WARNING

These are preliminary slides and may differ from what is actually presented in the lectures.

Additional material may be added and topics in these slides may not be covered.

Introduction

Specification and Verification I

- This course is on the specification and verification of software
- It is a prerequisite for the course entitled:
  - *Specification and Verification II*
  - which concerns hardware
- The two courses are really a single course

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) – google if curious

Syllabus

- **Program specification:**
  - partial and total correctness
- **Hoare notation**
- **Axioms and rules of Floyd-Hoare Logic**
  - discussion of soundness and completeness
- **Mechanised program verification**
  - verification conditions
- **Program refinement**
- **Higher order logic**
- **Semantic embedding in higher order logic**
- **Related notions**
  - weakest preconditions and strongest postconditions
  - VDM
  - dynamic 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:

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

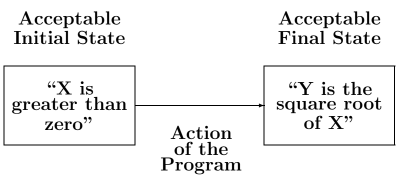
Commands:

$C ::= \mathtt{SKIP}$
$\quad \mid \ V \ \mathtt{:=} \ E$
$\quad \mid \ V(E_1) \ \mathtt{:=} \ E_2$
$\quad \mid \ C_1 \ \mathtt{;} \ C_2$
$\quad \mid \ \mathtt{IF} \ B \ \mathtt{THEN} \ C$ **(redundant)**
$\quad \mid \ \mathtt{IF} \ B \ \mathtt{THEN} \ C_1 \ \mathtt{ELSE} \ C_2$
$\quad \mid \ \mathtt{BEGIN\ VAR} \ V_1 \ \mathtt{;} \ \mathtt{..} \ \mathtt{VAR} \ V_n \ \mathtt{;} \ C \ \mathtt{END}$
$\quad \mid \ \mathtt{WHILE} \ B \ \mathtt{DO} \ C$
$\quad \mid \ \mathtt{FOR} \ V \ \mathtt{:=} \ E_1 \ \mathtt{UNTIL} \ E_2 \ \mathtt{DO} \ C$

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 as some writers use this word for commands
  - e.g. '`FOR`-statement' wrong here, use '`FOR`-command'

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

Specifications as the Basis of Contracts

- **A formal specification can be used as the basis for requirements analysis and procurement**
  - **is this what the customer intended?**
  - **see link** *Design by Contract* **on course web page**
- **"I want a program that swaps the values in X and Y"**
- **$\{\mathtt{X=x} \wedge \mathtt{Y=y}\}\ C\ \{\mathtt{X=y} \wedge \mathtt{Y=x}\}$**
- **The command**

  ```
  BEGIN R:=X; X:=Y; Y:=R END
  ```

  would fulfil the specification and so the contract
- **The command `BEGIN X:=Y; Y:=X END` would not**
- **How do we determine whether a program fulfils the contract?**

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 $(\mathtt{X}+1)^2 = \mathtt{X}^2 + 2 \times \mathtt{X} + 1$ 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
- 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."**

Exercise: debug the proof of "1=2" below

1.	$1/1 = 1/1$	Reflexivity of equality
2.	$-(1/1) = -(1/1)$	Negate both sides
3.	$-1/1 = 1/-1$	Use $-(n/m) = (-n)/m = n/(-m)$
4.	$\sqrt{-1/1} = \sqrt{1/-1}$	Square root of both sides
5.	$\sqrt{-1}/\sqrt{1} = \sqrt{1}/\sqrt{-1}$	Use $\sqrt{m/n} = \sqrt{m}/\sqrt{n}$ (since $(m/n)^2 = m^2/n^2$)
6.	$i/1 = 1/i$	As $i = \sqrt{-1}$ and $1 = \sqrt{1}$
7.	$i/2 = 1/2i$	Halving both sides
8.	$i/2 + 3/2i = 1/2i + 3/2i$	Adding same thing to both sides
9.	$i^2/2 + 3i/2i = i/2i + 3i/2i$	Multiplying both sides by i
10.	$-1/2 + 3/2 = 1/2 + 3/2$	Using $i^2 = -1$ and cancelling
11.	$1 = 2$	Adding the fractions

Formal Specification

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
- $\{\mathtt{X} = 1\}$ `WHILE T DO SKIP` $\{\mathtt{Y} = 2\}$ – **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
- $[\mathtt{X} = 1]$ `Y:=X; WHILE T DO SKIP` $[\mathtt{Y} = 1]$
  - this says that the execution of `Y:=X;WHILE T DO SKIP` terminates when started in a state satisfying $\mathtt{X} = 1$
  - after which $\mathtt{Y} = 1$ 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

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

A More Complicated Example

-
  ```
  {T}
   BEGIN                              ⎫
     R:=X;                            ⎪
     Q:=0;                            ⎬ C
     WHILE Y≤R DO                     ⎪
       BEGIN R:=R-Y; Q:=Q+1 END       ⎪
   END                                ⎭
  {R < Y ∧ X = R + (Y × Q)}
  ```
- **This is** $\{T\}\ C\ \{R < Y \ \wedge\ X = R + (Y \times Q)\}$
  - 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 [T] C [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 SKIP`
- **Another?**
  - `IF X >= Y THEN X := Y ELSE SKIP`
- **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**

  ⊢ {T} C {Y=max(X,Y)}

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

  ⊢ {X=x ∧ 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!**

Sorting

- **Suppose $\texttt{C}_{sort}$ is a command that is intended to sort the first N elements of an array**
- **To specify this formally, let $\texttt{SORTED}(A, N)$ mean**

$$A(1) \leq A(2) \leq \ldots \leq A(N)$$

- **A first attempt to specify that $\texttt{C}_{sort}$ sorts is**

$$\{1 \leq \texttt{N}\}\ \texttt{C}_{sort}\ \{\texttt{SORTED(A,N)}\}$$

- **Not enough:**
  - **`SORTED(A,N)` can be achieved by simply zeroing the first `N` elements of `A`**

Permutation Required

- It is necessary to require that the sorted array is a rearrangement, or permutation, of the original array

- To formalise this, let `PERM`(A, A', N) mean that

  $A(1), A(2), \ldots, A(N)$

  is a rearrangement of

  $A'(1), A'(2), \ldots, A'(N)$

- An improved specification that $\texttt{C}_{sort}$ sorts:

  $\{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a}\}\ \texttt{C}_{sort}\ \{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)}\}$

Still Not Correct ...

- The following specification is true

  ```
  {1≤N}
   N:=1
  {SORTED(A,N) ∧ PERM(A,a,N)}
  ```

- Must say explicitly that `N` is unchanged

- A better specification is thus:

  $\{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a} \wedge \texttt{N=n}\}$
  $\texttt{C}_{sort}$
  $\{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)} \wedge \texttt{N=n}\}$

- Is this the correct specification?

Summary

- We have given a notation for specifying
  - the partial correctness of programs
  - the total correctness of programs
- Specifications can still be incorrect
  - however, their meaning is precise
- We next describe the predicate calculus:
  - the precise language upon which they are based

Review of Predicate Calculus

- Program states are specified with *first-order logic* (FOL)
- Knowledge of this is assumed (brief review given now)
- Later *higher-order logic* (HOL) will be:
  - used to *semantically embed* Floyd-Hoare logic
  - used for hardware specification
- 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
- In higher-order logic statements are a special case of terms
  - because truth-values are values

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:**

$$\begin{array}{c} \texttt{X}, \quad \texttt{y}, \quad \texttt{Z}, \\ \texttt{1}, \quad \texttt{2}, \quad \texttt{325}, \\ \texttt{-X}, \quad \texttt{-(X+1)}, \quad \texttt{(x}\times\texttt{y)+Z}, \\ \sqrt{\texttt{(1+x}^2\texttt{)}}, \quad \texttt{X!}, \quad \texttt{sin(x)}, \quad \texttt{rem(X,Y)} \end{array}$$

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

Atomic Statements

- **Examples of atomic statements are**

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

- **`T` and `F` are atomic statements that are always true and false**
- **Other atomic statements are built from terms using *predicates*, e.g.**

