Hoare Logic

COMP2600 — Formal Methods for Software Engineering

Presented by Jinbo Huang
(most slides by Ranald Clouston)

Australian National University
Let’s start with a video...

http://www.youtube.com/watch?v=H61d_OkIht4
What was that all about?

_Nulka_ is a ‘soft-kill’ decoy missile, leading homing missiles away from ships to splash harmlessly in the water.

It was developed right here in Australia by **BAE** and **DSTO**.

It is now fitted as standard on many of the most expensive and heavily manned vessels in the Australian, US, and other navies.

*What would happen if Nulka failed?*
When failure is not an option

[We were] building a missile that must be trusted, that must work every time, and is a single shot device, and people’s lives depend on it.

Jim McDowell, Managing Director, BAE Systems

http://www.youtube.com/watch?v=Rkw1PtVxPDE

But how do we ensure that a system like Nulka lives up to this standard and literally never fails?

This is a problem of 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.
- 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.

In the case of Nulka, the flight control unit software was verified formally, partly using the Altran Praxis tool SPARK.
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.
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** – $x := e$

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

**Sequencing** – $S_1; S_2$

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

where $b$ is an expression with 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 \( x = y \).
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 a **assertion** saying something of interest about the **state after** the code is executed.
Assertions – Preconditions and Postconditions

In this course all our **variables** will store **numbers only**.

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

- $x = 3$
- $x = y$
- $x \neq y$
- $x > 0$
- $x \leq \left(y^2 + 1\frac{3}{4}\right)$
- etc...
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

Hoare logic will allow us to make claims such as:

If \((x > 0)\) is true \textit{before} \(y := 0 - x\) is executed
then \((y < 0 \land x \neq y)\) is true \textit{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 \textit{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 \}\n
\textbf{fact} := 1;

\textbf{while} (n > 0) \textbf{do}

\textbf{fact} := \textbf{fact} \times n;

n := n - 1;

\{ \textbf{fact} = n! \}
A Larger Hoare Triple (Revised)

\{ n \geq 0 \}

\text{fact} := 1; \\
i := n; \\
\textbf{while } (i > 0) \textbf{ do} \\
\text{fact} := \text{fact} * i; \\
i := i - 1 \\
\{ \text{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

\[
\{ x = 1 \} \text{ while } x=1 \text{ do } y:=2 \quad \{ x = 3 \}
\]

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 an extension to Hoare logic.
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 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.
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:

$$\{ Q(e) \} \ x := e \ { Q(x) \}$$
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 $x$ - before the assignment is executed.

You might ask if this rule is backwards and should be

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

But this is wrong: if we tried to apply this ‘axiom’ to the precondition $x = 0$ and code fragment $x := 1$ we’d get

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

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

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**

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 correct

Recall that the assignment axiom of Hoare Logic is:

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

Why is it so?

- Let \( v \) be the value assigned — i.e. \( v \) is 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!).
Weak and Strong Predicates

A predicate \( P \) is **stronger** than \( Q \) if it is the case that \( P \) 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

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

says more about the code than:

\[
\{ x = 5 \} \ x := x + 1 \ {x > 1} 
\]

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 2/6)

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

- The rule:

\[
P_s \Rightarrow P_w \quad \{P_w\} \quad S \quad \{Q\}
\]

\[
\{P_s\} \quad S \quad \{Q\}
\]

- An instance:

\[
(y = 2) \Rightarrow (y > 0) \quad \{y > 0\} \quad x := y \quad \{x > 0\}
\]

\[
\{y = 2\} \quad x := y \quad \{x > 0\}
\]
Proof rule for Weakening Postconditions (Rule 3/6)

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

- The rule:

\[
\begin{align*}
\{P\} & \quad S \quad \{Q_s\} & \quad Q_s \Rightarrow Q_w \\
\hline
\{P\} & \quad S \quad \{Q_w\}
\end{align*}
\]

- An instance:

\[
\begin{align*}
\{x > 2\} & \quad x := x + 1 \quad \{x > 3\} & \quad (x > 3) \Rightarrow (x > 1) \\
\hline
\{x > 2\} & \quad x := x + 1 \quad \{x > 1\}
\end{align*}
\]

