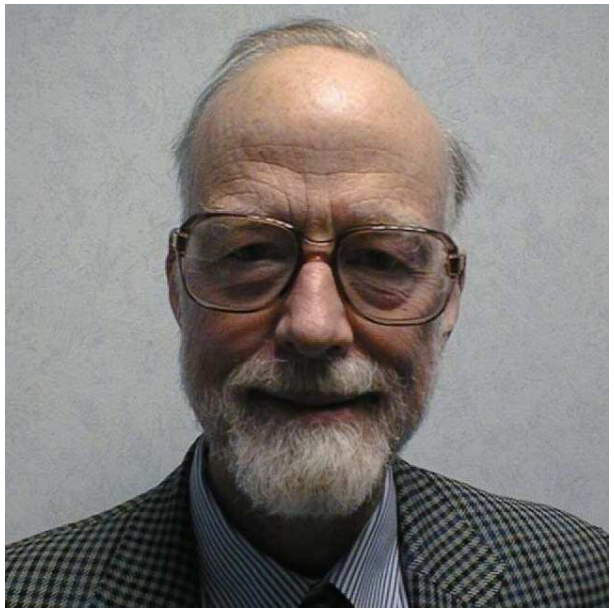
Hoare Logic

http://www.cl.cam.ac.uk/~mjcg/HoareLogic.html

- **Program specification using Hoare notation**
- **Axioms and rules of Hoare Logic**
- **Soundness and completeness**
- **Mechanised program verification**
- **Pointers, the frame problem and separation logic**



Prof. Mike Gordon

Program Specification and Verification

- **This course is about *formal* ways of specifying and validating software**
- **This contrasts with *informal* methods:**
  - **natural language specifications**
  - **testing**
- **Formal methods are *not* a panacea**
  - **formally verified designs may still not work**
  - **can give a false sense of security**
- **Assurance versus debugging**
  - **formal verification (FV) can reveal hard-to-find bugs**
  - **can also be used for assurance e.g. "proof of correctness"**
  - **Microsoft use FV for debugging, NSA use FV for assurance**
- **Goals of course:**
  - **enable you to understand and criticise 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
- It may be better to have something that works most of the time than nothing at all
- 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 course is concerned with Floyd-Hoare Logic**
  - **also known just as Hoare Logic**
- **Floyd-Hoare Logic is a method of reasoning mathematically about *imperative* programs**
- **It is the basis of mechanized program verification systems**
  - **the architecture of these will be described later**
- **Industrial program development methods like SPARK use ideas from Floyd-Hoare Logic to obtain high assurance**
- **Developments to the logic still under active development**
  - **e.g. separation logic (reasoning about pointers)**
  - **2/3 of 2010 BCS Distinguished Dissertation awards concerned separation logic**

A Little Programming Language

Expressions:

$E ::= \quad N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$

Boolean expressions:

$B ::= \quad \texttt{T} \mid \texttt{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$

Commands:

$C ::= \quad V \texttt{ := } E$
$\quad\mid\quad C_1 \texttt{ ; } C_2$
$\quad\mid\quad \texttt{IF } B \texttt{ THEN } C_1 \texttt{ ELSE } C_2$
$\quad\mid\quad \texttt{WHILE } B \texttt{ DO } C$

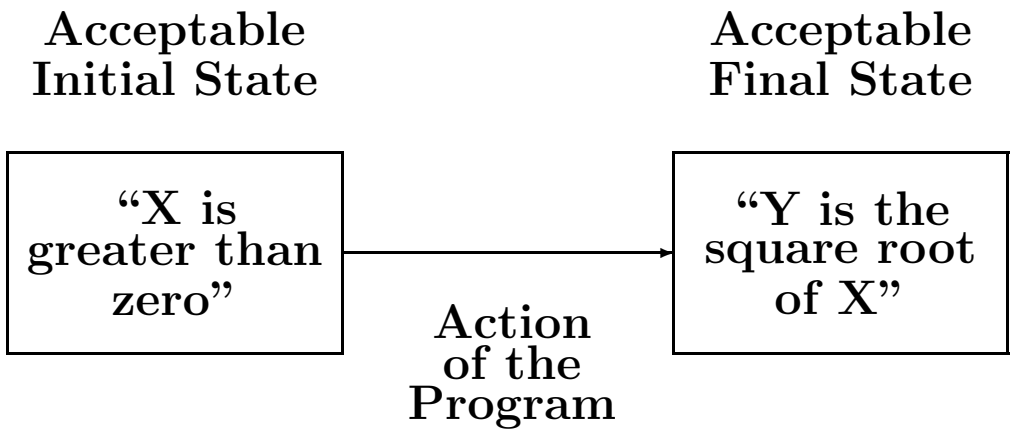
Some Notation

- **Programs are built out of *commands* like assignments, conditionals, while-loops etc**

- **The terms 'program' and 'command' are synonymous**
  - **the former generally used for commands representing complete algorithms**

- **The term 'statement' is used for conditions on program variables that occur in correctness specifications**
  - **potential for confusion: some people use this word for commands**

Specification of Imperative Programs



Hoare's notation

- **C.A.R. Hoare introduced the following notation called a** *partial correctness specification* **for specifying what a program does:**

$$\{P\}\ C\ \{Q\}$$

where:

  - C **is a command**
  - P **and** Q **are conditions on the program variables used in** C

- **Conditions on program variables will be written using standard mathematical notations together with** *logical operators* **like:**

  - $\wedge$ **('and'),** $\vee$ **('or'),** $\neg$ **('not'),** $\Rightarrow$ **('implies')**

- **Hoare's original notation was** $P\ \{C\}\ Q$ **not** $\{P\}\ C\ \{Q\}$**, but the latter form is now more widely used**

Meaning of Hoare's Notation

- $\{P\}\ C\ \{Q\}$ **is true if**
  - **whenever** C **is executed in a state satisfying** P
  - **and** *if* **the execution of** C **terminates**
  - **then the state in which** C **terminates satisfies** Q
- **Example:** $\{\texttt{X} = 1\}\ \texttt{X:=X+1}\ \{\texttt{X} = 2\}$
  - P **is the condition that the value of** `X` **is** `1`
  - Q **is the condition that the value of** `X` **is** `2`
  - C **is the assignment command** `X:=X+1`
    - **i.e. 'X becomes X+1'**
- $\{\texttt{X} = 1\}\ \texttt{X:=X+1}\ \{\texttt{X} = 2\}$ **is true**
- $\{\texttt{X} = 1\}\ \texttt{X:=X+1}\ \{\texttt{X} = 3\}$ **is false**

