Hoare logic

Lecture 3: Formalising the semantics of Hoare logic

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2018/19

Recap

In the previous lecture, we specified and verified some example programs using the syntactic rules of Hoare logic that we introduced in the first lecture.

In this lecture, we will prove the soundness of the syntactic rules, and look at some other properties of Hoare logic.

Semantics of Hoare logic

Recall: to define a Hoare logic, we need four main components:

- the programming language that we want to reason about:
  its syntax and dynamic semantics;
- an assertion language for defining state predicates:
  its syntax and an interpretation;
- an interpretation $\models$ of Hoare triples;
- a (sound) syntactic proof system $\vdash$ for deriving Hoare triples.

Dynamic semantics of WHILE

Dynamic semantics of WHILE

The dynamic semantics of WHILE will be given in the form of a small-step operational semantics (as in Part IB Semantics).

The states of the small-step operational semantics, called configurations, are pairs of a command C and a stack s.
We will abuse terminology, and also refer to s as the state.

The step relation $\langle C, s\rangle \to \langle C', s'\rangle$ expresses that configuration $\langle C, s\rangle$ can take a small step to become configuration $\langle C', s'\rangle$.

We will write $\to^*$ for the reflexive, transitive closure of $\to$.

Dynamic semantics of WHILE

Stacks are functions from variables to integers:

$$s \in \mathit{Stack} \stackrel{\mathit{def}}{=} \mathit{Var} \to \mathbb{Z}$$

These are **total** functions, and define the current value of every program variable and auxiliary variable.

This models WHILE with arbitrary precision integer arithmetic. A more realistic model might use 32-bit integers and require reasoning about overflow, etc.

Dynamic semantics of expressions: first approach

We could have two small-step reduction relations for arithmetic expressions and boolean expressions, $\langle E, s\rangle \to \langle E', s'\rangle$ and $\langle B, s\rangle \to \langle B', s'\rangle$:

$$\langle X, s\rangle \to \langle s(X), s\rangle \qquad \frac{N_1 + N_2 = N}{\langle N_1 + N_2, s\rangle \to \langle N, s\rangle}$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \to \langle E_1' + E_2, s'\rangle} \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle N_1 + E_2, s\rangle \to \langle N_1 + E_2', s'\rangle}$$

$$\ldots$$

Dynamic semantics of expressions: our approach

However, expressions in WHILE do not change the stack, and do not get stuck:

$$\forall E, s.\ \exists N.\ \langle E, s\rangle \to^* \langle N, s\rangle$$

(and the equivalent for B).

We take advantage of this, and specify the dynamic semantics of expressions in a way which makes our setup easier (the results are the same, but execution can take fewer steps):

We use functions $\mathcal{E}[\![E]\!](s)$ and $\mathcal{B}[\![B]\!](s)$ to evaluate arithmetic expressions and boolean expressions in a given stack s, respectively.

Semantics of expressions

$\mathcal{E}[\![E]\!](s)$ evaluates arithmetic expression E to an integer in stack s:

$$\mathcal{E}[\![-]\!](=) : \mathit{Exp} \times \mathit{Stack} \to \mathbb{Z}$$

$$\begin{aligned}
\mathcal{E}[\![N]\!](s) &\stackrel{def}{=} N \\
\mathcal{E}[\![X]\!](s) &\stackrel{def}{=} s(X) \\
\mathcal{E}[\![E_1 + E_2]\!](s) &\stackrel{def}{=} \mathcal{E}[\![E_1]\!](s) + \mathcal{E}[\![E_2]\!](s) \\
&\vdots
\end{aligned}$$

This semantics is too simple to handle operations such as division, which fails to evaluate to an integer on some inputs.

For example, if $s(X) = 3$ and $s(Y) = 0$, then
$\mathcal{E}[\![X + 2]\!](s) = \mathcal{E}[\![X]\!](s) + \mathcal{E}[\![2]\!](s) = 3 + 2 = 5$, and
$\mathcal{E}[\![Y + 4]\!](s) = \mathcal{E}[\![Y]\!](s) + \mathcal{E}[\![4]\!](s) = 0 + 4 = 4$.

