Hoare Logic and Model Checking

Christopher Pulte cp526
University of Cambridge

CST Part II – 2022/2023

Acknowledgements

These slides closely follow Jean Pichon-Pharabod's slides from the CST Part II – 2020/2021 course, tweaked, which are in turn heavily based on previous versions by Mike Gordon, Alan Mycroft, and Kasper Svendsen.

Thanks to Julia Bibik, Mistral Contrastin, John Fawcett, Craig Ferguson, Victor Gomes, Joe Isaacs, Hrutvik Kanabar, Neel Krishnaswami, Dylan McDermott, Ian Orton, Peter Rugg, Peter Sewell, Ben Simner, Domagoj Stolfa, Ross Tooley, and Conrad Watt for remarks and reporting mistakes.

Motivation

We often fail to write programs that meet our expectations:

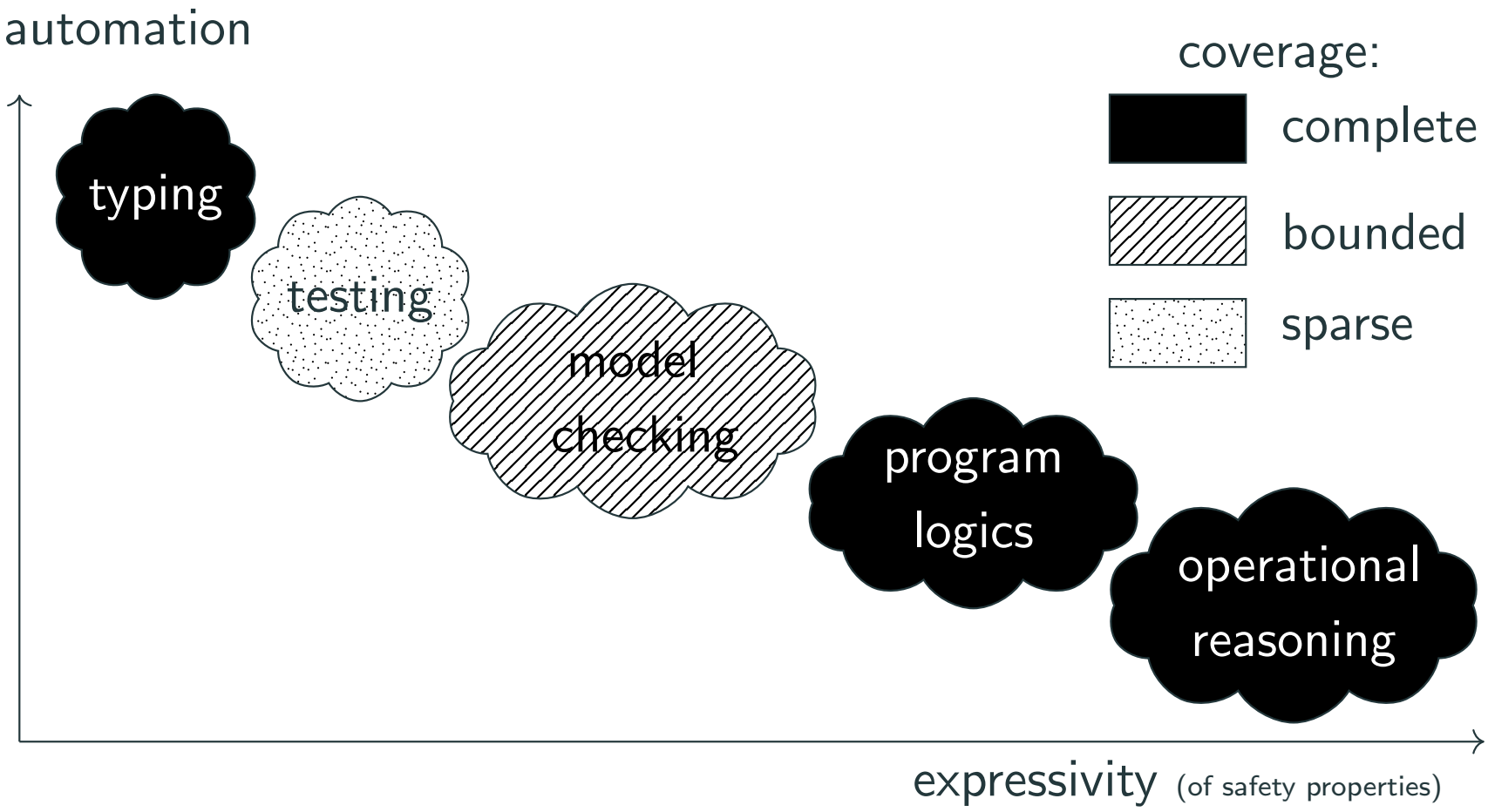
- we fail to write programs that meet their specification;
- we fail to write specifications that meet our expectations.