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

- **`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:**

$\neg$	(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. $(1 < 0) \Rightarrow (2 + 2 = 3)$
  - 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 $P \Leftrightarrow Q$**
  - $P = Q$ used in higher-order logic
  - $P \equiv Q$ used for definitions
  - also $P =_{def} Q$ used for definitions

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 $\forall n.\ n+0 = n$

Example

- Let us abbreviate

$$x_1 \leq x_2\ \wedge\ x_2 \leq x_3\ \wedge\ \ \ldots\ \ \wedge\ x_{n-1} \leq x_n$$

  by

$$x_1 \leq x_2 \leq x_3 \leq\ \ldots\ \leq x_{n-1} \leq x_n$$

- Recall that $\texttt{SORTED}(A, N)$ means that

$$A(1) \leq A(2) \leq \ldots \leq A(N)$$

- The definition of $\texttt{SORTED}$ can now be given formally

$$\begin{array}{l}\texttt{SORTED}(A, N)\ \equiv \\ \quad \forall I\ J.\ 1 \leq I \leq J \leq N\ \Rightarrow\ A(I) \leq A(J)\end{array}$$

Comment on Definition of SORTED

- The symbol $\equiv$ is synonymous with $\Leftrightarrow$ and means 'if and only if'
  - it is used to introduce the definitions of constants
- Notice that A and N are unbound variables in

$$\begin{array}{l}\texttt{SORTED}(A, N)\ \equiv \\ \quad \forall I\ J.\ 1 \leq I \leq J \leq N\ \Rightarrow\ A(I) \leq A(J)\end{array}$$

- Unbound variables are regarded as implicitly universally quantified
- The definition of $\texttt{SORTED}$ could thus have been written as

$$\begin{array}{l}\forall A\ N.\ \texttt{SORTED}(A, N)\ \equiv \\ \qquad \forall I\ J.\ 1 \leq I \leq J \leq N\ \Rightarrow\ A(I) \leq A(J)\end{array}$$

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

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

$PERM(A, a, N)$

- $PERM(A, a, N)$ easy to define if one can quantify over functions

```
PERM(A,a,N) =
 ∃f.  (∀n. n ≤ N ⇒ f(n) ≤ N) ∧
      (∀m n. m ≤ N ∧ n ≤ N ∧ f(m) = f(n)  ⇒  m = n) ∧
      (∀n. n ≤ N ⇒ a(n) = A(f(n)))
```

- Exercise: specify $PERM(A, a, N)$ in first-order logic

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

Invalid 'Proofs'

- It is quite easy to come up with plausible rules for reasoning about programs that are actually unsound
  - some examples for `FOR`-commands are given later
- Proofs of correctness of computer programs are often very intricate and formal methods are needed to ensure that they are valid
- It is thus important to make fully explicit the reasoning principles being used, so that their soundness can be analysed
- Soundness and completeness will be discussed informally later
- Here is a bogus proof

1.	$\sqrt{-1\times-1}$	$=\sqrt{-1\times-1}$	Reflexivity of $=$.
2.	$\sqrt{-1\times-1}$	$=(\sqrt{-1})\times(\sqrt{-1})$	Distributive law of $\sqrt{\ }$ over $\times$.
3.	$\sqrt{-1\times-1}$	$=(\sqrt{-1})^2$	Definition of $()^2$.
4.	$\sqrt{-1\times-1}$	$=-1$	definition of $\sqrt{\ }$.
5.	$\sqrt{1}$	$=-1$	As $-1\times-1=1$.
6.	1	$=-1$	As $\sqrt{1}=1$.

- See link *Some classic fallacies* on course web page

Proof Rules For Partial Correctness

Overview

- The proof rules that follow constitute an *axiomatic semantics* of our programming language (more on "axiomatic semantics" later)

E ::= N | V | $E_1 + E_2$ | $E_1 - E_2$ | $E_1 \times E_2$ | ...

B ::= `T` | `F` | E_1=E_2 | $E_1 \leq E_2$ | ...

```
C ::= SKIP
    | V := E
    | V(E1) := E2
    | C1 ; C2
    | IF B THEN C                (redundant)
    | IF B THEN C1 ELSE C2
    | BEGIN VAR V1 ; .. VAR V1 ; C END
    | WHILE B DO C
    | FOR V := E1 UNTIL E2 DO C
```

- Arrays and `FOR`-commands are tricky and discussed in a later lecture
- One-armed conditionals will be used to illustrate derived semantics

Judgements

- Three kinds of things that could be true or false:
  - statements of mathematics, e.g. $(\texttt{X}+1)^2 = \texttt{X}^2 + 2 \times \texttt{X} + 1$
  - 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 $\texttt{X}+1$, $\sqrt{2}$ etc. which denote values (usually numbers)
- Symbols S, S_1, ... , S_n stand for arbitrary statements
  - these are conditions like $\texttt{X} < \texttt{Y}$, $\texttt{X}^2 = 1$ 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

Skip

- Syntax: `SKIP`
- Semantics: the state is unchanged

The Skip Axiom

$$\vdash\ \{P\}\ \texttt{SKIP}\ \{P\}$$

- It is an axiom schema
  - P can be instantiated with arbitrary predicate calculus formulae (statements)
- Instances of the skip axiom are:
  - $\vdash\ \{\texttt{Y} = 2\}\ \texttt{SKIP}\ \{\texttt{Y} = 2\}$
  - $\vdash\ \{\texttt{T}\}\ \texttt{SKIP}\ \{\texttt{T}\}$
  - $\vdash\ \{\texttt{R=X+(Y}\times\texttt{Q)}\}\ \texttt{SKIP}\ \{\texttt{R=X+(Y}\times\texttt{Q)}\}$

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: $(\mathtt{X}{+}1 > \mathtt{X})[\mathtt{Y}{+}\mathtt{Z}/\mathtt{X}] \;=\; ((\mathtt{Y}{+}\mathtt{Z}){+}1 > \mathtt{Y}{+}\mathtt{Z})$
  - 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

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

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)

The Backwards Fallacy

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

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

  - 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

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

  - 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

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

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

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

- Simplifying the postcondition

$$\begin{array}{l}
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\exists v.\ \mathtt{X} = \mathtt{X+1}[v/\mathtt{X}]\ \wedge\ \mathtt{X=1}[v/\mathtt{X}]\} \\
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\exists v.\ \mathtt{X} = v + \mathtt{1}\ \wedge\ v = \mathtt{1}\} \\
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\exists v.\ \mathtt{X} = \mathtt{1} + \mathtt{1}\ \wedge\ v = \mathtt{1}\} \\
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\mathtt{X} = \mathtt{1} + \mathtt{1}\ \wedge\ \exists v.\ v = \mathtt{1}\} \\
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\mathtt{X} = \mathtt{2}\ \wedge\ \mathtt{T}\} \\
\vdash\ \{\mathtt{X=1}\}\ \mathtt{X:=X+1}\ \{\mathtt{X} = \mathtt{2}\}
\end{array}$$

- Forwards Axiom equivalent to standard one but harder to use

Precondition Strengthening

- Recall that

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

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

- 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
  - $\vdash\ \{\mathtt{X}+\mathtt{1}=\mathtt{n}+\mathtt{1}\}\ \mathtt{X}:=\mathtt{X}+\mathtt{1}\ \{\mathtt{X}=\mathtt{n}+\mathtt{1}\}$
    - from earlier slide
- It follows by precondition strengthening that

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

- 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

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

Validity

- Important to establish the validity of axioms and rules
- Can 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!
- Won't go into this topic here (some discussion later)
- The Assignment Axiom is not valid for 'real' programming languages
  - In an early PhD on Hoare Logic G. Ligler showed that the assignment axiom can fail to hold in six different ways for the language Algol 60

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'**

  `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:=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