Semantics of boolean expressions

$\mathcal{B}[\![B]\!](s)$ evaluates boolean expression B to a boolean in stack s:

$$\mathcal{B}[\![-]\!](=) : BExp \times Stack \to \mathbb{B}$$

$$\mathcal{B}[\![\mathbf{T}]\!](s) \stackrel{def}{=} \top$$

$$\mathcal{B}[\![\mathbf{F}]\!](s) \stackrel{def}{=} \bot$$

$$\mathcal{B}[\![E_1 \leq E_2]\!](s) \stackrel{def}{=} \begin{cases} \top & \text{if } \mathcal{E}[\![E_1]\!](s) \leq \mathcal{E}[\![E_2]\!](s) \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

For example, if $s(X) = 3$ and $s(Y) = 0$, then
$\mathcal{B}[\![X + 2 \geq Y + 4]\!](s) = \mathcal{E}[\![X + 2]\!](s) \geq \mathcal{E}[\![Y + 4]\!](s) = 5 \geq 4 = \top$.

Small-step operational semantics of WHILE

$$\frac{\mathcal{E}[\![E]\!](s) = N}{\langle X := E, s\rangle \to \langle \textbf{skip}, s[X \mapsto N]\rangle}$$

$$\frac{}{\langle \textbf{skip}; C_2, s\rangle \to \langle C_2, s\rangle} \qquad \frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_1, s\rangle} \qquad \frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_2, s\rangle}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \textbf{while } B \textbf{ do } C, s\rangle \to \langle \textbf{skip}, s\rangle}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top}{\langle \textbf{while } B \textbf{ do } C, s\rangle \to \langle C; \textbf{while } B \textbf{ do } C, s\rangle}$$

Properties of WHILE

Safety and determinacy

A configuration $\langle C, s\rangle$ is stuck, written $\langle C, s\rangle \not\to$, when $\forall C', s'. \neg(\langle C, s\rangle \to \langle C', s'\rangle)$.

The dynamic semantics of WHILE is safe, in that a configuration is stuck exactly when its command is **skip**:

$$(\langle C, s\rangle \not\to) \Leftrightarrow C = \textbf{skip}$$

This is true for any syntactically well-formed command, without any further typing! (Because our language is very simple.)

The dynamic semantics of WHILE is deterministic:

$$\langle C, s\rangle \to \langle C', s'\rangle \wedge \langle C, s\rangle \to \langle C'', s''\rangle \;\Rightarrow\; C'' = C' \wedge s'' = s'$$

Non-termination

It is possible to have an infinite sequence of steps starting from a configuration $\langle C, s\rangle$: $\langle C, s\rangle$ has a non-terminating execution (also "can diverge"), written $\langle C, s\rangle \to^\omega$, when there exists a sequence of commands $(C_n)_{n\in\mathbb{N}}$ and a sequence of stacks $(s_n)_{n\in\mathbb{N}}$ such that

$$C_0 = C \wedge s_0 = s \wedge \forall n \in \mathbb{N}.\, \langle C_n, s_n\rangle \to \langle C_{n+1}, s_{n+1}\rangle$$

Note that

$$\langle C, s\rangle \to^\omega \;\Leftrightarrow\; \exists C', s'.\, \langle C, s\rangle \to \langle C', s'\rangle \wedge \langle C', s'\rangle \to^\omega$$

Because WHILE is safe and deterministic, a configuration can take steps to **skip** if and only if it does not diverge:

$$(\exists s'.\, \langle C, s\rangle \to^* \langle \textbf{skip}, s'\rangle) \Leftrightarrow \neg(\langle C, s\rangle \to^\omega)$$

This can break down with a non-deterministic language.

Substitution

We use $E_1[E_2/X]$ to denote E_1 with E_2 substituted for every occurrence of program variable X:

$$-[= / \equiv] : \mathit{Expr} \times \mathit{Expr} \times \mathit{Var} \to \mathit{Expr}$$

$$N[E_2/X] \stackrel{def}{=} N$$

$$Y[E_2/X] \stackrel{def}{=} \begin{cases} \text{if } Y = X & E_2 \\ \text{if } Y \neq X & Y \end{cases}$$

$$(E_a + E_b)[E_2/X] \stackrel{def}{=} (E_a[E_2/X]) + (E_b[E_2/X])$$

$$\vdots$$

For example, $(X + (Y \times 2))[3 + Z/Y] = X + ((3 + Z) \times 2)$.

