Hoare Logic

(Slides modified from Mike Gordon, Xinyu Feng)

Outline

Program Specification and Verification

This class: *formal* ways of specifying and verifying software

This contrasts with *informal* methods:

- natural language specifications
- testing

Goals of this class:

- enable you to understand and criticize formal methods
- provide a stepping stone to current research

Testing

Testing can quickly find obvious bugs

- only trivial programs can be tested exhaustively
- the cases you do not test can still hide bugs
- coverage tools can help

How do you know what the correct test results should be?

Many industries' standards specify maximum failure rates

- e.g. fewer than 10^{-6} failures per second
- assurance that such rates have been achieved cannot be obtained by testing

Formal Methods

- Formal Specification: using mathematical notation to give a precise description of what a program should do
- Formal Verification: using precise rules to mathematically prove that a program satisfies a formal specification
- Formal Development (Refinement): developing programs in a way that ensures mathematically they meet their formal specifications

Formal Methods should be used in conjunction with testing, not as a replacement

Should we always use formal methods?

They can be expensive

- though can be applied in varying degrees of effort

There is a trade-off between expense and the need for correctness

For some applications, correctness is especially important

- nuclear reactor controllers
- car braking systems
- fly-by-wire aircraft
- software controlled medical equipment
- voting machines
- cryptographic code

Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible

Floyd-Hoare Logic

This class is concerned with Floyd-Hoare Logic

- also known just as Hoare Logic

Hoare Logic is a method of reasoning mathematically about *imperative* programs

It is the basis of mechanized program verification systems

Developments to the logic still under active development, e.g.

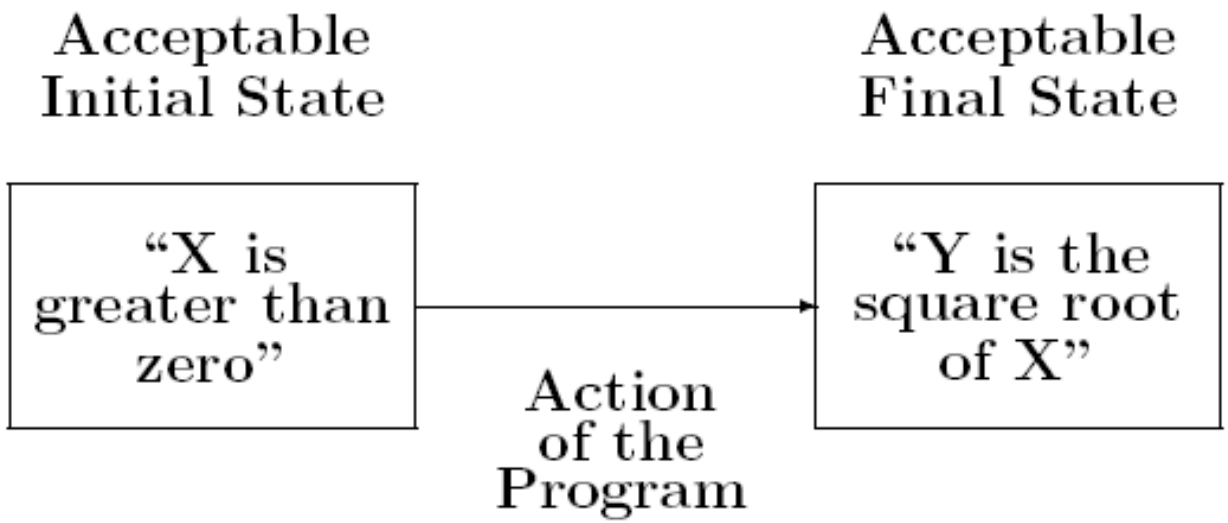
- separation logic (reasoning about pointers)
- concurrent program logics

A simple imperative language

$$
\begin{array}{rlcl}
\text{(IntExp)} & e & ::= & n \\
& & \mid & x \\
& & \mid & e + e \mid e - e \mid \ldots \\
\\
\text{(BoolExp)} & b & ::= & \textbf{true} \mid \textbf{false} \\
& & \mid & e = e \mid e < e \mid e > e \\
& & \mid & \neg b \mid b \wedge b \mid b \vee b \mid \ldots \\
\\
\text{(Comm)} & c & ::= & \textbf{skip} \\
& & \mid & x := e \\
& & \mid & c \,\textbf{;}\, c \\
& & \mid & \textbf{if } b \textbf{ then } c \textbf{ else } c \\
& & \mid & \textbf{while } b \textbf{ do } c \\
\\
\text{(State)} & \sigma & \in & \text{Var} \to \text{Int}
\end{array}
$$

Outline

Specifications of imperative programs



Hoare's notation (Hoare triples)

For a program c,

- partial correctness specification:

$$\{p\}c\{q\}$$

- total correctness specification:

$$[p]c[q]$$

Here p and q are *assertions*, i.e., conditions on the program variables used in c.

- p is called *precondition*, and q is called *postcondition*

Hoare's original notation was $p\{c\}q$ not $\{p\}c\{q\}$, but the latter form is now more widely used

Meanings of Hoare triples

A partial correctness specification $\{p\}c\{q\}$ is valid, iff

- if c is executed in a state initially satisfying p
- and *if* the execution of c terminates
- then the final state satisfies q

It is "partial" because for $\{p\}c\{q\}$ to be true it is *not* necessary for the execution of c to terminate when started in a state satisfying p

It is only required that *if* the execution terminates, then q holds

Meanings of Hoare triples

A partial correctness specification $\{p\}c\{q\}$ is valid, iff

- if c is executed in a state initially satisfying p
- and *if* the execution of c terminates
- then the final state satisfies q

A total correctness specification $[p]c[q]$ is valid, iff

- if c is executed in a state initially satisfying p
- then the execution of c terminates
- and the final state satisfies q

Informally: Total correctness = Termination + Partial correctness

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid
$\{x \leq 10\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid
$\{x \leq 10\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[x \leq 10]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	valid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid
$\{x \leq 10\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[x \leq 10]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	valid
$\{\mathbf{true}\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid
$\{x \leq 10\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[x \leq 10]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	valid
$\{\mathbf{true}\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[\mathbf{true}]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	invalid

Examples of program specs

$\{x = 1\}$	$x := x + 1$	$\{x = 2\}$	valid
$\{x = 1\}$	$x := x + 1$	$\{x = 3\}$	invalid
$\{x - y > 3\}$	$x := x - y$	$\{x > 2\}$	valid
$[x - y > 3]$	$x := x - y$	$[x > 2]$	valid
$\{x \leq 10\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[x \leq 10]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	valid
$\{\mathbf{true}\}$	**while** $x \neq 10$ **do** $x := x + 1$	$\{x = 10\}$	valid
$[\mathbf{true}]$	**while** $x \neq 10$ **do** $x := x + 1$	$[x = 10]$	invalid
$\{x = 1\}$	**while true do skip**	$\{x = 2\}$	valid

Total correctness

Informally: Total correctness = Termination + Partial correctness

Total correctness is the ultimate goal

- it is usually easier to show partial correctness and termination separately

Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of x

while $x > 1$ **do**
  if $odd(x)$ **then** $x := (3 * x) + 1$ **else** $x := x/2$

Logical variables

$$\{x = x_0 \wedge y = y_0\}\ r := x\,;x := y\,;y := r\ \{x = y_0 \wedge y = x_0\}$$

This says that *if* the execution of $r := x\,;x := y\,;y := r$ terminates (which it does), *then* the values of x and y are exchanged.

The variables x_0 and y_0 are used to name the initial values of program variables x and y.

- Used in assertions only. Not occur in the program.
- Have constant values.

They are called *logical variables*. (Sometimes it is also called *ghost variables*.)

More simple examples

- $\{x = x_0 \wedge y = y_0\}\ x := y\,;y := x\ \{x = y_0 \wedge y = x_0\}$
  - This says that $x := y\,;y := x$ exchanges the values of x and y
  - This is not valid

More simple examples

- $\{x = x_0 \wedge y = y_0\}\ x := y\,;y := x\ \{x = y_0 \wedge y = x_0\}$
  - This says that $x := y\,;y := x$ exchanges the values of x and y
  - This is not valid
- $\{\mathbf{true}\}c\{q\}$
  - This says that whenever c terminates, then q holds

More simple examples

- $\{x = x_0 \wedge y = y_0\}\ x := y\,;y := x\ \{x = y_0 \wedge y = x_0\}$
  - This says that $x := y\,;y := x$ exchanges the values of x and y
  - This is not valid
- $\{\mathbf{true}\}c\{q\}$
  - This says that whenever c terminates, then q holds
- $[\mathbf{true}]c[q]$
  - This says that c always terminates and ends in a state where q holds

More simple examples

- $\{x = x_0 \wedge y = y_0\}\ x := y\,;y := x\ \{x = y_0 \wedge y = x_0\}$
  - This says that $x := y\,;y := x$ exchanges the values of x and y
  - This is not valid
- $\{\mathbf{true}\}c\{q\}$
  - This says that whenever c terminates, then q holds
- $[\mathbf{true}]c[q]$
  - This says that c always terminates and ends in a state where q holds
- $\{p\}c\{\mathbf{true}\}$
  - This is valid for every condition p and every command c

More simple examples

- $\{x = x_0 \wedge y = y_0\}\ x := y\,;y := x\ \{x = y_0 \wedge y = x_0\}$
  - This says that $x := y\,;y := x$ exchanges the values of x and y
  - This is not valid
- $\{\mathbf{true}\}c\{q\}$
  - This says that whenever c terminates, then q holds
- $[\mathbf{true}]c[q]$
  - This says that c always terminates and ends in a state where q holds
- $\{p\}c\{\mathbf{true}\}$
  - This is valid for every condition p and every command c
- $[p]c[\mathbf{true}]$
  - This says that c terminates if initially p holds
  - It says nothing about the final state

Specification can be tricky

"The program must set y to the maximum of x and y"

Specification can be tricky

"The program must set y to the maximum of x and y"

$[\textbf{true}]c[y = max(x, y)]$

Specification can be tricky

"The program must set y to the maximum of x and y"

$[\mathbf{true}]c[y = max(x, y)]$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**

Specification can be tricky

"The program must set y to the maximum of x and y"

$[\mathbf{true}]c[y = max(x, y)]$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**
- Another?
  - **if** $x \geq y$ **then** $x := y$ **else skip**

Specification can be tricky

"The program must set y to the maximum of x and y"

$[\mathbf{true}]c[y = max(x, y)]$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**
- Another?
  - **if** $x \geq y$ **then** $x := y$ **else skip**
- Or even?
  - $y := x$

Specification can be tricky

"The program must set y to the maximum of x and y"

$[\textbf{true}]c[y = max(x, y)]$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**
- Another?
  - **if** $x \geq y$ **then** $x := y$ **else skip**
- Or even?
  - $y := x$

Later you will be able to prove that these programs are all "correct"...

The postcondition $y = max(x, y)$ says "y is the maximum of x and y *in the final state*"

Specification can be tricky

"The program must set y to the maximum of x and y"

The intended specification was probably *not* properly captured by

$$[\mathbf{true}]c[y = max(x, y)]$$

The correct formalization of what was intended is probably

$$[x = x_0 \wedge y = y_0]c[y = max(x_0, y_0)]$$

The lesson

- it is easy to write the wrong specification!
- a proof system will not help since the incorrect programs could have been proved "correct"
- testing would have helped!

A quick review of predicate logic

Predicate logic (a.k.a. first-order logic) forms the basis for program specification

- It is used to describe the acceptable initial states, and intended final states of programs

$$
\begin{array}{llll}
\text{(IntExp)} & e & ::= & n \\
& & \mid & x \qquad \text{(program variables, logical variables)} \\
& & \mid & e + e \mid e - e \mid \ldots \\
\\
\text{(Assn)} & p, q & ::= & \textbf{true} \mid \textbf{false} \\
& & \mid & e = e \mid e < e \mid e > e \mid \ldots \qquad \text{(predicates)} \\
& & \mid & \neg p \mid p \wedge p \mid p \vee p \mid p \Rightarrow p \\
& & \mid & \forall x.\, p \mid \exists x.\, p
\end{array}
$$

Derivation of assertions

$\vdash p$: there exists a proof or derivation of p following the inference rules.

