Utrecht University

Logic for Computer Science

15 – Hoare logic

Paige Randall North

(based on Wouter Swierstra's slides)

University of Utrecht

Last time

Operational semantics

This lecture

Hoare logic

Syntax of programming languages

Last time we thought about the programming language defined by (in BNF notation):

$$
\begin{aligned}
e &::= n \mid x \mid e + e \mid e \times e \mid \ldots \\
b &::= \text{true} \mid \text{false} \mid b \mid\mid b \mid b \;\&\&\; b \mid e < e \mid \ldots \\
p &::= x := e \quad\mid\quad p; p \quad\mid\quad \text{if } b \text{ then } p \text{ else } p \text{ fi} \quad\mid\quad \text{while } b \text{ do } p \text{ od}
\end{aligned}
$$

Operational semantics of programming language

And we saw operational semantics:

$$\frac{[\![e]\!](\sigma) = n}{\langle x := e, \sigma \rangle \ \rightarrow \ \sigma[x \mapsto n]} \text{ Assignment}$$

$$\frac{[\![b]\!](\sigma) = \text{true} \qquad \langle p_1, \sigma \rangle \rightarrow \sigma'}{\langle \text{ if } b \text{ then } p_1 \text{ else } p_2 \text{ fi }, \sigma \rangle \rightarrow \sigma'} \text{ If-true} \qquad \frac{[\![b]\!](\sigma) = \text{false} \qquad \langle p_2, \sigma \rangle \rightarrow \sigma'}{\langle \text{ if } b \text{ then } p_1 \text{ else } p_2 \text{ fi }, \sigma \rangle \rightarrow \sigma'} \text{ If-false}$$

$$\frac{\langle p_1 , \sigma \rangle \rightarrow \sigma' \qquad \langle p_2 , \sigma' \rangle \rightarrow \sigma''}{\langle (p_1 ; p_2) , \sigma \rangle \rightarrow \sigma''} \text{ Seq}$$

$$\frac{[\![b]\!](\sigma) = \mathit{false}}{\langle \text{ while } b \text{ do } p , \sigma \rangle \rightarrow \sigma} \text{ While-false}$$

$$\frac{[\![b]\!](\sigma) = \mathit{true} \qquad \langle p , \sigma \rangle \rightarrow \sigma' \qquad \langle \text{ while } b \text{ do } p \text{ od} , \sigma' \rangle \rightarrow \sigma''}{\langle \text{ while } b \text{ do } p , \sigma \rangle \rightarrow \sigma''} \text{ While-true}$$

From operational semantics to logic

These operational semantics determine how a program is executed from a given initial state σ.

But consider the following mini-program:

```
if x < y then r := x else r := y
```

Can we prove that after execution `r` will store the minimum of `x` and `y`?

This requires reasoning about *all* possible states – rather than *one* initial state.

To perform this kind of reasoning, we need a logic to reason about **all possible** executions.

From operational semantics to logic

These operational semantics determine how a program is executed from a given initial state σ.

But consider the following mini-program:

```
if x < y then r := x else r := y
```

Can we prove that after execution `r` will store the minimum of `x` and `y`?

This requires reasoning about *all* possible states – rather than *one* initial state.

To perform this kind of reasoning, we need a logic to reason about **all possible** executions.

This motivates the shift from *operational semantics* to *program logic*.

Specifications

A **formal specification** is a mathematical description of what a program should do.

Such a specification ignores many important details, such as the *non-functional* requirements about how fast the program is, the language used for its implementation, the development cost, etc.

Instead, we use a formal specification to answer one question:

Is this program doing what it should?

Specificiations

We will give specifications in the form of a pre- and post-condition that are predicates on our states.

Intuitively, the precondition captures the assumptions the program makes about the initial state;

The postcondition expresses the properties that are guaranteed to hold after the program has finished executing.

Notation

To define our logic for reasoning about programs, we introduce the following notation:

$\{ P \}$	p	$\{ Q \}$
pre-condition	programme	post-condition

For each state σ that satisfies the precondition *P*,

if executing $\langle p, \sigma \rangle$ terminates in some final state τ, then τ must satisfy *Q*.

