Program logics, second lecture

Variables and loops: Hoare logic

Xavier Leroy

2021-03-11

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Foundations of Hoare logic

Hoare triples

"Weak" triples:

$$\underset{\text{precondition}}{\{\,P\,\}}\;\underset{\text{command}}{c}\;\underset{\text{postcondition}}{\{\,Q\,\}}$$

Intuitive meaning:

"If command c, started in an initial state satisfying P, terminates, then the final state satisfies Q."

Later we'll see "strong" triples $[\,P\,]\;c\;[\,Q\,]$ that guarantee termination: "command c, started in an initial state satisfying P, always terminates, and the final state satisfies Q."

IMP: a small imperative language with structured control

Arithmetic expressions:

$a ::= \mathtt{x}$	program variables
$\mid 0 \mid 1 \mid \ldots$	constants
$\mid a_1 + a_2 \mid a_1 \times a_2 \mid \ldots$	operations

Boolean expressions:

$b ::= a_1 \leq a_2 \mid \ldots$	comparisons
$\mid b_1 \ \mathtt{and}\ b_2 \mid \mathtt{not}\ b \mid \ldots$	connectives

Commands:

$c ::= \mathtt{skip}$	empty command
$\mid \mathtt{x} := a$	assignment
$\mid c_1; c_2$	sequence
$\mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$	conditional
$\mid \mathtt{while}\ b\ \mathtt{do}\ c$	loop

The rules of weak Hoare logic for IMP

One rule for each kind of command.

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

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

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

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

Generic rules

The consequence rule:

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

Can also be presented as two rules: one that strengthens the precondition, another that weakens the 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: the top rule is derivable from the bottom two rules, and conversely.

An example of a derivation

$$
\dfrac{\top \Rightarrow 0 = 0 \land 1 = 1 \qquad \dfrac{\begin{array}{c}\{\, 0 = 0 \land 1 = 1 \,\}\ \mathrm{x} := 0\ \{\, \mathrm{x} = 0 \land 1 = 1 \,\} \\ \{\, \mathrm{x} = 0 \land 1 = 1 \,\}\ \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}\end{array}}{\{\, 0 = 0 \land 1 = 1 \,\}\ \mathrm{x} := 0; \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}}}{\{\, \top \,\}\ \mathrm{x} := 0; \mathrm{y} := 1\ \{\, \mathrm{x} = 0 \land \mathrm{y} = 1 \,\}}
$$

An example of a derivation

A more compact notation, as an IMP program annotated with assertions:

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

Verifying a “real” program: Euclidean division

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

$\mathtt{r} := \mathtt{a};$

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

$\mathtt{q} := 0;$

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

$\mathtt{while}\ \mathtt{r} \geq \mathtt{b}\ \mathtt{do}$

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

$\quad \mathtt{r} := \mathtt{r} - \mathtt{b};$

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

$\quad \mathtt{q} := \mathtt{q} + 1$

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

$\mathtt{done}$

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

Exercise: computing the maximum of two values (Mike Gordon)

1. Write an IMP program that sets $\mathtt{x}$ to the maximum of the values of $\mathtt{x}$ and of $\mathtt{y}$.

2. Specify this program in Hoare logic.

3. Verify the program against this specification.

Exercise: computing the maximum of two values (Mike Gordon)

1. Write an IMP program that sets $\mathtt{x}$ to the maximum of the values of $\mathtt{x}$ and of $\mathtt{y}$.

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

2. Specify this program in Hoare logic.

3. Verify the program against this specification.

Exercise: computing the maximum of two values (Mike Gordon)

1. Write an IMP program that sets $\mathtt{x}$ to the maximum of the values of $\mathtt{x}$ and of $\mathtt{y}$.

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

2. Specify this program in Hoare logic.

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

3. Verify the program against this specification.

Exercise: computing the maximum of two values (Mike Gordon)

1. Write an IMP program that sets x to the maximum of the values of x and of y.

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

2. Specify this program in Hoare logic.

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

3. Verify the program against this specification.

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

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

We conclude using the rule for conditional.

Is this the correct specification?

Many programs satisfy this specification…

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

Is this the correct specification?

