Methodologies for Software Processes

Lecture 5- Introduction to Program Verification

(The lecture slides and the notes are taken from Prof. Mike Gordon from Cambridge University)

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
-

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?
-

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 CS Distinguished Dissertation awards concerned separation logic

A Little 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:

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

Commands:

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

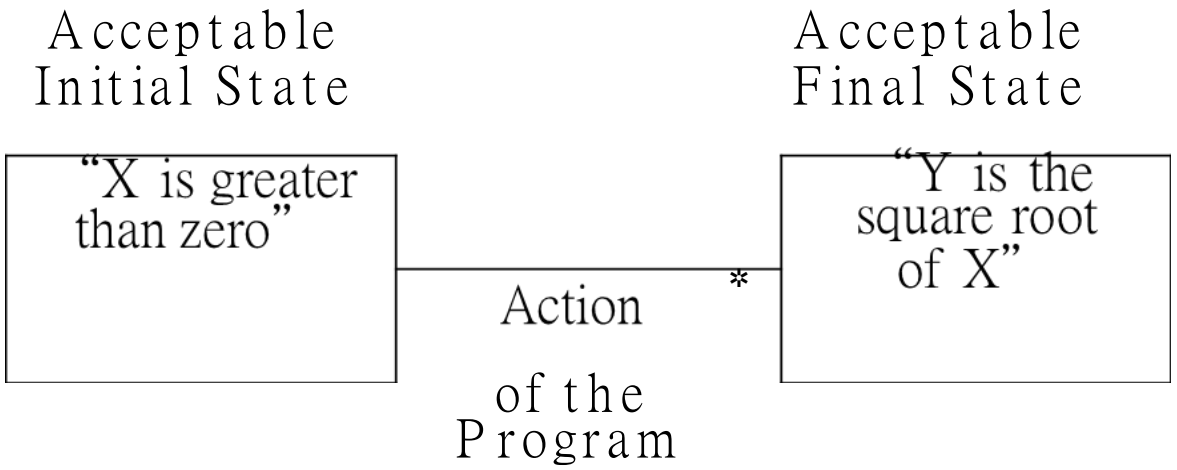
Some Notation

- 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: $\{X = 1\}\ X:=X+1\ \{X = 2\}$
  - 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'
- $\{X = 1\}\ X:=X+1\ \{X = 2\}$ is true
- $\{X = 1\}\ X:=X+1\ \{X = 3\}$ is false

Formal versus Informal Proof

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

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

Hoare's Verification Grand Challenge

① 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."

① 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.

① "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

Partial Correctness Specification

- An expression $\{P\}\ C\ \{Q\}$ is called a *partial correctness specification*
  - P is called its *precondition*
  - Q its *postcondition*
- $\{P\}\ C\ \{Q\}$ 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's execution terminates satisfies Q
- These specifications are '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
- `{X = 1} WHILE T DO X := X {Y = 2}` – this specification is true!

Total Correctness Specification

- A stronger kind of specification is a *total correctness specification*
  - there is no standard notation for such specifications
  - we shall use [*P*] *C* [*Q*]
- A total correctness specification [*P*] *C* [*Q*] is true if and only if
  - whenever *C* is executed in a state satisfying *P* the execution of *C* terminates
  - after *C* terminates *Q* holds
- [X = 1] Y:=X; WHILE T DO X := X [Y = 1]
  - this says that the execution of Y:=X;WHILE T DO X:=X terminates when started in a state satisfying X = 1
  - after which Y = 1 will hold
  - this is clearly false

Total Correctness

- Informally:

  Total correctness = Termination + Partial correctness

- Total correctness is the ultimate goal
  - 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 DIV 2
```

  - X DIV 2 evaluates to the result of rounding down X/2 to a whole number
  - the Collatz conjecture is that this terminates with X=1

-

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

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

Some Easy Exercises

- When is $[\mathsf{T}]\ C\ [\mathsf{T}]$ true?

- Write a partial correctness specification which is true if and only if the command C has the effect of multiplying the values of X and Y and storing the result in X

- Write a specification which is true if the execution of C always halts when execution is started in a state satisfying P

Specification can be Tricky

- "The program must set Y to the maximum of X and Y"
  - [T] C [Y = 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+1)`, `(x×y)+Z`,
`X!`, `sin(x)`, `rem(X,Y)`

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

Atomic Statements

① Examples of atomic statements are

$$T, \quad F, \quad X = 1, \quad R < Y, \quad X = R+(Y \times Q)$$

① T and F are atomic statements that are always true and false

① Other atomic statements are built from terms using *predicates*, e.g.

$$ODD(X), \quad PRIME(3), \quad X = 1, \quad (X+1)^2 \geq x^2$$

① ODD and PRIME are examples of predicates

① = and ≥ are examples of *infixed* predicates

① X, 1, 3, X+1, $(X+1)^2$, x^2 are terms in above atomic statements

Compound statements

- Compound statements are built up from atomic statements using:

  $\neg$ (not)
  $\wedge$ (and)
  $\vee$ (or)
  $\Rightarrow$ (implies)
  $\Leftrightarrow$ (if and only if)

  - 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. $(1 < 0) \Rightarrow (2 + 2 = 3)$
- $P \Leftrightarrow Q$ 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

$\neg P \wedge Q$	is equivalent to	$(\neg P) \wedge Q$
$P \wedge Q \Rightarrow R$	is equivalent to	$(P \wedge Q) \Rightarrow R$
$P \wedge Q \Leftrightarrow \neg R \vee S$	is equivalent to	$(P \wedge Q) \Leftrightarrow ((\neg R) \vee S)$

Universal quantification

① If S is a statement and x a variable

① Then $\forall x.\ S$ means:

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

① The statement

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

abbreviates

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

① 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

① 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

① If S is a statement and x a variable

① Then $\exists x.\ S$ means

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

① 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* $\vdash S_1, \ldots, \vdash S_n$
  - 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

① 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

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

Commands

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

Judgements

- Three kinds of things that could be true or false:
  - statements of mathematics, e.g. $(X + 1)^2 = X^2 + 2 \times X + 1$
  - partial correctness specifications $\{P\}\ C\ \{Q\}$
  - total correctness specifications $[P]\ C\ [Q]$
- These three kinds of things are examples of *judgements*
  - a logical system gives rules for proving judgements
  - Floyd-Hoare logic provides rules for proving partial correctness specifications
  - the laws of arithmetic provide ways of proving statements about integers
- $\vdash S$ means statement S can be proved
  - how to prove predicate calculus statements assumed known
  - this course covers axioms and rules for proving *program correctness statements*

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

Substitution Notation

- $Q[E/V]$ is the result of replacing all occurrences of V in Q by E
  - read $Q[E/V]$ as 'Q with E for V'
  - for example: `(X+1 > X)[Y+Z/X]` = `((Y+Z)+1 > Y+Z)`
  - ignoring issues with bound variables for now (e.g. variable capture)
- Same notation for substituting into terms, e.g. $E_1[E_2/V]$
- Think of this notation as the 'cancellation law'

$$V[E/V] = E$$

  which is analogous to the cancellation property of fractions

$$v \times (e/v) = e$$

- Note that $Q[x/V]$ doesn't contain V (if $V \neq x$)

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
  - $\vdash \{E = x\}\ V := E\ \{V = x\}$
  - $\vdash \{Y = 2\}\ X := 2\ \{Y = X\}$
  - $\vdash \{X + 1 = n + 1\}\ X := X + 1\ \{X = n + 1\}$
  - $\vdash \{E = E\}\ X := E\ \{X = E\}$ (if X does not occur in E)

The Backwards Fallacy

- Many people feel the assignment axiom is 'backwards'
- One common erroneous intuition is that it should be

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

  - where $P\ [V/E]$ denotes the result of substituting V for E in P
  - this has the false consequence $\vdash \{X=0\}\ X:=1\ \{X=0\}$
    (since $(X=0)\ [X/1]$ is equal to $(X=0)$ as 1 doesn't occur in $(X=0)$)

- Another erroneous intuition is that it should be

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

  - this has the false consequence $\vdash \{X=0\}\ X:=1\ \{1=0\}$
    (which follows by taking P to be $X=0$, V to be X and E to be 1)

A Forwards Assignment Axiom (Floyd)

- This is the original semantics of assignment due to Floyd

$$\vdash \{P\}\ V := E\ \{\exists v.\ V = E[v/V] \wedge P[v/V]\}$$

  - where v is a new variable (i.e. doesn't equal V or occur in P or E)

- Example instance

$$\vdash \{X=1\}\ X := X+1\ \{\exists v.\ X = X+1[v/X] \wedge X=1[v/X]\}$$

- Simplifying the postcondition

$$\vdash \{X=1\}\ X := X+1\ \{\exists v.\ X = X+1[v/X] \wedge X=1[v/X]\}$$
$$\vdash \{X=1\}\ X := X+1\ \{\exists v.\ X = v + 1 \wedge v = 1\}$$
$$\vdash \{X=1\}\ X := X+1\ \{\exists v.\ X = 1 + 1 \wedge v = 1\}$$
$$\vdash \{X=1\}\ X := X+1\ \{X = 1 + 1 \wedge \exists v.\ v = 1\}$$
$$\vdash \{X=1\}\ X := X+1\ \{X = 2 \wedge T\}$$
$$\vdash \{X=1\}\ X := X+1\ \{X = 2\}$$

- Forwards Axiom equivalent to standard one but harder to use

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

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

- Note the two hypotheses are different kinds of judgements

Example

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

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

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

Postcondition weakening

① 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

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

Validity

- Important to establish the validity of axioms and rules
- Later will give a *formal semantics* of our little programming language
  - then *prove* axioms and rules of inference of Floyd-Hoare logic are sound
  - this will only increase our confidence in the axioms and rules to the extent that we believe the correctness of the formal semantics!
- The Assignment Axiom is not valid for 'real' programming languages

Expressions with Side-effects

- The validity of the assignment axiom depends on expressions not having side effects

- Suppose that our language were extended so that it contained the 'block expression'

$$\texttt{BEGIN Y:=1; 2 END}$$

  - this expression has value 2, but its evaluation also 'side effects' the variable Y by storing 1 in it

- If the assignment axiom applied to block expressions, then it could be used to deduce

$$\vdash \{Y=0\}\ X:=\texttt{BEGIN Y:=1; 2 END}\ \{Y=0\}$$

  - since $(Y=0)[E/X] = (Y=0)$ (because X does not occur in $(Y=0)$)

  - this is clearly false; after the assignment Y will have the value 1

An Example Formal Proof

① Here is a little formal proof

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

① The rules precondition strengthening and postcondition weakening are sometimes called the *rules of consequence*

The sequencing rule

- Syntax: C_1; · · · ;C_n
- Semantics: the commands C_1, · · · , 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) $\vdash \{X{=}x \wedge Y{=}y\}\ R{:=}X\ \{R{=}x \wedge Y{=}y\}$

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

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

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

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

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

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

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

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

① 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

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

① Then by the conditional rule it follows that

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

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 ¬(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

① From the previous slide

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

① It is easy to deduce that

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

① Hence by the sequencing rule and postcondition weakening

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

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 ways of finding proofs and organising them:
  - finding invariants
  - derived rules
  - backwards proofs
  - annotating programs prior to proof
- Then we see how to automate program verification
  - the automation mechanises some of these ideas