Hoare Logic

COMP2600 — Formal Methods for Software Engineering

Rajeev Goré

Australian National University
Semester 2, 2016 (Slides courtesy of Ranald Clouston)
Australian Capital Territory Election Software

- Australian Capital Territory elections use Single Transferable Voting

- If we have $C$ candidates for $S < C$ vacancies you rank all candidates in order of your preference: Gough, Vladimir, ... , Tony

- Counting such ballots proceeds in rounds where in each round we count up the votes for each candidate, elect or eliminate one candidate and then transfer the surplus votes to the next preferred candidate
Australian Capital Territory Election Software

• Australian Capital Territory elections use Single Transferable Voting

• If we have $C$ candidates for $S < C$ vacancies you rank all candidates in order of your preference: John, Mary, ... , Tony

• Counting such ballots proceeds in rounds where in each round we count up the votes for each candidate, elect or eliminate one candidate and then transfer the surplus votes to the next preferred candidate

• The ACT Electoral Commission uses a computer program to count the votes cast for elections

• Is it safe?
**Australian Capital Territory Election Software**

- Australian Capital Territory elections use Single Transferable Voting
- If we have $C$ candidates for $S < C$ vacancies you rank all candidates in order of your preference: John, Mary, ..., Tony
- Counting such ballots proceeds in rounds where in each round we count up the votes for each candidate, elect or eliminate one candidate and then transfer the surplus votes to the next preferred candidate
- The ACT Electoral Commission uses a computer program to count the votes cast for elections
- Is it safe?
- Bugs? We have found three: all could have changed the outcome!
- Verification
What gets verified?

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

How do we do it?

- Informally analysing the code
- Testing:
  “Program testing can be used to show the presence of bugs, but never to show their absence!” - Edsger Dijkstra
- 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 (partial) correctness of program.
- Formal proofs are mechanically checkable.
- Good practice for ordinary programming.

Why not verify formally?

- Time consuming.
- Expensive.
Formal or Informal?

The question of whether to verify formally or not ultimately comes down to how disastrous occasional failure would be.

The website of UK-based formal software engineers Altran Praxis [http://www.altran-praxis.com](http://www.altran-praxis.com) showcases many of the industries that are most likely to take the formal route.
Verification for Functional Languages

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 last week.
Verification for 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*. 
C. A. R. (Tony) Hoare

The inventor of this week’s logic is also famous for inventing the **Quicksort** algorithm in 1960 - when he was just 26! A quote:

> Computer programming is an *exact science* in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely *deductive reasoning*.
Logic = Syntax and (Semantics or Calculus)

Syntax: special language from which we build formulae

Semantics: the way in which we assign truth to formulae via models

Validity: fundamental notion from semantics

Calculus: symbol manipulation rules which allows us to construct proofs

Provable: fundamental notion from calculus

Soundness and Completeness: provable if and only if valid

CPL Syntax: connectives $\land$, $\lor$, $\Rightarrow$, $\neg$ and atomic formulae $p, q, r \cdots$

CPL Semantics: model assigns true/false to each atomic formula and truth tables allow us to lift this to truth value of larger formulae

ND calculus: a finite set of rules, and regulations on how to use them
### Classical Prop Logic = Syntax and (Semantics or Calculus)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic formulae</td>
<td>truth values</td>
<td>Natural Deduction</td>
</tr>
<tr>
<td>$p, q, r \cdots$</td>
<td>${true, false}$</td>
<td></td>
</tr>
<tr>
<td>connectives</td>
<td>truth tables</td>
<td>rules of inference</td>
</tr>
<tr>
<td>$\land, \lor, \to, \neg$</td>
<td>satisfiable</td>
<td>rule instance</td>
</tr>
<tr>
<td>formulae</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A, B, C \cdots$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Soundness and Completeness: formula $A$ provable if and only if $A$ valid
Hoare Logic = Syntax and (Semantics or Calculus)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPL formulae</td>
<td>CPL semantics</td>
<td>(ND)</td>
</tr>
<tr>
<td>Hoare Triple ( {P} S {Q} )</td>
<td>if pre-state satisfies ( P ) and program ( S ) terminates then post-state satisfies ( Q )</td>
<td>rules of proof</td>
</tr>
<tr>
<td>( {x = 1} \ x := x + 1 \ {x = 2} )</td>
<td>if pre-state satisfies ( x = 1 ) and program ( x := x + 1 ) terminates then post-state satisfies ( x = 2 )</td>
<td>proof rule?</td>
</tr>
</tbody>
</table>
Hoare Logic: A Simple Imperative Programming Language

Syntax of the programming language component $S$ from a Hoare triple

To prove things about programs, we first need to fix a programming language.