Many programs satisfy this specification…

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

This is the wrong specification! We wanted to say

> *The value of* $\mathtt{x}$ *at the end of the program is the maximum of the values of* $\mathtt{x}$ *and* $\mathtt{y}$ *at the beginning of the program.*

Auxiliary variables

One solution is to use mathematical variables $\alpha, \beta, \ldots$, distinct from program variables $\mathtt{x}, \mathtt{y}, \ldots$:

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

These auxiliary variables are universally quantified implicitly before the triple:

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

Ghost variables

Alternative: in the specification, use variables from the programming language that do not occur in the program being specified:

$$\{\, \mathtt{x} = \mathtt{z} \,\}\; c \;\{\, \mathtt{x} = \max(\mathtt{z}, \mathtt{y}) \,\} \qquad \text{where } \mathtt{z} \text{ not free in } c$$

These ghost variables $\mathtt{z}$ preserve their values during execution of c, enabling the postcondition to talk about the state “before”.

“Strong” Hoare logic (total correctness)

“Strong” triples:

$$\begin{array}{ccc} & [\,P\,]\;c\;[\,Q\,] & \\ \nearrow & \uparrow & \nwarrow \\ \text{precondition} & \text{command} & \text{postcondition} \end{array}$$

Intuitive meaning:

“Command c, started in an initial state satisfying P, terminates in a final state satisfying Q.”

The rules of strong Hoare logic for IMP

Only loops can cause non-termination
$\Rightarrow$ for all other IMP constructors, the "strong" rules are similar to the "weak" rules.

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

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

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

Verifying termination of a loop

Using a variant: an expression a, with nonnegative values, that decreases at every loop iteration.

$$\frac{\forall\alpha,\ [\,P \wedge b \wedge a = \alpha\,]\; c\; [\,P \wedge 0 \leq a < \alpha\,]}{[\,P\,]\ \mathtt{while}\ b\ \mathtt{do}\ c\ [\,P \wedge \neg b\,]}$$

The loop must terminate after at most N iterations, where N is the initial value of variant a.

Note: with nested loops, the termination of each loop is verified independently of the other loops.
(Unlike in Turing 1949 and Floyd 1967.)

Verifying termination of Euclidean division

The variant is the variable $\mathtt{r}$.

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

$\mathtt{r} := a;$

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

$q := 0;$

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

$\mathtt{while}\ \mathtt{r} \ge b\ \mathrm{do}$

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

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

$\quad \mathtt{r} := \mathtt{r} - b;$

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

$\quad q := q + 1$

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

$\mathtt{done}$

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

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

Extension to several variants

We can use several variants $a_1, \dots, a_n$
and a well-founded order $\prec$ over n-tuples of integers.

(Typically, lexicographic order.)

$$\frac{\forall \alpha_1, \dots, \alpha_n, \quad [\, P \land b \land (a_1, \dots, a_n) = (\alpha_1, \dots \alpha_n) \,] \; c \; [\, P \land (a_1, \dots, a_n) \prec (\alpha_1, \dots \alpha_n) \,]}{[\, P \,] \; \texttt{while} \; b \; \texttt{do} \; c \; [\, P \land \neg b \,]}$$

Adding rules to the logic

A derived rule: conditional without `else`

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

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

Proof.

Here is the derivation:

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

$\square$

A derived rule: the `do...while` loop

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

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

Proof.

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

$\square$

A derived rule: the `for` counted loop

Notation: if h, i are two distinct variables,

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

We can derive a strong triple that guarantees loop termination, provided the loop body c contains no assignments to i nor to h.

$$\frac{[\, P \wedge i \leq h \,]\ c\ [\, P[i \leftarrow i+1] \,] \qquad i, h \text{ not assigned to in } c}{[\, P[i \leftarrow \ell] \,]\ \texttt{for}\ i = \ell\ \texttt{to}\ h\ \texttt{do}\ c\ [\, P \wedge i > h \,]}$$

The variant is the expression $h - i + 1$, which decreases by 1 at each iteration.

A derived rule: Floyd-style assignment

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

Proof.