- **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

- **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}`

Blocks

- Syntax: `BEGIN VAR` V_1`;` $\cdots$ `VAR` V_n`;` C `END`
- Semantics: command C is executed, then the values of $V_1, \cdots, V_n$ are restored to the values they had before the block was entered
  - the initial values of $V_1, \cdots, V_n$ inside the block are unspecified
- Example: `BEGIN VAR R; R:=X; X:=Y; Y:=R END`
  - the values of `X` and `Y` are swapped using `R` as a temporary variable
  - this command does *not* have a side effect on the variable `R`

The Block Rule

- The block rule takes care of local variables

The block rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P\}\ \texttt{BEGIN VAR}\ V_1;\ \ldots;\ \texttt{VAR}\ V_n;\ C\ \texttt{END}\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

- Note that the block rule is regarded as including the case when there are no local variables (the '$n = 0$' case)

The Side Condition

- The syntactic condition that none of the variables $V_1, \ldots, V_n$ occur in P or Q is an example of a *side condition*
- From
  ⊢ `{X=x ∧ Y=y} R:=X; X:=Y; Y:=R {Y=x ∧ X=y}`
  it follows by the block rule that
  ⊢ `{X=x ∧ Y=y}  BEGIN VAR R; R:=X; X:=Y; Y:=R END   {Y=x ∧ X=y}`
  since `R` does not occur in `X=x ∧ Y=y` or `X=y ∧ Y=x`
- However from
  ⊢ `{X=x ∧ Y=y} R:=X; X:=Y {R=x ∧ X=y}`
  one *cannot* deduce
  ⊢ `{X=x ∧ Y=y}  BEGIN VAR R; R:=X; X:=Y END  {R=x ∧ X=y}`
  since `R` occurs in `R=x ∧ X=y`

Exercises

- Consider the specification

  `{X=x} BEGIN VAR X; X:=1 END {X=x}`

  Can this be deduced from the rules given so far?
  (i) if so, give a proof of it
  (ii) if not, explain why not and suggest additional rules and/or axioms to enable it to be deduced
- Is the following true?

  ⊢ `{X=x ∧ Y=y} X:=X+Y; Y:=X-Y; X:=X-Y {Y=x ∧ X=y}`

  - if so prove it
  - if not, give the circumstances when it fails
- Show

  ⊢ `{X=R+(Y×Q)} BEGIN R:=R-Y; Q:=Q+1 END {X=R+(Y×Q)}`

Two-armed 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`
- **One-armed conditional is defined by:**
  - `IF` S `THEN` C $\;=_{define}\;$ `IF` S `THEN` C `ELSE SKIP`

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≥Y ⇒ X = max(X,Y)) ∧ (¬(X≥Y) ⇒ Y = max(X,Y))`

  it follows that

  $\vdash$ `{T ∧ X≥Y} MAX:=X {MAX=max(X,Y)}`

  and

  $\vdash$ `{T ∧ ¬(X≥Y)} MAX:=Y {MAX=max(X,Y)}`
- **Then by the conditional rule it follows that**

  $\vdash$ `{T} IF X≥Y THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}`
- **A one-armed conditional rule is derived later**

`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**

```
⊢ {X=R+(Y×Q)∧Y≤R} BEGIN R:=R-Y; Q:=Q+1 END {X=R+(Y×Q)}
```

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

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

Example

- **From the previous slide**

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

- **It is easy to deduce that**

```
⊢ {T} R:=X; Q:=0 {X=R+(Y×Q)}
```

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

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

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 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
   BEGIN Y:=Y×X; X:=X-1 END
{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 ∧ Y=1}
 WHILE X<N DO
   BEGIN X:=X+1; Y:=Y×X END
{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 ¬(`X<N`)
- Ah Ha! Invariant needed: $\boxed{\texttt{Y = X!} \wedge \texttt{X}\leq\texttt{N}}$
- At end $\texttt{X} \leq \texttt{N} \wedge \neg(\texttt{X<N}) \Rightarrow \texttt{X=N}$
- Often need to strenthen invariants to get them to work
  - typical to add stuff to 'carry along' like `X≤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 $\vdash$ {P} C {$Q_1 \wedge Q_2$} to be proved by proving separately that both $\vdash$ {P} C {Q_1} and also that $\vdash$ {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

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
    - $\vdash$ {T} R := X {R = X}
- Rather than:
  - by the assignment axiom
    - $\vdash$ {X = X} R := X {R = X}
  - by precondition strengthening with $\vdash$ T ⇒ X=X
    - $\vdash$ {T} R := X {R = X}

Derived rules for finding proofs

- Suppose the goal is to prove $\{\mathit{Precondition}\}\ \mathit{Command}\ \{\mathit{Postcondition}\}$
- If there were a rule of the form

$$\frac{\vdash\ H_1,\ \cdots,\ \vdash\ H_n}{\vdash\ \{P\}\ C\ \{Q\}}$$

then we could instantiate
$P \mapsto \mathit{Precondition},\ C \mapsto \mathit{Command},\ Q \mapsto \mathit{Postcondition}$
to get instances of $H_1,\ \cdots, H_n$ as subgoals

- Some of the rules are already in this form e.g. the sequencing rule
- We will derive rules of this form for all commands
- Then we use these derived rules for mechanising Hoare Logic proofs

Derived Rules

- We will establish derived rules for all commands

$$\frac{\ldots}{\vdash\ \{P\}\ \texttt{SKIP}\ \{Q\}}$$

$$\frac{\ldots}{\vdash\ \{P\}\ V\texttt{:=}E\ \{Q\}}$$

$$\frac{\ldots}{\vdash\ \{P\}\ C_1\texttt{;}C_2\ \{Q\}}$$

$$\frac{\ldots}{\vdash\ \{P\}\ \texttt{BEGIN VAR}\ V_1\texttt{;}\ \ldots\texttt{;}\ \texttt{VAR}\ V_n\texttt{;}\ C\ \texttt{END}\ \{Q\}}$$

$$\frac{\ldots}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

$$\frac{\ldots}{\vdash\ \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{Q\}}$$

- These support 'backwards proof' starting from a goal $\{P\}\ C\ \{Q\}$

The Derived Skip Rule

Derived Skip Rule

$$\frac{\vdash\ P \Rightarrow Q}{\vdash\ \{P\}\ \texttt{SKIP}\ \{Q\}}$$

- Justifying proof schema:

1. $\vdash\ P\ \Rightarrow\ Q$ By assumption.
2. $\vdash\ \{Q\}\ \texttt{SKIP}\ \{Q\}$ By the skip axiom.
3. $\vdash\ \{P\}\ \texttt{SKIP}\ \{Q\}$ By precondition strengthening with 1 and 2.

- This schematic proof shows that any inference with the Derived Skip Rule can be replaced by three inferences using primitive rules
- Derived rules are 'proof subroutines'
  - the schema metavariables (P, Q above) are the parameters of the subroutine
  - the derivation schema is the body of the subroutine

The Derived Assignment Rule

- An example 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.

- Can generalise this proof to a proof schema:

1. $\vdash\ \{Q[E/V]\}\ V\texttt{:=}E\ \{Q\}$ By the assignment axiom.
2. $\vdash\ P\ \Rightarrow\ Q[E/V]$ By assumption.
3. $\vdash\ \{P\}\ C\ \{Q\}$ By precondition strengthening.

- This proof schema justifies:

Derived Assignment Rule

$$\frac{\vdash\ P \Rightarrow Q[E/V]}{\vdash\ \{P\}\ V\texttt{:=}E\ \{Q\}}$$

- Note: $Q[E/V]$ is the weakest liberal precondition $wlp(V\texttt{:=}E, Q)$
- Example proof above can now be done in one less step

1. $\vdash\ \texttt{R=X}\ \Rightarrow\ \texttt{R=X} \wedge \texttt{0=0}$ By pure logic.
2. $\vdash\ \{\texttt{R=X}\}\ \texttt{Q:=0}\ \{\texttt{R=X} \wedge \texttt{Q=0}\}$ By derived assignment.