We will not have need of this postcondition weakening rule for a while...
Proof rule for Sequencing (Rule 4/6)

Imperative programs consist of a sequence of statements, affecting the state one after the other:

\[
\begin{array}{c}
\{P\} \quad S_1 \quad \{Q\} \\
\{Q\} \quad S_2 \quad \{R\}
\end{array}
\]

\[
\{P\} \quad S_1; \quad S_2 \quad \{R\}
\]

Example Instance:

\[
\begin{array}{c}
\{x > 2\} \quad x := x + 1 \quad \{x > 3\} \\
\{x > 3\} \quad x := x + 2 \quad \{x > 5\}
\end{array}
\]

\[
\{x > 2\} \quad x := x + 1; \quad x := x + 2 \quad \{x > 5\}
\]
Laying out a proof

For the sake of the sanity of your markers, it would help if you all used the same layout for your proofs:

1. \(\{x + 2 > 5\} \ x := x + 2 \ \{x > 5\}\)  
   (Assignment)

2. \(\{x > 3\} \ x := x + 2 \ \{x > 5\}\)  
   (1, Precondition Equivalence)

3. \(\{x + 1 > 3\} \ x := x + 1 \ \{x > 3\}\)  
   (Assignment)

4. \(\{x > 2\} \ x := x + 1 \ \{x > 3\}\)  
   (3, Precondition Equivalence)

5. \(\{x > 2\} \ x := x + 1; x := x + 2 \ \{x > 5\}\)  
   (2, 4, Sequencing)

Note the numbered proof steps and justifications.
How do we get the Condition in the Middle?

In the rule

$$\{P\} \ S_1 \ \{Q\} \ \{Q\} \ S_2 \ \{R\} \ \{P\} \ S_1; \ S_2 \ \{R\}$$

Our precondition $P$ and postcondition $R$ will be given to us, but how do we come up with the $Q$?

By starting with our goal $R$ and working backwards, as usual!
An example with precondition strengthening

Say we wanted to prove

\[
\{ x = 3 \} \quad x := x + 1; \ x := x + 2 \quad \{ x > 5 \}
\]

The first five steps will be the same as those we’ve seen:

5. \( \{ x > 2 \} \quad x := x + 1; \ x := x + 2 \quad \{ x > 5 \} \) \hspace{1cm} \text{(See earlier slide)}

To which we add:

6. \( x = 3 \Rightarrow x > 2 \) \hspace{1cm} \text{(Basic arithmetic)}

7. \( \{ x = 3 \} \quad x := x + 1; \ x := x + 2 \quad \{ x > 5 \} \) \hspace{1cm} \text{(5, 6, Pre. Strength.)}
Validity of Rule for Sequences

Suppose the hypotheses \{P\} S_1 \{Q\} and \{Q\} S_2 \{R\} are true and let \sigma_0 be an arbitrary state that satisfies \( P \).

From the rule premises, we know that:

1. Executing \( S_1 \) on \( \sigma_0 \) must produce a state \( \sigma_1 \) that satisfies \( Q \).
2. Executing \( S_2 \) on \( \sigma_1 \) must produce a state \( \sigma_2 \) that satisfies \( R \).

But \( S_1; S_2 \) just means execute \( S_1 \) and then execute \( S_2 \).

So, when \( S_1; S_2 \) executes on \( \sigma_0 \), the resulting state will be state \( \sigma_2 \) which we know must satisfy \( R \).

What about non-termination?
Proof Rule for Conditionals (Rule 5/6)

$$\begin{align*}
\{P \land b\} & S_1 \{Q\} & \{P \land \neg b\} & S_2 \{Q\} \\
\{P\} & \text{if } b \text{ then } S_1 \text{ else } S_2 \{Q\}
\end{align*}$$

- When a conditional is executed, either $S_1$ or $S_2$ is executed.
- Therefore, if the conditional is to establish $Q$, both $S_1$ and $S_2$ must establish $Q$.
- Similarly, if the precondition for the conditional is $P$, then it must also be a precondition for the two branches $S_1$ and $S_2$.
- The choice between $S_1$ and $S_2$ depends on evaluating $b$ in the initial state, so we can also assume $b$ to be a precondition for $S_1$ and $\neg b$ to be a precondition for $S_2$. 
Example of Conditional Rule

