Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Acknowledgements

These slides are heavily based on previous versions by Mike Gordon, Alan Mycroft, and Kasper Svendsen.

Thanks to Mistral Contrastin, Victor Gomes, Joe Isaacs, Ian Orton, and Domagoj Stolfa for reporting mistakes.

Motivation

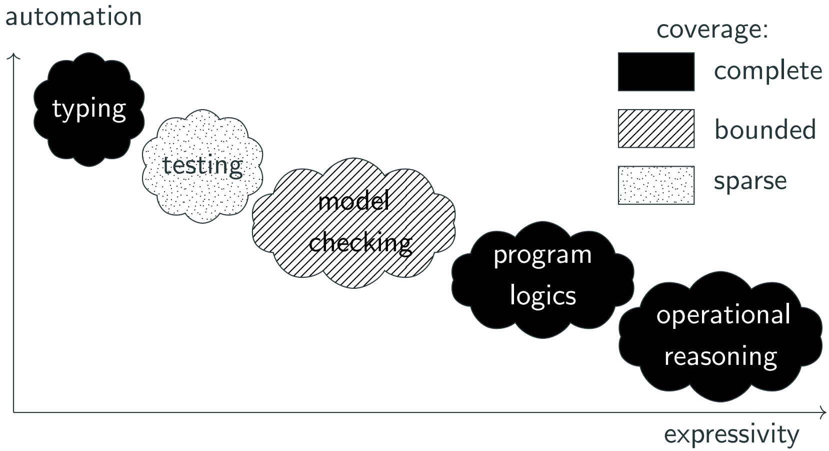
We often fail to write programs that meet our expectations, which we phrased in their specifications:

- we fail to write programs that meet their specification;
- we fail to write specifications that meet our expectations.

Addressing the former issue is called verification, and addressing the latter is called validation.

Background

There are many verification & validation techniques of varying coverage, expressivity, level of automation, ..., for example:



Choice of technique

More expressive and complete techniques lead to more confidence.

It is important to choose the right set of verification & validation techniques for the task at hand:

- verified designs may still not work;
- verification can give a false sense of security;
- verification can be very expensive and time-consuming.

More heavyweight techniques should be used together with testing, not as a replacement.

Course structure

This course is about two techniques, their underlying ideas, how to use them, and why they are correct:

- **Hoare logic** (Lectures 1-6);
- **Model checking** (Lectures 7-12).

These are not just techniques, but also ways of thinking about programs.

Lecture plan

Lecture 1: Informal introduction to Hoare logic

Lecture 2: Formal semantics of Hoare logic

Lecture 3: Examples, loop invariants, and total correctness

Lecture 4: Mechanised program verification

Lecture 5: Separation logic

Lecture 6: Examples in separation logic

Hoare logic

Hoare logic

Hoare logic is a formalism for relating the **initial** and **terminal** state of a program.

Hoare logic was invented in 1969 by Tony Hoare, inspired by earlier work of Robert Floyd.
There was little-known prior work by Alan Turing.

Hoare logic is still an active area of research.

Partial correctness triples

Hoare logic uses **partial correctness triples** (also “Hoare triples”) for specifying and reasoning about the behaviour of programs:

$$\{P\}\ C\ \{Q\}$$

is a logical statement about a command C,
where P and Q are state predicates:

- P is called the precondition, and describes the initial state;
- Q is called the postcondition, and describes the terminal state.

Components of a Hoare logic

To define a Hoare logic, we need four main components:

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

This lecture will introduce each component informally.
In the coming lectures, we will cover the formal details.

The WHILE language

Commands of the WHILE language

WHILE is the prototypical imperative language. Programs consist of commands, which include branching, iteration, and assignment:

$$
\begin{array}{rcl}
C & ::= & \textbf{skip} \\
  & \mid & C_1; C_2 \\
  & \mid & V := E \\
  & \mid & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \\
  & \mid & \textbf{while } B \textbf{ do } C
\end{array}
$$

Here, V is a variable, E is an arithmetic expression, which evaluates to an integer, and B is a boolean expression, which evaluates to a boolean.

States are mappings from variables to integers.

Expressions of the WHILE language

The grammar for arithmetic expressions and boolean expressions includes the usual arithmetic operations and comparison operators, respectively:

$$\begin{array}{lcll} E & ::= & N \mid V \mid E_1 + E_2 & \textit{arithmetic expressions} \\ & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \end{array}$$

$$\begin{array}{lcll} B & ::= & \mathbf{T} \mid \mathbf{F} \mid E_1 = E_2 & \textit{boolean expressions} \\ & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots & \end{array}$$

Note that expressions do not have side effects.

Assertions and specifications

The assertion language

Assertions (also "state predicates") $P, Q, \ldots$ include boolean expressions (which can contain program variables), combined using the usual logical operators: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$, $\exists$, $\ldots$

For instance, the predicate $X = Y + 1 \wedge Y > 0$ describes states in which the variable Y contains a positive value, and the value of X is equal to the value of Y plus 1.