Formal versus Informal Proof

- **Mathematics text books give** *informal proofs*
- **English arguments are used**
  - **proof of** $(\mathtt{X}+1)^2 = \mathtt{X}^2 + 2 \times \mathtt{X} + 1$

    "follows by the definition of squaring and distributivity laws"
- **Formal verification uses** *formal proof*
  - **the rules used are described and followed very precisely**
  - **formal proof has been used to discover errors in published informal ones**
- **Here is an example formal proof**

1.	$(\mathtt{X}+1)^2$	$= (\mathtt{X}+1) \times (\mathtt{X}+1)$	Definition of $()^2$.
2.	$(\mathtt{X}+1) \times (\mathtt{X}+1)$	$= (\mathtt{X}+1) \times \mathtt{X} + (\mathtt{X}+1) \times 1$	Left distributive law of $\times$ over $+$.
3.	$(\mathtt{X}+1)^2$	$= (\mathtt{X}+1) \times \mathtt{X} + (\mathtt{X}+1) \times 1$	Substituting line 2 into line 1.
4.	$(\mathtt{X}+1) \times 1$	$= \mathtt{X}+1$	Identity law for 1.
5.	$(\mathtt{X}+1) \times \mathtt{X}$	$= \mathtt{X} \times \mathtt{X} + 1 \times \mathtt{X}$	Right distributive law of $\times$ over $+$.
6.	$(\mathtt{X}+1)^2$	$= \mathtt{X} \times \mathtt{X} + 1 \times \mathtt{X} + \mathtt{X} + 1$	Substituting lines 4 and 5 into line 3.
7.	$1 \times \mathtt{X}$	$= \mathtt{X}$	Identity law for 1.
8.	$(\mathtt{X}+1)^2$	$= \mathtt{X} \times \mathtt{X} + \mathtt{X} + \mathtt{X} + 1$	Substituting line 7 into line 6.
9.	$\mathtt{X} \times \mathtt{X}$	$= \mathtt{X}^2$	Definition of $()^2$.
10.	$\mathtt{X} + \mathtt{X}$	$= 2 \times \mathtt{X}$	2=1+1, distributive law.
11.	$(\mathtt{X}+1)^2$	$= \mathtt{X}^2 + 2 \times \mathtt{X} + 1$	Substituting lines 9 and 10 into line 8.

The Structure of Proofs

- **A proof consists of a sequence of lines**
- **Each line is an instance of an** *axiom*
  - **like the definition of** $()^2$
- **or follows from previous lines by a** *rule of inference*
  - **like the substitution of equals for equals**
- **The statement occurring on the last line of a proof is the statement** *proved* **by it**
  - **thus** $(\mathtt{X}+\mathtt{1})^2 = \mathtt{X}^2 + \mathtt{2} \times \mathtt{X} + \mathtt{1}$ **is proved by the proof on the previous slide**
- **These are 'Hilbert style' formal proofs**
  - **can use a tree structure rather than a linear one**
  - **choice is a matter of convenience**

Formal proof is syntactic 'symbol pushing'

- Formal Systems reduce verification and proof to 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
- Thus computers can do formal verification
- Formal verification by hand generally not feasible
  - maybe hand verify high-level design, but not code
- Famous paper that's worth reading:
  - "Social processes and the proofs of theorems and programs".
    R. A. DeMillo, R. J. Lipton, and A. J. Perlis. CACM, May 1979
- Also see the book "Mechanizing Proof" by Donald MacKenzie

Hoare's Verification Grand Challenge

- **Bill Gates, keynote address at WinHec 2002**

  ``... software verification ... has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we are building tools that can do actual proof about the software and how it works in order to guarantee the reliability.''

- **Hoare has posed a challenge**

  The verification challenge is to achieve a significant body of verified programs that have precise external specifications, complete internal specifications, machine-checked proofs of correctness with respect to a sound theory of programming.

  The Deliverables

  A comprehensive theory of programming that covers the features needed to build practical and reliable programs.

  A coherent toolset that automates the theory and scales up to the analysis of large codes.

  A collection of verified programs that replace existing unverified ones, and continue to evolve in a verified state.

- **"You can't say anymore it can't be done! Here, we have done it."**

Hoare Logic and Verification Conditions

- **Hoare Logic is a deductive proof system for Hoare triples $\{P\}\ C\ \{Q\}$**
- **Can use Hoare Logic directly to verify programs**
  - **original proposal by Hoare**
  - **tedious and error prone**
  - **impractical for large programs**
- **Can 'compile' proving $\{P\}\ C\ \{Q\}$ to verification conditions**
  - **more natural**
  - **basis for computer assisted verification**
- **Proof of verification conditions equivalent to proof with Hoare Logic**
  - **Hoare Logic can be used to explain verification conditions**

Auxiliary Variables

- $\{$X=x $\wedge$ Y=y$\}$ R:=X; X:=Y; Y:=R $\{$X=y $\wedge$ Y=x$\}$
  - **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 **and** y**, which don't occur in the command and are used to name the initial values of program variables** X **and** Y
- **They are called** *auxiliary* **variables or** *ghost* **variables**
- **Informal convention:**
  - **program variable are upper case**
  - **auxiliary variable are lower case**

More simple examples

- $\{\texttt{X=x} \wedge \texttt{Y=y}\}$ `X:=Y; Y:=X` $\{\texttt{X=y} \wedge \texttt{Y=x}\}$
  - **this says that `X:=Y; Y:=X` exchanges the values of `X` and `Y`**
  - **this is not true**
- $\{\texttt{T}\}\ C\ \{Q\}$
  - **this says that whenever C halts, Q holds**
- $\{P\}\ C\ \{\texttt{T}\}$
  - **this specification is true for every condition P and every command C**
  - **because `T` is always true**
- $[P]\ C\ [\texttt{T}]$
  - **this says that C terminates if initially P holds**
  - **it says nothing about the final state**
- $[\texttt{T}]\ C\ [P]$
  - **this says that C always terminates and ends in a state where P holds**