We'll define this – once again – using inference rules. But's let look at some examples first.

Examples

- $\{ x = 3 \}$ `x := x + 1` $\{ x = 4 \}$

Unsurprising: if x = 3, after executing `x := x + 1`, we know x = 4.

- $\{ x = A \wedge y = B \}$ z:= x; x := y; y := z $\{ x = B \wedge y = A \}$

This is more interesting: it works for *any* values of A and B – this describes many possible executions, starting from some state for which the precondition holds.

- { true } while true do p := 0 od {p = 500 }

Note that the postcondition only makes a statement about the final state. If the program never terminates, it trivially satisfies any postcondition!

Examples

{ true }

```
x := 3;
p := 0;
i := 1;
while i <= x do
      p := p + i;
      i := i+1
od
```

{ p = 6 }

How can we write a derivation proving this? What are the *inference rules* that we can use?

Hoare logic

We'll go through a handful of inference rules for proving statements of the form $\{P\}\ p\ \{Q\}$.

Together these define a logic known as *Hoare logic* – named after Tony Hoare, a British computer scientist who pioneered the approach together with Edsger Dijkstra, Robert Floyd, and others.

Hoare logic – assignment

What rule should we use for assignment? We've seen one example:

$\{ x = 3 \}$ `x := x + 1` $\{ x = 4 \}$

We could generalise this:

$\{ x = N \}$ `x := x + 1` $\{ x = N + 1 \}$

Hoare logic – assignment

What rule should we use for assignment? We've seen one example:

$\{x = 3\}$ `x := x + 1` $\{x = 4\}$

We could generalise this:

$\{x = N\}$ `x := x + 1` $\{x = N + 1\}$

But what if we want to assign another expression than `x + 1`?

Or what if the pre- and postcoditions are not a simple equality?

Hoare logic – assignment

What rule should we use for assignment? We've seen one example:

$\{ x = 3 \}$ `x := x + 1` $\{ x = 4 \}$

We could generalise this:

$\{ x = N \}$ `x := x + 1` $\{ x = N + 1 \}$

But what if we want to assign another expression than `x + 1`?

Or what if the pre- and postcoditions are not a simple equality?

What's the most general rule?

Hoare logic – assignment

$$\frac{}{\{\ Q[x \backslash e]\ \} \quad \mathtt{x := e} \quad \{\ Q\ \}}\ \text{Assign}$$

- We write $Q[x \backslash e]$ for the result of replacing all the occurrences of x with e in Q.
- This rule seems backwards! It helps to read it back to front: in order for Q to hold after the assignment x := e, the precondition $Q[x \backslash e]$ should already hold.

Let's look at some examples. . .

Hoare logic – assignment

$$\frac{}{\{\ Q[x \backslash e]\ \}\quad \mathtt{x := e}\quad \{\ Q\ \}}\ \text{Assign}$$

Here are three different examples of this rule in action:

$$\frac{}{\{\ y = 3\ \}\ x := 3\ \{\ y = x\ \}}\ \text{Assign}$$

$$\frac{}{\{\ x = N + 1\ \}\ x := x - 1\ \{\ x = N\ \}}\ \text{Assign}$$

$$\frac{}{\{\ x + y = V\ \}\ z := x + y\ \{\ z = V\ \}}\ \text{Assign}$$

Hoare logic – conditional

$$\frac{????\qquad ????}{\{\,P\,\}\quad \text{if } b \text{ then } p_1 \text{ else } p_2 \quad \{\,Q\,\}}\ \text{If}$$

What happens when we execute an if statement?

We will continue executing either the 'then-branch' or the 'else-branch'; if both branches manage to end in a state satisfying Q, the entire if-statement will.

Hoare logic – conditional

$$\frac{\{\, P \wedge b \,\} \quad p_1 \quad \{Q\} \qquad \{\, P \wedge \neg b \,\} \quad p_2 \quad \{Q\}}{\{\, P \,\} \quad \text{if } b \text{ then } p_1 \text{ else } p_2 \quad \{\, Q \,\}} \text{ If}$$

By checking whether the guard *b* holds or not, we learn something. As a result, the precondition changes in both branches of the if-statement.

