# Logic for Computer Science

15 – Hoare logic

Wouter Swierstra

University of Utrecht

## Last time

**Natural deduction**

## This lecture

**Semantics of computer programs**

**A logic for reasoning about them**

## Syntax of programming language

The BNF notation can also be used to define *programming languages*:

$$e \quad ::= \quad n \mid x \mid e + e \mid e \times e \mid \ldots$$

$$b \quad ::= \quad \text{true} \mid \text{false} \mid b \,||\, b \mid b \,\&\&\, b \mid e < e \mid \ldots$$

$$p \quad ::= \quad x := e \quad \mid \quad p; p \quad \mid \quad \text{if } b \text{ then } p \text{ else } p \text{ fi} \quad \mid \quad \text{while } b \text{ do } p \text{ end}$$

We distinguish between (integer) expressions $e$ and boolean expressions $b$.

We haven't completely defined all the operations in $e$ and $b$ (such as negation, subtraction, etc.) – but this won't be the focus of this lecture.

We assume we have some set of variables $V$ – implicitly we use $x \in V$ throughout the above definitions; similarly, $n \in \mathbb{N}$, for instance.

## Semantics?

But this doesn’t say anything about a program’s behaviour...

How do these programs behave?

How can we define the **semantics** of programs?

If logic is about studying proofs – how can we prove a program is correct?

## Semantics for expressions

$$e \quad ::= \quad n \mid x \mid e + e \mid e \times e \mid \ldots$$

$$b \quad ::= \quad \text{true} \mid \text{false} \mid b \mid\mid b \mid b \; \&\& \; b \mid e < e \mid \ldots$$

In the previous lecture, I sketched how to give a semantics to propositional logic formulas;

We can use the same technique here...

## Semantics for expressions

$$e \quad ::= \quad n \mid x \mid e + e \mid e \times e \mid \ldots$$

$$b \quad ::= \quad \text{true} \mid \text{false} \mid b \mid\mid b \mid b \,\&\&\, b \mid e < e \mid \ldots$$

In the previous lecture, I sketched how to give a semantics to propositional logic formulas;

We can use the same technique here...

**Idea** We can write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean:

$$[\![e]\!] : \textbf{Int} \qquad [\![b]\!] : \textbf{Bool}$$

## Semantics for expressions

$$\begin{aligned} e &::= n \mid x \mid e + e \mid e \times e \mid \ldots \\ b &::= \text{true} \mid \text{false} \mid b_1 \mid\mid b_2 \mid b_1 \mathbin{\&\&} b_2 \mid e_1 < e_2 \mid \ldots \end{aligned}$$

**Idea** We can write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean.

But – this doesn't quite work: what is the value of `x + 3`?

## Semantics for expressions

$$e \quad ::= \quad n \mid x \mid e + e \mid e \times e \mid \ldots$$

$$b \quad ::= \quad \text{true} \mid \text{false} \mid b_1 \;||\; b_2 \mid b_1 \;\&\&\; b_2 \mid e_1 < e_2 \mid \ldots$$

**Idea** We can write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean.

But – this doesn't quite work: what is the value of `x + 3`?

This depends on the last value we assigned to the variable `x` – we need to keep track of the computer's memory.

## Memory

We can mode the contents of the computer's memory as a function $V \longrightarrow \textbf{Int}$ – this function tells us for each variable in $V$ what its current value is.

We can use this function to write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean.

$$[\![e]\!] : (V \longrightarrow \textbf{Int}) \longrightarrow \textbf{Int} \qquad [\![b]\!] : (V \longrightarrow \textbf{Int}) \longrightarrow \textbf{Bool}$$

Just as we saw for the semantics of propositional logic, we use this function to associate meaning with variables.

## Example

Previously we didn't know the meaning of `x + 3` – but what if we are given the current memory $\sigma : V \to \mathbf{Int}$ and we know that $\sigma(x) = 7$:

$$[\![x + 3]\!]_\sigma = [\![x]\!]_\sigma + [\![3]\!]_\sigma = \sigma(x) + 3 = 7 + 3 = 10$$

We can compute the integer associated with expressions and the boolean value associated with boolean expressions provided we know the current *state* of the computer's memory.

## Example

Previously we didn't know the meaning of `x + 3` – but what if we are given the current memory $\sigma : V \to \mathbf{Int}$ and we know that $\sigma(x) = 7$:

$$[\![x+3]\!]_\sigma = [\![x]\!]_\sigma + [\![3]\!]_\sigma = \sigma(x) + 3 = 7 + 3 = 10$$

We can compute the integer associated with expressions and the boolean value associated with boolean expressions provided we know the current *state* of the computer's memory.

**Note:** the semantics defined in this style is 'obvious' – but the addition symbol in $[\![x+3]\!]_\sigma$ and $[\![x]\!]_\sigma + [\![3]\!]_\sigma$ are very different! The first refers to the syntax of our language—it could equally well have been written `add(x,y)`, whereas the second is the addition function on **Int**.