Substitution property for expressions

We will use the following expression substitution property later:

$$\mathcal{E}[\![E_1[E_2/X]]\!](s) = \mathcal{E}[\![E_1]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])$$

The expression substitution property follows by induction on E_1.

Case $E_1 \equiv N$:

$$\mathcal{E}[\![N[E_2/X]]\!](s) = \mathcal{E}[\![N]\!](s) = N = \mathcal{E}[\![N]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Proof of substitution property: variable case

$$\mathcal{E}[\![E_1[E_2/X]]\!](s) = \mathcal{E}[\![E_1]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Case $E_1 \equiv Y$:

$$\mathcal{E}[\![Y[E_2/X]]\!](s)$$

$$= \begin{cases} \text{if } Y = X & \mathcal{E}[\![X[E_2/X]]\!](s) = \mathcal{E}[\![E_2]\!](s) = \mathcal{E}[\![X]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)]) \\ \text{if } Y \neq X & \mathcal{E}[\![Y]\!](s) = s(Y) = \mathcal{E}[\![Y]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)]) \end{cases}$$

$$= \mathcal{E}[\![Y]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Proof of substitution property: addition case

$$\mathcal{E}[\![E_1[E_2/X]]\!](s) = \mathcal{E}[\![E_1]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Case $E_1 \equiv E_a + E_b$:

$$\begin{aligned}
&\mathcal{E}[\![(E_a + E_b)[E_2/X]]\!](s) \\
&\quad = \mathcal{E}[\![(E_a[E_2/X]) + (E_b[E_2/X])]\!](s) \\
&\quad = \mathcal{E}[\![E_a[E_2/X]]\!](s) + \mathcal{E}[\![E_b[E_2/X]]\!](s) \\
&\quad = \mathcal{E}[\![E_a]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)]) + \mathcal{E}[\![E_b]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)]) \\
&\quad = \mathcal{E}[\![E_a + E_b]\!](s[X \mapsto \mathcal{E}[\![E_2]\!](s)])
\end{aligned}$$

Semantics of assertions

The language of assertions

Now, we have formally defined the dynamic semantics of the WHILE language that we wish to reason about.

The next step is to formalise the assertion language that we will use to describe and reason about states of WHILE programs.

We take the language of assertions to be (slight variation of) an instance of single-sorted first-order logic with equality (as in Part IB Logic and Proof).

The assertion language

The formal syntax of the assertion language is given below:

$$
\begin{array}{rcll}
\chi & ::= & X \mid x & \textit{variables} \\
t & ::= & \chi \mid f(t_1, ..., t_n) \quad n \geq 0 & \textit{terms} \\
P, Q & ::= & \bot \mid \top \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q & \textit{assertions} \\
 & \mid & \forall x.\, P \mid \exists x.\, P \mid t_1 = t_2 \mid p(t_1, ..., t_n) \quad n \geq 0 & \\
\neg P & \stackrel{def}{=} & P \Rightarrow \bot &
\end{array}
$$

Quantifiers quantify over terms, and only bind logical variables.

Here f and p range over an unspecified set of function symbols and predicate symbols, respectively, that includes (symbols for) the usual mathematical functions and predicates on integers.
In particular, we assume that they contain symbols that allows us to embed arithmetic expressions E as terms, and boolean expressions B as assertions.

Semantics of terms

$[\![t]\!](s)$ defines the semantics of a term t in a stack s:

$$[\![-]\!](=) : \mathit{Term} \times \mathit{Stack} \to \mathbb{Z}$$

$$[\![\chi]\!](s) \stackrel{def}{=} s(\chi)$$

$$[\![f(t_1, ..., t_n)]\!](s) \stackrel{def}{=} [\![f]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))$$

We assume that the appropriate function $[\![f]\!]$ associated to each function symbol f is provided along with the implicit signature.

In particular, we have $[\![E]\!](s) = \mathcal{E}[\![E]\!](s)$.