Derived Sequenced Assignment Rule

- **The following rule will be useful later**

Derived Sequenced Assignment Rule

$$\frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V := E\ \{Q\}}$$

- **Exercise: give a proof schema to justify this**
- **Intuitively work backwards:**
  - **push Q 'through' $V := E$, changing it to $Q[E/V]$**
- **Example: By the assignment axiom:**

```
⊢ {X=x∧Y=y} R:=X {R=x∧Y=y}
```

- **Hence by the sequenced assignment rule**

```
⊢ {X=x∧Y=y} R:=X; X:=Y {R=x∧X=y}
```

Backward Hoare & forward Floyd assignment axioms

- **Recall Hoare (backward) and Floyd (forward) assignment axioms**

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

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

- **Exercise 1 (easy): derive forward axiom from Hoare axiom**
  - **hint:** $P\ \Rightarrow\ \exists v.\ E = E[v/V]\ \wedge\ P[v/V]$
- **Exercise 2 (a bit harder): derive Hoare axiom from forward axiom**
  - **hint: if v is a new variable then** $P[E/V][v/V]\ =\ P[E[v/V]/V]$
- **Exercise 3: devise and justify a derived assignment rule based on the Floyd assignment axiom**

The Derived While Rule

Derived While Rule

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

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

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

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

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

Example

- **By the assignment axiom**

(i) $\vdash$ `{X=x∧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}`

- **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 ∧ Y=y} R:=X; X:=Y; Y:=R {Y=x ∧ X=y}`

Derived Block Rule

- **From the derived sequencing rule and the block rule the following rule for blocks can be derived**

The Derived Block Rule

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ \{P_1\}\ C_1\ \{Q_1\} & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ \{P_2\}\ C_2\ \{Q_2\} & \vdash\ Q_2 \Rightarrow P_3 \\ . & . \\ . & . \\ . & . \\ \vdash\ \{P_n\}\ C_n\ \{Q_n\} & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ \{P\}\ \texttt{BEGIN VAR}\ V_1;\ \ldots\ \texttt{VAR}\ V_m; C_1;\ \ldots\ ;\ C_n\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_m$ occur in P or Q.

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}`

- **Using the derived block rule, it can be deduced *in one step* from (i), (ii) and (iii) that**

```
⊢ {X=x ∧ Y=y}
    BEGIN VAR R; R:=X; X:=Y; Y:=R END
   {Y=x ∧ X=y}
```

Deriving Rules for New Commands

- **Suppose we define a one-armed conditional by:**
`IF` S `THEN` C $\equiv$ `IF` S `THEN` C `ELSE SKIP`

- **We can derive the following rule:**

The One-Armed Conditional Rule

$$\frac{\vdash\ \{P \wedge S\}\ C\ \{Q\} \qquad \vdash\ P \wedge \neg S \Rightarrow Q}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}}$$

- **Derivation**

1.	$\vdash\ P \wedge \neg S\ \Rightarrow\ Q$	Hypothesis
2.	$\vdash\ \{P \wedge \neg S\}$ `SKIP` $\{Q\}$	By derived `SKIP` rule
3.	$\vdash\ \{P \wedge S\}\ C\ \{Q\}$	Hypothesis
4.	$\vdash\ \{P\}$ `IF` S `THEN` C `ELSE SKIP` $\{Q\}$	By conditional rule
5.	$\vdash\ \{P\}$ `IF` S `THEN` C $\{Q\}$	By definition of one-armed conditional

Example

- **From**

```
 (i)  ⊢ {T ∧ X ≥ Y} Y:=X {Y=max(X,Y)}
(ii)  ⊢ T ∧ Y≥X ⇒ max(X,Y)=Y
```

- **Then by the derived one-armed conditional rule it follows that**

```
⊢ {T} IF X≥Y THEN Y:=X {Y=max(X,Y)}
```

Aside: linear versus tree structured proofs

- **Tree structured proofs are fine (e.g. for exams)**
  - advantages: familiar, maybe easier to construct
  - disadvantage: harder to present large proofs
- **Examples (proof schemas, not concrete proofs)**

$$\cfrac{\vdash\ P \Rightarrow Q[E/V] \qquad \cfrac{}{\vdash\ \{Q[E/V]\}\ V := E\ \{Q\}}\,ASS}{\vdash\ \{P\}\ V := E\ \{Q\}}\,PRE$$

$$\cfrac{\vdash\ P \Rightarrow Q \qquad \cfrac{}{\vdash\ \{Q\}\ \texttt{SKIP}\ \{Q\}}\,SKP}{\vdash\ \{P\}\ \texttt{SKIP}\ \{Q\}}\,PRE$$

$$\cfrac{\vdash\ \{P\}C\{Q[E/V]\} \qquad \cfrac{}{\vdash\ \{Q[E/V]\}\ V := E\ \{Q\}}\,ASS}{\vdash\ \{P\}\ C;V := E\ \{Q\}}\,SEQ$$

$$\cfrac{\cfrac{\vdash\ \{P \wedge S\}\ C\ \{Q\} \qquad \cfrac{\vdash\ P \wedge \neg S\ \Rightarrow\ Q}{\vdash\ \{P \wedge \neg S\}\ \texttt{SKIP}\ \{Q\}}\,DSKP}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \texttt{ELSE}\ \texttt{SKIP}\ \{Q\}}\,IF}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}}\,DEF$$

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 ∧ Y=y} R:=X; X:=Y; Y:=R {Y=x ∧ X=y}
```

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

```
{X=x ∧ Y=y} R:=X; X:=Y {R=x ∧ X=y}
```

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

```
{X=x ∧ Y=y} R:=X {R=x ∧ 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
     BEGIN R:=R-Y; Q:=Q+1 END
  {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
            BEGIN R:=R-Y; Q:=Q+1 END
         {X=R+(Y×Q) ∧ R<Y}
```

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

Example Continued (1)

- From previous slide:

```
(i)   ⊢ {T} R:=X; Q:=0 {R=X ∧ Q=0}
```

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

```
(iii)   ⊢ {T} R:=X {R=X ∧ 0=0}
```

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

```
⊢  T ⇒ X=X ∧ 0=0
```

- This is true by pure logic

Example continued (2)

- From an earlier slide:

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

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

```
(iv) R=X ∧ Q=0 ⇒ (X = R+(Y×Q))
(v) X = R+Y×Q ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)
```

and

```
       {X = R+(Y×Q) ∧ (Y≤R)}
(vi)    BEGIN R:=R-Y; Q:=Q+1 END
       {X=R+(Y×Q)}
```

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

Example Continued (3)

- To prove (vi), by the block rule, we must prove

```
       {X = R+(Y×Q) ∧ (Y≤R)}
(vii)   R:=R-Y; Q:=Q+1
       {X=R+(Y×Q)}
```

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

```
        {X=R+(Y×Q) ∧ (Y≤R)}
(viii)   R:=R-Y
        {X=R+(Y×(Q+1))}
```

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

(ix) `X=R+(Y×Q) ∧ Y≤R ⇒ (X = (R-Y)+(Y×(Q+1)))`

- This is true by arithmetic
- Exercise: Construct the forwards proof that corresponds to this backwards proof

Annotations

- The sequencing rule introduces a new statement R

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

- To apply this backwards, one needs to find a suitable statement R
- If C_2 is V:=E then sequenced assignment gives $Q[E/V]$ for R
- If C_2 isn't an assignment then need some other way to choose R
- Similarly, to use the derived While rule, must invent an invariant

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}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ←—P1
   WHILE Y≤R DO {X = R+Y×Q} ←—P2
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{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

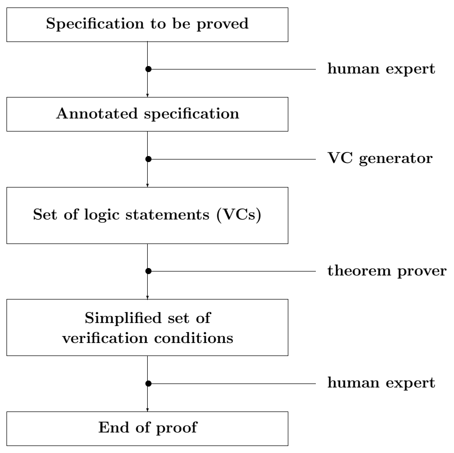
- The architecture of a simple program verifier will be described
- Justified with respect to the rules of Floyd-Hoare logic
- It is clear that
  - proofs are long and boring, even if the program being verified is quite simple
  - lots of fiddly little details to get right, many of which are trivial,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

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

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)
  - used to validate proof carrying code (PCC)
  - provides a verification front end to different provers (see *Why* system)

Architecture of a Verifier


Specification to be proved
human expert
Annotated specification
VC generator
Set of logic statements (VCs)
theorem prover
Simplified set of verification conditions
human expert
End of proof

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
- $\boxed{1}$ The program C is *annotated* by inserting statements (*assertions*) expressing conditions that are meant to hold at intermediate points
  - this step is tricky and needs intelligence and a good understanding of how the program works
  - automating it is an artificial intelligence problem
- $\boxed{2}$ 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
- $\boxed{3}$ 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}
 BEGIN
   R:=X;
   Q:=0;
   WHILE Y≤R DO
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

Example

- Step $\boxed{1}$ is to insert annotations P_1 and P_2