Informal semantics of partial correctness triples

The partial correctness triple $\{P\}\ C\ \{Q\}$ holds if and only if:

- assuming C is executed in an initial state satisfying P,
- and assuming moreover that this execution terminates,
- then the terminal state of the execution satisfies Q.

For instance,

- $\{X = 1\}\ X := X + 1\ \{X = 2\}$ holds;
- $\{X = 1\}\ X := X + 1\ \{X = 3\}$ does not hold.

Partial correctness

Partial correctness triples are called **partial** because they only specify the intended behaviour of terminating executions.

For instance, $\{X = 1\}$ **while** $X > 0$ **do** $X := X + 1$ $\{X = 0\}$ holds, because the given program never terminates when executed from an initial state where X is 1.

Hoare logic also features total correctness triples that strengthen the specification to require termination.

Informal semantics of total correctness

There is no standard notation for total correctness triples; we will use $[P]\ C\ [Q]$.

The total correctness triple $[P]\ C\ [Q]$ holds if and only if:

- assuming C is executed in an initial state satisfying P,
- then the execution terminates,
- and the terminal state satisfies Q.

Total correctness

The following total correctness triple does not hold:

$$[X = 1]\ \textbf{while}\ X > 0\ \textbf{do}\ X := X + 1\ [X = 0]$$

- the loop never terminates when executed from an initial state where X is positive.

The following total correctness triple does hold:

$$[X = 0]\ \textbf{while}\ X > 0\ \textbf{do}\ X := X + 1\ [X = 0]$$

- the loop always terminates immediately when executed from an initial state where X is zero.

Total correctness, partial correctness, and termination

Informally: total correctness = partial correctness + termination.

It is often easier to show partial correctness and termination separately.

Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of X:

```
while $X > 1$ do
    if $ODD(X)$ then $X := 3 \times X + 1$ else $X := X\ DIV\ 2$
```

Microsoft's T2 tool is used to prove termination of systems code.

Examples of specifications

Corner cases of partial correctness triples

$\{\bot\}\ C\ \{Q\}$

- this says nothing about the behaviour of C, because $\bot$ never holds for any initial state.

$\{\top\}\ C\ \{Q\}$

- this says that whenever C halts, Q holds.

$\{P\}\ C\ \{\top\}$

- this holds for every precondition P and command C, because $\top$ always holds in the terminate state.

Corner cases of total correctness triples

$[P]\ C\ [\top]$

- this says that C always terminates when executed from an initial state satisfying P.

$[\top]\ C\ [Q]$

- this says that C always terminates, and ends up in a state where Q holds.

The need for auxiliary variables

How can we specify that a program C computes the maximum of two variables X and Y, and stores the result in a variable Z?

Is this a good specification for C?

$$\{\top\}\ C\ \{(X \leq Y \Rightarrow Z = Y) \wedge (Y \leq X \Rightarrow Z = X)\}$$

No! Take C to be

$$X := 0; Y := 0; Z := 0$$

Then C satisfies the above specification!

The postcondition should refer to the **initial** values of X and Y.

Auxiliary variables

In Hoare logic, we use **auxiliary variables** (also “ghost variables”, or “logical variables”), which are not allowed not occur in the program, to refer to the initial values of variables in postconditions.

Notation: program variables are uppercase, and auxiliary variables are lowercase. v ranges over auxiliary variables, and concrete values are $x, y, \ldots$.

For instance, $\{X = x \wedge Y = y\}\ C\ \{X = y \wedge Y = x\}$ expresses that if C terminates, then it exchanges the values of variables X and Y.

Formal proof system for Hoare logic

Hoare logic

We will now introduce a natural deduction proof system for partial correctness triples due to Tony Hoare.

The logic consists of a set of **inference rule schemas** for deriving consequences from premises.

If S is a statement, we will write $\vdash S$ to mean that the statement S is derivable. We will have two derivability judgements:

- $\vdash P$, for derivability of assertions; and
- $\vdash \{P\}\ C\ \{Q\}$, for derivability of partial correctness triples.

Inference rule schemas

The inference rule schemas of Hoare logic will be specified as follows:

$$\frac{\vdash S_1 \qquad \cdots \qquad \vdash S_n}{\vdash S}$$

This expresses that S may be deduced from assumptions S_1, ..., S_n.

These are schemas that may contain meta-variables.

Proof trees

A proof tree for $\vdash S$ in Hoare logic is a tree with $\vdash S$ at the root, constructed using the inference rules of Hoare logic, where all nodes are shown to be derivable (so leaves require no further derivations):

$$\dfrac{\dfrac{\dfrac{}{\vdash S_1} \qquad \dfrac{}{\vdash S_2}}{\vdash S_3} \qquad \dfrac{}{\vdash S_4}}{\vdash S}$$

We typically write proof trees with the root at the bottom.