## Semantics for statements

$$
\begin{aligned}
p \quad ::= \quad & x := e \\
& \mid p; p \\
& \mid \text{if } b \text{ then } p \text{ else } p \\
& \mid \text{while } b \text{ do } p
\end{aligned}
$$

How should I define a semantics?

A statement such as:

$$x := 17$$

doesn't return any interesting result – but rather *modifies the state of our program*

## Semantics for statements

$$
\begin{aligned}
p \quad ::= \quad & x := e \\
& \mid p; p \\
& \mid \text{if } b \text{ then } p \text{ else } p \\
& \mid \text{while } b \text{ do } p
\end{aligned}
$$

How should I define a semantics?

A statement such as:

$$x := 17$$

doesn’t return any interesting result – but rather *modifies the state of our program*

Any semantics for our language should carefully describe how the state changes...

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

We start execution from some begin state – let's assume that the variables `x`, `p` and `i` all start as 0,1,2 respectively. That is initially we're in a state $\sigma$ which satisfies:

$$\sigma(\mathtt{x}) = 0 \quad \sigma(\mathtt{p}) = 1 \quad \sigma(\mathtt{i}) = 2$$

Now let's run this program step by step…

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(\mathtt{x}) = 0 \quad \sigma(\mathtt{p}) = 1 \quad \sigma(\mathtt{i}) = 2$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(x) = 0 \quad \sigma(p) = 1 \quad \sigma(i) = 2$$

$$\rightarrow$$

$$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 2$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 2$

$\rightarrow$

$\sigma(x) = 3 \quad \sigma(p) = 0 \quad \sigma(i) = 2$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(x) = 3 \quad \sigma(p) = 0 \quad \sigma(i) = 2$$

$$\to$$

$$\sigma(x) = 3 \quad \sigma(p) = 0 \quad \sigma(i) = 1$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(x) = 3 \quad \sigma(p) = 0 \quad \sigma(i) = 1$$

$$\rightarrow$$

$$\sigma(x) = 3 \quad \sigma(p) = 0 \quad \sigma(i) = 1$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 0 \quad \sigma(\mathtt{i}) = 1$

$\rightarrow$

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 1 \quad \sigma(\mathtt{i}) = 1$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 1$

$\rightarrow$

$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 2$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 2$$

$\rightarrow$

$$\sigma(x) = 3 \quad \sigma(p) = 1 \quad \sigma(i) = 2$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 1 \quad \sigma(\mathtt{i}) = 2$$

$$\rightarrow$$

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 3 \quad \sigma(\mathtt{i}) = 2$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 3 \quad \sigma(\mathtt{i}) = 2$

$\rightarrow$

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 3 \quad \sigma(\mathtt{i}) = 3$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 3 \quad \sigma(\mathtt{i}) = 3$$

$\rightarrow$

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 3 \quad \sigma(\mathtt{i}) = 3$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(x) = 3 \quad \sigma(p) = 3 \quad \sigma(i) = 3$$

$$\rightarrow$$

$$\sigma(x) = 3 \quad \sigma(p) = 6 \quad \sigma(i) = 3$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 3$

$\rightarrow$

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 4$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 4$$

$\rightarrow$

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 4$$

## Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

Our program terminates in the following state:

$$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 4$$

## Making this more precise...

This gives some idea of how a program is executed.

But this example raises some interesting questions:

- What would have happened if we would have used a different initial state? Would the results have been the same?
- Does every program terminate in a finite number of steps?
- Can our program ‘go wrong’ somehow – dividing by zero or accessing unallocated memory?

My goal isn’t to answer all these questions – but just to highlight the kind of issues you need to address when making the semantics of programming languages precise.

Let’s try to give a mathematical account of program execution.

## Modelling state

We model the current state of our computer's memory (storing the value of all our variables) as function:

$$\sigma : V \to \textbf{Int}$$

If we want to know the value of a given variable $x$, we can simply look it up $\sigma(x)$;

We will sometimes also need to *update* the current memory.

We write $\sigma[x \mapsto n]$ for the memory that is the same as $\sigma$ for all variables in $V$ **except** $x$, where it stores the value $n$.

In other words, this updates the current memory at one location, setting the value for $x$ to $n$.

## The meaning of our programs

Using the *inference rule notation* from the previous lecture, we can formalize the semantics of our language.

The key idea is that we define a relation on While $\times$ State $\times$ State, written as follows:

$$\langle p, \sigma \rangle \to \sigma'$$

This relation holds when executing the program p from the initial state $\sigma$ terminates in the final state $\sigma'$.

This formalizes the example we had a few slides ago, where we 'stepped through' the execution of a program to compute the final state after executing a program.

This relation gives an **operational semantics** for our programs, describing how to execute a program step by step.

## Operational semantics

$$p \quad ::= \quad x := e \quad | \quad p; p \quad | \quad \text{if } b \text{ then } p \text{ else } p \text{ fi} \quad | \quad \text{while } b \text{ do } p \text{ end}$$

We have four language constructs – we'll only need very few rules to describe their behaviour (in contrast to, say, natural deduction rules for propositional logic).