Hoare logic – conditional

$$\frac{\{\,P \wedge b\,\}\quad p_1 \quad \{Q\} \qquad \{\,P \wedge \neg b\,\}\quad p_2 \quad \{Q\}}{\{\,P\,\}\quad \text{if } b \text{ then } p_1 \text{ else } p_2 \quad \{\,Q\,\}}\ \text{If}$$

By checking whether the guard b holds or not, we learn something. As a result, the precondition changes in both branches of the if-statement.

Question

Use the two rules we have seen so far to show that:

$$\{\,0 \leqslant \text{x} \leqslant 5\,\}\quad \text{if x} < 5 \text{ then x := x+1 else x := 0 fi} \quad \{\,0 \leqslant \text{x} \leqslant 5\,\}$$

Hoare logic – composition

$$\frac{\{\, P \,\}\ \ p_1\ \ \{R\} \qquad \{\, R \,\}\ \ p_2\ \ \{Q\}}{\{\, P \,\}\ \ \ p_1\,;p_2\ \ \ \{\, Q \,\}}\ \text{Seq}$$

The rule for composition of programs is beautiful – it may remind you of function composition.

If we know that P holds of our initial state, we can run p_1 to reach a state satisfying R;

But now we can run p_2 on this state, to produce a state satisfying Q.

Hoare logic – bookkeeping

$$\frac{\{\,P\,\}\;\; p_1 \;\;\{R\} \qquad \{\,R\,\}\;\; p_2 \;\;\{Q\}}{\{\,P\,\}\;\; p_1 ; p_2 \;\;\{\,Q\,\}}\;\text{Seq}$$

If you look at this rule though, you may need to be very lucky to be able to use it: the postcondition of p_1 and precondition of p_2 must match **exactly**...

This rarely happens in larger derivations.

To still be able to use such rules, we need an additional 'bookkeeping' rule.

Hoare logic – consequence

$$\frac{P' \Rightarrow P \qquad \{P\}\ p\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ p\ \{Q'\}}\ \text{Consequence}$$

The **rule of consequence** states that we can change the pre- and postcondition provided:

- the **precondition** is **stronger** – that is, $P' \Rightarrow P$;
- the **postcondition** is **weaker** – that is, $Q \Rightarrow Q'$;

We can justify this rule by thinking back to what a statement of the form $\{P\}\ p\ \{Q\}$ means:

Hoare logic

For each state σ that satisfies the precondition P,

if executing $\langle p, \sigma\rangle$ terminates in some final state τ, then τ must satisfy Q.

Hoare logic – while

$$\frac{\{\,??? \wedge b\,\}\quad p \quad \{???\}}{\{\,P\,\}\quad \text{while } b \text{ do } p \text{ end} \quad \{\,??? \wedge \neg b\,\}}\ \text{While}$$

The general structure of the rule for loops should be along these lines:

- some precondition *P* should hold initially;
- the loop body may assume that the guard *b* is true;
- after completion, we know that the guard *b* is no longer true.

But how should we fill in the question marks?

Hoare logic – while

$$\frac{\{\, P \wedge b \,\} \quad p \quad \{???\}}{\{\, P \,\} \quad \text{while } b \text{ do } p \text{ end} \quad \{\, ??? \wedge \neg b \,\}}\ \text{While}$$

When we first enter the loop body, we know that *P* still holds.

Hoare logic – while

$$\frac{\{\, P \wedge b \,\} \quad p \quad \{P\}}{\{\, P \,\} \quad \text{while } b \text{ do } p \text{ end} \quad \{\, ??? \wedge \neg b \,\}} \text{While}$$

After completing the loop body, we may need to execute the loop body again (and again and again and again).

The precondition of *p* should *continue to hold during execution*.

Hoare logic – while

$$\frac{\{\, P \wedge b \,\} \;\; p \;\; \{P\}}{\{\, P \,\} \quad \text{while } b \text{ do } p \text{ end} \quad \{\, P \wedge \neg b \,\}} \text{ While}$$

After running the loop body over and over again, the postcondition of the entire while statement says that both P and $\neg b$ hold.