A More Complicated Example

- $\{\mathtt{T}\}$

  $\left.\begin{array}{l}\mathtt{R:=X;}\\ \mathtt{Q:=0;}\\ \mathtt{WHILE\ Y}\leq\mathtt{R\ DO}\\ \quad\mathtt{(R:=R-Y;\ Q:=Q+1)}\end{array}\right\}\ \mathtt{C}$

  $\{\mathtt{R} < \mathtt{Y} \ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$

- **This is** $\{\mathtt{T}\}\ \mathtt{C}\ \{\mathtt{R} < \mathtt{Y} \ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$
  - **where** `C` **is the command indicated by the braces above**
  - **the specification is true if whenever the execution of** `C` **halts, then** `Q` **is quotient and** `R` **is the remainder resulting from dividing** `Y` **into** `X`
  - **it is true (even if** `X` **is initially negative!)**
  - **in this example** `Q` **is a program variable**
  - **don't confuse** `Q` **with the metavariable** Q **used in previous examples to range over postconditions (Sorry: my bad notation!)**

Specification can be Tricky

- **"The program must set `Y` to the maximum of `X` and `Y`"**
  - $[\texttt{T}]\ C\ [\texttt{Y} = \texttt{max(X,Y)}]$
- **A suitable program:**
  - `IF X >= Y THEN Y := X ELSE X := X`
- **Another?**
  - `IF X >= Y THEN X := Y ELSE X := X`
- **Or even?**
  - `Y := X`
- **Later you will be able to prove that these programs are "correct"**
- **The postcondition "`Y=max(X,Y)`" says "`Y` is the maximum of `X` and `Y` *in the final state*"**

Specification can be Tricky (ii)

- **The intended specification was probably *not* properly captured by**

  $\vdash$ {T} C {Y=max(X,Y)}

- **The correct formalisation of what was intended is probably**

  $\vdash$ {X=x $\wedge$ Y=y} C {Y=max(x,y)}

- **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!**

Review of Predicate Calculus

- **Program states are specified with *first-order logic* (FOL)**
- **Knowledge of this is assumed (brief review given now)**
- **In first-order logic there are two separate syntactic classes**
  - **Terms (or expressions): these denote values (e.g. numbers)**
  - **Statements (or formulae): these are either true or false**

Terms (Expressions)

- **Statements are built out of *terms* which denote *values* such as numbers, strings and arrays**
- **Terms, like `1` and `4 + 5`, denote a fixed value, and are called *ground***
- **Other terms contain *variables* like `x`, `X`, `y`, `X`, `z`, `Z` etc**
- **We use conventional notation, e.g. here are some terms:**

`X`, `y`, `Z`,

`1`, `2`, `325`,

`-X`, `-(X+1)`, `(x`$\times$`y)+Z`,

$\sqrt{\texttt{(1+x}^2\texttt{)}}$, `X!`, `sin(x)`, `rem(X,Y)`

- **Convention:**
  - **program variables are uppercase**
  - **auxiliary (i.e. logical) variables are lowercase**

Atomic Statements

- **Examples of atomic statements are**

$$\texttt{T}, \qquad \texttt{F}, \qquad \texttt{X} = \texttt{1}, \qquad \texttt{R} < \texttt{Y}, \qquad \texttt{X} = \texttt{R+(Y}\times\texttt{Q)}$$

- **`T` and `F` are atomic statements that are always true and false**

- **Other atomic statements are built from terms using *predicates*, e.g.**

$$\texttt{ODD(X)}, \qquad \texttt{PRIME(3)}, \qquad \texttt{X} = \texttt{1}, \qquad \texttt{(X+1)}^2 \geq \texttt{x}^2$$

- **`ODD` and `PRIME` are examples of predicates**

- **$=$ and $\geq$ are examples of *infixed* predicates**

- **`X`, `1`, `3`, `X+1`, $\texttt{(X+1)}^2$, $\texttt{x}^2$ are terms in above atomic statements**

Compound statements

- **Compound statements are built up from atomic statements using:**

$$\begin{array}{ll} \neg & \text{(not)} \\ \wedge & \text{(and)} \\ \vee & \text{(or)} \\ \Rightarrow & \text{(implies)} \\ \Leftrightarrow & \text{(if and only if)} \end{array}$$

  - **The single arrow $\rightarrow$ is commonly used for implication instead of $\Rightarrow$**

- **Suppose P and Q are statements, then**
  - **$\neg P$ is true if P is false, and false if P is true**
  - **$P \wedge Q$ is true whenever both P and Q are true**
  - **$P \vee Q$ is true if either P or Q (or both) are true**
  - **$P \Rightarrow Q$ is true if whenever P is true, then Q is true**
  - **$P \Leftrightarrow Q$ is true if P and Q are either both true or both false**

More on Implication

- **By convention we regard $P \Rightarrow Q$ as being true if P is false**
- **In fact, it is common to regard $P \Rightarrow Q$ as equivalent to $\neg P \vee Q$**
- **Some philosophers disagree with this treatment of implication**
  - **since any implication $A \Rightarrow B$ is true if A is false**
  - **e.g.** $(1 < 0) \;\Rightarrow\; (2 + 2 = 3)$
  - **search web for "paradoxes of implication"**
- **$P \Leftrightarrow Q$ is equivalent to $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$**
- **Sometimes write $P = Q$ or $P \equiv Q$ for $P \Leftrightarrow Q$**

Precedence

- **To reduce the need for brackets it is assumed that**
  - **$\neg$ is more binding than $\wedge$ and $\vee$**
  - **$\wedge$ and $\vee$ are more binding than $\Rightarrow$ and $\Leftrightarrow$**
- **For example**

$$\begin{array}{lll} \neg P \wedge Q & \text{is equivalent to} & (\neg P) \wedge Q \\ P \wedge Q \Rightarrow R & \text{is equivalent to} & (P \wedge Q) \Rightarrow R \\ P \wedge Q \Leftrightarrow \neg R \vee S & \text{is equivalent to} & (P \wedge Q) \Leftrightarrow ((\neg R) \vee S) \end{array}$$

Universal quantification

- **If S is a statement and x a variable**
- **Then $\forall x.\ S$ means:**

  'for all values of x, the statement S is true'