(Proof of formulas within predicate logic assumed known.)

Semantics of assertions

$\sigma \models p$: p holds (is true) in σ, or σ satisfies p

$$\sigma \models \textbf{true} \qquad \text{always}$$

$$\sigma \models \textbf{false} \qquad \text{never}$$

$$\sigma \models e_1 = e_2 \quad \textit{iff} \quad [\![e_1]\!]_{intexp}\,\sigma = [\![e_2]\!]_{intexp}\,\sigma$$

$$\sigma \models \neg p \quad \textit{iff} \quad \neg(\sigma \models p)$$

$$\sigma \models p \wedge q \quad \textit{iff} \quad (\sigma \models p) \wedge (\sigma \models q)$$

$$\sigma \models p \vee q \quad \textit{iff} \quad (\sigma \models p) \vee (\sigma \models q)$$

$$\sigma \models p \Rightarrow q \quad \textit{iff} \quad (\sigma \models p) \Rightarrow (\sigma \models q)$$

Universal quantification and existential quantification

$$\sigma \models \forall x.\, p \quad \textit{iff} \quad \forall n.\, \sigma\{x \rightsquigarrow n\} \models p$$

$$\sigma \models \exists x.\, p \quad \textit{iff} \quad \exists n.\, \sigma\{x \rightsquigarrow n\} \models p$$

- $\forall x.\, p$ means

  "for all values of x, the assertion p is true"

- $\exists x.\, p$ means

  "for some value of x, the assertion p is true"

Validity of assertions

- p holds in σ (i.e. $\sigma \models p$)
- p is valid: for all σ, p holds in σ
- p is unsatisfiable: $\neg p$ is valid

Summary

Predicate logic forms the basis for program specification

It is used to describe the states of programs

We will next look at how to prove programs meet their specifications

Outline

Floyd-Hoare logic

To construct formal proofs of partial correctness specifications, axioms and rules of inference are needed

This is what Floyd-Hoare logic provides

- the formulation of the deductive system is due to Hoare
- some of the underlying ideas originated with Floyd

A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an axiom of the logic or follows from earlier lines by a rule of inference of the logic

- proofs can also be trees, if you prefer

A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion

Judgments

Three kinds of things that can be judgments

- predicate logic formulas, e.g. $x + 1 > x$
- partial correctness specification $\{p\}c\{q\}$
- total correctness specification $[p]c[q]$

$\vdash J$ means J can be proved

- how to prove predicate logic formulas assumed known
- Hoare logic provides axioms and rules for proving program correctness specifications

Recall our simple imperative language

$$
\begin{array}{rlll}
\text{(IntExp)} & e & ::= & n \\
& & \mid & x \\
& & \mid & e + e \mid e - e \mid \ldots \\
\\
\text{(BoolExp)} & b & ::= & \textbf{true} \mid \textbf{false} \\
& & \mid & e = e \mid e < e \mid e > e \\
& & \mid & \neg b \mid b \wedge b \mid b \vee b \mid \ldots \\
\\
\text{(Comm)} & c & ::= & \textbf{skip} \\
& & \mid & x := e \\
& & \mid & c \,\textbf{;}\, c \\
& & \mid & \textbf{if } b \textbf{ then } c \textbf{ else } c \\
& & \mid & \textbf{while } b \textbf{ do } c
\end{array}
$$

The assignment rule of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as})$$

The most central aspect of imperative languages is reduced to simple syntactic formula substitution!

The assignment rule of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as})$$

The most central aspect of imperative languages is reduced to simple syntactic formula substitution!

Examples:

$$\{y + 1 = 42\} \quad x := y + 1 \quad \{x = 42\}$$

$$\{42 = 42\} \quad x := 42 \quad \{x = 42\}$$

$$\{x - y > 3\} \quad x := x - y \quad \{x > 3\}$$

$$\{x + 1 > 0\} \quad x := x + 1 \quad \{x > 0\}$$

The assignment rule of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as})$$

Many people feel the assignment axiom is “backwards”.

One common erroneous intuition is that it should be

$$\frac{}{\{p\}\ x := e\ \{p[x/e]\}}$$

It leads to nonsensical examples like:

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

The assignment rule of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as})$$

Many people feel the assignment axiom is “backwards”.

Another erroneous intuition is that it should be

$$\frac{}{\{p\}\ x := e\ \{p[e/x]\}}$$

It leads to nonsensical examples like:

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

The assignment rule of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as})$$

Many people feel the assignment axiom is “backwards”.

A third kind of erroneous intuition is that it should be

$$\frac{}{\{p\}x := e\{p \wedge x = e\}}$$

It leads to nonsensical examples like:

$$\{x = 5\} \quad x := x + 1 \quad \{x = 5 \wedge x = x + 1\}$$

A forward assignment rule (due to Floyd)

$$\frac{}{\{p\}\ x := e\ \{\exists v.\ x = e[v/x] \wedge p[v/x]\}}\ (\textsc{as-fw})$$

Here v is a fresh variable (i.e. doesn't equal x or occur in p or e)

A forward assignment rule (due to Floyd)

$$\frac{}{\{p\}\ x := e\ \{\exists v.\ x = e[v/x] \wedge p[v/x]\}}\ (\textsc{as-fw})$$

Here v is a fresh variable (i.e. doesn't equal x or occur in p or e)

Example

$$\{x = 1\}\ x := x + 1\ \{\exists v.\ x = (x + 1)[v/x] \wedge (x = 1)[v/x]\}$$

A forward assignment rule (due to Floyd)

$$\frac{}{\{p\}\ x := e\ \{\exists v.\ x = e[v/x] \wedge p[v/x]\}}\ (\textsc{as-fw})$$

Here v is a fresh variable (i.e. doesn't equal x or occur in p or e)

Example

$$\{x = 1\}\ x := x + 1\ \{\exists v.\ x = (x + 1)[v/x] \wedge (x = 1)[v/x]\}$$

Simplifying the postcondition

$$\{x = 1\}\ x := x + 1\ \{\exists v.\ x = v + 1 \wedge v = 1\}$$

$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$

Forward rule equivalent to the standard one but harder to use

Strengthening precondition and weakening postcondition

Strengthening precedent (SP):

$$\frac{p \Rightarrow q \qquad \{q\}c\{r\}}{\{p\}c\{r\}} \ (\textsc{sp})$$

Weakening consequent (WC):

$$\frac{\{p\}c\{q\} \qquad q \Rightarrow r}{\{p\}c\{r\}} \ (\textsc{wc})$$

Note the two hypotheses are different kinds of judgments.

Strengthening precondition and weakening postcondition

Example:

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

Here n is a logical variable.

Proof:

1. $x = n \Rightarrow x + 1 = n + 1$ (Predicate Logic)
2. $\{x + 1 = n + 1\}\ x := x + 1\ \{x = n + 1\}$ AS
3. $\{x = n\}\ x := x + 1\ \{x = n + 1\}$ SP, 1, 2

Strengthening precondition and weakening postcondition

Example:

$$\{r = x\}\ z := 0\ \{r = x + (y * z)\}$$

1. $r = x \Rightarrow r = x \wedge 0 = 0$ (Predicate Logic)
2. $\{r = x \wedge 0 = 0\}\ z := 0\ \{r = x \wedge z = 0\}$ AS
3. $\{r = x\}\ z := 0\ \{r = x \wedge z = 0\}$ SP, 1, 2
4. $r = x \wedge z = 0 \Rightarrow r = x + (y * z)$ (Predicate Logic)
5. $\{r = x\}\ z := 0\ \{r = x + (y * z)\}$ WC, 3, 4

The consequence rule

The rules (SP) and (WC) are sometimes merged to the consequence rule.

$$\frac{p \Rightarrow p' \qquad \{p'\}c\{q'\} \qquad q' \Rightarrow q}{\{p\}c\{q\}} \ (\textsc{conseq})$$

Note that this rule is not syntax directed.

The sequential composition rule

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1 \,;\, c_2\{q\}}\ (\mathrm{sc})$$

Example

$$\{x = x_0 \wedge y = y_0\}\ r := x; x := y; y := r\ \{y = x_0 \wedge x = y_0\}$$

Proof:

1. $\{x = x_0 \wedge y = y_0\}\ r := x\ \{r = x_0 \wedge y = y_0\}$ AS
2. $\{r = x_0 \wedge y = y_0\}\ x := y\ \{r = x_0 \wedge x = y_0\}$ AS
3. $\{r = x_0 \wedge x = y_0\}\ y := r\ \{y = x_0 \wedge x = y_0\}$ AS
4. $\{x = x_0 \wedge y = y_0\}\ r := x; x := y;\ \{r = x_0 \wedge x = y_0\}$ SC, 1, 2
5. $\{x = x_0 \wedge y = y_0\}\ r := x; x := y; y := r\ \{y = x_0 \wedge x = y_0\}$ SC, 4, 3

Example

$$\{y > 3\}\ x := 2 * y; x := x - y\ \{x \geq 4\}$$

Proof:

1.	$\{x - y \geq 4\}x := x - y\{x \geq 4\}$	AS
2.	$\{2y - y \geq 4\}x := 2 * y\{x - y \geq 4\}$	AS
3.	$y > 3 \Rightarrow (2 * y) - y \geq 4$	(Predicate Logic)
4.	$\{y > 3\}x := 2 * y\{x - y \geq 4\}$	SP, 2, 3
5.	$\{y > 3\}\ x := 2 * y; x := x - y\ \{x \geq 4\}$	SC, 1, 4

The skip rule

$$\frac{}{\{p\}\textbf{skip}\{p\}}\ (\textsc{sk})$$

The conditional rule

$$\frac{\{p \wedge b\}c_1\{q\} \qquad \{p \wedge \neg b\}c_2\{q\}}{\{p\}\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\{q\}} \ (\textsc{cd})$$

The conditional rule

$$\frac{\{p \wedge b\}c_1\{q\} \qquad \{p \wedge \neg b\}c_2\{q\}}{\{p\}\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\{q\}} \ (\textsc{cd})$$

Example:

$$\{\textbf{true}\} \textbf{ if } x < y \textbf{ then } z := y \textbf{ else } z := x \ \{z = \max(x, y)\}$$

Proof (next slide)

The conditional rule – example

1. $\{y = \max(x, y)\}\ z := y\ \{z = \max(x, y)\}$ AS
2. $\{x = \max(x, y)\}\ z := x\ \{z = \max(x, y)\}$ AS
3. $\mathbf{true} \wedge x < y \Rightarrow y = \max(x, y)$
4. $\mathbf{true} \wedge \neg(x < y) \Rightarrow x = \max(x, y)$
5. $\{\mathbf{true} \wedge x < y\}\ z := y\ \{z = \max(x, y)\}$ SP, 1, 3
6. $\{\mathbf{true} \wedge \neg(x < y)\}\ z := x\ \{z = \max(x, y)\}$ SP, 2, 4
7. $\{\mathbf{true}\}\ \mathbf{if}\ x < y\ \mathbf{then}\ z := y\ \mathbf{else}\ z := x\ \{z = \max(x, y)\}$ CD, 5, 6

Review the “tricky-spec” example at the beginning

$\{\textbf{true}\}c\{y = max(x, y)\}$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**
- Another?
  - **if** $x \geq y$ **then** $x := y$ **else skip**
- Or even?
  - $y := x$

Now let’s prove that these are all correct.

Review the “tricky-spec” example at the beginning

$\{\textbf{true}\}c\{y = max(x, y)\}$

- A suitable program:
  - **if** $x \geq y$ **then** $y := x$ **else skip**
- Another?
  - **if** $x \geq y$ **then** $x := y$ **else skip**
- Or even?
  - $y := x$

Now let’s prove that these are all correct.

$\{x = x_0 \wedge y = y_0\}c\{y = max(x_0, y_0)\}$

Now let’s show that the first program above satisfies this spec.

More rules

$$\frac{\{p\}c\{q\} \qquad \{p'\}c\{q'\}}{\{p \wedge p'\}c\{q \wedge q'\}}\ (\textsc{ca}) \qquad\qquad \frac{\{p\}c\{q\} \qquad \{p'\}c\{q'\}}{\{p \vee p'\}c\{q \vee q'\}}\ (\textsc{da})$$

These rules are useful for splitting a proof into independent bits

- prove $\{p\}c\{q_1 \wedge q_2\}$ by separately proving both $\{p\}c\{q_1\}$ and $\{p\}c\{q_2\}$

Any proof with these rules could be done without using them

- i.e. they are theoretically redundant (proof omitted)
- however, useful in practice

All the rules till now also hold for total correctness.

