# Hoare Logic

## COMP2600 — Formal Methods for Software Engineering

Rajeev Goré

Australian National University

Semester 2, 2014 (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` 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**.

## 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 $\wedge, \vee, \Rightarrow, \neg$ and atomic formulae $p, q, r \cdots$

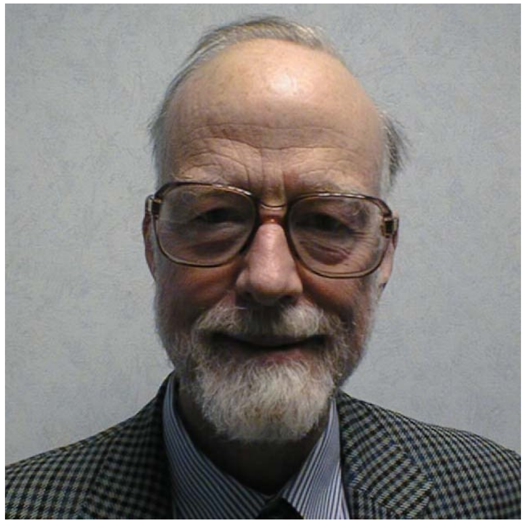
**CPL Semantics:** model assigns true/false to each atomic formula and truth tables allow us to lift this to truth value of larger formulae

**ND calculus:** a finite set of rules, and regulations on how to use them

## 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**.



## A Very Simple Imperative Language

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 –** $\mathtt{x} := \mathtt{e}$

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

**Sequencing –** $\mathtt{S}_1 ; \mathtt{S}_2$

**Conditional –** $\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2$

> where $\mathtt{b}$ is an expression with variables, arithmetic and logic that returns a boolean (true or false), e.g. $\mathtt{y} < 0$, $\mathtt{x} \neq \mathtt{y} \wedge \mathtt{z} = 0$...

**While –** $\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{S}$

## A Note on (the lack of) Aliasing

Suppose we had the code fragment

$$\mathtt{x} := \mathtt{y}$$

This looks up the number value of $\mathtt{y}$ and copies it into the piece of memory pointed to by $\mathtt{x}$. 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

$$\mathtt{x} := \mathtt{x} + 1$$

then after that it is no longer the case that $x = y$.

## Syntax: Three Components of Hoare Logic Assertions

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 Assertions – 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 Assertions – 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.

- $x = 4 \wedge y = 2$;
- $x < 0 \vee y < 0$;
- $x > y \Rightarrow x = 2 * y$;
- $True$;
- $False$.

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:

> 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-x` is a code fragment;
- $(y < 0 \wedge x \neq y)$ 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

$\{\mathtt{n} \geq 0\}$

```
fact := 1;
while (n>0)
    fact := fact * n;
    n := n-1;
```

$\{\mathtt{fact} = \mathtt{n}!\}$

Question - what if $\mathtt{n} < 0$?

## 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

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

is ***true*** in Hoare logic semantics

## 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**

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

## The Assignment Axiom – Intuition

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

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

You might ask if this rule is backwards and should be

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

But this is wrong: if we tried to apply this 'axiom' to the precondition $x = 0$ and code fragment $\texttt{x} := 1$ we'd get

$$\{x = 0\}\ \texttt{x} := 1\ \{1 = 0\}\ !$$

instead of

$$\{x = 0\}\ \texttt{x} := 1\ \{x = 1\}$$

## 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

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

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:

$$\{y = 2\}\ \texttt{x:=2}\ \{y = x\}$$

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:

$$\{y = x + 1\}\ \texttt{x:=x+1}\ \{y = x\}$$

## Example 3 of $\{Q(e)\}$ `x := e` $\{Q(x)\}$

How might we try to prove

$$\{y > 0\}\ \texttt{x:=y+3}\ \{x > 3\}\ ?$$

Start with the postcondition $x > 3$ and apply the axiom:

$$\{y + 3 > 3\}\ \texttt{x:=y+3}\ \{x > 3\}$$

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:

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

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**.

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

***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

$$\{y = 2\}\ \texttt{x:=y}\ \{x > 0\}\ ?$$

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

$$\{y > 0\}\ \texttt{x:=y}\ \{x > 0\}$$

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!).