- **The statement**

  $$\forall x_1\ x_2\ \ldots\ x_n.\ S$$

  abbreviates

  $$\forall x_1.\ \forall x_2.\ \ldots\ \forall x_n.\ S$$

- **It is usual to adopt the convention that any unbound (i.e. *free*) variables in a statement are to be regarded as implicitly universally quantified**
- **For example, if n is a variable then the statement $n+0 = n$ is regarded as meaning the same as $\forall n.\ n + 0 = n$**

Existential quantification

- **If S is a statement and x a variable**
- **Then $\exists x.\ S$ means**

  'for some value of x, the statement S is true'

- **The statement**

  $$\exists x_1\ x_2\ \ldots\ x_n.\ S$$

  abbreviates

  $$\exists x_1.\ \exists x_2.\ \ldots\ \exists x_n.\ S$$

Summary

- **Predicate calculus forms the basis for program specification**
- **It is used to describe the acceptable initial states, and intended final states of programs**
- **We will next look at how to prove programs meet their specifications**
- **Proof of theorems within predicate calculus assumed known!**

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**

Notation for Axioms and Rules

- **If S is a statement, $\vdash S$ means S has a proof**
  - **statements that have proofs are called *theorems***
- **The axioms of Floyd-Hoare logic are specified by *schemas***
  - **these can be *instantiated* to get particular partial correctness specifications**
- **The inference rules of Floyd-Hoare logic will be specified with a notation of the form**

$$\frac{\vdash S_1, \ \ldots \ , \ \vdash S_n}{\vdash S}$$

  - **this means the *conclusion* $\vdash S$ may be deduced from the *hypotheses* $\vdash S_1$, ... , $\vdash S_n$**
  - **the hypotheses can either all be theorems of Floyd-Hoare logic**
  - **or a mixture of theorems of Floyd-Hoare logic and theorems of mathematics**

An example rule

The sequencing rule

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1;C_2\ \{R\}}$$

- **If a proof has lines matching** $\vdash \{P\}\ C_1\ \{Q\}$ **and** $\vdash \{Q\}\ C_2\ \{R\}$
- **One may deduce a new line** $\vdash \{P\}\ C_1;C_2\ \{R\}$
- **For example if one has deduced:**
  - $\vdash$ `{X=1} X:=X+1 {X=2}`
  - $\vdash$ `{X=2} X:=X+1 {X=3}`
- **One may then deduce:**
  - $\vdash$ `{X=1} X:=X+1; X:=X+1 {X=3}`
- **Method of verification conditions (VCs) generates** *proof obligation*
  - $\vdash$ `X=1` $\Rightarrow$ `X+(X+1)=3`
    - **VCs are handed to a theorem prover**
    - **"Extended Static Checking" (ESC) is an industrial example**

Reminder of our little programming language

- **The proof rules that follow constitute an *axiomatic semantics* of our programming language**

Expressions

$E \ ::= \ N \ \mid \ V \ \mid \ E_1 + E_2 \ \mid \ E_1 - E_2 \ \mid \ E_1 \times \ E_2 \ \mid \ \ldots$

Boolean expressions

$B \ ::= \ \texttt{T} \ \mid \ \texttt{F} \ \mid \ E_1 = E_2 \ \mid \ E_1 \leq E_2 \ \mid \ \ldots$

Commands

$C \ ::= \ V \ \texttt{:=} \ E$	**Assignments**
$\mid \ C_1 \ \texttt{;} \ C_2$	**Sequences**
$\mid \ \texttt{IF} \ B \ \texttt{THEN} \ C_1 \ \texttt{ELSE} \ C_2$	**Conditionals**
$\mid \ \texttt{WHILE} \ B \ \texttt{DO} \ C$	**WHILE-commands**

Syntactic Conventions

- **Symbols V, V_1, ... , V_n stand for arbitrary variables**
  - **examples of particular variables are `X`, `R`, `Q` etc**
- **Symbols E, E_1, ... , E_n stand for arbitrary expressions (or terms)**
  - **these are things like `X` $+ 1$, $\sqrt{2}$ etc. which denote values (usually numbers)**
- **Symbols S, S_1, ... , S_n stand for arbitrary statements**
  - **these are conditions like `X` $<$ `Y`, `X`$^2 = 1$ etc. which are either true or false**
  - **will also use P, Q, R to range over pre and postconditions**
- **Symbols C, C_1, ... , C_n stand for arbitrary commands**

The Assignment Axiom (Hoare)

- **Syntax:** V `:=` E
- **Semantics: value of V in final state is value of E in initial state**
- **Example:** `X:=X+1` **(adds one to the value of the variable** `X`**)**

The Assignment Axiom

$$\vdash \{Q[E/V]\}\ V := E\ \{Q\}$$

Where V is any variable, E is any expression, Q is any statement.

- **Instances of the assignment axiom are**
  - $\vdash \{\mathtt{E} = \mathtt{x}\}\ \mathtt{V} := \mathtt{E}\ \{\mathtt{V} = \mathtt{x}\}$
  - $\vdash \{\mathtt{Y} = 2\}\ \mathtt{X} := 2\ \{\mathtt{Y} = \mathtt{X}\}$
  - $\vdash \{\mathtt{X} + 1 = \mathtt{n} + 1\}\ \mathtt{X} := \mathtt{X} + 1\ \{\mathtt{X} = \mathtt{n} + 1\}$
  - $\vdash \{E = E\}\ \mathtt{X} := E\ \{\mathtt{X} = E\}$ **(if** `X` **does not occur in** E**)**

Precondition Strengthening

- **Recall that**

$$\frac{\vdash\ S_1,\ \ldots\ ,\ \vdash\ S_n}{\vdash\ S}$$

means $\vdash\ S$ **can be deduced from** $\vdash\ S_1,\ \ldots\ ,\ \vdash\ S_n$

- **Using this notation, the rule of precondition strengthening is**

Precondition strengthening

$$\frac{\vdash\ P \Rightarrow P', \qquad \vdash\ \{P'\}\ C\ \{Q\}}{\vdash\ \{P\}\ C\ \{Q\}}$$

- **Note the two hypotheses are different kinds of judgements**

Example

- **From**
  - $\vdash$ `X=n` $\Rightarrow$ `X+1=n+1`
    - **trivial arithmetical fact**
  - $\vdash \{\mathtt{X}+1=\mathtt{n}+1\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$
    - **from earlier slide**
- **It follows by precondition strengthening that**

$$\vdash \{\mathtt{X}=\mathtt{n}\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$$