The **while** rule

Partial correctness of **while**:

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg b\}}\ (\textsc{whp})$$

i is called *loop invariant*.

It says that

- if executing c *once* preserves the truth of i, then executing c *any number of times* also preserves the truth of i
- after a **while** command has terminated, the test must be false

Example

$$\{x \leq 10\}\ \textbf{while}\ x \neq 10\ \textbf{do}\ x := x + 1\ \{x = 10\}$$

Proof:

1. $\{x + 1 \leq 10\}\ x := x + 1\ \{x \leq 10\}$ AS
2. $x \leq 10 \wedge x \neq 10 \Rightarrow x + 1 \leq 10$
3. $\{x \leq 10 \wedge x \neq 10\}\ x := x + 1\ \{x \leq 10\}$ SP, 1, 2
4. $\{x \leq 10\}\ \textbf{while}\ x \neq 10\ \textbf{do}\ x := x + 1\ \{x \leq 10 \wedge \neg(x \neq 10)\}$ WHP, 3
5. $x \leq 10 \wedge \neg(x \neq 10) \Rightarrow x = 10$
6. $\{x \leq 10\}\ \textbf{while}\ x \neq 10\ \textbf{do}\ x := x + 1\ \{x = 10\}$ WC, 4, 5

Another example

Compute x/y. Store the quotient in z and the remainder in r.

$\{\textbf{true}\}$

$r := x;$

$z := 0;$

while $y \leq r$ **do**

  $r := r - y;$

  $z := z + 1;$

$\{r < y \wedge x = r + y * z\}$

Loop invariant:

$$x = r + y * z$$

How does one find loop invariant?

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg b\}}\ (\textsc{whp})$$

Look at the facts:

- invariant i must hold initially
- with the negated test $\neg b$ the invariant i must establish the result
- when the test b holds, the body must leave the invariant i unchanged

Think about how the loop works – the invariant should say that:

- what has been done so far together with what remains to be done
- holds at each iteration of the loop
- and gives the desired result when the loop terminates

Finding loop invariant – example of factorial

$\{x = 0 \wedge n > 0 \wedge f = 1\}$

while $x < n$ **do**

$\quad x := x + 1$ **;** $f := f * x$**;**

$\{x = n \wedge f = n!\}$

Finding loop invariant – example of factorial

$$\{x = 0 \wedge n > 0 \wedge f = 1\}$$

while $x < n$ **do**

$$\quad x := x + 1 \,;\, f := f * x;$$

$$\{x = n \wedge f = n!\}$$

An invariant is $f = x!$

But!! At end we need $f = n!$, but (WHP) rule only gives $\neg(x < n)$

So, invariant needed: $f = x! \wedge x \leq n$

- At end $x \leq n \wedge \neg(x < n) \Rightarrow x = n$

Often need to strengthen invariants to get them to work

- typical to add stuff to "carry along" like $x \leq n$

Finding loop invariant – another factorial

$$\{x = n \wedge n > 0 \wedge f = 1\}$$

while $x > 0$ **do**

$$f := f * x \,;\, x := x - 1;$$

$$\{x = 0 \wedge f = n!\}$$

Think how the loop works

- f stores the result so far
- $x!$ is what remains to be computed
- $n!$ is the desired result

Finding loop invariant – another factorial

$$\{x = n \wedge n > 0 \wedge f = 1\}$$

while $x > 0$ **do**

$$f := f * x \,;\, x := x - 1;$$

$$\{x = 0 \wedge f = n!\}$$

Think how the loop works

- f stores the result so far
- $x!$ is what remains to be computed
- $n!$ is the desired result

An invariant is $f * x! = n!$

- "result so far" $*$ "stuff to be done" = "desired result"
- decrease in x combines with increase in f to make invariant

Finding loop invariant – another factorial

$$\{x = n \wedge n > 0 \wedge f = 1\}$$

while $x > 0$ **do**

$$f := f * x \,;\, x := x - 1;$$

$$\{x = 0 \wedge f = n!\}$$

An invariant is $f * x! = n!$

But!! At end we need $x = 0$, but (WHP) rule only gives $\neg(x > 0)$

So, we have to strengthen invariant: $(f * x! = n!) \wedge x \geq 0$

- At end $x \geq 0 \wedge \neg(x > 0) \Rightarrow x = 0$

Proving partial correctness

$$\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{x = 10\}$$

Proof:

1. $\{\textbf{true} \wedge x \neq 10\}\ \textbf{skip}\ \{\textbf{true} \wedge x \neq 10\}$ — SK
2. $\textbf{true} \wedge x \neq 10 \Rightarrow \textbf{true}$
3. $\{\textbf{true} \wedge x \neq 10\}\ \textbf{skip}\ \{\textbf{true}\}$ — WC, 1, 2
4. $\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{\textbf{true} \wedge \neg(x \neq 10)\}$ — WHP, 3
5. $\textbf{true} \wedge \neg(x \neq 10) \Rightarrow x = 10$
6. $\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{x = 10\}$ — WC, 4, 5

Proving partial correctness

Another example:

1. $\{\mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{true}\}$ SK
2. $\mathbf{true} \wedge \mathbf{true} \Rightarrow \mathbf{true}$
3. $\{\mathbf{true} \wedge \mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{true}\}$ SP, 1, 2
4. $\{\mathbf{true}\}\ \mathbf{while\ true\ do\ skip}\ \{\mathbf{true} \wedge \neg\mathbf{true}\}$ WHP, 3

The **while** rule for total correctness

The **while** commands are the only commands in our simple language that can cause non-termination

- they are thus the only kind of command with a non-trivial termination rule

The idea behind the **while** rule for total correctness is

- to prove **while** *b* **do** *c* terminates
- show that some non-negative metric (e.g. a loop counter) decreases on each iteration of *c*
- this decreasing metric is called a *variant*

The **while** rule for total correctness

$$\frac{[i \wedge b \wedge (e = x_0)]\ c\ [i \wedge (e < x_0)] \qquad i \wedge b \Rightarrow e \geq 0}{[i]\ \textbf{while}\ b\ \textbf{do}\ c\ [i \wedge \neg b]}\ (\textsc{wht})$$

Here $x_0 \notin fv(c) \cup fv(e) \cup fv(i) \cup fv(b)$.

x_0 is a *logical variable*. e is the *variant* of the loop.

Basic idea:

- The first premise: the metric is decreased by execution of c.
- The second premise: when the metric becomes negative, b is false, and the loop terminates (invariant i is always satisfied).

The **while** rules

Partial correctness of **while**:

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg b\}}\ (\textsc{whp})$$

Total correctness of **while**:

$$\frac{[i \wedge b \wedge (e = x_0)]\ c\ [i \wedge (e < x_0)] \qquad i \wedge b \Rightarrow e \geq 0}{[i]\ \textbf{while}\ b\ \textbf{do}\ c\ [i \wedge \neg b]}\ (\textsc{wht})$$

where $x_0 \notin fv(c) \cup fv(e) \cup fv(i) \cup fv(b)$.

This is the major rule that distinguishes the logic for total correctness from partial correctness.

Proving total correctness

$$[x \leq 10] \textbf{ while } x \neq 10 \textbf{ do } x := x + 1 \; [x = 10]$$

1. $\{x + 1 \leq 10 \wedge 10 - (x + 1) < z\} \; x := x + 1 \; \{x \leq 10 \wedge 10 - x < z\}$ AS
2. $x \leq 10 \wedge x \neq 10 \wedge 10 - x = z \Rightarrow x + 1 \leq 10 \wedge 10 - (x + 1) < z$
3. $\{x \leq 10 \wedge x \neq 10 \wedge 10 - x = z\} \; x := x + 1 \; \{x \leq 10 \wedge 10 - x < z\}$ SP
4. $x \leq 10 \wedge x \neq 10 \Rightarrow 10 - x \geq 0$
5. $[x \leq 10] \textbf{ while } x \neq 10 \textbf{ do } x := x + 1 \; [x \leq 10 \wedge \neg(x \neq 10)]$ WHT
6. $x \leq 10 \wedge \neg(x \neq 10) \Rightarrow x = 10$
7. $[x \leq 10] \textbf{ while } x \neq 10 \textbf{ do } x := x + 1 \; [x = 10]$ WC

Termination specifications

Informally, Total correctness = Termination + Partial correctness

This informal equation can be represented by the following rules:

$$\frac{\{p\}\ c\ \{q\} \qquad [p]\ c\ [\mathbf{true}]}{[p]\ c\ [q]}$$

$$\frac{[p]\ c\ [q]}{\{p\}\ c\ \{q\}} \qquad\qquad \frac{[p]\ c\ [q]}{[p]\ c\ [\mathbf{true}]}$$

Besides,

$$\frac{\{p\}\ c\ \{q\} \qquad \text{if } c \text{ contains no } \mathbf{while} \text{ commands}}{[p]\ c\ [q]}$$

All rules

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as}) \qquad \frac{}{\{p\}\textbf{skip}\{p\}}\ (\textsc{sk})$$

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1\ ;c_2\{q\}}\ (\textsc{sc}) \qquad \frac{\{p \wedge b\}c_1\{q\} \qquad \{p \wedge \neg b\}c_2\{q\}}{\{p\}\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\{q\}}\ (\textsc{cd})$$

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg b\}}\ (\textsc{whp})$$

$$\frac{[i \wedge b \wedge (e = x_0)]\ c\ [i \wedge (e < x_0)] \qquad i \wedge b \Rightarrow e \geq 0}{[i]\ \textbf{while}\ b\ \textbf{do}\ c\ [i \wedge \neg b]}\ (\textsc{wht})$$

$$\frac{p \Rightarrow q \qquad \{q\}c\{r\}}{\{p\}c\{r\}}\ (\textsc{sp}) \qquad \frac{\{p\}c\{q\} \qquad q \Rightarrow r}{\{p\}c\{r\}}\ (\textsc{wc})$$

$$\frac{\{p\}c\{q\} \qquad \{p'\}c\{q'\}}{\{p \wedge p'\}c\{q \wedge q'\}}\ (\textsc{ca}) \qquad \frac{\{p\}c\{q\} \qquad \{p'\}c\{q'\}}{\{p \vee p'\}c\{q \vee q'\}}\ (\textsc{da})$$

Total correctness of factorial

$$[x = 0 \wedge n > 0 \wedge f = 1]$$

while $x < n$ **do**

$$x := x + 1 \,;\, f := f * x;$$

$$[x = n \wedge f = n!]$$

Loop invariant:

$$i \stackrel{\text{def}}{=} (f = x! \wedge x \leq n)$$

Loop variant:

$$e \stackrel{\text{def}}{=} (n - x)$$

Another factorial

The program:

$$c \stackrel{\text{def}}{=} f := 1 \,;\, \textbf{while } x > 0 \textbf{ do } (f := f * x \,;\, x := x - 1)$$

The specification:

$$[x = n]\ c\ [x < 0 \vee f = n!]$$

Proof: we first prove the following sub-goals:

$$(G1)[x < 0]\ c\ [x < 0]$$
$$(G2)[x = n \wedge x \geq 0]\ c\ [f = n!]$$

Then we apply the (DA) and (SP) rules.