Write $Q \stackrel{def}{=} \exists x_0, \mathtt{x} = a[\mathtt{x} \leftarrow x_0] \wedge P[\mathtt{x} \leftarrow x_0]$.

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

Indeed, $\begin{array}{rcl} Q[\mathtt{x} \leftarrow a] & = & \exists x_0, a = a[\mathtt{x} \leftarrow x_0] \wedge P[\mathtt{x} \leftarrow x_0][\mathtt{x} \leftarrow a] \\ & = & \exists x_0, a = a[\mathtt{x} \leftarrow x_0] \wedge P[\mathtt{x} \leftarrow x_0] \end{array}$

and it suffices to take $x_0 = \mathtt{x}$. $\square$

Some admissible rules

Conjunction, disjunction, quantification:

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

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

Proof.

Induction on c and inversion on the derivations of $\{\, P_1 \,\}\; c\; \{\, Q_1 \,\}$, $\{\, P_2 \,\}\; c\; \{\, Q_2 \,\}$, etc. $\square$

Growing the programming language

Unstructured control (`goto`)

Goto considered harmful … or not?

Commands: $c ::= \ldots \mid \texttt{goto}\ \ell \mid \ell : c$

We need to associate an invariant $L(\ell)$ to each label ℓ.

The triples become $L \vdash \{\, P \,\}\ c\ \{\, Q \,\}$.

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

Nondeterminism

Enable programs to have several different behaviors.

$$
\begin{array}{lll}
c ::= \dots & & \\
\quad \mid c_1 \mathbin{[\!]} c_2 & & \text{execute either } c_1 \text{ or } c_2 \\
\quad \mid x := \texttt{choose}(N) & & \text{set } x \text{ to a number between 0 and } N-1 \\
\quad \mid \texttt{havoc}\ x & & \text{set } x \text{ to an arbitrary number}
\end{array}
$$

The other constructions can be derived from `havoc`:

$$
\begin{array}{rcl}
x := \texttt{choose}(N) & \approx & \texttt{havoc}\ x;\ x := x \bmod N \\
c_1 \mathbin{[\!]} c_2 & \approx & x := \texttt{choose}(2); \texttt{if}\ x = 0\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \\
x := \texttt{choose}(N) & \approx & x := 0 \mathbin{[\!]} x := 1 \mathbin{[\!]} \cdots \mathbin{[\!]} x := N-1
\end{array}
$$

Rules for nondeterminism

The rule for choice:

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

The axiom for `choose`:

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

or

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

The axiom for `havoc`:

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

or

$$\{\,Q[\mathtt{x} \leftarrow \mathtt{y}]\,\}\ \mathtt{havoc\ x}\ \{\,Q\,\} \quad \text{if } \mathtt{y} \text{ does not occur in } Q$$

Run-time assertions

Run-time assertions introduce the possibility of run-time failure.

$c ::= \ldots$

$\quad\mid \texttt{assert}\ b$ (b is a Boolean expression; appropriate for run-time checking)

$\quad\mid \texttt{assert}\ A$ (A is a logical assertion; appropriate for static verification)

Verification must guarantee the absence of run-time failures.

Hence the rule:

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

Errors in arithmetic computations

Evaluating an arithmetic expression a or Boolean expression b can also fail at run-time: integer division by zero, arithmetic overflow, etc.

We can characterize absence of failures as a predicate `Def`:

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

(etc.)

In the rules of the logic, we add preconditions to guarantee that all expressions evaluate without errors.

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

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

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

Connections with semantics: soundness of Hoare logic

Which logic to state and work with assertions?

Hoare's viewpoint:

- A "bespoke" logic
- that "talks" about program variables (`x`, ...) and programming language operators (+, `and`, ...)
- An assertion = a proposition of this bespoke logic.

A more practical viewpoint:

- An "off the shelf" logic,
  typically first-order logic + arithmetic.
- "Talks" about program variables and programming language operators via a translation.
- An assertion = a predicate on the memory store.

The meaning of assertions

A store s associates a value to each program variable.