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

In practice, verification and validation feed back into each other.

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:

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

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.

Hoare logic

Lecture plan

Lecture 1: Informal introduction to Hoare logic

Lecture 2: Examples, loop invariants, and mechanisation

Lecture 3: Formal semantics and properties of Hoare logic

Lecture 4: Introduction to separation logic

Lecture 5: Verifying abstract data types in separation logic

Lecture 6: Extending 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 in 1949.

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:

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

Here, X 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, $\mathit{Var} \to \mathbb{Z}$.

Expressions of the WHILE language

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

$$
\begin{array}{lcll}
E & ::= & N \mid X \mid E_1 + E_2 & \textit{arithmetic expressions} \\
 & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
 & & & \\
B & ::= & \mathbf{T} \mid \mathbf{F} \mid E_1 = E_2 & \textit{boolean expressions} \\
 & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots &
\end{array}
$$

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 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 variable X contains a value that is is equal to the value that Y contains plus 1.

Informal semantics of partial correctness triples

The partial correctness triple $\{P\}\ C\ \{Q\}$ holds semantically, written $\models \{P\}\ C\ \{Q\}$, 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,

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

Partial correctness

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

For instance, $\models \{X = 1\}$ **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.

Later we will see that it is also possible to have total correctness triples that strengthen partial correctness triples to require termination.

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.

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 to occur in the program, to refer to the initial values of variables in postconditions. We call the variables that can occur in programs **program variables**.

Notation: program variables are uppercase, and auxiliary variables are lowercase.

Using auxiliary variables, we can specify C with
$\{X = x \wedge Y = y\}\ C\ \{(x \leq y \Rightarrow Z = y) \wedge (y \leq x \Rightarrow Z = x)\}$.

Using auxiliary variables

The previous specification still allows C to change X and Y, which may not be what we want. We can prevent that with

$$\{X = x \wedge Y = y\}\ C\ \left\{ \begin{array}{l} X = x \wedge Y = y \wedge \\ (x \leq y \Rightarrow Z = y) \wedge (y \leq x \Rightarrow Z = x) \end{array} \right\}$$

Using auxiliary variables, we can express that if C terminates, then it exchanges the values of variables X and Y:

$$\{X = x \wedge Y = y\}\ C\ \{X = y \wedge Y = x\}$$

Examples of partial correctness triples

C computes the Euclidian division of X by Y into Q and R:

$$\{X = x \wedge Y = y \wedge x \geq 0 \wedge y > 0\}\ C\ \{x = Q \times y + R \wedge 0 \leq R < y\}$$

C tests whether P is prime:

$$\{P = p \wedge p > 0\}\ C\ \left\{ \begin{array}{l} (R = 0 \Rightarrow \exists q, r.\, q > 1 \wedge r > 1 \wedge p = q \times r) \wedge \\ (R = 1 \Rightarrow \forall q, r.\, q > 1 \wedge r > 1 \Rightarrow p \neq q \times r) \end{array} \right\}$$

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:

- $\vdash_{FOL} P$, 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

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

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

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

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

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

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

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

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

For instance,

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

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

The rule of consequence

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

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

Note: the $\vdash_{FOL} P \Rightarrow Q$ 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:

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

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

The conditional rule

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

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 $\vdash \{\top \wedge X \geq Y\}\ Z := X\ \{Z = max(X, Y)\}$ and $\vdash \{\top \wedge \neg(X \geq Y)\}\ Z := Y\ \{Z = max(X, Y)\}$.

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.

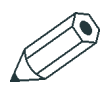
We will return to be problem of finding loop invariants.