- **Note that `n` is an *auxiliary* (or *ghost*) variable**

Postcondition weakening

- **Just as the previous rule allows the precondition of a partial correctness specification to be strengthened, the following one allows us to weaken the postcondition**

Postcondition weakening

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \qquad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

An Example Formal Proof

- **Here is a little formal proof**

1.	$\vdash \{\texttt{R=X} \wedge \texttt{0=0}\}\ \texttt{Q:=0}\ \{\texttt{R=X} \wedge \texttt{Q=0}\}$	By the assignment axiom
2.	$\vdash \texttt{R=X} \Rightarrow \texttt{R=X} \wedge \texttt{0=0}$	By pure logic
3.	$\vdash \{\texttt{R=X}\}\ \texttt{Q:=0}\ \{\texttt{R=X} \wedge \texttt{Q=0}\}$	By precondition strengthening
4.	$\vdash \texttt{R=X} \wedge \texttt{Q=0} \Rightarrow \texttt{R=X+(Y} \times \texttt{Q)}$	By laws of arithmetic
5.	$\vdash \{\texttt{R=X}\}\ \texttt{Q:=0}\ \{\texttt{R=X+(Y} \times \texttt{Q)}\}$	By postcondition weakening

- **The rules precondition strengthening and postcondition weakening are sometimes called the** *rules of consequence*

The sequencing rule

- **Syntax:** $C_1\,;\ \cdots\ ;C_n$
- **Semantics: the commands $C_1, \cdots, C_n$ are executed in that order**
- **Example:** `R:=X; X:=Y; Y:=R`
  - **the values of `X` and `Y` are swapped using `R` as a temporary variable**
  - **note *side effect*: value of `R` changed to the old value of `X`**

The sequencing rule

$$\frac{\vdash\ \{P\}\ C_1\ \{Q\},\qquad \vdash\ \{Q\}\ C_2\ \{R\}}{\vdash\ \{P\}\ C_1;C_2\ \{R\}}$$

Example Proof

Example: By the assignment axiom:

(i) $\vdash$ {X=x∧Y=y} R:=X {R=x∧Y=y}

(ii) $\vdash$ {R=x∧Y=y} X:=Y {R=x∧X=y}

(iii) $\vdash$ {R=x∧X=y} Y:=R {Y=x∧X=y}

Hence by (i), (ii) and the sequencing rule

(iv) $\vdash$ {X=x∧Y=y} R:=X; X:=Y {R=x∧X=y}

Hence by (iv) and (iii) and the sequencing rule

(v) $\vdash$ {X=x∧Y=y} R:=X; X:=Y; Y:=R {Y=x∧X=y}

Conditionals

- **Syntax:** `IF` S `THEN` C_1 `ELSE` C_2
- **Semantics:**
  - **if the statement S is true in the current state, then C_1 is executed**
  - **if S is false, then C_2 is executed**
- **Example:** `IF X<Y THEN MAX:=Y ELSE MAX:=X`
  - **the value of the variable `MAX` it set to the maximum of the values of `X` and `Y`**

The Conditional Rule

The conditional rule

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \qquad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

- **From Assignment Axiom + Precondition Strengthening and**

  $\vdash$ (X$\geq$Y $\Rightarrow$ X = max(X,Y)) $\wedge$ ($\neg$(X$\geq$Y) $\Rightarrow$ Y = max(X,Y))

  it follows that

  $\vdash$ {T $\wedge$ X$\geq$Y} MAX:=X {MAX=max(X,Y)}

  and

  $\vdash$ {T $\wedge$ $\neg$(X$\geq$Y)} MAX:=Y {MAX=max(X,Y)}

- **Then by the conditional rule it follows that**

  $\vdash$ {T} IF X$\geq$Y THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}

WHILE-commands

- **Syntax:** `WHILE` S `DO` C
- **Semantics:**
  - if the statement S is true in the current state, then C is executed and the `WHILE`-command is repeated
  - if S is false, then nothing is done
  - thus C is repeatedly executed until the value of S becomes false
  - if S never becomes false, then the execution of the command never terminates
- **Example:** `WHILE` $\neg$`(X=0) DO X:= X-2`
  - if the value of `X` is non-zero, then its value is decreased by `2` and then the process is repeated
- **This `WHILE`-command will terminate (with `X` having value `0`) if the value of `X` is an even non-negative number**
  - in all other states it will not terminate

Invariants

- **Suppose** $\vdash \{P \wedge S\}\ C\ \{P\}$
- P **is said to be an** *invariant* **of** C **whenever** S **holds**
- **The** `WHILE`**-rule says that**
  - **if** P **is an invariant of the body of a** `WHILE`**-command whenever the test condition holds**
  - **then** P **is an invariant of the whole** `WHILE`**-command**
- **In other words**
  - **if executing** C *once* **preserves the truth of** P
  - **then executing** C *any number of times* **also preserves the truth of** P
- **The** `WHILE`**-rule also expresses the fact that after a** `WHILE`**-command has terminated, the test must be false**
  - **otherwise, it wouldn't have terminated**

The WHILE-Rule

The WHILE-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

- **It is easy to show**

  $\vdash$ {X=R+(Y×Q)∧Y≤R} R:=R-Y; Q:=Q+1 {X=R+(Y×Q)}

- **Hence by the WHILE-rule with P = 'X=R+(Y×Q)' and S = 'Y≤R'**

  $\vdash$ {X=R+(Y×Q)}
  WHILE Y≤R DO
  (R:=R-Y; Q:=Q+1)
  {X=R+(Y×Q) ∧ ¬(Y≤R)}

Example

- **From the previous slide**

$$\vdash \{X{=}R{+}(Y{\times}Q)\}$$
$$\quad \texttt{WHILE } Y{\leq}R \texttt{ DO}$$
$$\quad\quad \texttt{(R:=R-Y; Q:=Q+1)}$$
$$\{X{=}R{+}(Y{\times}Q) \wedge \neg(Y{\leq}R)\}$$

- **It is easy to deduce that**

$$\vdash \{T\}\ \texttt{R:=X; Q:=0}\ \{X{=}R{+}(Y{\times}Q)\}$$

- **Hence by the sequencing rule and postcondition weakening**

$$\vdash \{T\}$$
$$\quad \texttt{R:=X;}$$
$$\quad \texttt{Q:=0;}$$
$$\quad \texttt{WHILE } Y{\leq}R \texttt{ DO}$$
$$\quad\quad \texttt{(R:=R-Y; Q:=Q+1)}$$
$$\{R{<}Y \wedge X{=}R{+}(Y{\times}Q)\}$$

How does one find an invariant?