Semantics of assertions

$[\![P]\!]$ defines the set of stacks that satisfy the assertion P:

$$[\![-]\!] : \mathit{Assertion} \to \mathcal{P}(\mathit{Stack})$$

$$[\![\bot]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \bot\} = \emptyset$$

$$[\![\top]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \top\} = \mathit{Stack}$$

$$[\![P \vee Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \vee s \in [\![Q]\!]\} = [\![P]\!] \cup [\![Q]\!]$$

$$[\![P \wedge Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \wedge s \in [\![Q]\!]\} = [\![P]\!] \cap [\![Q]\!]$$

$$[\![P \Rightarrow Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \Rightarrow s \in [\![Q]\!]\}$$

(continued)

Semantics of assertions (continued)

$$[\![t_1 = t_2]\!] \stackrel{def}{=} \{s \in Stack \mid [\![t_1]\!](s) = [\![t_2]\!](s)\}$$

$$[\![p(t_1, ..., t_n)]\!] \stackrel{def}{=} \{s \in Stack \mid [\![p]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))\}$$

$$[\![\forall x.\, P]\!] \stackrel{def}{=} \{s \in Stack \mid \forall N.\, s[x \mapsto N] \in [\![P]\!]\}$$

$$[\![\exists x.\, P]\!] \stackrel{def}{=} \{s \in Stack \mid \exists N.\, s[x \mapsto N] \in [\![P]\!]\}$$

We assume that the appropriate predicate $[\![p]\!]$ associated to each predicate symbol p is provided along with the implicit signature.

In particular, we have $[\![B]\!] = \{s \mid \mathcal{B}[\![B]\!](s) = \top\}$.

We could write $s \models P$ for $s \in [\![P]\!]$.

Substitutions

We use $t[E/X]$ and $P[E/X]$ to denote t and P with E substituted for every occurrence of program variable X, respectively.

Since our quantifiers bind logical variables, and all free variables in E are program variables, there is no issue with variable capture:

$$(\forall x.\, P)[E/X] \stackrel{def}{=} \forall x.\, (P[E/X])$$

$$\vdots$$

Substitution property

The term and assertion semantics satisfy a similar substitution property to the expression semantics:

- $[\![t[E/X]]\!](s) = [\![t]\!](s[X \mapsto \mathcal{E}[\![E]\!](s)])$
- $s \in [\![P[E/X]]\!] \Leftrightarrow s[X \mapsto \mathcal{E}[\![E]\!](s)] \in [\![P]\!]$

They are easily provable by induction on t and P, respectively: the former by using the substitution property for expressions, and the latter by using the former. (Exercise)

The latter property will be useful in the proof of soundness of the syntactic assignment rule.

Semantics of Hoare triples

Semantics of partial correctness triples

Now that we have formally defined the dynamic semantics of WHILE and our assertion language, we can define the formal meaning of our triples.

A partial correctness triple asserts that if the given command terminates when executed from an initial state that satisfies the precondition, then the terminal state must satisfy the postcondition:

$$\models \{P\}\ C\ \{Q\} \quad \stackrel{def}{=} \quad \forall s, s'.\, s \in [\![P]\!] \wedge \langle C, s\rangle \rightarrow^* \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![Q]\!]$$

Without safety, we would have to worry about getting stuck without reaching **skip**.

Soundness of Hoare logic

Soundness of Hoare logic

Theorem (Soundness)
If $\vdash \{P\}\ C\ \{Q\}$ *then* $\models \{P\}\ C\ \{Q\}$.

Soundness expresses that any triple derivable using the syntactic proof system holds semantically.

Soundness can be proved by induction on the $\vdash \{P\}\ C\ \{Q\}$ derivation:

- it suffices to show, for each inference rule, that if each hypothesis holds semantically, then the conclusion holds semantically.

Soundness of the assignment rule

$$\models \{P[E/X]\}\ X := E\ \{P\}$$

Assume $s \in [\![P[E/X]]\!]$ and $\langle X := E, s\rangle \to^* \langle \textbf{skip}, s'\rangle$.

From the substitution property, it follows that
$s[X \mapsto \mathcal{E}[\![E]\!](s)] \in [\![P]\!]$.

From inversion on the steps, there exists an N such that
$\mathcal{E}[\![E]\!](s) = N$ and $s' = s[X \mapsto N]$, so $s' = s[X \mapsto \mathcal{E}[\![E]\!](s)]$.

Hence, $s' \in [\![P]\!]$.

Soundness of the loop rule

If $\models \{P \wedge B\}\ C\ \{P\}$ then $\models \{P\}$ **while** B **do** $C\ \{P \wedge \neg B\}$

How can we get past the fact that the loop step rule defines the steps of a loop in terms of the steps of a loop?

We will prove $\models \{P\}$ **while** B **do** $C\ \{P \wedge \neg B\}$ by proving a modified version of the property.