- Assignments – $x := e$
- Sequential composition – $p; p$
- Conditionals – if $b$ then $p$ else $p$ fi
- Loops – while $b$ do $p$ end

## Assignment

$$\frac{[\![e]\!]_\sigma = n}{\langle x := e, \sigma\rangle \;\to\; \sigma[x \mapsto n]}\ \text{Assignment}$$

There is one rule for handling assignment.

The assignment statement always terminates in one step.

Starting in the state $\sigma$, executing $x := e$ produces a new state, $\sigma[\mathrm{x} \mapsto \mathrm{n}]$, that updates the memory location for $x$ to store the value of $e$.

For example, given a state $\sigma$ satisfying $\sigma(y) = 3$, we can execute the command $x := y + 2$ by:

$$\frac{[\![y + 2]\!]_\sigma = 5}{\langle x := y + 2, \sigma\rangle \;\to\; \sigma[x \mapsto 5]}$$

## Conditionals

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

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

There are two rules for evaluating if-then-else statements:

- if the guard *b* is true, we continue evaluating the then branch, leaving the state unchanged;
- if the guard *b* is false, we continue evaluating the else branch, leaving the state unchanged;

## Example

Suppose we start from a state σ satisfying σ(x) = 3 and σ(y) = 10

We can execute the following program:

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

Using the previous derivation rules:

$$\dfrac{[\![x < y]\!](\sigma) = \text{true} \qquad \dfrac{[\![x]\!](\sigma) = 3}{\langle\, r := x \,,\, \sigma \,\rangle \to \sigma[\mathrm{r} \mapsto 3]}\ \text{Assignment}}{\langle\, \text{if } x < y \text{ then } r := x \text{ else } r := y \text{ fi} \,,\, \sigma \,\rangle \to \sigma[\mathrm{r} \mapsto 3]}$$

## Sequential composition

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

The rule for sequential composition chains together the execution of two programs, $p_1$ and $p_2$:

- if executing $p_1$ from the initial state $\sigma$ terminates in a state $\sigma'$;
- and executing $p_2$ from $\sigma'$ terminates in a state $\sigma''$;
- the composite program $p_1 ; p_2$ terminates in $\sigma''$.

## Loops

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

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

We need two rules to handle loops:

- if the guard $b$ is false, we do not enter the loop body or change the state – but rather halt in the current state $\sigma$;
- if the guard $b$ is true, we execute the loop body $p$, and then continue executing the main while loop.

## Operational semantics

These six rules determine precisely how a program is executed.

Given any initial state $\sigma$ and program $p$, we can repeatedly apply these rules to determine if the program terminates or not.

This formalizes the process I went through with large example I did at the beginning of the lecture: showing how to evaluate an example program from some initial state.

## Labelled transition systems

This semantics forms a **labelled transition system**:

- the set of states are the current program $p$ and memory $\sigma$;
- our operational semantics determine the transition relation between our states;
- if we extend our language with other effects, such as opening a window, writing to stdout, or asking the user to input a character – we can add further *actions* to our system to observe these effects.

## From operational semantics to logic

These operational semantics determine how a program is executed from a given initial state $\sigma$.

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 minimal value 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 $\sigma$.

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 minimal value 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 $\sigma$ that satisfies the precondition *P*,

if executing $\langle p, \sigma \rangle$ terminates in some final state $\tau$, then $\tau$ 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 give 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

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

- 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

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

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:

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

## Hoare logic – composition

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

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

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

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 $\sigma$ that satisfies the precondition $P$,

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

## Hoare logic – while

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

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

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

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

## Hoare logic – while

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

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

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

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

$$\frac{\{ P \wedge b \} \quad p \quad \{P\}}{\{ P \} \quad \text{while } b \text{ do } p \text{ od} \quad \{ P \wedge \neg b \}} \text{ While}$$

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

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:

$\{ x \geq 5 \}$ `while x > 5 do x := x - 1 od` $\{ x \geq 5 \wedge x \leqslant 5 \}$

## Example

**Question**

Give a derivation of the following statement:

$\{ x \geq 5 \}$ `while x > 5 do x := x - 1 od` $\{ x \geq 5 \wedge x \leqslant 5 \}$

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

## 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 $\sigma$ such that $P(\sigma)$, if $\langle\, \mathrm{p}\,,\, \sigma\,\rangle \longrightarrow \tau$ then $Q(\tau)$

**Completeness** For all states $\sigma$ and $\tau$ and programs $p$, such that $\langle\, \mathrm{p}\,,\, \sigma\,\rangle \longrightarrow \tau$. 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 $\sigma$ such that $P(\sigma)$, if $\langle\, p\,,\ \sigma\,\rangle \rightarrow \tau$ then $Q(\tau)$

**Completeness** For all states $\sigma$ and $\tau$ and programs $p$, such that $\langle\, p\,,\ \sigma\,\rangle \rightarrow \tau$. 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.