Store $\quad s ::= \text{variable} \to \text{value}$

An assertion P (mentioning program variables $\mathtt{x}, \mathtt{y}, \ldots$) is interpreted as a predicate on the store s:

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

Example

The assertion $0 \leq \mathtt{x} < \mathtt{y}$ denotes the predicate $\lambda s.\ 0 \leq s(\mathtt{x}) < s(\mathtt{y})$.

Semantics of expressions

We assume given a denotational semantics for expressions of the language: each expression a is interpreted as a function $[\![a]\!] : \text{store} \to \text{value}$. Typically:

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

Operators $\oplus$, $\otimes$ denote addition and multiplication of the programming language. For example, for arithmetic modulo 2^32:

$$n_1 \oplus n_2 = \mathtt{norm}(n_1 + n_2) \qquad n_1 \otimes n_2 = \mathtt{norm}(n_1 \times n_2)$$
$$\mathtt{norm}(n) = n \bmod 2^{32} \qquad \text{(unsigned integers)}$$
$$\mathtt{norm}(n) = (n + 2^{31}) \bmod 2^{32} - 2^{31} \quad \text{(signed integers)}$$

Substitution in assertions

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

In the assignment rule, what does $Q[x \leftarrow a]$ mean?

It is the predicate $[\![Q]\!]\ s$ where $[\![x]\!]\ s$ (that is, $s(x)$) is replaced by $[\![a]\!]\ s$.

Example:

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

By construction, we have

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

This provides semantic justification for Hoare's assignment rule.

Errors in arithmetic expressions

To model arithmetic errors (e.g. division by zero), we can add a special denotation $\mathtt{err}$:

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

The substituted assertion $Q[x \leftarrow a]$ requires $[\![a]\!]\, s \neq \mathtt{err}$:

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

The $\mathtt{Def}(a)$ assertion must guarantee $[\![a]\!]\, s \neq \mathtt{err}$.

This provides semantic justification for the modified assignment rule

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

Semantics of commands

A semantics for commands must account for

- divergence (non-termination) (`while` loops, ...)
- run-time errors ($1/0$, run-time assertions)
- nondeterminism (`choose`, `havoc`, $c_1 \mathbin{[\!]} c_2$)

We take an operational semantics based on a reduction relation:

$$
\begin{array}{ccc}
c/s & \rightarrow & c'/s' \\
c/s & \rightarrow & \mathtt{err}
\end{array}
$$

c: command	one step	c': residual command
s: store "before"	of execution	s': store "after"
		`err`: run-time error

Reduction rules for IMP

(see my lecture of 2019-11-28)

$$
\begin{array}{rcll}
(x := a)/s & \to & \texttt{skip}/s[x \leftarrow [\![a]\!]\, s] & \\
(\texttt{skip}; c_2)/s & \to & c_2/s & \\
(c_1; c_2)/s & \to & (c_1'; c_2)/s' & \text{if } c_1/s \to c_1'/s' \\
(c_1; c_2)/s & \to & \texttt{err} & \text{if } c_1/s \to \texttt{err} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s & \to & c_1/s & \text{if } [\![b]\!]\, s \text{ is true} \\
(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s & \to & c_2/s & \text{if } [\![b]\!]\, s \text{ is false} \\
(\texttt{while}\ b\ \texttt{do}\ c)/s & \to & \texttt{skip}/s & \text{if } [\![b]\!]\, s \text{ is false} \\
(\texttt{while}\ b\ \texttt{do}\ c)/s & \to & (c; \texttt{while}\ b\ \texttt{do}\ c)/s & \text{if } [\![s]\!]\, b \text{ is true} \\
(\texttt{havoc}\ x)/s & \to & \texttt{skip}/s[x \leftarrow n] & \text{for any } n \\
(\texttt{assert}\ A)/s & \to & \texttt{skip}/s & \text{if } [\![A]\!]\, s \text{ is true} \\
(\texttt{assert}\ A)/s & \to & \texttt{err} & \text{if } [\![A]\!]\, s \text{ is false}
\end{array}
$$

Reduction sequences

The possible behaviors of a command c correspond to sequences of reductions for c/s.