Example

Example: integer square root algorithm

We can use these rules to verify a very inefficient integer square root algorithm:

$$\{X = x \wedge x \geq 0\}$$
$$S = 0; \textbf{while } (S+1) \times (S+1) \leq X \textbf{ do } S := S + 1$$
$$\{S \times S \leq x \wedge x < (S+1) \times (S+1)\}$$



Example: integer square root algorithm

We can use the following invariant:

$$X = x \wedge x \geq 0 \wedge S \times S \leq x$$

$$\nabla_1 = \dfrac{\dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \wedge S \times S \leq x \wedge (S+1) \times (S+1) \leq X \Rightarrow X = x \wedge x \geq 0 \wedge (S+1) \times (S+1) \leq X} \quad \dfrac{}{\vdash \{X = x \wedge x \geq 0 \wedge (S+1) \times (S+1) \leq X\}\ S := S+1\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}} \quad \dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \wedge S \times S \leq x \Rightarrow X = x \wedge x \geq 0 \wedge S \times S \leq x}}{\vdash \{X = x \wedge x \geq 0 \wedge S \times S \leq x \wedge (S+1) \times (S+1) \leq X\}\ S := S+1\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}}$$

$$\nabla_2 = \dfrac{\dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \wedge S \times S \leq x \Rightarrow X = x \wedge x \geq 0 \wedge S \times S \leq x} \quad \dfrac{\dfrac{\nabla_1}{\vdash \{X = x \wedge x \geq 0 \wedge S \times S \leq x \wedge (S+1) \times (S+1) \leq X\}\ S := S+1\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}}}{\vdash \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}\ \textbf{while}\ (S+1) \times (S+1) \leq X\ \textbf{do}\ S := S+1\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x \wedge \neg((S+1) \times (S+1) \leq X)\}} \quad \dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \wedge S \times S \leq x \wedge \neg((S+1) \times (S+1) \leq X) \Rightarrow S \times S \leq x \wedge x < (S+1) \times (S+1)}}{\vdash \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}\ \textbf{while}\ (S+1) \times (S+1) < X\ \textbf{do}\ S := S+1\ \{S \times S \leq x \wedge x < (S+1) \times (S+1)\}}$$

$$\dfrac{\dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \Rightarrow X = x \wedge x \geq 0 \wedge 0 \times 0 \leq x} \quad \dfrac{}{\vdash \{X = x \wedge x \geq 0 \wedge 0 \times 0 \leq x\}\ S := 0\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}} \quad \dfrac{\vdots}{\vdash_{FOL} X = x \wedge x \geq 0 \wedge S = 0 \Rightarrow X = x \wedge x \geq 0 \wedge S \times S \leq x}}{\vdash \{X = x \wedge x \geq 0\}\ S := 0\ \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}} \quad \dfrac{\nabla_2}{\vdash \{X = x \wedge x \geq 0 \wedge S \times S \leq x\}\ \textbf{while}\ (S+1) \times (S+1) \leq X\ \textbf{do}\ S := S+1\ \{S \times S \leq x \wedge x < (S+1) \times (S+1)\}}$$
$$\vdash \{X = x \wedge x \geq 0\}\ S := 0; \textbf{while}\ (S+1) \times (S+1) \leq X\ \textbf{do}\ S := S+1\ \{S \times S \leq x \wedge x < (S+1) \times (S+1)\}$$

Not very practical... we will see how to fix that in the next lecture.

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:

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

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

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

Conclusion

Applications

- Facebook's bug-finding Infer tool:
  `http://fbinfer.com/`
- The Rust programming language:
  `https://www.rust-lang.org/`
- Verification of the seL4 microkernel assembly:
  `https://entropy2018.sciencesconf.org/data/myreen.pdf`

Tools

- For Hoare Logic:
  - Why3 `http://why3.lri.fr/`
    Sedgewick in Why3:
    `http://pauillac.inria.fr/~levy/why3/index.html`
  - Boogie `https://github.com/boogie-org/boogie`

- For separation logic:
  - VeriFast `https://github.com/verifast/verifast`
  - The Iris higher-order concurrent separation logic framework, implemented and verified in a proof assistant:
    `http://iris-project.org/`

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 defines Hoare triples on top of it to reason about the programs.

In the next lecture, we will use Hoare logic to reason about example 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.