We write $\langle C, s\rangle \rightarrow^k \langle C', s'\rangle$ to mean $\langle C, s\rangle$ can take k steps, where $k \geq 0$, to reach $\langle C', s'\rangle$.

Soundness of the loop rule: base case

If (IH) $\forall s, s'.\, s \in [\![P \wedge B]\!] \wedge \langle C, s\rangle \to^* \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P]\!]$,
then $\forall n > 0.\, \forall k < n.\, \forall s, s'.\, s \in [\![P]\!] \wedge$
$\langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P \wedge \neg B]\!]$

We can prove this by a (nested) induction on n:

Case 1: assume $s \in [\![P]\!]$, $k < 1$, and
$\langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle$.

Then **while** B **do** $C =$ **skip**, so we have a contradiction.

Soundness of the loop rule: inductive case

If (IH) $\forall s, s'.\, s \in [\![P \wedge B]\!] \wedge \langle C, s\rangle \to^* \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P]\!]$,
then $\forall n > 0.\, \forall k < n.\, \forall s, s'.\, s \in [\![P]\!] \wedge$
$\langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P \wedge \neg B]\!]$

Case $n+1$: assume $s \in [\![P]\!]$, $k < n+1$,
$\langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle$, and
(nIH) $\forall k < n.\, \forall s, s'.\, s \in [\![P]\!] \wedge \langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle \Rightarrow$
$s' \in [\![P \wedge \neg B]\!]$.

If $k = 0$, it is as before.
If $k = 1$, B must have evaluated to false: $\mathcal{B}[\![B]\!](s) = \bot$ and $s' = s$.
Since $\mathcal{B}[\![B]\!](s) = \bot$, $s \notin [\![B]\!]$, so $s \in [\![B]\!] \Rightarrow s \in [\![\bot]\!]$, so
$s \in [\![B \Rightarrow \bot]\!]$, so $s \in [\![\neg B]\!]$. Therefore, $s \in [\![P \wedge \neg B]\!]$.
Hence, $s' = s \in [\![P \wedge \neg B]\!]$.

Soundness of the loop rule: inductive case (continued)

If (IH) $\forall s, s'.\, s \in [\![P \wedge B]\!] \wedge \langle C, s\rangle \to^* \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P]\!]$, then $\forall n > 0.\, \forall k < n.\, \forall s, s'.\, s \in [\![P]\!] \wedge \langle \textbf{while } B \textbf{ do } C, s\rangle \to^k \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![P \wedge \neg B]\!]$

If $k > 1$, B must have evaluated to true: $\mathcal{B}[\![B]\!](s) = \top$, and there exists s^*, k_1, and k_2 such that $\langle C, s\rangle \to^{k_1} \langle \textbf{skip}, s^*\rangle$, $\langle \textbf{while } B \textbf{ do } C, s^*\rangle \to^{k_2} \langle \textbf{skip}, s'\rangle$, and $k = k_1 + k_2 + 2$.
Since $\mathcal{B}[\![B]\!](s) = \top$, $s \in [\![B]\!]$. Therefore, $s \in [\![P \wedge B]\!]$.

From the outer induction hypothesis IH, it follows that $s^* \in [\![P]\!]$, and so by the inner induction hypothesis nIH, $s' \in [\![P \wedge \neg B]\!]$.

Other properties of Hoare logic

Completeness

Completeness is the converse property of soundness:
If $\models \{P\}\ C\ \{Q\}$ then $\vdash \{P\}\ C\ \{Q\}$.

Our Hoare logic inherits the incompleteness of arithmetic and is therefore **not** complete.

Completeness

To see why, assume that $\models \{P\}\ C\ \{Q\} \Rightarrow \vdash \{P\}\ C\ \{Q\}$.

We can then show that our assertion logic is complete:
Assume $\models P$, that is, $\forall s.\, s \in [\![P]\!]$.
Then $\models \{\top\}$ **skip** $\{P\}$.
Using completeness, we can derive $\vdash \{\top\}$ **skip** $\{P\}$.
Then, by examining that derivation, we have a derivation of $\vdash_{FOL} \top \Rightarrow P$, and hence a derivation of $\vdash_{FOL} P$.

But the assertion logic includes arithmetic, and is therefore **not** complete, so we have a contradiction.

Relative completeness

The previous argument showed that because the assertion logic is not complete, then neither is Hoare logic.