Another factorial (G1)

$[x < 0]$

$f := 1;$

$[x < 0]$

while $x > 0$ **do**

$\quad f := f * x ; x := x - 1;$

$[x < 0]$

Another factorial (G2)

$[x = n \wedge x \geq 0]$

$f := 1;$

while $x > 0$ **do**

$\quad f := f * x ; x := x - 1;$

$[f = n!]$

Another factorial (G2)

$$[x = n \wedge x \geq 0 \wedge f = 1]$$

while $x > 0$ **do**

$$f := f * x \,;\, x := x - 1;$$

$$[f = n!]$$

Loop invariant: $(f * x! = n!) \wedge x \geq 0$

Another factorial (G2)

$$[x = n \wedge x \geq 0 \wedge f = 1]$$

while $x > 0$ **do**

$$f := f * x \,;\, x := x - 1;$$

$$[f = n!]$$

Loop invariant: $(f * x! = n!) \wedge x \geq 0$

Loop variant: x

Example of division

Can we prove?

$$[\textbf{true}]$$

$$r := x;$$

$$z := 0;$$

$$\textbf{while } y \leq r \textbf{ do}$$

$$\quad r := r - y;$$

$$\quad z := z + 1;$$

$$[r < y \wedge x = r + y * z]$$

Loop invariant: $x = r + y * z$

Loop variant: r

Example of division

$[y > 0]$

$r := x;$

$z := 0;$

while $y \leq r$ **do**

$\quad r := r - y;$

$\quad z := z + 1;$

$[r < y \wedge x = r + y * z]$

Loop invariant: $(x = r + y * z) \wedge y > 0$

Loop variant: r

Summary

We have given:

- a notation for specifying what a program does
- a way of proving that it meets its specification

Now we look at some ways of organizing proofs that make it easier for humans to do the verification:

- derived rules
- annotating programs prior to proofs

Then we see how to automate program verification

- the automation mechanizes some of these ideas

Combining multiple proof steps

Proofs involve lots of tedious fiddly small steps

It is tempting to take shortcuts and apply several rules at once

- this increases the chance of making mistakes

Example:

- by assignment axiom & precondition strengthening

$$\{\mathbf{true}\}r := x\{r = x\}$$

Rather than:

- by the assignment axiom

$$\{x = x\}r := x\{r = x\}$$

- by precondition strengthening with $\mathbf{true} \Rightarrow x = x$

$$\{\mathbf{true}\}r := x\{r = x\}$$

Derived rule for assignment

$$\frac{p \Rightarrow q[e/x]}{\{p\}\ x := e\ \{q\}}$$

Derivation:

1.	$\{q[e/x]\}\ x := e\ \{q\}$	AS
2.	$p \Rightarrow q[e/x]$	assumption
3.	$\{p\}\ x := e\ \{q\}$	SP, 1, 2

Derived rule for assignment

$$\frac{p \Rightarrow q[e/x]}{\{p\}\ x := e\ \{q\}}$$

Example:

1.	$r = x \Rightarrow r = x \wedge 0 = 0$	predicate logic
2.	$\{r = x\}\ z := 0\ \{r = x \wedge z = 0\}$	derived assignment

One less step than the original proof:

1.	$\{r = x \wedge 0 = 0\}\ z := 0\ \{r = x \wedge z = 0\}$	AS
2.	$r = x \Rightarrow r = x \wedge 0 = 0$	predicate logic
3.	$\{r = x\}\ z := 0\ \{r = x \wedge z = 0\}$	SP, 1, 2

Derived rule for sequenced assignment

$$\frac{\{p\}\ c\ \{q[e/x]\}}{\{p\}\ c\,;x := e\ \{q\}}$$

Intuitively work backwards:

- push q "through" $x := e$, changing it to $q[e/x]$

Example:

1. $\{x = x_0 \wedge y = y_0\}\ r := x\ \{r = x_0 \wedge y = y_0\}$ AS
2. $\{x = x_0 \wedge y = y_0\}\ r := x\,;x := y\ \{r = x_0 \wedge x = y_0\}$ sequenced as

Derived **while** rule for partial correctness

$$\frac{p \Rightarrow i \qquad \{i \wedge b\}\ c\ \{i\} \qquad i \wedge \neg b \Rightarrow q}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}}$$

Example:

1. $x \leq 10 \wedge x \neq 10 \Rightarrow (x + 1) \leq 10$
2. $x \leq 10 \Rightarrow x \leq 10$
3. $\{x \leq 10 \wedge x \neq 10\}\ x := x + 1\ \{x \leq 10\}$ derived as, 1
4. $x \leq 10 \wedge \neg(x \neq 10) \Rightarrow x = 10$
5. $\{x \leq 10\}$ **while** $x \neq 10$ **do** $x := x + 1\ \{x = 10\}$ derived wh, 2, 3, 4

Derived **while** rule for total correctness

$$\frac{\begin{array}{l} p \Rightarrow i \\ i \wedge b \Rightarrow e \geq 0 \\ i \wedge \neg b \Rightarrow q \\ [i \wedge b \wedge (e = x_0)]\ c\ [i \wedge (e < x_0)] \end{array}}{[p]\ \textbf{while}\ b\ \textbf{do}\ c\ [q]}$$

where $x_0 \notin fv(p) \cup fv(q) \cup fv(c) \cup fv(e) \cup fv(i) \cup fv(b)$

Derived rule for multiple sequential composition

$$\frac{\begin{array}{c} p_0 \Rightarrow q_0 \\ \{q_0\}c_0\{p_1\} \quad p_1 \Rightarrow q_1 \\ \ldots \\ \{q_{n-1}\}c_{n-1}\{p_n\} \quad p_n \Rightarrow q_n \end{array}}{\{p_0\}c_0 \,;\, \ldots \,;\, c_{n-1}\{q_n\}} \; (\textsc{msq}_n)$$

Derivation of $\textsc{msq}_1$:

1.	$p_0 \Rightarrow q_0$	assumption
2.	$\{q_0\}c_0\{p_1\}$	assumption
3.	$\{p_0\}c_0\{p_1\}$	sp, 1, 2
4.	$p_1 \Rightarrow q_1$	assumption
5.	$\{p_0\}c_0\{q_1\}$	wc, 3, 4

$\textsc{msq}_n$ derived from $\textsc{msq}_{n-1}$ and sc.

Derived rule for multiple sequential composition

$$\frac{\begin{array}{c} p_0 \Rightarrow q_0 \\ \{q_0\}c_0\{p_1\} \quad p_1 \Rightarrow q_1 \\ \ldots \\ \{q_{n-1}\}c_{n-1}\{p_n\} \quad p_n \Rightarrow q_n \end{array}}{\{p_0\}c_0 \,;\, \ldots \,;\, c_{n-1}\{q_n\}} \; (\mathsf{MSQ}_n)$$

Example:

1. $\{x = x_0 \wedge y = y_0\}\ r := x\ \{r = x_0 \wedge y = y_0\}$ **AS**
2. $\{r = x_0 \wedge y = y_0\}\ x := y\ \{r = x_0 \wedge x = y_0\}$ **AS**
3. $\{r = x_0 \wedge x = y_0\}\ y := r\ \{y = x_0 \wedge x = y_0\}$ **AS**
4. $\{x = x_0 \wedge y = y_0\}\ r := x \,;\, x := y \,;\, y := r\ \{y = x_0 \wedge x = y_0\}$ MSQ_3

Annotations

The sequential composition rule introduces a new assertion r

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1 \,;\, c_2\{q\}} \text{ (sc)}$$

To apply this rule, one needs to find a suitable assertion r

If c_2 is $x := e$, then the sequenced assignment gives $q[e/x]$ for r

If c_2 isn't an assignment, then need some other way to choose r

Similarly, to use the (WHP) rule, must invent an invariant

Annotate first

It is helpful to think up these assertions before you start the proof and then annotate the program with them

- the information is then available when you need it in the proof
- this can help avoid you being bogged down in details
- the annotation should be true whenever control reaches that point

Annotate first

For example, the following program could be annotated at the points p_1 and p_2 indicated by the arrows:

$\{\textbf{true}\}$

$r := x;$

$z := 0;$

$\{r = x \wedge z = 0\} \qquad \leftarrow p_1$

while $y \leq r$ **do** $\quad \{x = r + y * z\} \quad \leftarrow p_2$

$\quad r := r - y;$

$\quad z := z + 1;$

$\{r < y \wedge x = r + y * z\}$

Summary

We have looked at two ways of organizing proofs that make it easier for humans to apply them:

- deriving “bigger step” rules
- annotating programs

Next we see how these techniques can be used to mechanize program verification

Outline

Automated program verification

We will describe the architecture of a simple program verifier

Justified with respect to the rules of Hoare logic

It is clear that

- proofs are long and boring, even if the program being verified is quite simple
- lots of fiddly little details to get right, many of which are trivial, e.g.

$$(r = x \wedge z = 0) \Rightarrow (x = r + y * z)$$

Automation

Goal: automate the routine bits of proofs in Hoare logic

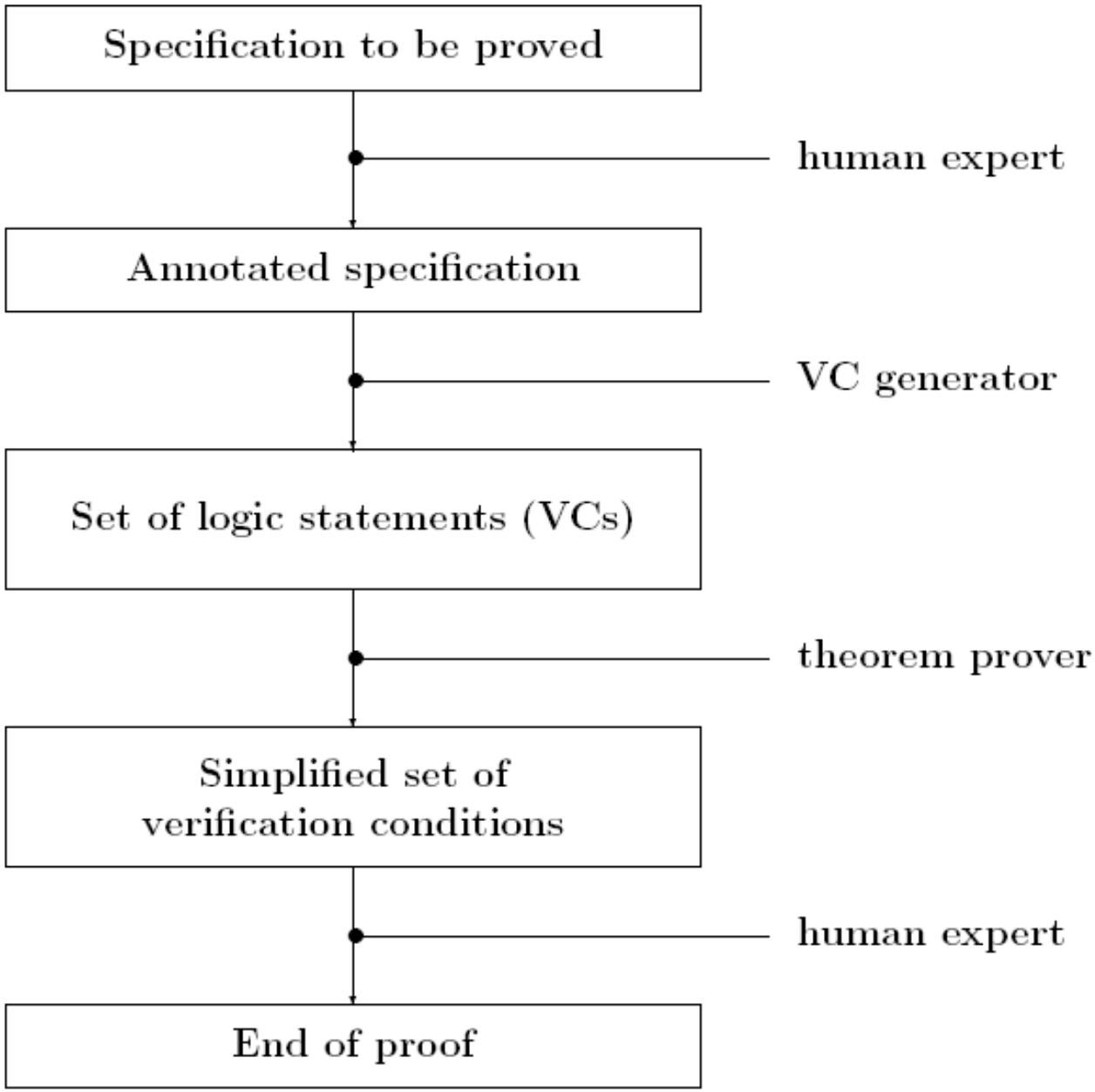
Unfortunately, logicians have shown that it is impossible in principle to design a *decision procedure* to decide automatically the truth or falsehood of an arbitrary mathematical formula

This does not mean that one cannot have procedures that will prove many *useful* theorems

- the non-existence of a general decision procedure merely shows that one cannot hope to prove *everything* automatically
- in practice, it is quite possible to build a system that will mechanize the boring and routine aspects of verification

