}\ c\ \{\{ Q \}\}$, peut-on dériver $\{ P \}\ c\ \{ Q \}$?
> Si $[[P]]\ c\ [[Q]]$, peut-on dériver $[P]\ c\ [Q]$?

Logique de Hoare et calculabilité

La question de la complétude a été beaucoup étudiée dans les années 1970 en raison du lien suivant entre logique de Hoare et calculabilité :

Corollaire (de la correction sémantique)

Si $\{ \top \}\ c\ \{ \bot \}$ *est dérivable, alors* c *ne termine pas.*

Si la logique de Hoare était complète, on aurait une équivalence : $\{ \top \}\ c\ \{ \bot \}$ est dérivable si et seulement si c ne termine pas.

Incomplétude de la logique pour un langage Turing-complet

Rappel : l'ensemble des énoncés dérivables dans un système d'axiomes et de règles est récursivement énumérable (r.e.).

L'ensemble des triplets $\{ P \}\ c\ \{ Q \}$ dérivables est donc r.e.

L'ensemble des triplets $\{ \top \}\ c\ \{ \bot \}$ dérivables est donc r.e.
(en $\ll$filtrant$\gg$ l'énumération de tous les triplets).

Si la logique est complète, l'ensemble des programmes c qui ne terminent pas est donc r.e.

Par conséquent, si la logique est complète, le problème de l'arrêt est décidable !

Une analyse du problème

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Que signifient les prémisses $P \Rightarrow P'$ et $Q' \Rightarrow Q$?

- ≪Implications dérivables dans une logique formelle.≫ L'ensemble de ces implications est r.e., donc $\{\, P \,\}\ c\ \{\, Q \,\}$ est r.e., et la logique est incomplète.
- ≪Implications vraies (dans tous les modèles).≫ Alors $\{\, P \,\}\ c\ \{\, Q \,\}$ n'est pas r.e. et la logique est complète (transparents suivants).

Complétude relative

(Stephen A. Cook, *Soundness and completeness of an axiom system for program verification*, SIAM J. Comput., 1978)

On peut montrer que la logique de Hoare est complète si la même logique «ambiante» est utilisée

- pour interpréter les implications $P \Rightarrow P'$, $Q' \Rightarrow Q$ dans la règle de conséquence ;
- pour définir le triplet sémantique $\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \stackrel{def}{=} \forall s,\ [\![P]\!]\ s \Rightarrow \mathtt{Safe}\ c\ s\ Q.$

Plus faible précondition sémantique

On définit la plus faible précondition sémantique (*weakest (liberal) precondition*) de la commande c avec postcondition Q :

$$\mathtt{wpsem}\ c\ Q \quad \stackrel{def}{=} \quad \lambda s.\ \mathtt{Safe}\ c\ s\ Q$$

Par définition du triplet sémantique, on a

$$\{\{\ P\ \}\}\ c\ \{\{\ Q\ \}\} \quad \text{si et seulement si} \quad P \Rightarrow \mathtt{wpsem}\ c\ Q$$

Lemme (la plus faible précondition sémantique est dérivable)

$\{\ \mathtt{wpsem}\ c\ Q\ \}\ c\ \{\ Q\ \}$ *est dérivable en logique de Hoare.*

Démonstration.

Récurrence sur c et «inversions» sur le prédicat $\mathtt{Safe}$, p.ex. $\mathtt{Safe}\ (c_1; c_2)\ s\ Q$ implique $\mathtt{Safe}\ c_1\ s\ (\mathtt{wpsem}\ c_2\ Q)$. □

Complétude relative

Théorème (complétude relative)

Si $\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, Q \,\}\!\}$ *est démontrable dans une logique L, alors* $\{\, P \,\}\ c\ \{\, Q \,\}$ *est dérivable dans la logique de Hoare utilisant la logique L pour les implications de la règle de conséquence.*

Démonstration.

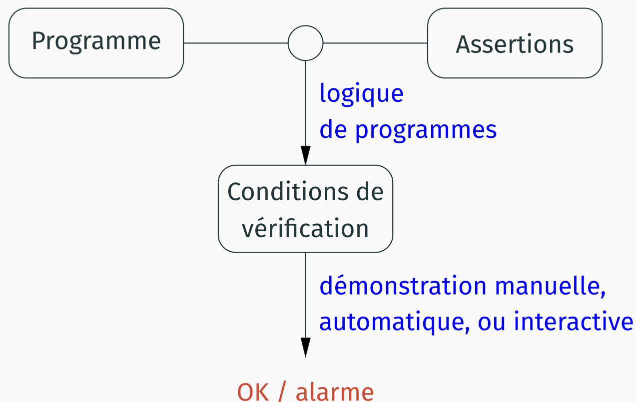
Par hypothèse $\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, Q \,\}\!\}$, on a $P \Rightarrow \texttt{wpsem}\ c\ Q$.