- Termination with final store s': reductions to $\texttt{skip}/s'$

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

- Termination on an error: reductions to $\texttt{err}$

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

- Divergence: infinite reduction sequence

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

- Going wrong: (cannot happen if relation $\to$ is complete)

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

Soundness of the logic w.r.t. reduction semantics

Intuitive interpretation of triples:

$\{ P \} \; c \; \{ Q \}$	"command c, started in an initial store s satisfying P, executes without run-time errors, and if it terminates, the final store satisfies Q"
$[P] \; c \; [Q]$	"command c, started in an initial store s satisfying P, always terminates without run-time errors, and the final store satisfies Q"

Is this true of all the possible executions of c/s according to the reduction semantics?

Semantic soundness of the weak logic

Theorem (Semantic soundness of the weak logic)

Assume $\{\, P \,\}\ c\ \{\, Q \,\}$. *Let* s *be a store such that* $[\![P]\!]\ s$.

1. *Safety: it is impossible that* $c/s \xrightarrow{*} \texttt{err}$
2. *Partial correctness: if* $c/s \xrightarrow{*} \texttt{skip}/s'$, *then* $[\![Q]\!]\ s'$.

We now outline several proofs. The first proof is inspired by soundness proofs for type systems.

Semantic soundness of the weak logic

Lemma (Immediate safety and preservation)

Assume $\{\, P \,\}\ c\ \{\, Q \,\}$ *and* $[\![P]\!]\ s$.

1. *Immediate safety:* $c/s \not\to \mathtt{err}$
2. *Preservation: if* $c/s \to c'/s'$, *there exists a precondition* P' *such that* $\{\, P' \,\}\ c'\ \{\, Q \,\}$ *and* $[\![P']\!]\ s'$.

Proof.

By case analysis on reduction rules $c/s \to \ldots$ and inversion on the derivation of $\{\, P \,\}\ c\ \{\, Q \,\}$. □

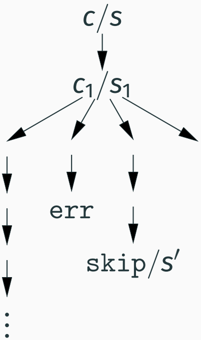
Semantic soundness of the weak logic

Semantic soundness follows easily:

1. Safety: assume by way of contradiction that
   $c/s \xrightarrow{*} c'/s' \rightarrow \texttt{err}$.
   By preservation, there exists P' s.t. $\{\, P' \,\}\ c'\ \{\, Q \,\}$ and $[\![P']\!]\ s'$.
   By immediate safety, $c'/s' \not\rightarrow \texttt{err}$. Contradiction.

2. Partial correctness: assume $c/s \xrightarrow{*} \texttt{skip}/s'$.
   By preservation, there exists P' such that $\{\, P' \,\}\ \texttt{skip}\ \{\, Q \,\}$
   and $[\![P']\!]\ s'$.
   By inversion on $\{\, P' \,\}\ \texttt{skip}\ \{\, Q \,\}$, we have $P' \Rightarrow Q$.
   Therefore, $[\![Q]\!]\ s'$, as expected.

The tree of reductions



One run of the program = one branch of the tree.

The program always terminates
= all branches are finite
= the tree can be described by an inductive predicate

Termination as an inductive predicate

$\texttt{Term}\ c\ s\ Q$: “command c, started in store s, always terminates, and the final store satisfies Q”.

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

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

This is an inductive predicate. Therefore, it does not hold if an infinite sequence of reductions exist.

Semantic soundness of the strong logic

The semantic triple: "if the initial store satisfies P, command c terminates in a store satisfying Q"

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

We show that this definition satisfies the axioms and the inference rules of Hoare logic:

- $[[\, P \,]]\ \mathtt{skip}\ [[\, P \,]]$
- If $[[\, P \,]]\ c_1\ [[\, Q \,]]$ and $[[\, Q \,]]\ c_2\ [[\, R \,]]$ then $[[\, P \,]]\ c_1; c_2\ [[\, R \,]]$
- etc.

Theorem (Semantic soundness of the strong logic)

If $[\, P \,]\ c\ [\, Q \,]$ is derivable, then $[[\, P \,]]\ c\ [[\, Q \,]]$ holds.