```
{T}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ←— P₁
   WHILE Y≤R DO {X = R+Y×Q} ←— P₂
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

- The annotations P_1 and P_2 state conditions which are intended to hold *whenever* control reaches them

Example Continued

```
{T}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ←— P₁
   WHILE Y≤R DO {X = R+Y×Q} ←— P₂
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

- Control only reaches the point at which P_1 is placed once
- It reaches P_2 each time the `WHILE` body is executed
  - whenever this happens `X=R+Y×Q` holds, even though the values of `R` and `Q` vary
  - P_2 is an *invariant* of the `WHILE`-command

Generating and Proving Verification Conditions

- **Step $\boxed{2}$ will generate the following four verification conditions**

  (i) `T` $\Rightarrow$ `(X=X` $\wedge$ `0=0)`
  (ii) `(R=X` $\wedge$ `Q=0)` $\Rightarrow$ `(X = R+(Y×Q))`
  (iii) `(X = R+(Y×Q))` $\wedge$ `Y≤R)` $\Rightarrow$ `(X = (R-Y)+(Y×(Q+1)))`
  (iv) `(X = R+(Y×Q))` $\wedge$ $\neg$`(Y≤R)` $\Rightarrow$ `(X = R+(Y×Q)` $\wedge$ `R<Y)`

- **Notice that these are statements of arithmetic**
  - the constructs of our programming language have been 'compiled away'
- **Step $\boxed{3}$ consists in proving the four verification conditions**
  - easy with modern automatic theorem provers

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` (and `FOR`) 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}
 BEGIN
   Y:=1; ⟵①
   WHILE X≠0 DO ⟵②
     BEGIN Y:=Y×X; X:=X-1 END
 END
{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**

A VC Generation Program

- **The algorithm for generating verification conditions is *recursive* on the structure of commands**
- **The rule just given corresponds to the recursive program clause:**

  VC $(C(C_1, C_2))$ = `[`vc_1, … ,vc_n`]`**@ (VC** C_1**) @ (VC** C_2**)**
- **The 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
  - only one rule applies in each case so 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 $\vdash \{P\}C\{Q\}$ is a theorem of the logic

Justification of Verification Conditions

- **The argument that the verification conditions are sufficient will be by *induction* on the structure of C**
- **Such inductive arguments have two parts**
  - show the result holds for atomic commands: skip and assignments
  - show that when C is not an atomic command, then if the result holds for the constituent commands of C (this is called the *induction hypothesis*), then it holds also for C
- **The first of these parts is called the *basis* of the induction**
- **The second is called the *step***
- **The basis and step entail that the result holds for all commands**

VC for SKIP

The Skip command

The single verification condition generated by

$$\{P\}\ \texttt{SKIP}\ \{Q\}$$

is

$$P \ \Rightarrow\ Q$$

- **Example: The verification condition for**

  `{X=0} SKIP {X=0}`

  is

  `X=0` $\Rightarrow$ `X=0`

  (which is clearly true)

Justification of `SKIP` VC

- We must show that if the VCs of $\{P\}$ `SKIP` $\{Q\}$ are provable then $\vdash\ \{P\}$ `SKIP` $\{Q\}$

- Proof:
  - Assume $P \Rightarrow Q$ is provable since it is the VC (i.e. assume $\vdash\ P \Rightarrow Q$)
  - From derived Skip rule it follows that $\vdash\ \{P\}$ `SKIP` $\{Q\}$ as required

VC for Assignments

Assignment commands

The single verification condition generated by

$$\{P\}\ V\texttt{:=}E\ \{Q\}$$

is

$$P\ \Rightarrow\ Q\texttt{[}E/V\texttt{]}$$

- Example: The verification condition for

  `{X=0} X:=X+1 {X=1}`

  is

  `X=0` $\Rightarrow$ `(X+1)=1`

  (which is clearly true)

- Note: $Q\texttt{[}E/V\texttt{]}\ =\ \texttt{wlp}(\texttt{"}V\texttt{:=}E\texttt{"}, Q)$

Justification of Assignment VC

- We must show that if the VCs of $\{P\}\ V := E\ \{Q\}$ are provable then $\vdash\ \{P\}\ V := E\ \{Q\}$

- Proof:
  - Assume $\vdash\ P \Rightarrow Q\texttt{[}E/V\texttt{]}$ as it is the VC
  - From derived assignment rule it follows that $\vdash\ \{P\}\ V := E\ \{Q\}$

VCs for Two-Armed Conditional

Two-armed conditional

The verification conditions generated from

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$$

are

(i) the verification conditions generated by

$$\{P\ \wedge\ S\}\ C_1\ \{Q\}$$

(ii) the verifications generated by

$$\{P\ \wedge\ \neg S\}\ C_2\ \{Q\}$$

- Example: The verification conditions for

  `{T} IF X≥Y THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}`

  are

  (i) the VCs for `{T ∧ X≥Y} MAX:=X {MAX=max(X,Y)}`

  (ii) the VCs for `{T ∧ ¬(X≥Y)} MAX:=Y {MAX=max(X,Y)}`

Justification for the Conditional VCs (1)

- Must show that if VCs of
  $\{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$
  are provable, then
  $\vdash$ $\{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$

- Proof:
  - Assume the VCs $\{P \wedge S\}\ C_1\ \{Q\}$ and $\{P \wedge \neg S\}\ C_2\ \{Q\}$
  - The inductive hypotheses tell us that if these VCs are provable then the corresponding Hoare Logic theorems are provable
  - i.e. by induction $\vdash \{P \wedge S\}\ C_1\ \{Q\}$ and $\vdash \{P \wedge \neg S\}\ C_2\ \{Q\}$
  - Hence by the conditional rule $\vdash$ $\{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$

Review of Annotated Sequences

- If $C_1;C_2$ is properly annotated, then either

  $\boxed{\text{Case 1:}}$ it is of the form $C_1;\{R\}C_2$ and C_2 **is not an assignment**

  $\boxed{\text{Case 2:}}$ it is of the form $C;V := E$

- And C, C_1 and C_2 are properly annotated

VCs for Sequences

Sequences

1. The verification conditions generated by

$$\{P\}\ C_1\ \{R\}\ C_2\ \{Q\}$$

   (where C_2 is not an assignment) are the union of:

   (a) the verification conditions generated by $\{P\}\ C_1\ \{R\}$

   (b) the verifications generated by $\{R\}\ C_2\ \{Q\}$

2. The verification conditions generated by

$$\{P\}\ C;V\texttt{:=}E\ \{Q\}$$

   are the verification conditions generated by $\{P\}\ C\ \{Q[E/V]\}$

Example

- The verification conditions generated from

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

- Are those generated by

  $\{$`X=x` $\wedge$ `Y=y`$\}$ `R:=X; X:=Y` $\{$`(X=y` $\wedge$ `Y=x)[R/Y]`$\}$

- This simplifies to

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

- The verification conditions generated by this are those generated by

  $\{$`X=x` $\wedge$ `Y=y`$\}$ `R:=X` $\{$`(X=y` $\wedge$ `R=x)[Y/X]`$\}$

- Which simplifies to

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

Example Continued

- The only verification condition generated by

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

  is

  `X=x` $\wedge$ `Y=y` $\Rightarrow$ `(Y=y` $\wedge$ `R=x)[X/R]`

- Which simplifies to

  `X=x` $\wedge$ `Y=y` $\Rightarrow$ `Y=y` $\wedge$ `X=x`

- Thus the single verification condition from

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

  is

  `X=x` $\wedge$ `Y=y` $\Rightarrow$ `Y=y` $\wedge$ `X=x`

Justification of VCs for Sequences (1)

- $\boxed{\text{Case 1:}}$ If the verification conditions for

  $\{P\}\ C_1\ ;\ \{R\}\ C_2\ \{Q\}$

  are provable

- Then the verification conditions for

  $\{P\}\ C_1\ \{R\}$

  and

  $\{R\}\ C_2\ \{Q\}$

  must both be provable

- Hence by induction

  $\vdash\ \{P\}\ C_1\ \{R\}$ and $\vdash\ \{R\}\ C_2\ \{Q\}$

- Hence by the sequencing rule

  $\vdash\ \{P\}\ C_1;C_2\ \{Q\}$

Justification of VCs for Sequences (2)

- $\boxed{\text{Case 2:}}$ If the verification conditions for

  $\{P\}\ C;V := E\ \{Q\}$

  are provable, then the verification conditions for

  $\{P\}\ C\ \{Q[E/V]\}$

  are also provable

- Hence by induction

  $\vdash\ \{P\}\ C\ \{Q[E/V]\}$

- Hence by the derived sequenced assignment rule

  $\vdash\ \{P\}\ C;V := E\ \{Q\}$

VCs for Blocks

Blocks

The verification conditions generated by

$$\{P\}\ \texttt{BEGIN VAR}\ V_1;\ \ldots\ ;\texttt{VAR}\ V_n;C\ \texttt{END}\ \{Q\}$$

are

(i) the verification conditions generated by $\{P\}C\{Q\}$, and

(ii) none of V_1, $\ldots$, V_n occur in either P or Q.

Syntactic Conditions for Blocks

- **Generating verification conditions from blocks involves checking a syntactic condition**
  - **the local variables do not occur in the precondition or postcondition**
- **The need for this is clear from the side condition in the block rule**
- **Example: the verification conditions for**