Par le lemme, on peut dériver $\{\, \texttt{wpsem}\ c\ Q \,\}\ c\ \{\, Q \,\}$.

On conclut $\{\, P \,\}\ c\ \{\, Q \,\}$ par la règle de conséquence. □

Vers l’automatisation : un calcul des plus faibles préconditions

La vérification déductive (rappel)



Comment engendrer les conditions de vérification ?

Comment minimiser la quantité d'assertions à fournir ?

Programmes complètement annotés

$\{0 \leq a\} \Rightarrow \{a = b \cdot 0 + a \ \wedge\ 0 \leq a\}$

```
r := a;
```

$\{a = b \cdot 0 + r \ \wedge\ 0 \leq r\}$

```
q := 0;
```

$\{a = b \cdot q + r \ \wedge\ 0 \leq r\}$

```
while r ≥ b do
```

$\{a = b \cdot q + r \ \wedge\ 0 \leq r \ \wedge\ r \geq b\} \Rightarrow$
$\{a = b \cdot (q+1) + (r - b) \ \wedge\ 0 \leq r - b\}$

```
    r := r − b;
```

$\{a = b \cdot (q+1) + r \ \wedge\ 0 \leq r\}$

```
    q := q + 1
```

$\{a = b \cdot q + r \ \wedge\ 0 \leq r\}$

```
done
```

$\{a = b \cdot q + r \ \wedge\ 0 \leq r \ \wedge\ r < b\} \Rightarrow$
$\{q = a/b \ \wedge\ r = a \bmod b\}$

Conditions de vérification : les étapes «⇒» où on applique la règle de conséquence.

Réduire la quantité d’annotations à fournir

Pour vérifier un sous-programme c, il suffit de fournir

- la précondition P
- la postcondition Q
- un invariant de boucle Inv pour chaque boucle dans c.

Les autres assertions logiques et les conditions de vérification s’obtiennent alors par calcul de plus faibles préconditions ou de plus fortes postconditions.

Plus faible précondition

La plus faible précondition (*weakest precondition*) d'une commande c avec postcondition Q est une assertion $\texttt{wp}\ c\ Q$ telle que

- c'est une précondition : $[\,\texttt{wp}\ c\ Q\,]\ c\ [\,Q\,]$;
- c'est la plus faible : si $[\,P\,]\ c\ [\,Q\,]$ alors $P \Rightarrow \texttt{wp}\ c\ Q$.

Par conséquent :

$$[\,P\,]\ c\ [\,Q\,] \quad \text{si et seulement si} \quad P \Rightarrow \texttt{wp}\ c\ Q$$

Intuition : les hypothèses nécessaires pour que le code c calcule bien le résultat décrit par la postcondition Q.

Intuition originale (Dijkstra, 1975) : synthétiser le code c par raffinement à partir de sa postcondition Q.

Autres «transformateurs de prédicats»

Plus faible précondition libérale $\texttt{wlp}\ c\ Q$

(*weakest liberal precondition*)

Comme $\texttt{wp}$ mais ne garantit pas la terminaison :

$$\{\,P\,\}\ c\ \{\,Q\,\} \quad \text{si et seulement si} \quad P \Rightarrow \texttt{wlp}\ c\ Q$$

Plus forte postcondition (libérale) $\texttt{sp}\ P\ c$ $\texttt{slp}\ P\ c$

(*strongest (liberal) postcondition*)

$$[\,P\,]\ c\ [\,Q\,] \quad \text{si et seulement si} \quad \texttt{sp}\ P\ c \Rightarrow Q$$

$$\{\,P\,\}\ c\ \{\,Q\,\} \quad \text{si et seulement si} \quad \texttt{slp}\ P\ c \Rightarrow Q$$

Intuition : exécution symbolique de c à partir d'un état satisfaisant P.

Calculer la plus faible précondition

Une caractérisation non effective : $\texttt{wlp}\ c\ Q = \bigvee \{ P \mid \{\, P \,\}\ c\ \{\, Q \,\} \}$

Pour les programmes sans boucles, une définition récursive sur c :

$$
\begin{aligned}
\texttt{wlp}\ \texttt{skip}\ Q &= Q \\
\texttt{wlp}\ (x := a)\ Q &= Q[x \leftarrow a] \\
\texttt{wlp}\ (c_1; c_2)\ Q &= \texttt{wlp}\ c_1\ (\texttt{wlp}\ c_2\ Q) \\
\texttt{wlp}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ Q &= (b \land \texttt{wlp}\ c_1\ Q) \lor (\neg b \land \texttt{wlp}\ c_2\ Q) \\
\texttt{wlp}\ (\texttt{havoc}\ x)\ Q &= \forall n,\ Q[x \leftarrow n] \\
\texttt{wlp}\ (\texttt{assert}\ A)\ Q &= A \land Q
\end{aligned}
$$

(Les mêmes équations sont valides pour $\texttt{wp}$.)