Semantic triples instead of axiomatic triples

Many authors no longer provide an axiomatization of the triples $[\,P\,]\ c\ [\,Q\,]$. Instead, they take the semantic definition directly:

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

Then, they show the axioms and the inference rules of the logic as lemmas that hold for this definition.

This makes it possible to reason over programs like in Hoare logic, but with semantic soundness being guaranteed by construction.

This is not in the “axiomatic” spirit of Hoare (1969),
but simplifies the formalization and the addition of new rules *a posteriori*.

Semantic triples for weak logic

Can we follow the same approach for a weak program logic?

Yes, if we replace the predicate $\texttt{Term}\ c\ s\ Q$ by a predicate $\texttt{Safe}\ c\ s\ Q$ stating that executions of c/s do not terminate on an error, and that if they terminate, the final state satisfies Q.

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

A coinductive definition of `Safe`

Just like the `Term` predicate is naturally inductive, the `Safe` predicate is naturally coinductive:

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

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

A coinductive predicate supports derivations that are infinitely deep. Hence, `Safe` $c\ s\ Q$ holds if c/s diverges without errors (by infinitely many applications of the second rule).

A step-indexed definition of $\texttt{Safe}$

Instead of coinduction, we can use step-indexing:

$$\texttt{Safe}\ c\ s\ Q \quad \stackrel{def}{=} \quad \forall n, \texttt{Safe}^n\ c\ s\ Q$$

The inductive predicate $\texttt{Safe}^n\ c\ s\ Q$ means that executions of c/s do not cause errors during the first n execution steps, and satisfy Q if they terminate in n steps or less.

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

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

Semantic triples for the weak logic

Either with the coinductive definition of $\texttt{Safe}$ or with the step-indexed definition, the weak semantic triple

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

satisfies the axioms and the inference rules of weak Hoare logic:

- $\{\{\, P \,\}\}\ \texttt{skip}\ \{\{\, P \,\}\}$
- If $\{\{\, P \,\}\}\ c_1\ \{\{\, Q \,\}\}$ and $\{\{\, Q \,\}\}\ c_2\ \{\{\, R \,\}\}$ then $\{\{\, P \,\}\}\ c_1; c_2\ \{\{\, R \,\}\}$
- If $\{\{\, P \wedge b \,\}\}\ c\ \{\{\, P \,\}\}$ then $\{\{\, P \,\}\}\ \texttt{while}\ b\ \texttt{do}\ \ c\ \{\{\, P \wedge \neg b \,\}\}$
- etc.

Semantic soundness of the weak logic

As a corollary, we obtain another proof of semantic soundness for the weak logic:

Theorem (Semantic soundness of the weak logic)

If $\{\, P \,\}\ c\ \{\, Q \,\}$ *is derivable, then* $\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, Q \,\}\!\}$ *holds.*

Proof.

By induction on the derivation of $\{\, P \,\}\ c\ \{\, Q \,\}$. $\square$

Connections with semantics: completeness of Hoare logic

Completeness of Hoare logic

The converse of semantic soundness:

> Can any property of the executions of a program c be expressed as a triple $\{ P \}\ c\ \{ Q \}$ and derived in Hoare logic?

Using semantic triples, we can make the question more precise:

> If $\{\{ P \}\}\ c\ \{\{ Q \}\}$ holds, can we derive $\{ P \}\ c\ \{ Q \}$?
> If $[[P]]\ c\ [[Q]]$ holds, can we derive $[P]\ c\ [Q]$?

The completeness issue was the topic of many studies in the 1970's because of the following connection between Hoare logic and computability:

Corollary (of semantic soundness)

If $\{\top\}\ c\ \{\bot\}$ is derivable, then c does not terminate.

If Hoare logic was complete, we would have an equivalence: $\{\top\}\ c\ \{\bot\}$ is derivable if and only if c does not terminate.

Incompleteness of the logic for a Turing-complete language

Reminder: the set of propositions that can be derived in a system of axioms and inference rules is recursively enumerable (r.e.).

The set of derivable triples $\{\, P \,\}\ c\ \{\, Q \,\}$ is r.e.