However, Hoare logic is **relatively complete** for our simple language:

- Relative completeness expresses that any failure to derive $\vdash \{P\}\ C\ \{Q\}$ for a statement that holds semantically can be traced back to a failure to prove $\vdash_{FOL} R$ for some valid arithmetic statement R.

In practice, completeness is not that important, and there is more focus on nice, usable rules.

Decidability

Finally, Hoare logic is not decidable: there there does not exist a computable function f such that

$$f(P, C, Q) = \top \quad \Leftrightarrow \quad \models \{P\}\ C\ \{Q\}$$

$\models \{\top\}\ C\ \{\bot\}$ holds if and only if C does not terminate.
Moreover, we can encode Turing machines in WHILE.
Hence, since the Halting problem is undecidable, so is Hoare logic.

Other perspectives on Hoare triples

Other perspectives on Hoare triples

So far, we have assumed P, C, and Q were given, and focused on proving $\vdash \{P\}\ C\ \{Q\}$.

Recall, if P and Q are assertions, P is stronger than Q, and Q is weaker than P, when $\vdash_{FOL} P \Rightarrow Q$.

If we are given P and C, can we infer a Q?
Is there a best such Q, $sp(P, C)$? ('strongest postcondition')

Symmetrically, if we are given C and Q, can we infer a P?
Is there a best such P, $wlp(C, Q)$? ('weakest liberal precondition')

Are there functions wlp and sp such that

$$(\vdash_{FOL} P \Rightarrow wlp(C, Q)) \Leftrightarrow \vdash \{P\}\ C\ \{Q\} \Leftrightarrow (\vdash_{FOL} sp(P, C) \Rightarrow Q)$$

Terminology

We write *wlp* and talk about weakest **liberal** precondition because we only consider partial correctness.

This has no relevance here because, as we will see, there is no effective general finite (first-order) formula for weakest preconditions, liberal or not, or strongest postconditions, for commands containing loops, so we will not consider weakest preconditions, liberal or not, for loops, so there is no difference between partial and total correctness.

Computing weakest liberal preconditions (except for loops)

Dijkstra gives rules for computing weakest liberal preconditions for deterministic loop-free code:

$$
\begin{aligned}
wlp(\textbf{skip}, Q) &= Q \\
wlp(X := E, Q) &= Q[E/X] \\
wlp(C_1; C_2, Q) &= wlp(C_1, wlp(C_2, Q)) \\
wlp(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, Q) &= (B \Rightarrow wlp(C_1, Q)) \wedge \\
&\quad (\neg B \Rightarrow wlp(C_2, Q))
\end{aligned}
$$

These rules are suggested by the relative completeness of the Hoare logic proof rules from the first lecture.

Example of weakest liberal precondition computation

$$
\begin{aligned}
& wlp(X := X + 1; Y := Y + X,\ \exists m, n.\, X = 2 \times m \wedge Y = 2 \times n) \\
= {} & wlp(X := X + 1,\ wlp(Y := Y + X,\ \exists m, n.\, X = 2 \times m \wedge Y = 2 \times n)) \\
= {} & wlp(X := X + 1,\ (\exists m, n.\, X = 2 \times m \wedge Y = 2 \times n)[Y + X/Y]) \\
= {} & wlp(X := X + 1,\ \exists m, n.\, X = 2 \times m \wedge Y + X = 2 \times n) \\
= {} & (\exists m, n.\, X = 2 \times m \wedge Y + X = 2 \times n)[X + 1/X] \\
= {} & \exists m, n.\, X + 1 = 2 \times m \wedge Y + (X + 1) = 2 \times n \\
\Leftrightarrow {} & \exists m, n.\, X = 2 \times m + 1 \wedge Y = 2 \times n
\end{aligned}
$$

Weakest preconditions for loops

While the following property holds for loops

$$
\begin{array}{l}
\mathit{wlp}(\textbf{while } B \textbf{ do } C, Q) \Leftrightarrow \\
\quad \mathit{wlp}(\textbf{if } B \textbf{ then } (C; \textbf{while } B \textbf{ do } C) \textbf{ else skip}, Q) \Leftrightarrow \\
\quad (B \Rightarrow \mathit{wlp}(C, \mathit{wlp}(\textbf{while } B \textbf{ do } C, Q))) \wedge (\neg B \Rightarrow Q)
\end{array}
$$

it does not define $\mathit{wlp}(\textbf{while } B \textbf{ do } C, Q)$ as a finite formula in first-order logic.