For this 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 built from variables and arithmetic that returns a number, e.g. $2 + 3, x \times y + 1$...

**Sequencing** – $S_1; S_2$

**Conditional** – if $b$ then $S_1$ else $S_2$

where $b$ is an expression built from variables, arithmetic and logic that returns a boolean (true or false), e.g. $y < 0, x \neq y \land z = 0$...

**While** – while $b$ do $S$
A Note on (the lack of) Aliasing

Suppose we had the code fragment

\[ x := y \]

This looks up the number value of \( y \) and copies it into the piece of memory pointed to by \( x \). Now they both have the same number value.

It does not make the variables \( x \) and \( y \) point to the same piece of memory.

So if the next line of the program is

\[ x := x + 1 \]

then after that it is no longer the case that \( x = y \).
Syntax: Three Components of Hoare Triples

1. A precondition

2. A code fragment

3. A postcondition

The precondition is an **assertion** saying something of interest about the **state before** the code is executed.

The postcondition is an **assertion** saying something of interest about the **state after** the code is executed.
Semantics: notion of a state

A state is determined by the values given to the program variables.

In this course all our variables will store numbers only.
Syntax of Preconditions and Postconditions

So all our assertions about the state will be built out of variables, numbers, and basic arithmetic relations:

- $x = 3$;
- $x = y$;
- $x \neq y$;
- $x > 0$;
- $x \leq (y^2 + 1\frac{3}{4})$;
- etc...
Syntax of Preconditions and Postconditions ctd.

We may want to make complicated claims about several different variables, so we will use propositional logic to combine the simple assertions, e.g.

- $x = 4 \land y = 2$;
- $x < 0 \lor y < 0$;
- $x > y \Rightarrow x = 2 \times y$;
- True;
- False.

The last two logical constructions - True and False - will prove particularly useful, as we’ll later see.
A Rough Guide to Hoare Logic Semantics

Hoare logic will allow us to make claims such as:

\[
\text{If } (x > 0) \text{ is true before } y := 0 - x \text{ is executed then } (y < 0 \land x \neq y) \text{ is true afterwards.}
\]

In this example,

- \((x > 0)\) is a precondition;
- \(y := 0 - x\) is a code fragment;
- \((y < 0 \land x \neq y)\) is a postcondition.

This particular assertion is intuitively **true**; we will need to learn Hoare logic before we can prove this, though!
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. \( \{ x > 0 \} \ y := 0-x \ \{ y < 0 \land x \neq y \} \)
A Larger Hoare Triple

\{n \geq 0\}

\texttt{fact := 1;}
\texttt{while (n>0)}
\quad \texttt{fact := fact * n;}
\quad \texttt{n := n-1}
\{\texttt{fact = n!}\}

Question - 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 semantics

ie if pre-state satisfies \( x = 1 \) *and* the while loop terminates then the post-state satisfies \( x = 3 \) (but the while loop does not terminate)
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?

We need a calculus:

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

(Just like ND for classical propositional logic . . . )

We will now turn to developing and applying a basic version of Hoare Logic.
The Assignment Axiom (Rule 1/6)

We will have one rule for each of our four kinds of statement (plus two other rules, as we’ll see).

First, we look at assignment.

Assignments change the state so we expect Hoare triples for assignments to reflect that change.

Suppose $Q(x)$ is a predicate involving a variable $x$, and that $Q(e)$ indicates the same formula with all occurrences of $x$ replaced by the expression $e$.

The assignment axiom of Hoare Logic: one line proof

$$\{Q(e)\} \ x := e \ {Q(x)}$$
The Assignment Axiom – Intuition

\[
\{Q(e)\} \ x := e \ {Q(x)}
\]

If we want \(x\) to have some property \(Q\) after the assignment, then that property must hold for the value \((e)\) assigned to \(x\) - \textit{before} the assignment is executed.

You might ask if this rule is \textit{backwards} and should be

\[
\{Q(x)\} \ x := e \ {Q(e)}
\]

But this is \textit{wrong}: if we tried to apply this ‘axiom’ to the precondition \(x = 0\) and code fragment \(x := 1\) we’d get

\[
\{x = 0\} \ x := 1 \ \{1 = 0\}
\]

which says “if \(x = 0\) initially and \(x := 1\) terminates then \(1 = 0\) finally”
Work from the Goal, ‘Backwards’

It may seem natural to start at the **precondition** and reason towards the **postcondition**, but this is *not* the best way to do Hoare logic.

Instead start with your **goal** (postcondition) and go ‘backwards’.

e.g. to apply the assignment axiom

\[
\{Q(e)\} \ x := e \ {Q(x)}
\]

take the postcondition, **copy** it across to the precondition, then **replace** all occurrences of \(x\) with \(e\).