```
{X=x ∧ Y=y}
BEGIN VAR R; R:=X; X:=Y; Y:=R END
{X=y ∧ Y=x}
```

- **Are the verification conditions generated by**

  `{X=x ∧ Y=y} R:=X; X:=Y; Y:=R {X=y ∧ Y=x}`

  and the syntactic check that `R` **not in** `{X=x ∧ Y=y}` **or** `{X=y ∧ Y=x}`
- **See previous example for verification conditions generated by this**

Justification of VCs for Blocks

- **If the verification conditions for**

  $\{P\}$ `BEGIN VAR` V_1`;`$\ldots$`;VAR` V_n`;`C `END` $\{Q\}$

  are provable
- **Then the verification conditions for**

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

  are provable and $V_1, \ldots, V_n$ **do not occur in** P **or** Q
- **By induction**

  $\vdash \{P\}\ C\ \{Q\}$
- **Hence by the block rule**

  $\vdash$ $\{P\}$ `BEGIN VAR` V_1`;`$\ldots$`;VAR` V_n`;`C `END` $\{Q\}$

VCs for `WHILE`-Commands

- **A correctly annotated specification of a `WHILE`-command has the form**

  $\{P\}$ `WHILE` S `DO` $\{R\}$ C $\{Q\}$
- **The annotation** R **is called an invariant**

`WHILE`-commands

The verification conditions generated from

$\{P\}$ `WHILE` S `DO` $\{R\}$ C $\{Q\}$

are

(i) $P \Rightarrow R$

(ii) $R \wedge \neg S \Rightarrow Q$

(iii) the verification conditions generated by $\{R \wedge S\}\ C\{R\}$

Example

- **The verification conditions for**

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

are:

(i) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

(ii) `X = R+Y×Q ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)`

together with the verification condition for

```
{X = R+(Y×Q) ∧ (Y≤R)}
 BEGIN R:=R-Y; Q:=Q+1 END
{X=R+(Y×Q)}
```

which consists of the single condition

(iii) `X = R+(Y×Q) ∧ (Y≤R) ⇒ X = (R-Y)+(Y×(Q+1))`

Example Summarised

- By previous transparency

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

if