The standard approach to this will be described now

Architecture of a verifier



Architecture of a verifier

Input: Hoare triple, or *annotated specification*

- users may need to insert some intermediate assertions

The system generates a set of purely mathematical formulas called *verification conditions* (VCs)

If the verification conditions are provable, then the original specification can be deduced from the axioms and rules of Hoare logic

The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically

- if it fails, advice is sought from the user

Verification conditions (VCs)

The three steps in proving $\{p\}c\{q\}$ with a verifier

1. The program c is annotated by inserting assertions that are meant to hold at intermediate points
   - tricky: needs intelligence and good understanding of how the program works
   - automating it is an artificial intelligence problem
2. A set of logic formulas called verification conditions (VCs) is then generated from the annotated specification
   - this is purely mechanical and easily done by a program
3. The verification conditions are proved
   - needs automated theorem proving (i.e. artificial intelligence)

Verification conditions (VCs)

The three steps in proving $\{p\}c\{q\}$ with a verifier

1. The program c is annotated by inserting assertions that are meant to hold at intermediate points
   - tricky: needs intelligence and good understanding of how the program works
   - automating it is an artificial intelligence problem
2. A set of logic formulas called verification conditions (VCs) is then generated from the annotated specification
   - this is purely mechanical and easily done by a program
3. The verification conditions are proved
   - needs automated theorem proving (i.e. artificial intelligence)

To improve automated verification one can try to

- reduce the number and complexity of the annotations required
- increase the power of the theorem prover
- still a research area

Validity of VCs

Step 2 generates VCs. It will be shown that

- if one can prove all the verification conditions generated from $\{p\}c\{q\}$, then $\vdash \{p\}c\{q\}$

Step 2 converts a verification problem into a conventional mathematical problem

The process will be illustrated with:

$\{\textbf{true}\}$
$r := x;$
$z := 0;$
while $y \leq r$ **do**
  $r := r - y\,; z := z + 1;$
$\{r < y \wedge x = r + y * z\}$

Example

Step 1 is to insert annotations

$$
\begin{array}{l}
\{\textbf{true}\} \\
r := x\textbf{;} \\
z := 0\textbf{;} \\
\{r = x \wedge z = 0\} \qquad \leftarrow p_1 \\
\textbf{while } y \leq r \textbf{ do} \quad \{x = r + y * z\} \quad \leftarrow p_2 \\
\quad r := r - y\textbf{;} \\
\quad z := z + 1\textbf{;} \\
\{r < y \wedge x = r + y * z\}
\end{array}
$$

The annotations p_1 and p_2 are assertions which are intended to hold whenever control reaches them

Example

$\{\textbf{true}\}$
$r := x;$
$z := 0;$
$\{r = x \wedge z = 0\} \qquad \leftarrow p_1$
while $y \leq r$ **do** $\quad \{x = r + y * z\} \quad \leftarrow p_2$
$\quad r := r - y\,; z := z + 1;$
$\{r < y \wedge x = r + y * z\}$

Control only reaches p_1 once

Control reaches p_2 each time the loop body is executed

- whenever this happens, p_2 holds, even though the values of r and z vary
- p_2 is an invariant of the **while** command

Generating and proving VCs

Step 2 will generate the following four verification conditions

(1) $\textbf{true} \Rightarrow x = x \wedge 0 = 0$

(2) $r = x \wedge z = 0 \Rightarrow (x = r + (y * z))$

(3) $(x = r + (y * z)) \wedge y \leq r \Rightarrow (x = (r - y) + (y * (z + 1))$

(4) $(x = r + (y * z)) \wedge \neg(y \leq r) \Rightarrow r < y \wedge (x = r + (y * z))$

Notice that these are statements of arithmetic

- the constructs of our programming language have been "compiled away"

Step 3 consists in proving the four verification conditions

- easy with modern automatic theorem provers

Annotation of commands

An annotated command is a command with assertions embedded within it

A command is properly annotated if assertions have been inserted at the following places

(1) before each command c_i in the command sequence $c_1 \,;\, c_2 \,;\, \ldots \,;\, c_n$ if c_i is not an assignment command

(2) after the word **do** in **while** commands

The inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs

Can reduce number of annotations using weakest preconditions (discuss later)

Annotation of specifications

$\{p\}c\{q\}$ is properly annotated if c is a properly annotated command

Example: To be properly annotated, assertions should be inserted at points p_1 and p_2 of the specification below

$$
\begin{array}{l}
\{x = n \wedge x \geq 0\} \\
f := 1; \\
\{p_1\} \\
\textbf{while } x > 0 \textbf{ do} \quad \{p_2\} \\
\quad f := f * x; x := x - 1; \\
\{x = 0 \wedge f = n!\}
\end{array}
$$

Suitable assertions would be

$$
\begin{array}{ll}
p_1 : & \{f = 1 \wedge x = n \wedge x \geq 0\} \\
p_2 : & \{(f * x! = n!) \wedge x \geq 0\}
\end{array}
$$

Verification condition generation

The algorithm for generating VCs from an annotated specification $\{p\}c\{q\}$ is *recursive* on the structure of c

We will describe it command by command using rules of the form:

- The VCs for $C(c_1, c_2)$ are
  - $vc_1, \ldots, vc_n$
  - together with the VCs for c_1 and those for c_2
- $VC(C(c_1, c_2)) = \{vc_1, \ldots, vc_n\} \cup VC(c_1) \cup VC(c_2)$

Rules are chosen so that only one VC rule applies in each case

- applying them is then purely mechanical
- the choice is based on the syntax
- VC generation is deterministic

Justification of VCs

This process will be justified by showing that $\vdash \{p\}c\{q\}$ if all the verification conditions can be proved

We will prove that for any c,

- assuming the VCs of $\{p\}c\{q\}$ are provable
- then $\vdash \{p\}c\{q\}$ can be derived with the logic

Justification of VCs

This process will be justified by showing that $\vdash \{p\}c\{q\}$ if all the verification conditions can be proved

We will prove that for any c,

- assuming the VCs of $\{p\}c\{q\}$ are provable
- then $\vdash \{p\}c\{q\}$ can be derived with the logic

Proof by *induction* on the structure of c

- base case: show the result holds for atomic commands, i.e. **skip** and assignments
- inductive step: show that when c is not an atomic command, then if the result holds for the constituent commands of c (induction hypothesis), then it holds also for c

Thus the result holds for all commands

VCs for **skip**

The single verification condition generated by

$$\{p\}\ \textbf{skip}\ \{q\}$$

is

$$p \Rightarrow q$$

Example: The VC for

$$\{x = 0\}\ \textbf{skip}\ \{x = 0\}$$

is

$$x = 0 \Rightarrow x = 0$$

(which is clearly true)

Justification of VCs for **skip**

We must show that
if the VCs of $\{p\}$ **skip** $\{q\}$ are provable, then $\vdash \{p\}$ **skip** $\{q\}$

Proof:

- Assume $p \Rightarrow q$ as it is the VC
- By (SK) rule and (WC) rule, $\vdash \{p\}$ **skip** $\{q\}$

VCs for assignments

The single verification condition generated by

$$\{p\}\ x := e\ \{q\}$$

is

$$p \Rightarrow q[e/x]$$

Example: The VC for

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

is

$$x = 0 \Rightarrow (x + 1) = 1$$

(which is clearly true)

Justification of VCs for assignments

We must show that
if the VCs of $\{p\}\ x := e\ \{q\}$ are provable, then $\vdash \{p\}\ x := e\ \{q\}$

Proof:

- Assume $p \Rightarrow q[e/x]$ as it is the VC
- By derived assignment rule, $\vdash \{p\}\ x := e\ \{q\}$

VCs for conditionals

The verification conditions generated from

$$\{p\}\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\{q\}$$

are the union of

(1) the verification conditions generated by

$$\{p \wedge b\}c_1\{q\}$$

(2) the verification conditions generated by

$$\{p \wedge \neg b\}c_2\{q\}$$

VCs for conditionals

Example: The VCs for

$$\{\textbf{true}\}\ \textbf{if}\ x < y\ \textbf{then}\ z := y\ \textbf{else}\ z := x\ \{z = \max(x, y)\}$$

are the union of

(1) the VCs generated by

$$\{\textbf{true} \wedge x < y\}\ z := y\ \{z = \max(x, y)\}$$

(2) the VCs generated by

$$\{\textbf{true} \wedge \neg(x < y)\}\ z := x\ \{z = \max(x, y)\}$$

Justification of VCs for conditionals

We must show that
if the VCs of $\{p\}$**if** b **then** c_1 **else** $c_2\{q\}$ are provable, then
$\vdash \{p\}$**if** b **then** c_1 **else** $c_2\{q\}$

Proof:

- Assume the VCs of $\{p \wedge b\}c_1\{q\}$ and $\{p \wedge \neg b\}c_2\{q\}$
- The inductive hypotheses tell us that if these VCs are provable then the corresponding Hoare logic specs are provable
- i.e. $\vdash \{p \wedge b\}c_1\{q\}$ and $\vdash \{p \wedge \neg b\}c_2\{q\}$
- By (CD) rule, $\vdash \{p\}$**if** b **then** c_1 **else** $c_2\{q\}$

Review of properly annotated sequences

If $c_1 \,;\, \ldots \,;\, c_n$ is properly annotated, then it must be one of the forms

- $c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, \{r\}\, c_n$
- $c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, x := e$

where $c_1 \,;\, \ldots \,;\, c_{n-1}$ is properly annotated

VCs for sequences

- The verification conditions generated by

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, \{r\}\, c_n\{q\}$$

(where c_n is not an assignment) are the union of

(1) the verification conditions generated by

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{r\}$$

(2) the verification conditions generated by

$$\{r\}c_n\{q\}$$

- The verification conditions generated by

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, x := e\{q\}$$

are the verification conditions generated by

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{q[e/x]\}$$

Example

The VCs for

$$\{x = x_0 \wedge y = y_0\}\ r := x\,;x := y\,;y := r\ \{y = x_0 \wedge x = y_0\}$$

are those generated by

$$\{x = x_0 \wedge y = y_0\}\ r := x\,;x := y\ \{(y = x_0 \wedge x = y_0)[r/y]\}$$

which simplifies to

$$\{x = x_0 \wedge y = y_0\}\ r := x\,;x := y\ \{r = x_0 \wedge x = y_0\}$$

The VCs for this are those generated by

$$\{x = x_0 \wedge y = y_0\}\ r := x\ \{(r = x_0 \wedge x = y_0)[y/x]\}$$

which simplifies to

$$\{x = x_0 \wedge y = y_0\}\ r := x\ \{r = x_0 \wedge y = y_0\}$$

Example continued

The only VC for

$$\{x = x_0 \wedge y = y_0\}\ r := x\ \{r = x_0 \wedge y = y_0\}$$

is

$$(x = x_0 \wedge y = y_0) \Rightarrow (r = x_0 \wedge y = y_0)[x/r]$$

which simplifies to

$$(x = x_0 \wedge y = y_0) \Rightarrow (x = x_0 \wedge y = y_0)$$

Thus the single VC for

$$\{x = x_0 \wedge y = y_0\}\ r := x\ ;\ x := y\ ;\ y := r\ \{y = x_0 \wedge x = y_0\}$$

is

$$(x = x_0 \wedge y = y_0) \Rightarrow (x = x_0 \wedge y = y_0)$$

Justification of VCs for sequences (1)

If the VCs for

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, \{r\}\, c_n\{q\}$$

are provable, then the VCs for

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{r\} \quad \text{and} \quad \{r\}c_n\{q\}$$

must both be provable

Hence by induction hypothesis,

$$\vdash \{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{r\} \quad \text{and} \quad \vdash \{r\}c_n\{q\}$$

Hence by the (sc) rule

$$\vdash \{p\}c_1 \,;\, \ldots \,;\, c_n\{q\}$$

Justification of VCs for sequences (2)

If the VCs for

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, x := e\{q\}$$

are provable, then the VCs for

$$\{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{q[e/x]\}$$

are also provable

Hence by induction hypothesis,

$$\vdash \{p\}c_1 \,;\, \ldots \,;\, c_{n-1}\{q[e/x]\}$$

Hence by the derived sequenced assignment rule

$$\vdash \{p\}c_1 \,;\, \ldots \,;\, c_{n-1} \,;\, x := e\{q\}$$

VCs for **while** commands (partial correctness)