Formal proof system for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}} \qquad \frac{}{\vdash \{P[E/V]\}\ V := E\ \{P\}}$$

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

$$\frac{\vdash \{P \land B\}\ C_1\ \{Q\} \qquad \vdash \{P \land \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

$$\frac{\vdash \{P \land B\}\ C\ \{P\}}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \land \neg B\}}$$

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \vdash \{P_2\}\ C\ \{Q_2\} \qquad \vdash Q_2 \Rightarrow Q_1}{\vdash \{P_1\}\ C\ \{Q_1\}}$$

The skip rule

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

The **skip** rule expresses that any assertion that holds before **skip** is executed also holds afterwards.

P is a meta-variable ranging over an arbitrary state predicate.

For instance, $\vdash \{X = 1\}\ \textbf{skip}\ \{X = 1\}$.

The assignment rule

$$\frac{}{\vdash \{P[E/V]\}\ V := E\ \{P\}}$$

Here, $P[E/V]$ means the assertion P with the expression E substituted for all occurrences of the variable V.

For instance,

$$\vdash \{X + 1 = 2\}\ X := X + 1\ \{X = 2\}$$

$$\vdash \{Y + X = Y + 10\}\ X := Y + X\ \{X = Y + 10\}$$

The assignment rule

The assignment rule reads right-to-left; could we use another rule that reads more easily?

Consider the following plausible alternative assignment rule:

$$\frac{}{\vdash \{P\}\ V := E\ \{P[E/V]\}}$$

We can instantiate this rule to obtain the following triple, which does not hold:

$$\{X = 0\}\ X := 1\ \{1 = 0\}$$

The rule of consequence

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \vdash \{P_2\}\ C\ \{Q_2\} \qquad \vdash Q_2 \Rightarrow Q_1}{\vdash \{P_1\}\ C\ \{Q_1\}}$$

The rule of consequence allows us to strengthen preconditions and weaken postconditions.

Note: the $\vdash P \Rightarrow Q$ hypotheses are a different kind of judgment.

For instance, from $\vdash \{X + 1 = 2\}\ X := X + 1\ \{X = 2\}$,
we can deduce $\vdash \{X = 1\}\ X := X + 1\ \{X = 2\}$.

Sequential composition

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

If the postcondition of C_1 matches the precondition of C_2, we can derive a specification for their sequential composition.

For example, if we have deduced:

- $\vdash \{X = 1\}\ X := X + 1\ \{X = 2\}$
- $\vdash \{X = 2\}\ X := X \times 2\ \{X = 4\}$

we may deduce that $\vdash \{X = 1\}\ X := X + 1; X := X \times 2\ \{X = 4\}$.

The conditional rule

$$\frac{\vdash \{P \land B\}\ C_1\ \{Q\} \qquad \vdash \{P \land \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

For instance, to prove that

$$\vdash \{\top\}\ \textbf{if}\ X \geq Y\ \textbf{then}\ Z := X\ \textbf{else}\ Z := Y\ \{Z = max(X, Y)\}$$

it suffices to prove that $\vdash \{\top \land X \geq Y\}\ Z := X\ \{Z = max(X, Y)\}$ and $\vdash \{\top \land \neg(X \geq Y)\}\ Z := Y\ \{Z = max(X, Y)\}$.

The loop rule

$$\frac{\vdash \{P \wedge B\}\ C\ \{P\}}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \wedge \neg B\}}$$

The loop rule says that

- if P is an invariant of the loop body when the loop condition succeeds, then P is an invariant for the whole loop, and
- if the loop terminates, then the loop condition failed.

We will return to be problem of finding loop invariants.

(Redundant) Conjunction and disjunction rules

$$\frac{\vdash \{P_1\}\ C\ \{Q\} \qquad \vdash \{P_2\}\ C\ \{Q\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q\}}$$

$$\frac{\vdash \{P\}\ C\ \{Q_1\} \qquad \vdash \{P\}\ C\ \{Q_2\}}{\vdash \{P\}\ C\ \{Q_1 \wedge Q_2\}}$$

These rules are useful for splitting up proofs.

Any proof with these rules could be done without using them

- i.e. they are theoretically redundant (proof omitted),
- however, they are useful in practice.

Summary

Hoare logic is a formalism for reasoning about the behaviour of programs by relating their initial and terminal state.

It uses an assertion logic based on first-order logic to reason about program states, and extends this with Hoare triples to reason about the programs.

Papers of historical interest:

- C. A. R. Hoare. An axiomatic basis for computer programming. 1969.
- R. W. Floyd. Assigning meanings to programs. 1967.
- A. M. Turing. Checking a large routine. 1949.

In the next lecture, we will formalise the intuitions we gave today, and prove soundness of Hoare logic.