Hoare logic – while

$$\frac{\{\, P \land b \,\} \;\; p \;\; \{P\}}{\{\, P \,\} \quad \text{while } b \text{ do } p \text{ end} \quad \{\, P \land \neg b \,\}} \text{ While}$$

After running the loop body over and over again, the postcondition of the entire while statement says that both P and $\neg b$ hold.

We call P the **loop invariant** – it continues to hold throughout the execution of the while loop.

Example

Question

Give a derivation of the following statement:

$$\{x \geqslant 5\} \;\; \texttt{while x > 5 do x := x - 1 od} \;\; \{x \geqslant 5 \wedge x \leqslant 5\}$$

Example

Question

Give a derivation of the following statement:

$$\{ x \geqslant 5 \} \quad \texttt{while x > 5 do x := x - 1 od} \quad \{ x \geqslant 5 \wedge x \leqslant 5 \}$$

$$\dfrac{\dfrac{\{ x - 1 \geqslant 5 \} \quad x := x - 1 \quad \{ x \geqslant 5 \}}{\{ x \geqslant 5 \wedge x > 5 \} \quad x := x - 1 \quad \{ x \geqslant 5 \}}\ \text{Consq}}{\{ x \geqslant 5 \} \quad \text{while } x > 5 \text{ do } x := x - 1 \text{ end} \quad \{ x \geqslant 5 \wedge x \leqslant 5 \}}\ \text{While}$$

Hoare logic – soundness and completeness

How can we be sure that we chose the right set of inference rules?

Once again, we can show that these rules are **sound** and **complete** with respect to our operational semantics.

Soundness If we can prove $\{P\}\ p\ \{Q\}$ then for all states σ such that $P(\sigma)$, if $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \to \tau$ then $Q(\tau)$

Completeness For all states σ and τ and programs p, such that $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \to \tau$. Then for all preconditions P and postconditions Q for which $P(\sigma) \Rightarrow Q(\tau)$, there exists a derivation showing $\{P\}\ p\ \{Q\}$.

We can reason about all possible program behaviours using the rules of Hoare logic.

Hoare logic – soundness and completeness

How can we be sure that we chose the right set of inference rules?

Once again, we can show that these rules are **sound** and **complete** with respect to our operational semantics.

Soundness If we can prove $\{P\}\ p\ \{Q\}$ then for all states σ such that $P(\sigma)$, if $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \rightarrow \tau$ then $Q(\tau)$

Completeness For all states σ and τ and programs *p*, such that $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \rightarrow \tau$. Then for all preconditions *P* and postconditions *Q* for which $P(\sigma) \Rightarrow Q(\tau)$, there exists a derivation showing $\{P\}\ p\ \{Q\}$.

We can reason about all possible program behaviours using the rules of Hoare logic.

Put differently, we never need to *execute* code to prove its correctness.

From While to C#

We still need to consider a bucketload of missing features to turn our simple imperative language into a more realistic programming language:

- Classes, objects, inheritance, abstract classes, virtual methods, . . .
- Strings, arrays, and other richer types
- Exceptions;
- Concurrency;
- Recursion;
- Shared memory;
- Standard libraries;
- Compiler primitives;
- Foreign function interfaces;
- . . .

Program calculation

Problem

Given a precondition P and postcondition Q, find a program p such that $\{P\}\ p\ \{Q\}$ holds.

There is a rich field of research on **program calculation** that tries to solve this problem.

Approaches include the refinement calculus, pioneered by people such as Edsger Dijkstra, Tony Hoare, and many others.

Industrial strength program verification

Hoare logic (and its descendents) still form the basis of state-of-the-art verification tools:

- The Infer suite developed by Facebook;
- Automated theorem provers such as Dafny, Spec#, Key, and many others;

Computers are very, very good at computing and checking derivations in Hoare logic.

Instead of pages and pages of scribbles, this technology is catching bugs without ever having to run a single line of code.

Wrapping up

Please fill in the Caracal evaluation form!

Material

- Lecture notes - Chapter 3;
- Check the wikipedia page on Hoare Logic for lots more examples and explanation.