Suppose we wish to prove:

\[
\{ x > 2 \} \text{ if } x > 2 \text{ then } y := 1 \text{ else } y := -1 \{ y > 0 \}
\]

The proof rule for conditionals suggests we prove:

\[
\{ x > 2 \land x > 2 \} \ y := 1 \ \{ y > 0 \}
\]
\[
\{ x > 2 \land \neg(x > 2) \} \ y := -1 \ \{ y > 0 \}
\]

Simplifying the preconditions:

(1) \( \{ x > 2 \} \ y := 1 \ \{ y > 0 \} \)

(2) \( \{ False \} \ y := -1 \ \{ y > 0 \} \)
Example ctd

• For subgoal (1) the assignment axiom tells us that

\[ \{1 > 0\} \ y := 1 \ \{y > 0\} \]

Which is equivalent to

\[ \{\text{True}\} \ y := 1 \ \{y > 0\} \]

Now since \((x > 2) \implies \text{True}\) we can strengthen the precondition to \((x > 2)\) as required.

• For subgoal (2) the assignment axiom tells us that

\[ \{-1 > 0\} \ y := -1 \ \{y > 0\} \]

and \((-1 > 0)\) is equivalent to \(\text{False}\), as required.

QED.
Conditionals Without ‘Else’

The *conditional* rule is for code fragments of the form

\[
\text{if } b \text{ then } S_1 \text{ else } S_2
\]

How would we derive a rule for a conditional statement without *else*:

\[
\text{if } b \text{ then } S
\]

First note that this is equivalent to

\[
\text{if } b \text{ then } S \text{ else } x := x
\]

(It doesn’t matter much what \(x\) is here, or whether this variable is used anywhere else in the program!)
Conditionals Without ‘Else’ ctd.

How do we prove

\[
\begin{align*}
\{ P \land b \} & \quad S & \quad \{ Q \} & \quad \{ P \land \neg b \} & \quad x := x & \quad \{ Q \} \\
\{ P \} & \quad \textbf{if} \quad b \quad \textbf{then} \quad S & \quad \textbf{else} \quad x := x & \quad \{ Q \} 
\end{align*}
\]

Our \textbf{assignment} rule only gets us as far as

\[
\{ Q \} \quad x := x \quad \{ Q \}
\]

\textbf{Precondition strengthening} comes to the rescue, giving us the derived ‘rule’

\[
\begin{align*}
\{ P \land b \} & \quad S & \quad \{ Q \} & \quad (P \land \neg b) \Rightarrow Q \\
\{ P \} & \quad \textbf{if} \quad b \quad \textbf{then} \quad S & \quad \textbf{else} \quad x := x & \quad \{ Q \}
\end{align*}
\]
Proof Rule for While Loops (Rule 6/6)

\[
\begin{align*}
\{P \land b\} & \quad S & \{P\} \\
\{P\} & \quad \textbf{while} \quad b & \quad \textbf{do} & \quad S & \{P \land \neg b\}
\end{align*}
\]

- \(P\) is called the \textit{loop invariant}.
- \(P\) is true before we encounter the while statement, and remains true each time around the loop (although not necessarily midway \textit{during} execution of the loop body).
- If the loop terminates the control condition must be false, so \(\neg b\) appears in the postcondition.
- For the body of the loop \(S\) to execute, \(b\) needs to be true, so it appears in the precondition.
Validating the While Rule

Assume $P$ holds in the initial state.

If $b$ is false immediately then the program state is not changed, so $P \land \neg b$ holds.

Now assume the invariant holds: $\{P \land b\} S \{P\}$. $P$ therefore holds at the end of each iteration.

Assume that the loop terminates, and so $b$ eventually becomes false. Then (as above) the program state does not change from there, so $P \land \neg b$ holds.

*(What about non-termination?)*
Applying the While Rule

- The most difficult thing is to come up with our invariant. This requires intuition - there is no algorithm that will do it for you.