```
⊢ R=X ∧ Q=0 ⇒ (X = R+(Y×Q))
```

and

```
⊢ X = R+(Y×Q) ∧ ¬(Y≤R)⇒ (X = R+(Y×Q) ∧ R<Y)
```

and

```
⊢ X = R+(Y×Q) ∧ (Y≤R)  ⇒ X = (R-Y)+(Y×(Q+1))
```

Justification of WHILE VCs

- If the verification conditions for

  $\{P\}$ `WHILE` S `DO` $\{R\}$ C $\{Q\}$

  are provable, then

  $\vdash\ P \Rightarrow R$

  $\vdash\ (R\ \wedge\ \neg S)\ \Rightarrow\ Q$

  and the verification conditions for

  $\{R\ \wedge\ S\}\ C\ \{R\}$

  are provable

- By induction

  $\vdash\ \{R\ \wedge\ S\}\ C\ \{R\}$

- Hence by the derived `WHILE`-rule

  $\vdash\ \{P\}$ `WHILE` S `DO` C $\{Q\}$

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
  - it appeals to the derived rules
- All the techniques introduced earlier are used
  - backwards proof
  - derived rules
  - annotation

Dijkstra's weakest preconditions

- Weakest preconditions is a theory of refinement
  - idea is to calculate a program to achieve a postcondition
  - not a theory of post hoc verification
- Non-determinism a key idea in Dijksta's presentation
  - start with a non-deterministic high level pseudo-code
  - refine to deterministic and efficient code
- Weakest preconditions (wp) are for total correctness
- Weakest *liberal* preconditions (wlp) for partial correctness
- If C is a command and Q a predicate, then informally:
  - `wlp`$(C,Q)\ =$ '*The weakest predicate P such that $\{P\}\ C\ \{Q\}$*'
  - `wp`$(C,Q)\ =$ '*The weakest predicate P such that $[P]\ C\ [Q]$*'
- If P and Q are predicates then $Q\ \Rightarrow\ P$ means P is 'weaker' than Q

Rules for weakest preconditions

- **Relation with Hoare specifications:**

$$\{P\}\ C\ \{Q\} \quad\Leftrightarrow\quad P \Rightarrow \texttt{wlp}(C,Q)$$
$$[P]\ C\ [Q] \quad\Leftrightarrow\quad P \Rightarrow \texttt{wp}(C,Q)$$

- **Dijkstra gives rules for computing weakest preconditions:**

$$\begin{array}{lcl} \texttt{wp}(\texttt{SKIP},Q) & = & Q \\ \texttt{wp}(V\texttt{:=}E,Q) & = & Q[E/V] \\ \texttt{wp}(C_1\texttt{;}C_2,\ Q) & = & \texttt{wp}(C_1,\texttt{wp}(C_2,\ Q)) \\ \texttt{wp}(\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ Q) & = & (B\ \Rightarrow \texttt{wp}(C_1,Q))\ \wedge\ (\neg B\ \Rightarrow\ \texttt{wp}(C_2,Q)) \end{array}$$

  for deterministic loop-free code the same equations hold for `wlp`

- **Rule for `WHILE`-commands doesn't give a first order result**

- **Weakest preconditions closely related to verification conditions**

- **VCs for $\{P\}\ C\ \{Q\}$ are related to $P \Rightarrow \texttt{wlp}(C,Q)$**
  - **VCs use annotations to ensure first order formulas can be generated**

Sequencing example

- **Swapping variables:**

$$\begin{array}{l} \texttt{wlp}(\texttt{R:=X; X:=Y; Y:=R},(\texttt{Y}=x \wedge \texttt{X}=y)) \\ \quad = \texttt{wlp}(\texttt{R:=X},\ \texttt{wlp}(\texttt{X:=Y},\ \texttt{wlp}(\texttt{Y:=R},\ (\texttt{Y}=x \wedge \texttt{X}=y)))) \\ \quad = \texttt{wlp}(\texttt{R:=X},\ \texttt{wlp}(\texttt{X:=Y},\ (\texttt{Y}=x \wedge \texttt{X}=y)[\texttt{R/Y}])) \\ \quad = \texttt{wlp}(\texttt{R:=X},\ \texttt{wlp}(\texttt{X:=Y},\ (\texttt{R}=x \wedge \texttt{X}=y))) \\ \quad = \texttt{wlp}(\texttt{R:=X},\ (\texttt{R}=x \wedge \texttt{Y}=y)) \\ \quad = (\texttt{X}=x \wedge \texttt{Y}=y) \end{array}$$

- **So since $\{P\}\ C\ \{Q\} \Leftrightarrow P \Rightarrow \texttt{wlp}(C,Q)$**

  to prove

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

  just need to prove:

  $(\texttt{X}=x \wedge \texttt{Y}=y) \Rightarrow (\texttt{X}=x \wedge \texttt{Y}=y)$

  which is clearly true (instance of $S \Rightarrow S$)

Conditional example

- **Compute `wlp` of the maximum program:**

$$\begin{array}{l} \texttt{wlp}(\texttt{IF X<Y THEN MAX:=Y ELSE MAX:=X},(\texttt{MAX}=max(x,y)) \\ \quad = (\texttt{X<Y}\ \Rightarrow\ \texttt{wlp}(\texttt{MAX:=Y},\ (\texttt{MAX}=max(x,y)))) \\ \qquad \wedge \\ \qquad (\neg(\texttt{X<Y})\ \Rightarrow\ \texttt{wlp}(\texttt{MAX:=X},\ (\texttt{MAX}=max(x,y)))) \\ \quad = (\texttt{X<Y}\ \Rightarrow\ \texttt{Y}=max(x,y))\ \wedge\ (\neg(\texttt{X<Y})\ \Rightarrow\ \texttt{X}=max(x,y)) \\ \quad = \mathit{if}\ \texttt{X<Y}\ \mathit{then}\ \texttt{Y}=max(x,y)\ \mathit{else}\ \texttt{X}=max(x,y) \end{array}$$

- **So to prove**

  $\{\texttt{X}=x \wedge \texttt{Y}=y\}$ `IF X<Y THEN MAX:=X ELSE MAX:=Y` $\{\texttt{MAX}=max(x,y)\}$

  just prove:

  $(\texttt{X}=x \wedge \texttt{Y}=y) \Rightarrow (\texttt{X<Y}\ \Rightarrow\ \texttt{Y}=max(x,y))\ \wedge\ (\neg(\texttt{X<Y})\ \Rightarrow\ \texttt{X}=max(x,y))$

  which follows from the defining property of max

  $\vdash\ \ \forall x\ y.\ (x \geq y \Rightarrow x = max(x,y))\ \wedge\ (\neg(x \geq y) \Rightarrow y = max(x,y))$

Using `wlp` to improve verification condition method

- **If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $P \Rightarrow \texttt{wlp}(C,Q)$**
  - **no annotations needed in sequences!**

- **Cannot in general compute a finite formula for $\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)$**

- **The following holds**

  $\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)\ =\ \mathit{if}\ S\ \mathit{then}\ \texttt{wlp}(C,\ \texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q))\ \mathit{else}\ Q$

- **Above doesn't define $\texttt{wlp}(C,Q)$ as a finite statement**

- **We will describe a hybrid VC and `wlp` method**

$\texttt{wlp}$-based verification condition method

- We define $\texttt{awp}(C,Q)$ and $\texttt{wvc}(C,Q)$
  - $\texttt{awp}(C,Q)$ is a statement sort of approximating $\texttt{wlp}(C,Q)$
  - $\texttt{wvc}(C,Q)$ is a set of verification conditions
- If C is loop-free then
  - $\texttt{awp}(C,Q) \ = \ \texttt{wlp}(C,Q)$
  - $\texttt{wvc}(C,Q) \ = \ \{\}$
- Denote by $\bigwedge\mathcal{S}$ the conjunction of all the statements in $\mathcal{S}$
  - $\bigwedge\{\} = \texttt{T}$
  - $\bigwedge(\mathcal{S}_1 \cup \mathcal{S}_2) = (\bigwedge\mathcal{S}_1) \wedge (\bigwedge\mathcal{S}_2)$
- It will follow that $\bigwedge\texttt{wvc}(C,Q) \Rightarrow \{\texttt{awp}(C,Q)\}\ C\ \{Q\}$
- Hence to prove $\{P\}C\{Q\}$ it is sufficient to prove all the statements in $\texttt{wvc}(C,Q)$ and $P \Rightarrow \texttt{awp}(C,Q)$

Definition of $\texttt{awp}$

- Assume all $\texttt{WHILE}$-commands are annotated: $\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C$
- Define $\texttt{awp}$ recursively by:

$$
\begin{array}{ll}
\texttt{awp}(\texttt{SKIP},\ Q) & = Q \\
\texttt{awp}(V\ \texttt{:=}\ E,\ Q) & = Q[E/V] \\
\texttt{awp}(C_1\ ;\ C_2,\ Q) & = \texttt{awp}(C_1,\ \texttt{awp}(C_2,\ Q)) \\
\texttt{awp}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ Q) & = (S\ \wedge \texttt{awp}(C_1,\ Q)) \vee (\neg S \wedge \texttt{awp}(C_2,\ Q)) \\
\texttt{awp}(\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C,\ Q) & = R
\end{array}
$$

- Note:
  $(S\ \wedge \texttt{awp}(C_1,\ Q)) \vee (\neg S \wedge \texttt{awp}(C_2,\ Q)\ = \ \mathit{if}\ S\ \mathit{then}\ \texttt{awp}(C_1,\ Q)\ \mathit{else}\ \texttt{awp}(C_2,\ Q)$

Definition of $\texttt{wvc}$

- Assume all $\texttt{WHILE}$-commands are annotated: $\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C$
- Define $\texttt{wvc}$ recursively by:

$$
\begin{array}{ll}
\texttt{wvc}(\texttt{SKIP},\ Q) & = \{\} \\
\texttt{wvc}(V\ \texttt{:=}\ E,\ Q) & = \{\} \\