The `WHILE`-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

- **Look at the facts:**
  - **invariant P must hold initially**
  - **with the negated test $\neg S$ the invariant P must establish the result**
  - **when the test S holds, the body must leave the invariant P 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**

Example

- Consider a factorial program

```
{X=n ∧ Y=1}
 WHILE X≠0 DO
   (Y:=Y×X; X:=X-1)
{X=0 ∧ Y=n!}
```

- Look at the facts
  - initially `X=n` and `Y=1`
  - finally `X=0` and `Y=n!`
  - on each loop `Y` is increased and, `X` is decreased
- Think how the loop works
  - `Y` holds the result so far
  - `X!` is what remains to be computed
  - `n!` is the desired result
- The invariant is `X!×Y = n!`
  - 'stuff to be done' × 'result so far' = 'desired result'
  - decrease in `X` combines with increase in `Y` to make invariant

Related example

$\{X{=}0 \wedge Y{=}1\}$
WHILE $X{<}N$ DO $(X{:=}X{+}1;\ Y{:=}Y{\times}X)$
$\{Y{=}N!\}$

- **Look at the Facts**
  - **initially** $X{=}0$ **and** $Y{=}1$
  - **finally** $X{=}N$ **and** $Y{=}N!$
  - **on each iteration both** X **an** Y **increase:** X **by** 1 **and** Y **by** X
- **An invariant is** $Y = X!$
- **At end need** $Y = N!$**, but WHILE-rule only gives** $\neg(X{<}N)$
- **Ah Ha! Invariant needed:** $\boxed{Y = X! \wedge X{\leq}N}$
- **At end** $X \leq N \wedge \neg(X{<}N) \Rightarrow X{=}N$
- **Often need to strenthen invariants to get them to work**
  - **typical to add stuff to 'carry along' like** $X{\leq}N$

Conjunction and Disjunction

Specification conjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Specification disjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

- **These rules are useful for splitting a proof into independent bits**
  - **they enable** $\vdash \{\mathtt{P}\}\ \mathtt{C}\ \{\mathtt{Q}_1 \wedge \mathtt{Q}_2\}$ **to be proved by proving separately that both** $\vdash \{\mathtt{P}\}\ \mathtt{C}\ \{\mathtt{Q}_1\}$ **and also that** $\vdash \{\mathtt{P}\}\ \mathtt{C}\ \{\mathtt{Q}_2\}$
- **Any proof with these rules could be done without using them**
  - **i.e. they are theoretically redundant (proof omitted)**
  - **however, useful in practice**

Combining Multiple Steps

- Proofs involve lots of tedious fiddly small steps
  - similar sequences are used over and over again
- It is tempting to take short cuts and apply several rules at once
  - this increases the chance of making mistakes
- Example:
  - by assignment axiom & precondition strengthening
    - $\vdash \{\mathtt{T}\}\ \mathtt{R} := \mathtt{X}\ \{\mathtt{R} = \mathtt{X}\}$
- Rather than:
  - by the assignment axiom
    - $\vdash \{\mathtt{X} = \mathtt{X}\}\ \mathtt{R} := \mathtt{X}\ \{\mathtt{R} = \mathtt{X}\}$
  - by precondition strengthening with $\vdash\ \mathtt{T} \Rightarrow \mathtt{X{=}X}$
    - $\vdash \{\mathtt{T}\}\ \mathtt{R} := \mathtt{X}\ \{\mathtt{R} = \mathtt{X}\}$

The Derived While Rule

Derived While Rule

$$\frac{\vdash\ P \Rightarrow R \quad \vdash\ \{R\ \wedge\ S\}\ C\ \{R\} \quad \vdash\ R \wedge\ \neg S\ \Rightarrow Q}{\vdash\ \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{Q\}}$$

- **This follows from the While Rule and the rules of consequence**
- **Example: it is easy to show**

$\vdash$ R=X $\wedge$ Q=0 $\Rightarrow$ X=R+(Y$\times$Q)

$\vdash$ {X=R+(Y$\times$Q)$\wedge$Y$\leq$R} R:=R-Y; Q:=Q+1 {X=R+(Y$\times$Q)}

$\vdash$ X=R+(Y$\times$Q)$\wedge\neg$(Y$\leq$R) $\Rightarrow$ X=R+(Y$\times$Q)$\wedge\neg$(Y$\leq$R)

- **Then, by the derived While rule**

```
⊢ {R=X ∧ Q=0}
   WHILE Y≤R DO
     (R:=R-Y; Q:=Q+1)
 {X=R+(Y×Q) ∧ ¬(Y≤R)}
```

The Derived Sequencing Rule

- **The rule below follows from the sequencing and consequence rules**

The Derived Sequencing Rule

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ \{P_1\}\ C_1\ \{Q_1\} & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ \{P_2\}\ C_2\ \{Q_2\} & \vdash\ Q_2 \Rightarrow P_3 \\ \vdots & \vdots \\ \vdash\ \{P_n\}\ C_n\ \{Q_n\} & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ \{P\}\ C_1;\ \ldots\ ;\ C_n\ \{Q\}}$$

- **Exercise: why no derived conditional rule?**

Example

- **By the assignment axiom**

  (i) $\vdash$ {X=x$\wedge$Y=y} R:=X {R=x$\wedge$Y=y}

  (ii) $\vdash$ {R=x$\wedge$Y=y} X:=Y {R=x$\wedge$X=y}

  (iii) $\vdash$ {R=x$\wedge$X=y} Y:=R {Y=x$\wedge$X=y}

- **Using the derived sequencing rule, it can be deduced** *in one step* **from (i), (ii), (iii) and the fact that for any** P**:** $\vdash P \Rightarrow P$