A properly annotated specification for **while** has the form

$$\{p\}\ \textbf{while}\ b\ \textbf{do}\ \{i\}\ c\ \{q\}$$

The annotation i is the loop invariant

The verification conditions generated by

$$\{p\}\ \textbf{while}\ b\ \textbf{do}\ \{i\}\ c\ \{q\}$$

are the union of

(1) $p \Rightarrow i$

(2) $i \wedge \neg b \Rightarrow q$

(3) the verification conditions generated by $\{i \wedge b\}c\{i\}$

Example

The VCs for

$$\begin{array}{l} \{r = x \wedge z = 0\} \\ \textbf{while } y \leq r \textbf{ do} \quad \{x = r + y * z\} \\ \quad r := r - y \,;\, z := z + 1; \\ \{r < y \wedge x = r + y * z\} \end{array}$$

are the union of

(1) $(r = x \wedge z = 0) \Rightarrow (x = r + y * z)$

(2) $(x = r + y * z) \wedge \neg(y \leq r) \Rightarrow (r < y \wedge x = r + y * z)$

(3) the VCs for

$$\{x = r + y * z \wedge y \leq r\}\ r := r - y \,;\, z := z + 1\ \{x = r + y * z\}$$

which consists of the single condition

$$x = r + y * z \wedge y \leq r \quad \Rightarrow \quad x = (r - y) + y * (z + 1)$$

Justification of VCs for **while** (partial correctness)

If the VCs for

$$\{p\}\ \textbf{while}\ b\ \textbf{do}\ \{i\}\ c\ \{q\}$$

are provable, then

(1) $p \Rightarrow i$

(2) $i \wedge \neg b \Rightarrow q$

(3) the VCs generated by $\{i \wedge b\}c\{i\}$

are all provable

Hence by induction hypothesis,

$$\vdash \{i \wedge b\}c\{i\}$$

Hence by the derived **while** rule

$$\vdash \{p\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}$$

VCs for termination

Verification conditions are easily extended to total correctness

To generate total correctness verification conditions for **while** commands, it is necessary to add a variant as an annotation in addition to an invariant

No other extra annotations are needed for total correctness

VCs for **while**-free code same as for partial correctness

Annotations for **while** (total correctness)

A properly annotated total correctness specification of a **while** command has the form

$$[p]\ \textbf{while}\ b\ \textbf{do}\ \{i\}[e]\ c\ [q]$$

where i is the invariant and e is the variant

Note that the variant is intended to be a non-negative expression that decreases each round of the loop

The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them (as before)

VCs for **while** (total correctness)

The verification conditions generated by

$$[p]\ \textbf{while}\ b\ \textbf{do}\ \{i\}[e]\ c\ [q]$$

are the union of

(1) $p \Rightarrow i$

(2) $i \wedge \neg b \Rightarrow q$

(3) $i \wedge b \Rightarrow e \geq 0$

(4) the verification conditions generated by

$$\{i \wedge b \wedge e = x_0\}c\{i \wedge e < x_0\}$$

where $x_0 \notin fv(p) \cup fv(q) \cup fv(c) \cup fv(e) \cup fv(i) \cup fv(b)$

Example

The VCs for

$$
\begin{array}{l}
[r = x \wedge y > 0 \wedge z = 0] \\
\textbf{while } y \leq r \textbf{ do} \quad \{x = r + y * z \wedge y > 0\}[\,r\,] \\
\quad r := r - y\,;z := z + 1; \\
[r < y \wedge x = r + y * z]
\end{array}
$$

are the union of

(1) $(r = x \wedge y > 0 \wedge z = 0) \Rightarrow (x = r + y * z \wedge y > 0)$

(2) $x = r + y * z \wedge y > 0 \wedge \neg(y \leq r) \Rightarrow (r < y \wedge x = r + y * z)$

(3) $x = r + y * z \wedge y > 0 \wedge (y \leq r) \Rightarrow (r \geq 0)$

(4) the VCs for

$$
\begin{array}{l}
[x = r + y * z \wedge y > 0 \wedge y \leq r \wedge r = r_0] \\
r := r - y\,;z := z + 1 \\
[x = r + y * z \wedge y > 0 \wedge r < r_0]
\end{array}
$$

Example continued

The single VC for

$$[x = r + y * z \wedge y > 0 \wedge y \leq r \wedge r = r_0]$$
$$r := r - y \,;\, z := z + 1$$
$$[x = r + y * z \wedge y > 0 \wedge r < r_0]$$

is

$$x = r + y * z \wedge y > 0 \wedge y \leq r \wedge r = r_0$$
$$\Rightarrow \quad x = (r - y) + y * (z + 1) \;\wedge\; y > 0 \;\wedge\; (r - y) < r_0$$

Note: To prove $(r - y) < r_0$ we need to know $y > 0$

Summary

Have outlined the design of an automated program verifier

Annotated specifications “compiled to” mathematical statements

- if the statements (VCs) can be proved, the program is verified

Human help is required to give the annotations and prove the VCs

The algorithm was justified by an inductive proof

All the techniques introduced earlier are used

- derived rules
- annotation

Introduction to Dafny

Dafny is

- a programming language with built-in specification constructs;
- a program verifier for functional correctness of programs.

`http://research.microsoft.com/en-us/projects/dafny/`

Try it for fun!
`http://rise4fun.com/Dafny`

Examples in Dafny: pre- and post-conditions

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
    requires 0 < y
    ensures less < x < more
{
    more := x + y;
    less := x - y;
}
```

Result:

```
Dafny program verifier finished with 2 verified, 0
errors
```

Examples in Dafny: loop invariant

```
method test (x: int) returns (r: int)
        requires x <= 10
        ensures r == 10
{
        r := x;
        while r < 10
                invariant r <= 10
        {
                r := r + 1;
        }
}
```

Result:

```
Dafny program verifier finished with 2 verified, 0
errors
```

Examples in Dafny: loop invariant

```
method test (x: int) returns (r: int)
        requires x <= 10
        ensures r == 10
{
        r := x;
        while r < 10
                invariant r < 10
        {
                r := r + 1;
        }
}
```

Result:

```
This loop invariant might not hold on entry.
This loop invariant might not be maintained by the
loop.
```

Dafny proves total correctness

From the tutorial:

Dafny proves that code terminates, i.e. does not loop forever, by using decreases annotations. For many things, Dafny is able to guess the right annotations, but sometimes it needs to be made explicit. In fact, for all of the code we have seen so far, Dafny has been able to do this proof on its own, which is why we haven't seen the decreases annotation explicitly yet.

Examples in Dafny: decreases annotation

```
method test (x: int) returns (r: int)
        requires x <= 10
        ensures r == 10
{
        r := x;
        while r < 10
                invariant r <= 10
                decreases 10 – r
        {
                r := r + 1;
        }
}
```

Result:

```
Dafny program verifier finished with 2 verified, 0
errors
```

Outline

Soundness and completeness

The set of inference rules gives us a logic system. This kind of logic is called a program logic, which is designed specifically for program verification.

We use $\vdash \{p\}c\{q\}$ to represent that there is a derivation of $\{p\}c\{q\}$ following the rules.

We use $\models \{p\}c\{q\}$ to represent the meaning of $\{p\}c\{q\}$.

Soundness of the program logic:
If $\vdash \{p\}c\{q\}$, we have $\models \{p\}c\{q\}$.
If $\vdash [p]c[q]$, we have $\models [p]c[q]$.

Completeness of the program logic:
If $\models \{p\}c\{q\}$, we have $\vdash \{p\}c\{q\}$.
If $\models [p]c[q]$, we have $\vdash [p]c[q]$.

Soundness and completeness

Hoare logic is both sound and complete, provided that the underlying logic is!
(Often, the underlying logic is sound but incomplete.)

We will prove this now.

Roadmap

Review of predicate logic

- syntax
- semantics
- soundness and completeness

Formal semantics of Hoare triples

- preconditions and postconditions
- semantics of commands
- soundness of Hoare axioms and rules
- completeness and relative completeness

Review of predicate logic

Syntax:

$$
\begin{array}{llll}
\text{(IntExp)} & e & ::= & n \mid x \mid e + e \mid e - e \mid \ldots \\
\text{(Assn)} & p, q & ::= & \textbf{true} \mid \textbf{false} \\
 & & \mid & e = e \mid e < e \mid e > e \mid \ldots \qquad \text{(predicates)} \\
 & & \mid & \neg p \mid p \wedge p \mid p \vee p \mid p \Rightarrow p \\
 & & \mid & \forall x.\, p \mid \exists x.\, p
\end{array}
$$

Review of predicate logic

A more general version:

$$\begin{array}{lllll} \text{(IntExp)} & e & ::= & n \mid x \mid f(e, \ldots, e) \\ \\ \text{(Assn)} & p, q & ::= & \textbf{true} \mid \textbf{false} \mid P(e, \ldots, e) \\ & & \mid & \neg p \mid p \wedge p \mid p \vee p \mid p \Rightarrow p \\ & & \mid & \forall x.\, p \mid \exists x.\, p \end{array}$$

But we will use the simpler one.

Semantics of assertions

$$\sigma \models \textbf{true} \qquad \text{always}$$

$$\sigma \models \textbf{false} \qquad \text{never}$$

$$\sigma \models e_1 = e_2 \quad \textit{iff} \quad [\![e_1]\!]_{intexp}\,\sigma = [\![e_2]\!]_{intexp}\,\sigma$$

$$\sigma \models \neg p \quad \textit{iff} \quad \neg(\sigma \models p)$$

$$\sigma \models p \wedge q \quad \textit{iff} \quad (\sigma \models p) \wedge (\sigma \models q)$$

$$\sigma \models p \vee q \quad \textit{iff} \quad (\sigma \models p) \vee (\sigma \models q)$$

$$\sigma \models p \Rightarrow q \quad \textit{iff} \quad (\sigma \models p) \Rightarrow (\sigma \models q)$$

$$\sigma \models \forall x.\, p \quad \textit{iff} \quad \forall n.\, \sigma\{x \rightsquigarrow n\} \models p$$

$$\sigma \models \exists x.\, p \quad \textit{iff} \quad \exists n.\, \sigma\{x \rightsquigarrow n\} \models p$$

Validity of assertions

- p holds in σ: $\sigma \models p$
- p is valid: for all σ, p holds in σ. We will write $\models p$.
- p is unsatisfiable: $\neg p$ is valid

Soundness and completeness of predicate logic

Deductive system for predicate logic specifies $\vdash p$, i.e. p is provable

Soundness: if $\vdash p$ then $\models p$

- proof by induction on derivation of $\vdash p$

Completeness: if $\models p$ then $\vdash p$

- Gödel's incompleteness theorem: there exists no proof system for arithmetic in which all valid assertions are systematically derivable

Semantics of Hoare triples

Recall that $\{p\}c\{q\}$ is is valid, iff

- if c is executed in a state initially satisfying p
- and *if* the execution of c terminates
- then the final state satisfies q

p and q are predicate logic formula

Will formalize semantics of $\{p\}c\{q\}$ to express:

- if c is executed in a state σ such that $\sigma \models p$
- and *if* the execution of c starting in σ terminates in a state σ'
  - this is the semantics of c
- then $\sigma' \models q$

Review of small-step operational semantics

$$\frac{[\![e]\!]_{intexp}\,\sigma = n}{(x := e, \sigma) \longrightarrow (\textbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

$$\frac{(c_0,\, \sigma) \longrightarrow (c_0',\, \sigma')}{(c_0\,;c_1,\, \sigma) \longrightarrow (c_0'\,;c_1,\, \sigma')} \qquad \frac{}{(\textbf{skip}\,;c_1,\, \sigma) \longrightarrow (c_1,\, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \mathit{true}}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\, \sigma) \longrightarrow (c_0,\, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \mathit{false}}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\, \sigma) \longrightarrow (c_1,\, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \mathit{true}}{(\textbf{while } b \textbf{ do } c,\, \sigma) \longrightarrow (c\,;\textbf{while } b \textbf{ do } c,\, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \mathit{false}}{(\textbf{while } b \textbf{ do } c,\, \sigma) \longrightarrow (\textbf{skip},\, \sigma)}$$

Semantics of Hoare triples

$$\models \{p\}c\{q\} \;\; \textit{iff} \;\; \forall \sigma, \sigma'. \, (\sigma \models p) \wedge ((c, \sigma) \longrightarrow^* (\textbf{skip}, \sigma')) \Rightarrow (\sigma' \models q)$$

$$\models [p]c[q] \;\; \textit{iff} \;\; \forall \sigma. \, (\sigma \models p) \Rightarrow \exists \sigma'. \, ((c, \sigma) \longrightarrow^* (\textbf{skip}, \sigma')) \wedge (\sigma' \models q)$$

Soundness proof (for partial correctness)

Soundness: if $\vdash \{p\}c\{q\}$, then $\models \{p\}c\{q\}$.

Definition $\text{Safe}^n(c, \sigma, q)$:

- $\text{Safe}^0(c, \sigma, q)$ always holds;
- $\text{Safe}^{n+1}(c, \sigma, q)$ holds iff one of the following is true:
  - $c = \textbf{skip}$ and $\sigma \models q$; or
  - there exist c' and σ' such that $(c, \sigma) \longrightarrow (c', \sigma')$ and $\text{Safe}^n(c', \sigma', q)$

We say $\text{Safe}(c, \sigma, q)$ iff $\text{Safe}^n(c, \sigma, q)$ holds for all n.

Lemma 1:
For all σ, if $\sigma \models p$ implies $\text{Safe}(c, \sigma, q)$, then $\models \{p\}c\{q\}$.
Lemma 2:
If $\vdash \{p\}c\{q\}$, then for all σ such that $\sigma \models p$, we have $\text{Safe}(c, \sigma, q)$.

Soundness proof (Lemma 1)

To prove Lemma 1, we prove the following important lemmas:

Lemma (Progress):
If Safe(c, σ, q), then either c is **skip**, or there exist c' and σ' such that $(c, \sigma) \longrightarrow (c', \sigma')$.

Lemma (Preservation):
If Safe(c, σ, q) and $(c, \sigma) \longrightarrow (c', \sigma')$, then Safe$(c', \sigma', q)$.

Both lemmas trivially follow from the definition of Safe.

Soundness proof (Lemma 2)

Lemma 2:
If $\vdash \{p\}c\{q\}$, then for all σ such that $\sigma \models p$, we have $\text{Safe}(c, \sigma, q)$.

Proof by *induction* over the derivation of $\vdash \{p\}c\{q\}$.

Soundness proof (Lemma 2)

Lemma 2:
If $\vdash \{p\}c\{q\}$, then for all σ such that $\sigma \models p$, we have Safe(c, σ, q).

Proof by *induction* over the derivation of $\vdash \{p\}c\{q\}$.

- base case: show the result holds if the last step of the derivation is by axioms, i.e. the (SK) and (AS) rules
- inductive step: show that when the last step of the derivation is by applying a rule (with judgments as hypotheses), then if the result holds for the judgments in the hypotheses of the rule (induction hypothesis), then it holds also for the judgment in the conclusion

Thus the result holds for all derivations using the logic rules

You may also think this proof is by induction over the height of the derivation tree.

Recall the rules of Hoare logic

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\textsc{as}) \qquad \frac{}{\{p\}\mathbf{skip}\{p\}}\ (\textsc{sk})$$

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1\ ;c_2\{q\}}\ (\textsc{sc}) \qquad \frac{\{p \wedge b\}c_1\{q\} \qquad \{p \wedge \neg b\}c_2\{q\}}{\{p\}\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{q\}}\ (\textsc{cd})$$

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i \wedge \neg b\}}\ (\textsc{whp})$$

$$\frac{p \Rightarrow q \qquad \{q\}c\{r\}}{\{p\}c\{r\}}\ (\textsc{sp}) \qquad \frac{\{p\}c\{q\} \qquad q \Rightarrow r}{\{p\}c\{r\}}\ (\textsc{wc})$$

“Soundness” of individual rules

Lemma (AS):
For all σ such that $\sigma \models p[e/x]$, we have Safe$(x := e, \sigma, p)$.

“Soundness” of individual rules

Lemma (AS):
For all σ such that $\sigma \models p[e/x]$, we have $\text{Safe}(x := e, \sigma, p)$.

That is, we need to prove:
For all n, for all σ, if $\sigma \models p[e/x]$, then $\text{Safe}^n(x := e, \sigma, p)$.

“Soundness” of individual rules

Lemma (AS):
For all σ such that $\sigma \models p[e/x]$, we have $\text{Safe}(x := e, \sigma, p)$.

That is, we need to prove:
For all n, for all σ, if $\sigma \models p[e/x]$, then $\text{Safe}^n(x := e, \sigma, p)$.

Proof by induction over n.

- base case: $n = 0$. Trivial.
- inductive step: $n = k + 1$. ...

“Soundness” of individual rules

Lemma (AS):
For all σ such that $\sigma \models p[e/x]$, we have $\text{Safe}(x := e, \sigma, p)$.

That is, we need to prove:
For all n, for all σ, if $\sigma \models p[e/x]$, then $\text{Safe}^n(x := e, \sigma, p)$.

Proof by induction over n.

- base case: $n = 0$. Trivial.
- inductive step: $n = k + 1$. ...

Substitution Lemma:
If $\sigma \models p[e/x]$ and $[\![e]\!]_{intexp}\, \sigma = n$, then $\sigma\{x \rightsquigarrow n\} \models p$.

Review of substitution

$$
\begin{array}{lcl}
x[e/x] & = & e \\
y[e/x] & = & y \\
(e_0 + e_1)[e/x] & = & (e_0[e/x]) + (e_1[e/x])
\end{array}
$$

$$
\begin{array}{lcll}
(p \wedge q)[e/x] & = & (p[e/x]) \wedge (q[e/x]) & \\
(\forall x.\, p)[e/x] & = & \forall x.\, p & \\
(\forall y.\, p)[e/x] & = & \forall y.\, (p[e/x]) & \text{if } y \notin fv(e) \\
(\forall y.\, p)[e/x] & = & \forall z.\, (p[z/y][e/x]) & \text{if } y \in fv(e) \text{ and } z \text{ fresh} \\
\ldots & & &
\end{array}
$$

Examples:

$$(x < 0 \wedge \exists x.\, x \leq y)[y + 1/x] = y + 1 < 0 \wedge \exists x.\, x \leq y$$

$$(x < 0 \wedge \exists x.\, x \leq y)[x + 1/y] = x < 0 \wedge \exists z.\, z \leq x + 1$$

Substitution Lemma

If $\sigma \models p[e/x]$ and $[\![e]\!]_{intexp}\ \sigma = n$, then $\sigma\{x \rightsquigarrow n\} \models p$.

Proof: By induction over the structure of p.

“Soundness” of individual rules

Lemma (sc):
If

1. for all σ, if $\sigma \models p$, then Safe(c_1, σ, r);
2. for all σ, if $\sigma \models r$, then Safe(c_2, σ, q);

then, for all σ such that $\sigma \models p$, we have Safe$(c_1 \,;\, c_2, \sigma, q)$.

“Soundness” of individual rules

Lemma (WHP):
If

1. for all σ, if $\sigma \models i \wedge b$, then Safe$(c, \sigma, i)$

then, for all σ such that $\sigma \models i$, we've Safe$(\textbf{while } b \textbf{ do } c, \sigma, i \wedge \neg b)$.

“Soundness” of individual rules

Lemma (SP):
If

1. $p \Rightarrow q$;
2. for all σ, if $\sigma \models q$, then Safe(c, σ, r)

then, for all σ such that $\sigma \models p$, we have Safe(c, σ, r).

Other rules are similar.

Semantics and soundness based-on big-step semantics

The soundness can also be defined with respect to big-step semantics. The proof is simpler.

Incompleteness of Hoare logic

Soundness: if $\vdash \{p\}c\{q\}$ then $\models \{p\}c\{q\}$

Completeness: if $\models \{p\}c\{q\}$ then $\vdash \{p\}c\{q\}$

- to show this not possible, first observe that for any p,

$$\models \{\mathbf{true}\}\mathbf{skip}\{p\} \;\Leftrightarrow\; \models p$$

$$\vdash \{\mathbf{true}\}\mathbf{skip}\{p\} \;\Leftrightarrow\; \vdash p$$

  thus, if Hoare logic was complete, then contradicting Gödel's theorem
- alternative proof (using computability theory):
  $\models \{\mathbf{true}\}c\{\mathbf{false}\}$ iff c does not halt. But the halting problem is undecidable.

Relative completeness

Actual reason of incompleteness are rules (SP) and (wc) since they are based on the validity of implications within predicate logic.

Therefore: separation of proof system (Hoare logic) and assertion language (predicate logic)

One can show: if an “oracle” is available which decides whether a given assertion is valid, then all valid partial correctness properties can be systematically derived

$\Longrightarrow$ Relative completeness

Relative completeness

Theorem [Cook 1978]: Hoare Logic is *relatively complete*, i.e., if $\models \{p\}c\{q\}$ then $\Gamma \vdash \{p\}c\{q\}$ where $\Gamma = \{p \mid (\models p)\}$

Thus: if we know that a partial correctness property is valid, then we know that there is a corresponding derivation.

The proof uses the following concept: assume that, e.g., $\{p\}c_1 ; c_2\{q\}$ has to be derived. This requires an intermediate assertion r such that $\{p\}c_1\{r\}$ and $\{r\}c_2\{q\}$. How to find it?

Weakest precondition

Definition: Given command c and assertion q, the *weakest precondition* $\mathrm{wp}(c, q)$ is an assertion such that

$$\sigma \models \mathrm{wp}(c, q) \;\Leftrightarrow\; (\forall \sigma'.\, (c, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma') \Rightarrow \sigma' \models q)$$

Corollary: For all p, c and q,

$$\models \{p\}c\{q\} \;\Leftrightarrow\; \models (p \Rightarrow \mathrm{wp}(c, q))$$

Weakest precondition

Definition: An assertion language is called *expressive* if, for every c and q, the weakest precondition $\mathrm{wp}(c, q)$ is an assertion in the language.

The assertion language we have used is expressive.

Proof.

$$
\begin{aligned}
\mathrm{wp}(\textbf{skip}, q) &= q \\
\mathrm{wp}(x := e, q) &= q[e/x] \\
\mathrm{wp}(c_1 \,;\, c_2, q) &= \mathrm{wp}(c_1, \mathrm{wp}(c_2, q)) \\
\mathrm{wp}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, q) &= (b \wedge \mathrm{wp}(c_1, q)) \vee (\neg b \wedge \mathrm{wp}(c_2, q))
\end{aligned}
$$

For **while**, tricky encoding in first-order arithmetic using Gödel's β function (see Winskel's book *The formal semantics of programming languages: an introduction*)

Relative completeness

Lemma: For every c and q,

$$\vdash \{\mathrm{wp}(c, q)\}c\{q\}$$

Proof by induction over the structure of c.

Proof of Cook's Completeness Theorem. We have to show: $\models \{p\}c\{q\} \Rightarrow \Gamma \vdash \{p\}c\{q\}$ where $\Gamma = \{p \mid (\models p)\}$.

- From the above lemma, we know $\vdash \{\mathrm{wp}(c, q)\}c\{q\}$
- Since $\models \{p\}c\{q\}$, we know $\models p \Rightarrow \mathrm{wp}(c, q)$.
- Thus $p \Rightarrow \mathrm{wp}(c, q)$ is in Γ.
- By the (SP) rule, we have $\Gamma \vdash \{p\}c\{q\}$.

Summary

Hoare logic is sound

Hoare logic for our simple language is complete relative to an oracle

- oracle must be able to prove $p \Rightarrow \text{wp}(c, q)$
- $\text{wp}(c, q)$ must be expressible in assertion language

The incompleteness of the proof system for simple Hoare logic stems from the weakness of the proof system of the assertion language logic, not any weakness of the Hoare logic proof system.

Clarke showed relative completeness fails for complex languages

Outline

Formal semantics of a programming language

- Operational semantics
- Denotational semantics
- Axiomatic semantics

Axiomatic semantics

From wikipedia:

Axiomatic semantics is an approach based on mathematical logic to proving the correctness of computer programs. It is closely related to Hoare logic.

Axiomatic semantics define the meaning of a command in a program by describing its effect on assertions about the program state.

Example (Euclid)

7.1. (module rule)

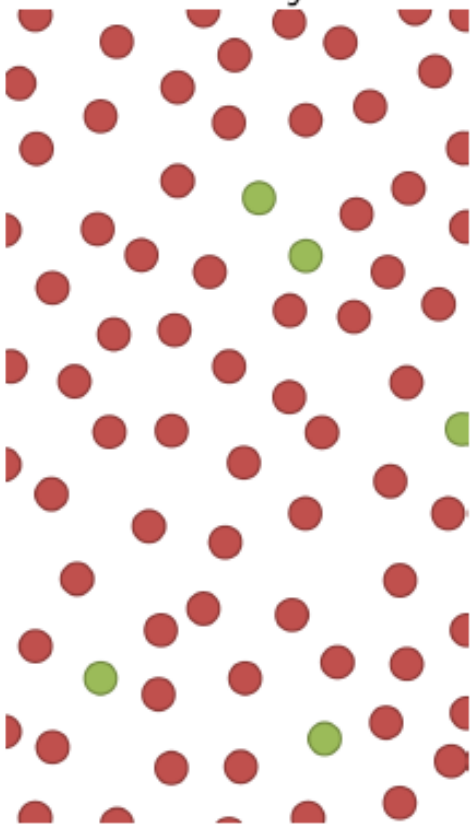
$$
\begin{array}{ll}
(1) & Q \supset Q0(A/t), \\
(2) & P1\{\textbf{const } K; \textbf{var } V; S_4\}\ Q4(A/t) \wedge Q, \\
(3) & P2(A/t) \wedge Q\{S_2\}\ Q2(A/t) \wedge Q, \\
(4) & \exists\, g1(P3(A/t) \wedge Q\{S_3\}\ Q3(A/t) \wedge g = g1(A, c, d)), \\
(5) & \exists\, g(P3(A/t) \wedge Q \supset Q3(A/t)), \\
(6) & P6(A/t) \wedge Q\{S_6\}\ Q1, \\
(7) & P \supset P1(a/c), \\
(8.1) & [Q0(a/c, x/t, x'/t') \supset (P2(x/t, x'/t', a2/x2, e2/e2, a/c) \wedge \\
 & \quad (Q2(x2\#/t, x'/t', a2\#/x2, e2/e2, a/c, y2\#/y2, a2/x2', y2/y2') \supset \\
 & \quad R1(x2\#/x, a2\#/a2, y2\#/y2)))\ \{x\,.\,p(a2, e2)\}\ R1 \wedge Q0(a/c, x/t, x'/t'), \\
(8.2) & (Q0(a/c, x/t) \supset P3(x/t, a3/e3, a/c)) \supset \\
 & \quad Q3(x/t, a3/e3, a/c, f(a3, d3)/g) \wedge Q0(a/c, x/t), \\
(8.3) & P1(a/c) \wedge (Q4(x4\#/t, x'/t', a/c, y4\#/y4, y4/y4') \supset R4(x4\#/x, y4\#/y4)) \\
 & \quad \{x\,.\,Initially\}\ R4 \wedge Q0(a/c, x/t, x'/t'), \\
(8.4) & (Q0(a/c, x/t, x'/t') \supset P6(x/t, x'/t', a/c)) \wedge (Q1(a/c, y6\#/y6, y6/y6') \supset \\
 & \quad R(y6\#/y6))\ \{x\,.\,Finally\}\ R] \\
 & \vdash \\
(8.5) & P(x\#/x)\ \{x\,.\,Initially; S; x\,.\,Finally\}\ R(x\#/x) \\
\hline
 & P\{\textbf{var } x: T(a); S\}\ R \wedge Q1
\end{array}
$$

From London et al.'s *Proof rules for the programming language Euclid* (1978)

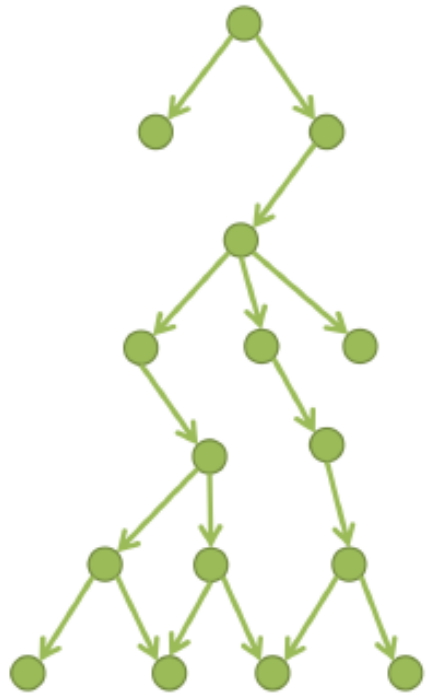
Another example (C11)

The C11 standard uses both styles of semantics:

An axiomatic style for concurrency



and an operational style for the rest



From Kyndylan Nienhuis

Another example (C11)

$$\forall a, b.\ sb(a,b) \implies \mathsf{tid}(a) = \mathsf{tid}(b) \qquad \text{(ConsSB)}$$

$$\mathsf{order}(\mathsf{iswrite}, mo) \wedge \forall \ell.\ \mathsf{total}(\mathsf{iswrite}_\ell, mo) \qquad \text{(ConsMO)}$$

$$\mathsf{order}(\mathsf{isSC}, sc) \wedge \mathsf{total}(\mathsf{isSC}, sc) \wedge (\mathsf{hb} \cup mo) \cap (\mathsf{isSC} \times \mathsf{isSC}) \subseteq sc \qquad \text{(ConsSC)}$$

$$\forall b.\ (\exists c.\ rf(b) = c) \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a,b) \qquad \text{(ConsRFdom)}$$

$$\forall a, b.\ rf(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \qquad \text{(ConsRF)}$$

$$\forall a, b.\ rf(b) = a \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \implies \mathsf{hb}(a,b) \qquad \text{(ConsRFna)}$$

$$\forall a, b.\ rf(b) = a \wedge \mathsf{isSC}(b) \implies \mathsf{imm}(\mathsf{scr}, a, b) \vee \neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a,x) \wedge \mathsf{imm}(\mathsf{scr}, x, b) \qquad \text{(SCReads)}$$

$$\nexists a.\ \mathsf{hb}(a,a) \qquad \text{(IrrHB)}$$

$$\nexists a, b.\ rf(b) = a \wedge \mathsf{hb}(b,a) \qquad \text{(ConsRFhb)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(b,a) \qquad \text{(CohWW)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(rf(b), rf(a)) \qquad \text{(CohRR)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(rf(b), a) \qquad \text{(CohWR)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(b, rf(a)) \qquad \text{(CohRW)}$$

$$\forall a, b.\ \mathsf{isrmw}(a) \wedge rf(a) = b \implies \mathsf{imm}(mo, b, a) \qquad \text{(AtRMW)}$$

$$\forall a, b, \ell.\ lab(a) = lab(b) = \mathsf{A}(\ell) \implies a = b \qquad \text{(ConsAlloc)}$$

where $\mathsf{order}(P,R) \stackrel{\text{def}}{=} (\nexists a.\ R(a,a)) \wedge (R^+ \subseteq R) \wedge (R \subseteq P \times P)$

$\mathsf{total}(P,R) \stackrel{\text{def}}{=} (\forall a, b.\ P(a) \wedge P(b) \implies a = b \vee R(a,b) \vee R(b,a))$

$\mathsf{imm}(R,a,b) \stackrel{\text{def}}{=} R(a,b) \wedge \nexists c.\ R(a,c) \wedge R(c,b)$

$\mathsf{scr}(a,b) \stackrel{\text{def}}{=} sc(a,b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a)$

Figure 5. Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(lab, sb, asw, rf, mo, sc)$.

From Vafeiadis et al.'s *Common compiler optimisations are invalid in the C11 memory model and what we can do about it* (2015)

Summary of Hoare Logic

Hoare logic is a deductive proof system for Hoare triples $\{p\}c\{q\}$.

Formal proof is syntactic "symbol pushing".

- The rules say "if you have a string of characters of this form, you can obtain a new string of characters of this other form"
- Even if you don't know what the strings are intended to mean, provided the rules are designed properly and you apply them correctly, you will get correct results (though not necessarily the desired result)

Hoare logic is compositional.

- The structure of a program's correctness proof mirrors the structure of the program itself.

Coq Implementations

When encoding a logic into a proof assistant such as Coq, a choice needs to be made between using a *shallow* and a *deep* embedding.

Deep embedding:

1. define a datatype representing the syntax for your logic
2. give a model of the syntax
3. prove that axioms about your syntax are sound with respect to the model

Shallow embedding: just start with a model, and prove entailments between formulas

`http://cstheory.stackexchange.com/questions/1370/shallow-versus-deep-embeddings`

From Software Foundations (Hoare.v)