The set of derivable triples $\{\, \top \,\}\ c\ \{\, \bot \,\}$ is r.e.
(by “filtering” the enumeration of all derivable triples).

If the logic is complete, the set of programs that do not terminate is r.e.

Consequently, if the logic is complete, the halting problem is decidable!

An analysis of the issue

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

What is the meaning of the premises $P \Rightarrow P'$ and $Q' \Rightarrow Q$?

- "Implications that can be derived in a formal logic."
  The set of these implications is r.e., hence $\{\,P\,\}\; c\; \{\,Q\,\}$ is r.e., and the logic is incomplete.
- "Implications that are true in a standard model."
  Then $\{\,P\,\}\; c\; \{\,Q\,\}$ is not r.e. and the logic is complete (next slides).

Relative completeness

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

We can show that Hoare logic is complete, provided the same "ambient" logic is used

- to interpret the implications $P \Rightarrow P'$, $Q' \Rightarrow Q$ in the consequence rule;
- to define semantic triples
  $\{\{\, P \,\}\}\; c \;\{\{\, Q \,\}\} \stackrel{def}{=} \forall s,\; [\![P]\!]\; s \Rightarrow \texttt{Safe}\; c\; s\; Q.$

Weakest semantic precondition

We define the weakest (liberal) semantic precondition of command c with postcondition Q:

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

By definition of semantic triples, we have

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} \quad \text{if and only if} \quad P \Rightarrow \mathtt{wpsem}\ c\ Q$$

Lemma (the weakest semantic precondition is derivable)

$\{\, \mathtt{wpsem}\ c\ Q \,\}\ c\ \{\, Q \,\}$ *is derivable in Hoare logic.*

Proof.

Induction over c and "inversion" lemmas on the $\mathtt{Safe}$ predicate, such as $\mathtt{Safe}\ (c_1; c_2)\ s\ Q$ implies $\mathtt{Safe}\ c_1\ s\ (\mathtt{wpsem}\ c_2\ Q)$. □

Relative completeness

Theorem (relative completeness)

If $\{\!\{\, P \,\}\!\}\; c\; \{\!\{\, Q \,\}\!\}$ is provable in logic L,
then $\{\, P \,\}\; c\; \{\, Q \,\}$ can be derived in Hoare logic,
using L for the implications in the consequence rule.

Proof.

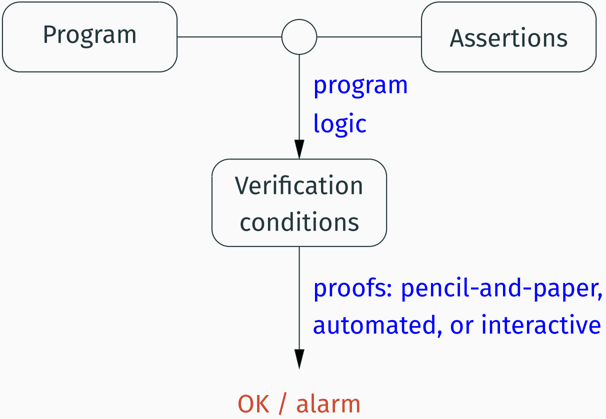
By hypothesis $\{\!\{\, P \,\}\!\}\; c\; \{\!\{\, Q \,\}\!\}$, we have $P \Rightarrow \texttt{wpsem}\; c\; Q$.

By the previous lemma, we can derive $\{\, \texttt{wpsem}\; c\; Q \,\}\; c\; \{\, Q \,\}$.

We conclude $\{\, P \,\}\; c\; \{\, Q \,\}$ with the consequence rule. $\square$

Towards automation:
weakest precondition calculus

Deductive verification (reminder)



How to generate the verification conditions?

How to minimize the amount of assertions to provide?

Fully-annotated programs

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

```
r := a;
```

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

```
q := 0;
```

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

```
while r ≥ b do
```

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

```
    r := r − b;
```

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

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

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

```
done
```

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

Verification conditions: the “$\Rightarrow$” steps where we apply the consequence rule.

Reducing the amount of annotations

To verify a program fragment c, it is enough to provide

- the precondition P
- the postcondition Q
- a loop invariant Inv for each loop in c.