  $\vdash$ {X=x $\wedge$ Y=y} R:=X; X:=Y; Y:=R {Y=x $\wedge$ X=y}

Forwards and backwards proof

- **Previously it was shown how to prove $\{P\}C\{Q\}$ by**
  - **proving properties of the components of C**
  - **and then putting these together, with the appropriate proof rule, to get the desired property of C**
- **For example, to prove $\vdash \{P\}C_1;C_2\{Q\}$**
- **First prove $\vdash \{P\}C_1\{R\}$ and $\vdash \{R\}C_2\{Q\}$**
- **then deduce $\vdash \{P\}C_1;C_2\{Q\}$ by sequencing rule**
- **This method is called *forward proof***
  - **move forward from axioms via rules to conclusion**
- **The problem with forwards proof is that it is not always easy to see what you need to prove to get where you want to be**
- **It is more natural to work backwards**
  - **starting from the goal of showing $\{P\}C\{Q\}$**
  - **generate subgoals until problem solved**

Example

- **Suppose one wants to show**

  $\{X{=}x \wedge Y{=}y\}\ R{:=}X;\ X{:=}Y;\ Y{:=}R\ \{Y{=}x \wedge X{=}y\}$

- **By the assignment axiom and derived sequenced assignment rule it is sufficient to show the subgoal**

  $\{X{=}x \wedge Y{=}y\}\ R{:=}X;\ X{:=}Y\ \{R{=}x \wedge X{=}y\}$

- **Similarly this subgoal can be reduced to**

  $\{X{=}x \wedge Y{=}y\}\ R{:=}X\ \{R{=}x \wedge Y{=}y\}$

- **This clearly follows from the assignment axiom**

Backwards versus Forwards Proof

- **Backwards proof just involves using the rules backwards**
- **Given the rule**

$$\frac{\vdash S_1 \quad \ldots \quad \vdash S_n}{\vdash S}$$

- **Forwards proof says:**
  - **if we have proved** $\vdash S_1 \ \ldots \ \vdash S_n$ **we can deduce** $\vdash S$
- **Backwards proof says:**
  - **to prove** $\vdash S$ **it is sufficient to prove** $\vdash S_1 \ \ldots \ \vdash S_n$
- **Having proved a theorem by backwards proof, it is simple to extract a forwards proof**

Example Backwards Proof

- **To prove**

```
⊢ {T}
   R:=X;
   Q:=0;
   WHILE Y≤R DO
      (R:=R-Y; Q:=Q+1)
 {X=R+(Y×Q) ∧ R<Y}
```

- **By the sequencing rule, it is sufficient to prove**

(i) `⊢ {T} R:=X; Q:=0 {R=X ∧ Q=0}`

(ii)
```
⊢ {R=X ∧ Q=0}
   WHILE Y≤R DO
      (R:=R-Y; Q:=Q+1)
  {X=R+(Y×Q) ∧ R<Y}
```

- **Where does** `{R=X ∧ Q=0}` **come from? (Answer later)**

Example Continued (1)

- **From previous slide:**

  (i) $\vdash \{\texttt{T}\}\ \texttt{R:=X; Q:=0}\ \{\texttt{R=X} \wedge \texttt{Q=0}\}$

- **To prove (i), by the sequenced assignment axiom, we must prove:**

  (iii) $\vdash \{\texttt{T}\}\ \texttt{R:=X}\ \{\texttt{R=X} \wedge \texttt{0=0}\}$

- **To prove (iii), by the derived assignment rule, we must prove:**

  $\vdash \quad \texttt{T} \Rightarrow \texttt{X=X} \wedge \texttt{0=0}$

- **This is true by pure logic**

Example continued (2)

- **From an earlier slide:**

(ii) $\vdash$ $\{R=X \wedge Q=0\}$
WHILE $Y \leq R$ DO
(R:=R−Y; Q:=Q+1)
$\{X=R+(Y \times Q) \wedge R<Y\}$

- **To prove (ii), by the derived while rule, we must prove:**

(iv) $R=X \wedge Q=0 \Rightarrow (X = R+(Y \times Q))$

(v) $X = R+Y \times Q \wedge \neg(Y \leq R) \Rightarrow (X = R+(Y \times Q) \wedge R<Y)$

and

(vi) $\{X = R+(Y \times Q) \wedge (Y \leq R)\}$
(R:=R−Y; Q:=Q+1)
$\{X=R+(Y \times Q)\}$

- **(iv) and (v) are proved by pure arithmetic**

Example Continued (3)

- **To prove (vi), we must prove**

$$\{\texttt{X = R+(Y}\times\texttt{Q)} \wedge \texttt{(Y}\leq\texttt{R)}\}$$
$$\textbf{(vii)} \quad \texttt{(R:=R-Y; Q:=Q+1)}$$
$$\{\texttt{X=R+(Y}\times\texttt{Q)}\}$$

- **To prove (vii), by the sequenced assignment rule, we must prove**

$$\{\texttt{X=R+(Y}\times\texttt{Q)} \wedge \texttt{(Y}\leq\texttt{R)}\}$$
$$\textbf{(viii)} \quad \texttt{R:=R-Y}$$
$$\{\texttt{X=R+(Y}\times\texttt{(Q+1))}\}$$

- **To prove (viii), by the derived assignment rule, we must prove**

$$\textbf{(ix)}\ \texttt{X=R+(Y}\times\texttt{Q)} \wedge \texttt{Y}\leq\texttt{R} \Rightarrow \texttt{(X = (R-Y)+(Y}\times\texttt{(Q+1)))}$$

- **This is true by arithmetic**

- **Exercise: Construct the forwards proof that corresponds to this backwards proof**