Note that the postcondition may have no, one, or many occurrences of \(x\) in it; all get replaced by \(e\) in the precondition.
Example 1 of \{Q(e)\} x := e \{Q(x)\}

Consider the code fragment $x := 2$ and suppose that the desired postcondition is $(y = x)$.

Our precondition is found by copying the postcondition $y = x$ over, then replacing our occurrence(s) of the variable $x$ with the expression $2$.

Formally:

\[
\{y = 2\} \ x := 2 \ \{y = x\}
\]

is an instance of the assignment axiom.
Example 2 of \( \{Q(e)\} \ x := e \ {Q(x)} \)

Consider the code fragment \( x := x + 1 \) and suppose that the desired postcondition is \( y = x \).

We proceed as in the last slide:

\[ \{y = x + 1\} \ x := x + 1 \ {y = x} \]
Example 3 of \( \{Q(e)\} \ x := e \ {Q(x)} \)

How might we try to prove

\[
\{y > 0\} \ x := y+3 \ {x > 3} \ ?
\]

Start with the postcondition \(x > 3\) and apply the axiom:

\[
\{y + 3 > 3\} \ x := y+3 \ {x > 3}
\]

Then use the fact that \(y + 3 > 3\) is equivalent to \(y > 0\) to get our result.

You can always replace predicates by equivalent predicates; just label your proof step with ‘precondition equivalence’, or ‘postcondition equivalence’.
Proving the Assignment Axiom sound w.r.t. semantics

Recall that the assignment axiom of Hoare Logic is:

$$\{Q(e)\} \ x := e \ {Q(x)}$$

Why is it so?

- Let $v$ be the value of expression $e$ in the initial state.
- If $Q(e)$ is true initially, then so is $Q(v)$.
- Since the variable $x$ has value $v$ after the assignment (and nothing else is changed in the state), $Q(x)$ must be true after that assignment.
The Assignment Axiom is Optimal

The Hoare triple in the assignment axiom is as strong as possible.

\{Q(e)\} \ x := e \ \{Q(x)\}

That is, if \(Q(x)\) holds after the assignment then \(Q(e)\) MUST have held before it.

Why?

- Suppose \(Q(x)\) is true after the assignment.
- If \(v\) is the value assigned, \(Q(v)\) is true after the assignment.
- Since it is only the value of \(x\) that is changed, and the predicate \(Q(v)\) does not involve \(x\), \(Q(v)\) must also be true before the assignment.
- Since \(v\) was the value of \(e\) before the assignment, \(Q(e)\) is true initially.
A non-example

What if we wanted to prove

\[ \{ y = 2 \} \ x := y \ { x > 0 } \ ? \]

This is clearly true. But our assignment axiom doesn’t get us there:

\[ \{ y > 0 \} \ x := y \ { x > 0 } \]

We cannot just replace \( y > 0 \) with \( y = 2 \) either - they are not equivalent.

We need a new Hoare logic rule that manipulates our preconditions (and while we’re at it, a rule for postconditions as well!).
# Hoare Logic = Syntax and (Semantics or Calculus)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPL predicates</td>
<td>CPL semantics</td>
<td>No ND!</td>
</tr>
</tbody>
</table>

- **Variables**: \( x, y, z \cdots \) arithmetic expressions \( 1 + 2, x < y, \cdots \) and predicates built from them
- **States**: map variables and expressions to values and predicates to true/false
- **Programming language**: \( :=; \text{if.then.while} \)
- **Hoare Triple**: \( \{P\} S \{Q\} \)
  - if pre-state satisfies \( P \) and \( S \) terminates then post-state satisfies \( Q \)

- **Rules of arithmetic** e.g. 
  - \( 1 + 2 = 3 \), \( 2^2 = 4 \), \( 6/3 = 2 \) etc
- **As usual**: one rule for each (seen rule for \( := \))
Summary of Lecture 1

Hoare triple syntax:

\[ \{ P \} \ S \ \{ Q \} \]

Proof Rule for Assignment:

\[ \{ Q(e) \} \ x := e \ \{ Q(x) \} \]

Hoare triple Semantics (meaning):
If \( P \) is true in the initial state and \( S \) terminates then \( Q \) will hold in the final state.

One line proof: Any triple that looks like this is (defined to be) provable and Soundness: is guaranteed to be true

Examples:

<table>
<thead>
<tr>
<th>Hoare triple</th>
<th>true?</th>
<th>Provable by assignment axiom?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x + 1 = 3 ) ( x := x + 1 \ { x = 3 } )</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>( y &gt; 2 ) ( x := y \ { x &gt; 2 } )</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>( y = 3 ) ( x := y \ { x &gt; 2 } )</td>
<td>yes</td>
<td>no ... need more rules ...</td>
</tr>
</tbody>
</table>