- The postcondition we get after applying our rule has form $P \land \neg b$. This might not be the same as the postcondition $Q$ we want! But if

$$ (P \land \neg b) \iff Q $$

we can replace $P \land \neg b$ with the equivalent $Q$ to complete our proof.

But remember postcondition weakening - we don’t even need $P \land \neg b$ to be equivalent to $Q$. It is enough to have

$$ P \land \neg b \implies Q $$
Example

Suppose we want to find a precondition $P$ such that:

$$\{P\} \text{while } (n > 0) \text{ do } n := n - 1 \{n = 0\}$$

We want a loop invariant $P$ such that

- if $P$ is true initially, $P$ remains true each time around the loop;
- $(P \land \neg(n > 0)) \Rightarrow (n = 0)$

So the following looks a reasonable loop invariant:

$$P \equiv (n \geq 0)$$

The premise of the while rule then follows from the assignment axiom...
Example, Formally

1. \( \{ n - 1 \geq 0 \} \quad n := n - 1 \quad \{ n \geq 0 \} \) \hspace{1cm} (Assignment)

2. \( \{ n \geq 0 \land n > 0 \} \quad n := n - 1 \quad \{ n \geq 0 \} \) \hspace{1cm} (1, Precond. Equiv.)

3. \( \{ n \geq 0 \} \quad \text{while } (n > 0) \text{ do } n := n - 1 \quad \{ n \geq 0 \land \neg(n > 0) \} \) \hspace{1cm} (2, While)

4. \( \{ n \geq 0 \} \quad \text{while } (n > 0) \text{ do } n := n - 1 \quad \{ n = 0 \} \) \hspace{1cm} (3, Postcond. Equiv.)

There are other invariants we could have chosen, such as \( n = 0 \) and \( False \).

The one we chose was ‘better’ because it gave us a \textit{weaker} precondition.
Let’s Prove a Program!

Program (with specification):

\[
\{ \text{True} \}
\]

\[
i := 0;
\]

\[
s := 0;
\]

\[
\text{while } (i \neq n) \text{ do }
\]

\[
i := i + 1;
\]

\[
s := s + (2i - 1)
\]

\[
\{ s = n^2 \}
\]

(The sum of the first \( n \) odd numbers is \( n^2 \))
A Very Informal Analysis

Let's look at some examples:

\[ 1 = 1 = 1^2 \]
\[ 1 + 3 = 4 = 2^2 \]
\[ 1 + 3 + 5 = 9 = 3^2 \]
\[ 1 + 3 + 5 + 7 = 16 = 4^2 \ldots \]

It looks OK - let's see if we can prove it!
How can we prove it?

First we need a loop invariant $P$.

Think about the postcondition and the control condition:

```plaintext
while (i ≠ n) do
  i:=i+1;
  s:=s+(2*i-1)
{s = n²}
```

From the while rule, we want $\ (P \land i = n) \Rightarrow (s = n²) \$

Think about the loop body. Each time, $i$ increments and $s$ moves on to the next square number.

**Loop invariant** $P \equiv (s = i²)$ seems plausible.
Check $P$ is an invariant:

Using the assignment axiom and the sequence rule:

1. $\{s + (2 \times i - 1) = i^2\} \quad s := s + (2 \times i - 1) \quad \{s = i^2\}$  
   (Assignment)

2. $\{s + (2 \times (i + 1) - 1) = (i + 1)^2\} \quad i := i + 1 \quad \{s + (2 \times i - 1) = i^2\}$  
   (Asst.)

3. $\{s = i^2\} \quad i := i + 1 \quad \{s + (2 \times i - 1) = i^2\}$  
   (2, Precond. Equiv.)

4. $\{s = i^2\} \quad i := i + 1; \quad s := s + (2 \times i - 1) \quad \{s = i^2\}$  
   (1, 3, Sequencing)

So far, so good. ($P$ is an invariant.)
Completing the Proof

1. Strengthen the precondition to match the While rule premise:

\[ \{(s = i^2) \land (i \neq n)\} \ i:=i+1; \ s:=s+(2*i-1) \ {s = i^2} \]

2. Now apply the While rule and postcondition weakening:

\[ \{s = i^2\} \text{ while } \ldots \ s:=s+(2*i-1) \ {s = n^2} \]

3. Check that the initialisation establishes the invariant:

\[ \{0 = 0^2\} \ i:=0; \ s:=0 \ {s = i^2} \]

4. \((0 = 0^2) \equiv True\), so putting it all together with Sequencing we have

\[ \{True\} \ Program \ {s = n^2} \]
What About Termination?

