

Hoare Logic

COMP2600 — Formal Methods for Software Engineering

Ranald Clouston

Australian National University
Semester 2, 2011

Introduction

Reliability is sometimes important

... and sometimes very important.

- Chemical plants
- Aircraft controls
- Pacemakers
- Railway networks, etc.

High reliability requires rigorous **verification**.

What gets verified?

- Hardware
- Compilers
- Programs
- Specifications, too ...

How do we do it?

- Informally analysing the code
- Testing
- *Formal verification*

Formal Program Verification

Formal program verification is about **proving** properties of programs using logic and mathematics.

In particular, it is about

- proving they meet their specifications
- proving that requirements are satisfied

Why verify formally?

- Proofs *guarantee* correctness.
- Formal proofs are mechanically checkable.

Why not verify formally?

- Time consuming.
- Expensive.

Example: Altran Praxis, <http://www.altran-praxis.com>.

Verification in Haskell

Haskell is a pure functional language, so:

- Equations defining functions *really are equations*
- Therefore, we can prove properties of Haskell programs using *standard mathematical techniques* such as:
 - substitution of equal terms
 - arithmetic
 - structural induction, etc.
- We saw some of this in week 3.

Verification in Imperative Languages

- Imperative languages are built around a *program state* (data stored in memory).
- Imperative programs are sequences of *commands that modify that state*.

To prove properties of imperative programs, we need

- A way of expressing assertions about program states.
- Rules for manipulating and proving those assertions.

These will be provided by **Hoare Logic**.

Three Components of Hoare Logic Assertions:

1. **A precondition**
2. **A code fragment**
3. **A postcondition**

The *precondition* is a **predicate** asserting something of interest about the *state before* the code is executed.

The *postcondition* is a **predicate** asserting something of interest about the *state after* the code is executed.

Assertions – Preconditions and Postconditions

Hoare logic will allow us to make *assertions* such as:

If $(x > 0)$ is true *before* $y := 0-x$ is executed
then $(y < 0 \wedge x \neq y)$ is true *afterwards*.

In this example,

- $(x > 0)$ is a precondition
- $(y < 0 \wedge x \neq y)$ is a postcondition

In this course our pre- and postconditions will all be built out of

- assertions about arithmetic and variables: $y < 0$, $x \neq y$...
- propositional logic connectives: \wedge, \vee ...
- *True, False*.

Hoare's Notation – the Definition

The **Hoare triple**:

$$\{P\} S \{Q\}$$

means:

If P is true in the initial state
and S terminates
then Q will hold in the final state.

Examples:

1. $\{x = 2\} x := x+1 \{x = 3\}$
2. $\{x = 2\} x := x+1 \{x = 5000\}$
3. $\{x > 0\} y := 0-x \{y < 0 \wedge x \neq y\}$

A Larger Hoare Triple

```
{n ≥ 0}
fact := 1;
while (n > 0)
  fact := fact * n;
  n := n - 1;
{fact = n!}
```

What if $n < 0$?

Partial Correctness

Hoare logic expresses *partial correctness*.

We say a program is *partially correct* if it gives the right answer whenever it terminates.

It never gives a wrong answer, but it may give no answer at all.

$\{P\} S \{Q\}$ does **NOT** imply that S terminates, even if P holds initially.

For example

$$\{x = 1\} \text{ while } x=1 \text{ do } y:=2 \{x = 3\}$$

is **true** in Hoare logic.

Partial Correctness is OK

Why not insist on termination?

- We **may not want** termination.
- It **simplifies the logic**.
- If necessary, **we can prove termination separately**.

We will come back to termination next week with the Weakest Precondition Calculus.

There's not much point writing stuff down unless you can do something with it...

We can use pre- and postconditions to specify the effect of a code fragment on the state, but how do we **prove or disprove** a Hoare Triple specification?

- Is $\{P\} S \{Q\}$ **true** or **false**?

We need a **logic** or a **calculus**:

- a collection of *rules and procedures* for (formally) manipulating the (language of) triples.

(Just like algebra, just like logic...)

We will now turn to developing and applying a basic version of Hoare Logic.

Weak and Strong Conditions

A condition P is **stronger** than Q in the cases where P implies Q .

(Similarly Q is **weaker** than P .)

If P is stronger than Q then P is **more likely to be false** than Q .

A politician's example:

- *I will keep unemployment below 3%* is **stronger** than
- *I will keep unemployment below 15%*.
- The **strongest** possible statement is *False*, i.e. *I will keep unemployment below 0%*.
- The **weakest** possible statement is *True*, i.e. *I will keep unemployment at or below 100%*.

Strong Postconditions

- $(x = 6) \Rightarrow (x > 0)$ so $(x = 6)$ is **stronger** than $(x > 0)$
- The statement:

$$\{x = 5\} x := x + 1 \{x = 6\}$$

says **more** about the code than:

$$\{x = 5\} x := x + 1 \{x > 0\}$$

If a *postcondition* Q_1 is **stronger** than Q_2 ,

then $\{P\} S \{Q_1\}$ is a **stronger** statement than $\{P\} S \{Q_2\}$.

Weak Preconditions

- The condition $(x > 0)$ says less about a state than the condition $(x = 5)$. It is the *weaker* condition.

- but the statement

$$\{x > 0\} \text{ x := x + 1 } \{x > 1\}$$

says **more** about the code than:

$$\{x = 5\} \text{ x := x + 1 } \{x > 1\}$$

If a *precondition* P_1 is *weaker* than P_2 , then $\{P_1\}S\{Q\}$ is *stronger* than $\{P_2\}S\{Q\}$.

(Usually we are interested in strong postconditions and weak preconditions, because they say more about the code.)

Proof rule for Strengthening Preconditions (Rule 1/6)

It is safe to make a *precondition* more *specific* (*stronger*).

- The rule:

$$\frac{\{P_w\} S \{Q\} \quad P_s \rightarrow P_w}{\{P_s\} S \{Q\}}$$

- An instance:

$$\frac{\{x > 2\} \text{ x := x + 1 } \{x > 3\} \quad (x = 4) \rightarrow (x > 2)}{\{x = 4\} \text{ x := x + 1 } \{x > 3\}}$$

Proof rule for Weakening Postconditions (Rule 2/6)

It is safe to *weaken* a *postcondition* so it says *less*.

- The rule:

$$\frac{\{P\} S \{Q_s\} \quad Q_s \rightarrow Q_w}{\{P\} S \{Q_w\}}$$

- An instance:

$$\frac{\{x > 2\} \text{ x := x + 1 } \{x > 3\} \quad (x > 3) \rightarrow (x > 1)}{\{x > 2\} \text{ x := x + 1 } \{x > 1\}}$$

A Very Simple Imperative Language

We will need a Hoare logic rule for each kind of imperative command.

To this end we'll define a little language with four different kinds of statement.

Assignment – $x := e$

where x is a variable and e is an expression with variables and arithmetic that returns a number, e.g. $2 + 3$, $x * y + 1$...

Sequencing – $S_1; S_2$

Conditional – if b then S_1 else S_2

where b is an expression with variables and arithmetic that returns a *boolean* (true or false), e.g. $y < 0$, $x \neq y$...

While – while b do S