There is no general finite formula for $\mathit{wlp}(\textbf{while } B \textbf{ do } C, Q)$ in first-order logic. (Otherwise, it would be easy to find invariants!)

Verification condition generation

We can now sketch the design of a verification condition generation algorithm.

(1) The precondition needs to imply the approximate weakest liberal precondition induced by the provided loop invariants.

(2) Moreover, the provided loop invariants need to be actual loop invariants, and, together with the guard not holding, need to imply the loop postcondition.

These can be computed mutually recursively.

Summary

We have defined a dynamic semantics for the WHILE language, and a formal semantics for a Hoare logic for WHILE.

We have shown that the syntactic proof system from the first lecture is sound with respect to this semantics, but not complete.

Supplementary reading on soundness and completeness:

- Glynn Winskel. The Formal Semantics of Programming Languages: An Introduction. Chapters 6–7.
- Software Foundations, Benjamin C. Pierce et al.

In the next lecture, we will look at extending Hoare logic to reason about pointers.

Not examinable: Verification condition generation

We can define annotated programs:

$$\begin{array}{rcl} \mathcal{C} & ::= & \textbf{skip} \\ & | & \mathcal{C}_1; \mathcal{C}_2 \\ & | & X := E \\ & | & \textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \\ & | & \textbf{while } B \textbf{ do } \{I\}\ \mathcal{C} \end{array}$$

and an erasure function:

$$|\textbf{skip}| \stackrel{def}{=} \textbf{skip}$$

$$|\mathcal{C}_1; \mathcal{C}_2| \stackrel{def}{=} |\mathcal{C}_1|; |\mathcal{C}_2|$$

$$|X := E| \stackrel{def}{=} X := E$$

$$|\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2| \stackrel{def}{=} \textbf{if } B \textbf{ then } |\mathcal{C}_1| \textbf{ else } |\mathcal{C}_2|$$

$$|\textbf{while } B \textbf{ do } \{I\}\ \mathcal{C}| \stackrel{def}{=} \textbf{while } B \textbf{ do } |\mathcal{C}|$$

Not examinable: Computing verification conditions 1/2

We can then define our verification condition generation function

$$VC(P, \mathcal{C}, Q) \stackrel{def}{=} \{P \Rightarrow awlp(\mathcal{C}, Q)\} \cup VCaux(\mathcal{C}, Q)$$

using (1) an approximation of weakest liberal precondition that approximates loops using the provided invariants

$$awlp(\textbf{skip}, Q) \stackrel{def}{=} Q$$

$$awlp(X := E, Q) \stackrel{def}{=} Q[E/X]$$

$$awlp(\mathcal{C}_1; \mathcal{C}_2, Q) \stackrel{def}{=} awlp(\mathcal{C}_1, awlp(\mathcal{C}_2, Q))$$

$$awlp(\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2, Q) \stackrel{def}{=} (B \Rightarrow awlp(C_1, Q)) \wedge (\neg B \Rightarrow awlp(C_2, Q))$$

$$awlp(\textbf{while } B \textbf{ do } \{I\}\ \mathcal{C}, Q) \stackrel{def}{=} I$$

Not examinable: Computing verification conditions 2/2

(2) an auxiliary function that collects side-conditions of loops:

$$VCaux(\textbf{skip}, Q) \stackrel{def}{=} \emptyset$$

$$VCaux(X := E, Q) \stackrel{def}{=} \emptyset$$

$$VCaux(\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2, Q) \stackrel{def}{=} VCaux(\mathcal{C}_1, Q) \cup VCaux(\mathcal{C}_2, Q)$$

$$VCaux(\mathcal{C}_1; \mathcal{C}_2, Q) \stackrel{def}{=} VCaux(\mathcal{C}_1, awlp(\mathcal{C}_2, Q)) \cup VCaux(\mathcal{C}_2, Q)$$

$$VCaux(\textbf{while } B \textbf{ do } \{I\}\ \mathcal{C}, Q) \stackrel{def}{=} \{I \wedge \neg B \Rightarrow Q, I \wedge B \Rightarrow awlp(\mathcal{C}, I)\} \cup VCaux(\mathcal{C}, I)$$