Remember that Hoare Logic is for *Partial Correctness*.

Consider the code `while 1+1 = 2 do x:=0`. This will loop forever!

However you can still prove things about it, e.g.

\[
\begin{align*}
\{ True \} & \quad \text{while } 1+1 = 2 \quad \text{do } x:=0 \quad \{ False \}
\end{align*}
\]

(left as an exercise).

There are separate techniques to show termination. For example, one technique is to define a **loop variant** that changes every time we go around the loop but cannot change forever (e.g. a decreasing natural number).
Are the Rules Complete?

We have defined rules for a very simple language; new rules would be needed if we wanted e.g. for loops, arrays, exceptions...

But are our rules the ‘right’ ones for our little language?

- We have focused on soundness (i.e. every provable Hoare triple is true).
- Although we showed that each of the rules is sound, there are some assumptions we haven’t really discussed.
- With the same assumptions, the rules are also (relatively) complete\(^a\) for the language.

\(^a\) A logic is complete if every true expression is provable in the logic
What Are These Assumptions?

- The language we use for pre- and postconditions is **expressive** (i.e., can express *weakest liberal precondition* for all \( S \{ Q \} \)).

- We assumed no **aliasing** of variables. (In most real languages we can have multiple names for the one piece of memory.)

How is **aliasing** a problem?

- Suppose \( x \) and \( y \) refer to the same cell of memory.
  - We get \( \{ y + 1 = 5 \land y = 5 \} \ x := y + 1 \ \{ x = 5 \land y = 5 \} \)
  - i.e. \( \{ y = 4 \land y = 5 \} \ x := y + 1 \ \{ x = 5 \land y = 5 \} \)

so our assignment rule is no longer as strong as possible.
Reflections on Program Proof

What we’ve learned in these lectures:

Reasoning about assignment, sequencing, and conditionals is mostly mechanical:

- The biggest risk is that a ‘typo’ leads us into error.
- Simplifying, weakening and strengthening assertions can also be tricky.

Reasoning about while, however, requires creativity:

- Hoare: finding an appropriate invariant.
- Total correctness: finding an appropriate variant.
Reflections on Program Proof - Nulka

How do these lectures stack up to industrial formal methods?

Looking at the example of the flight control unit of the Nulka decoy missile (remember that?) we can see that there’s a lot more to it!

- We must be precise about what we want to prove about our program.
  
  Such **formal specifications** for Nulka were written in the language **Z**.
  
  Specification with Z is the topic of the next block of lectures.

- A larger and more impure language than the ‘toy’ language of the last five lectures was needed for more convenient programming.

  **SPARK Ada**, a ‘safe’ sublanguage of **Ada**, was used.
Reflections on Program Proof - Nulka ctd.

- ‘Proving the program correct’ in one go at the end is obviously impractical. So a constant cycle of programming and verification is needed. This was monitored via the requirement management tool **DOORS**.

- Many of the fiddly aspects of program proof were automated with **SPARK Examiner**. I’m afraid you can’t do this with your assignment!

- The program was then extensively tested ‘the old-fashioned way’, particularly in conjunction with the actual hardware.

(Ranald interviewed **Neville Pickering**, Senior Software Engineer at BAE Australia, before preparing this slide.)
Reflections on Program Proof - the Big Picture

Clearly there is much to learn before you can build your own decoy missile, nuclear power plant, etc!

Of course Nulka is just one example of formal methods in action. Different problems will require different approaches.

- The use of finite state automata in Weeks 8 and 9
- Guest lectures in Week 12

Still, these lectures give the foundations which more sophisticated formal methods are built on, and should help you to think about the programs you develop in a more traditional setting also.
References

The textbook has material on Hoare Logic


Some nice online notes with lots of examples:


A comprehensive early history of Hoare Logic appears in