Plus faible précondition libérale pour une boucle

Non calculable en général : $\texttt{wlp}\ (\texttt{while}\ b\ \texttt{do}\ c)\ Q = \bigvee_i P_i$
avec $P_0 = \neg b \wedge Q$ et $P_{i+1} = b \wedge \texttt{wlp}\ c\ P_i$.

On demande au programmeur d'annoter chaque boucle par son invariant *Inv*. Alors :

$$\texttt{wlp}\ (\texttt{while}^{Inv}\ b\ \texttt{do}\ c)\ Q = Inv$$

à condition que

$$b \wedge Inv \Rightarrow \texttt{wlp}\ c\ Inv \qquad \text{et} \qquad \neg b \wedge Inv \Rightarrow Q$$

Pour calculer `wp`, il faut aussi annoter la boucle par le variant qui en garantit la terminaison.

Un semi-algorithme de vérification déductive

Pour vérifier $\{ P \} \; c \; \{ Q \}$, supposant annotées les boucles de c :

1. Calculer $\texttt{wlp}\; c\; Q$ et les conditions de vérification $\texttt{vc}\; c\; Q$:

$$\begin{aligned}
\texttt{vc}\; (\texttt{while}^{Inv}\; b\; \texttt{do}\; c)\; Q &= (b \land Inv \Rightarrow \texttt{wlp}\; c\; Inv) \\
&\quad \land (\neg b \land Inv \Rightarrow Q) \\
&\quad \land \texttt{vc}\; c\; Inv \\
\texttt{vc}\; \texttt{skip}\; Q &= \top \\
\texttt{vc}\; (c_1; c_2)\; Q &= \texttt{vc}\; c_1\; (\texttt{wlp}\; c_2\; Q) \land \texttt{vc}\; c_2\; Q
\end{aligned}$$

   et de même pour les autres constructions du langage.

2. Démontrer $(P \Rightarrow \texttt{wlp}\; c\; Q) \land \texttt{vc}\; c\; Q$, qui est une formule de logique usuelle, à l'aide d'un démonstrateur automatique.

Calculer et vérifier la plus forte postcondition libérale

Pour mémoire, les équations pour `slp` :

$$\begin{aligned}
\texttt{slp}\ P\ \texttt{skip} &= P \\
\texttt{slp}\ P\ (x := a) &= \exists x_0, \mathrm{x} = a[\mathrm{x} \leftarrow x_0] \wedge P[\mathrm{x} \leftarrow x_0] \\
\texttt{slp}\ P\ (c_1; c_2) &= \texttt{slp}\ (\texttt{slp}\ P\ c_1)\ c_2 \\
\texttt{slp}\ P\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &= \texttt{slp}\ (b \wedge P)\ c_1 \vee \texttt{slp}\ (\neg b \wedge P)\ c_2 \\
\texttt{slp}\ P\ (\texttt{while}^{Inv}\ b\ \texttt{do}\ c) &= \neg b \wedge Inv \\
\texttt{slp}\ P\ (\texttt{havoc}\ \mathrm{x}) &= \exists x_0, P[x \leftarrow x_0] \\
\texttt{slp}\ P\ (\texttt{assert}\ A) &= A \wedge P
\end{aligned}$$

et les conditions de vérification non triviales :

$$\begin{aligned}
\texttt{vc}\ P\ (\texttt{while}^{Inv}\ b\ \texttt{do}\ c) &= (P \Rightarrow Inv) \wedge (\texttt{sp}\ (b \wedge Inv)\ c \Rightarrow Inv) \\
&\quad \wedge \texttt{vc}\ (b \wedge Inv)\ c \\
\texttt{vc}\ P\ (\texttt{assert}\ A) &= P \Rightarrow A
\end{aligned}$$

Point d’étape

Point d'étape sur la logique de Hoare

Un formalisme très riche.

Deux visions complémentaires :

- La vision axiomatique : les règles définissent le langage.
- La vision opérationnelle : les règles sont des théorèmes qui facilitent le raisonnement sur les exécutions des programmes.

S'étend assez facilement à de nombreuses structures de contrôle (`goto`, `break`, `return`, exceptions, procédures, ...).

Quid des structures de données ? (→ prochain cours)

Bibliographie

Bibliographie

Deux présentations de la logique de Hoare :

- H. R. Nielson et F. Nielson, *Semantics with Applications : an appetizer*, Springer, 2007, ch. 9 et 10.
  (Suit l'approche opérationnelle.)
- G. Winskel, *The Formal Semantics of Programming Languages*, MIT Press, 1993, ch. 6 et 7.
  (Suit l'approche axiomatique classique. Discussion très complète de la question de la complétude.)

Mécanisations de la logique de Hoare :

- Le développement Coq correspondant à ce cours :
  `https://github.com/xavierleroy/cdf-program-logics`
- T. Nipkow et G. Klein, *Concrete Semantics*, chap. 12.