```
Definition Assertion := state → Prop.
```

```
Definition as1 : Assertion := fun st ⇒ st X = 3.
Definition as2 : Assertion := fun st ⇒ st X ≤ st Y.
Definition as3 : Assertion :=
  fun st ⇒ st X = 3 ∨ st X ≤ st Y.
Definition as4 : Assertion :=
  fun st ⇒ st Z × st Z ≤ st X ∧
            ¬ (((S (st Z)) × (S (st Z))) ≤ st X).
Definition as5 : Assertion := fun st ⇒ True.
Definition as6 : Assertion := fun st ⇒ False.
```

```
Definition assert_implies (P Q : Assertion) : Prop :=
  ∀st, P st → Q st.

Notation "P ⇾ Q" :=
  (assert_implies P Q) (at level 80) : hoare_spec_scope.
Open Scope hoare_spec_scope.
```

From Software Foundations (Hoare.v)

```
Definition hoare_triple
           (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀st st',
       c / st ⇓ st' →
       P st →
       Q st'.


Notation "{{ P }} c {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level)
  : hoare_spec_scope.
```

From Software Foundations (Hoare.v)

```
Definition assn_sub X a P : Assertion :=
  fun (st : state) ⇒
    P (update st X (aeval st a)).

Notation "P [ X |-> a ]" := (assn_sub X a P) (at level 10).
```

```
Theorem hoare_asgn : ∀Q X a,
  {{Q [X ↦ a]}} (X ::= a) {{Q}}.
Proof.
  unfold hoare_triple.
  intros Q X a st st' HE HQ.
  inversion HE. subst.
  unfold assn_sub in HQ. assumption. Qed.
```

From Software Foundations (Hoare.v)

```
Theorem hoare_consequence_pre : ∀(P P' Q : Assertion) c,
  {P'} c {Q} →
  P ->> P' →
  {P} c {Q}.
```

□

```
Theorem hoare_consequence_post : ∀(P Q Q' : Assertion) c,
  {P} c {Q'} →
  Q' ->> Q →
  {P} c {Q}.
```

□

From Software Foundations (HoareAsLogic.v)

```
Inductive hoare_proof : Assertion → com → Assertion → Type
:=
  | H_Skip : ∀P,
      hoare_proof P (SKIP) P
  | H_Asgn : ∀Q V a,
      hoare_proof (assn_sub V a Q) (V ::= a) Q
  | H_Seq : ∀P c Q d R,
      hoare_proof P c Q → hoare_proof Q d R → hoare_proof P
(c;;d) R
  | H_If : ∀P Q b c1 c2,
    hoare_proof (fun st ⇒ P st ∧ bassn b st) c1 Q →
    hoare_proof (fun st ⇒ P st ∧ ~(bassn b st)) c2 Q →
    hoare_proof P (IFB b THEN c1 ELSE c2 FI) Q
  | H_While : ∀P b c,
    hoare_proof (fun st ⇒ P st ∧ bassn b st) c P →
    hoare_proof P (WHILE b DO c END) (fun st ⇒ P st ∧ ¬ (bassn
b st))
  | H_Consequence : ∀(P Q P' Q' : Assertion) c,
    hoare_proof P' c Q' →
    (∀st, P st → P' st) →
    (∀st, Q' st → Q st) →
    hoare_proof P c Q.
```

From Software Foundations (HoareAsLogic.v)

```
Theorem hoare_proof_sound : ∀P c Q,
  hoare_proof P c Q → {{P}} c {{Q}}.
```