Annotate First

- **It is helpful to think up these statements 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**
- **Example, the following program could be annotated at the points P_1 and P_2 indicated by the arrows**

```
{T}
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P1
   WHILE Y≤R DO {X = R+Y×Q} ⟵P2
     (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

Summary

- We have looked at three ways of organizing proofs that make it easier for humans to apply them:
  - deriving "bigger step" rules
  - backwards proof
  - annotating programs
- Next we see how these techniques can be used to mechanize program verification

NEW TOPIC: Mechanizing Program Verification

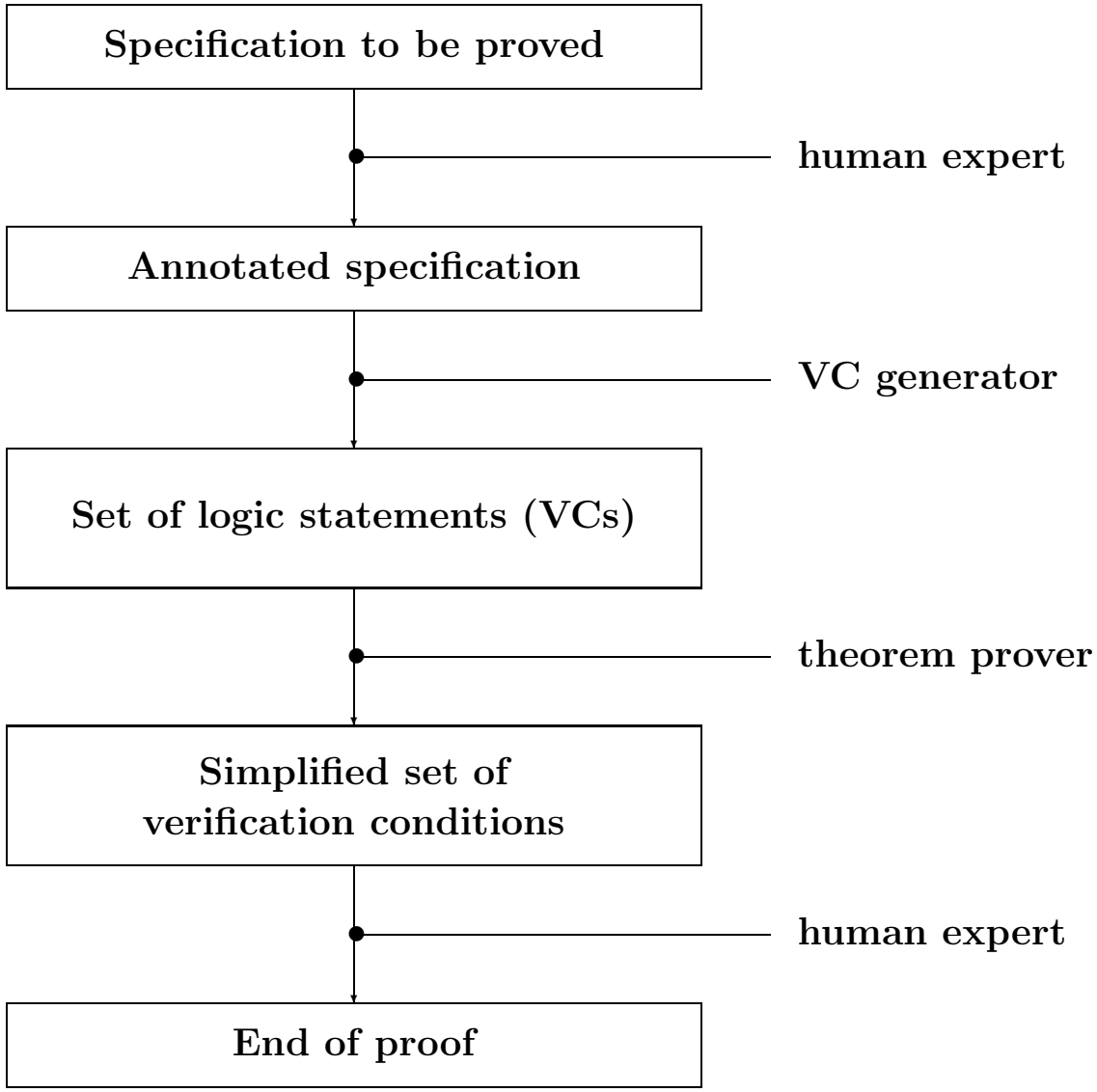
- The architecture of a simple program verifier will be described
- Justified with respect to the rules of Floyd-Hoare logic
- It is clear that
  - proofs are long and boring, even if the program being verified is quite simple
  - lots of fiddly little details to get right, many of which are trivial, e.g.

$$\vdash \ (\mathtt{R}{=}\mathtt{X} \ \wedge \ \mathtt{Q}{=}\mathtt{0}) \ \Rightarrow \ (\mathtt{X} = \mathtt{R} + \mathtt{Y}{\times}\mathtt{Q})$$

Mechanization

- **Goal:** automate the routine bits of proofs in Floyd-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 statement
- 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 in the course
  - ideas very old (JC King's 1969 CMU PhD, Stanford verifier in 1970s)
  - used by program verifiers (e.g. Gypsy and SPARK verifier)
  - provides a verification front end to different provers (see *Why* system)

Architecture of a Verifier



Commentary

- **Input: a Hoare triple annotated with mathematical statements**
  - **these annotations describe relationships between variables**
- **The system generates a set of purely mathematical statements called** *verification conditions* **(or 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

- **The three steps in proving $\{P\}C\{Q\}$ with a verifier**
- **1 The program C is *annotated* by inserting statements (*assertions*) expressing conditions 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 statements 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. more 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 Verification Conditions

- **It will be shown that**
  - **if one can prove all the verification conditions generated from $\{P\}C\{Q\}$**
  - **then $\vdash \{P\}C\{Q\}$**
- **Step $\boxed{2}$ converts a verification problem into a conventional mathematical problem**
- **The process will be illustrated with:**

```
{T}
   R:=X;
   Q:=0;
   WHILE Y≤R DO
      (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

Annotation of Commands

- **An annotated command is a command with statements (*assertions*) embedded within it**

- **A command is *properly annotated* if statements have been inserted at the following places**

  (i) before C_2 in $C_1;C_2$ if C_2 is *not* an assignment command

  (ii) 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 (see later)**

Annotation of Specifications

- **A properly annotated specification is a specification $\{P\}C\{Q\}$ where C is a properly annotated command**

- **Example: To be properly annotated, assertions should be at points ① and ② of the specification below**

```
{X=n}
   Y:=1; ⟵①
   WHILE X≠0 DO ⟵②
     (Y:=Y×X; X:=X-1)
{X=0 ∧ Y=n!}
```

- **Suitable statements would be**

at ①: {Y = 1 ∧ X = n}
at ②: {Y×X! = n!}

Verification Condition Generation

- **The VCs generated from an annotated specification $\{P\}C\{Q\}$ are obtained by considering the various possibilities for C**
- **We will describe it command by command using rules of the form:**
- **The VCs for $C(C_1, C_2)$ are**
  - vc_1, ... , vc_n
  - **together with the VCs for C_1 and those for C_2**
- **Each VC rule corresponds to either a primitive or derived rule**