The other logical assertions and the verification conditions can then be obtained by computing weakest preconditions or strongest postconditions.

Weakest precondition

The weakest precondition of a command c with postcondition Q is an assertion $\texttt{wp}\ c\ Q$ such that

- it is a precondition: $[\, \texttt{wp}\ c\ Q\,]\ c\ [\, Q\,]$;
- it is the weakest: if $[\, P\,]\ c\ [\, Q\,]$ then $P \Rightarrow \texttt{wp}\ c\ Q$.

Consequently:

$$[\, P\,]\ c\ [\, Q\,] \quad \text{if and only if} \quad P \Rightarrow \texttt{wp}\ c\ Q$$

Intuition: $\texttt{wp}\ c\ Q$ are the necessary hypotheses for program c to compute a result that satisfies the postcondition Q.

Original intuition (Dijkstra, 1975): synthesizing the program c by refinement from its postcondition Q.

Other “predicate transformers”

Weakest liberal precondition $\texttt{wlp}\ c\ Q$

Like `wp` but does not guarantee termination:

$$\{ P \}\ c\ \{ Q \} \quad \text{if and only if} \quad P \Rightarrow \texttt{wlp}\ c\ Q$$

Strongest (liberal) postcondition $\texttt{sp}\ P\ c \quad \texttt{slp}\ P\ c$

$$[P]\ c\ [Q] \quad \text{if and only if} \quad \texttt{sp}\ P\ c \Rightarrow Q$$

$$\{ P \}\ c\ \{ Q \} \quad \text{if and only if} \quad \texttt{slp}\ P\ c \Rightarrow Q$$

Intuition: symbolic execution of c from a state satisfying P.

Computing weakest preconditions

A non-effective characterization: $\texttt{wlp}\ c\ Q = \bigvee \{P \mid \{\,P\,\}\ c\ \{\,Q\,\}\}$

For programs without loops, a definition by induction over c:

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

(These equations are valid for `wp` as well.)

Weakest liberal precondition for a loop

Not computable in general: $\texttt{wlp}\ (\texttt{while}\ b\ \texttt{do}\ c)\ Q = \bigvee_i P_i$ with $P_0 = \neg b \wedge Q$ and $P_{i+1} = b \wedge \texttt{wlp}\ c\ P_i$.

We ask the programmer to annotate each loop by its invariant *Inv*. In this case,

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

provided that

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

To compute wp, the programmer should also annotate the loop by the variant that guarantees termination.

A semi-algorithm for deductive verification

To verify $\{\, P \,\}\ c\ \{\, Q \,\}$, assuming all loops in c are annotated:

1. Compute $\mathtt{wlp}\ c\ Q$ and the following verification conditions $\mathtt{vc}\ c\ Q$

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

   and likewise for the other language constructs.

2. Prove $(P \Rightarrow \mathtt{wlp}\ c\ Q) \land \mathtt{vc}\ c\ Q$, which is a proposition in ordinary logic, using an automated theorem prover.

Computing and verifying the strongest liberal postcondition

For reference, the equations defining `slp`:

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

and the nontrivial verification conditions:

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

Summary

Summary so far

Hoare logics are a rich formalism.

Two complementary viewpoints:

- The axiomatic viewpoint: the rules of the logic define the language.
- The operational viewpoint: the rules are theorems that simplify reasoning about program executions.

Extends rather easily to a great many control structures (`goto`, `break`, `return`, exceptions, procedures, …).

What about data structures? (→ next lecture)

References

References

Two presentations of Hoare logic:

- H. R. Nielson and F. Nielson, *Semantics with Applications: an appetizer*, Springer, 2007, ch. 9 and 10.
  (Follows the operational approach.)
- G. Winskel, *The Formal Semantics of Programming Languages*, MIT Press, 1993, ch. 6 and 7.
  (Follows Hoare's axiomatic approach. Comprehensive discussion of completeness issues.)

Mechanizing Hoare logic:

- The companion Coq development for this lecture:
  `https://github.com/xavierleroy/cdf-program-logics`
- T. Nipkow and G. Klein, *Concrete Semantics*, ch. 12.