\texttt{wvc}(C_1\ ;\ C_2,\ Q) & = \texttt{wvc}(C_1, \texttt{awp}(C_2, Q)) \cup \texttt{wvc}(C_2, Q) \\
\texttt{wvc}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ Q) & = \texttt{wvc}(C_1,\ Q) \cup \texttt{wvc}(C_2,\ Q) \\
\texttt{wvc}(\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C,\ Q) & = \{R \wedge \neg S \Rightarrow Q,\ R \wedge S \Rightarrow \texttt{awp}(C, R)\} \\
 & \quad \cup\, \texttt{wvc}(C, R)
\end{array}
$$

Correctness of $\texttt{wlp}$-based verification conditions

- Theorem: $\bigwedge\texttt{wvc}(C,Q) \Rightarrow \{\texttt{awp}(C,Q)\}\ C\ \{Q\}$. Proof by Induction on C
  - $\bigwedge\texttt{wvc}(\texttt{SKIP},Q) \Rightarrow \{\texttt{awp}(C,Q)\}\ C\ \{Q\}$ is $\texttt{T} \Rightarrow \{Q\}\ \texttt{SKIP}\ \{Q\}$
  - $\bigwedge\texttt{wvc}(V\texttt{:=}E,Q) \Rightarrow \{\texttt{awp}(C,Q)\}\ C\ \{Q\}$ is $\texttt{T} \Rightarrow \{Q[E/V]\}\ V\ \texttt{:=}\ E\ \{Q\}$
  - $\bigwedge\texttt{wvc}(C_1;C_2,Q) \Rightarrow \{\texttt{awp}(C_1;C_2,Q)\}\ C_1;C_2\ \{Q\}$ is
    $\bigwedge(\texttt{wvc}(C_1,\texttt{awp}(C_2,Q)) \cup \texttt{wvc}(C_2,Q)) \Rightarrow \{\texttt{awp}(C_1,\ \texttt{awp}(C_2,Q))\}\ C_1;C_2\ \{Q\}$.
    By induction $\bigwedge\texttt{wvc}(C_2,Q) \Rightarrow \{\texttt{awp}(C_2,Q)\}\ C_1\ \{Q\}$
    and $\bigwedge\texttt{wvc}(C_1,\texttt{awp}(C_2,Q)) \Rightarrow \{\texttt{awp}(C_1,\texttt{awp}(C_2,Q))\}\ C_2\ \{\texttt{awp}(C_2,Q)\}$,
    hence result by the Sequencing Rule.
  - $\bigwedge\texttt{wvc}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,Q)$
    $\Rightarrow \{\texttt{awp}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,Q)\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$
    is $\bigwedge(\texttt{wvc}(C_1,\ Q) \cup \texttt{wvc}(C_2,\ Q))$
    $\Rightarrow \{(S\ \wedge \texttt{awp}(C_1,\ Q)) \vee (\neg S \wedge \texttt{awp}(C_2,\ Q)\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$.
    By induction $\bigwedge\texttt{wvc}(C_1,Q) \Rightarrow \{\texttt{awp}(C_1,Q)\}\ C_1\ \{Q\}$
    and $\bigwedge\texttt{wvc}(C_2,Q) \Rightarrow \{\texttt{awp}(C_2,Q)\}\ C_2\ \{Q\}$. Strengthening preconditions
    gives $\bigwedge\texttt{wvc}(C_1,Q) \Rightarrow \{\texttt{awp}(C_1,Q) \wedge S\}\ C_1\ \{Q\}$
    and $\bigwedge\texttt{wvc}(C_2,Q) \Rightarrow \{\texttt{awp}(C_2,Q) \wedge \neg S\}\ C_2\ \{Q\}$, hence
    $\bigwedge\texttt{wvc}(C_1,Q) \Rightarrow \{((S\ \wedge \texttt{awp}(C_1,\ Q)) \vee (\neg S \wedge \texttt{awp}(C_2,\ Q))) \wedge S\}\ C_1\ \{Q\}$
    and $\bigwedge\texttt{wvc}(C_2,Q) \Rightarrow \{((S\ \wedge \texttt{awp}(C_1,\ Q)) \vee (\neg S \wedge \texttt{awp}(C_2,\ Q))) \wedge \neg S\}\ C_2\ \{Q\}$,
    hence result by the Conditional Rule.
  - $\bigwedge\texttt{wvc}(\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C,Q) \Rightarrow \{\texttt{awp}(\texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C,Q)\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$
    is $\bigwedge(\{R \wedge \neg S \Rightarrow Q,\ R \wedge S \Rightarrow \texttt{awp}(C,R)\} \cup \texttt{wvc}(C,R)) \Rightarrow \{R\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$.
    By induction $\bigwedge\texttt{wvc}(C,R) \Rightarrow \{\texttt{awp}(C,R)\}\ C\ \{R\}$, hence result by $\texttt{WHILE}$-Rule.

Example

$\boxed{\mathtt{wvc}(\mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C,\ Q)\ =\ (R,\ \{R \wedge \neg S \Rightarrow Q,\ R \wedge S \Rightarrow \mathtt{awp}(C,R)\} \cup \mathtt{wvc}(C,R))}$

$\mathtt{wvc}(\mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q})$ **(analyse body)**
$\quad = (\mathtt{wlp}(\mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}), \{\})$
$\quad = (\mathtt{X} = \mathtt{R\text{-}Y} + \mathtt{Y} \times \mathtt{Q+1},\ \{\})$

$\mathtt{wvc}(\mathtt{WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y})$
$\quad = (\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q},$
$\qquad \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y},$
$\qquad\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R\text{-}Y} + \mathtt{Y} \times \mathtt{Q+1}\} \cup \{\})$

$\mathtt{wvc}(\mathtt{Q:=0;WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;\ Q:=Q+1},\ \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y})$
$\quad = (\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{0},$
$\qquad \{\} \cup \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y},$
$\qquad\qquad \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R\text{-}Y} + \mathtt{Y} \times \mathtt{Q+1}\})$

$\mathtt{wvc}(\mathtt{R=X;Q:=0;WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;\ Q:=Q+1},\ \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y})$
$\quad = (\mathtt{X} = \mathtt{X} + \mathtt{Y} \times \mathtt{0},$
$\qquad \{\} \cup \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R{+}Y{\times}Q} \wedge \mathtt{R{<}Y},$
$\qquad\qquad \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R\text{-}Y} + \mathtt{Y} \times \mathtt{Q+1}\})$

As $\mathtt{X} = \mathtt{X} + \mathtt{Y} \times \mathtt{0}$ is T the correctness theorem for $\mathtt{wvc}$ gives

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

Strongest postconditions

- **Define $\mathtt{sp}(C,P)$ to be 'strongest' Q such that $\{P\}\ C\ \{Q\}$**
  - partial correctness: $\{P\}\ C\ \{\mathtt{sp}(C,P)\}$
  - strongest means if $\{P\}\ C\ \{Q\}$ then $\mathtt{sp}(C,P) \Rightarrow Q$
- **Note that $\mathtt{wlp}$ goes 'backwards', but $\mathtt{sp}$ goes 'forwards'**
  - verification condition for $\{P\}\ C\ \{Q\}$ is: $\mathtt{sp}(C,P) \Rightarrow Q$
- **By 'strongest' and Hoare logic postcondition weakening**
  - $\{P\}\ C\ \{Q\}$ **if and only if** $\mathtt{sp}(C,P) \Rightarrow Q$

Strongest postconditions for loop-free code

- **Only consider loop-free code**
- $\mathtt{sp}(\mathtt{SKIP},\ P)\ =\ P$
- $\mathtt{sp}(V\ \mathtt{:=}\ E,\ P)\ =\ \exists v.\ V = E\mathtt{[}v/V\mathtt{]} \wedge P\mathtt{[}v/V\mathtt{]}$
- $\mathtt{sp}(C_1\ \mathtt{;}\ C_2,\ P)\ =\ \mathtt{sp}(C_2,\ \mathtt{sp}(C_1,\ P))$
- $\mathtt{sp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,\ P)\ =\ \mathtt{sp}(C_1,\ P \wedge S)\ \vee\ \mathtt{sp}(C_2,\ P \wedge \neg S)$

- **$\mathtt{sp}(V\mathtt{:=}E,\ P)$ corresponds to Floyd assignment axiom**
- **Can dynamically prune conditionals because $\mathtt{sp}(C,\mathtt{F}) = \mathtt{F}$**

Sequencing example

- $\mathtt{sp}(\mathtt{R:=X;\ X:=Y;\ Y:=R},\ \mathtt{X} = x \wedge \mathtt{Y} = y)$

$$\begin{aligned}
&= \mathtt{sp}(\mathtt{Y:=R},\ \mathtt{sp}(\mathtt{X:=Y},\ \mathtt{sp}(\mathtt{R:=X},\ \mathtt{X} = x \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ \mathtt{sp}(\mathtt{X:=Y},\ (\exists v.\ \mathtt{R} = \mathtt{X}\mathtt{[}v/\mathtt{R}\mathtt{]} \wedge (\mathtt{X} = x \wedge \mathtt{Y} = y)\mathtt{[}v/\mathtt{R}\mathtt{]})))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ \mathtt{sp}(\mathtt{X:=Y},\ (\exists v.\ \mathtt{R} = \mathtt{X} \wedge (\mathtt{X} = x \wedge \mathtt{Y} = y))))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ \mathtt{sp}(\mathtt{X:=Y},\ (\mathtt{R} = \mathtt{X} \wedge \mathtt{X} = x \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\exists v.\ \mathtt{X} = \mathtt{Y}\mathtt{[}v/\mathtt{X}\mathtt{]} \wedge (\mathtt{R} = \mathtt{X} \wedge \mathtt{X} = x \wedge \mathtt{Y} = y)\mathtt{[}v/\mathtt{X}\mathtt{]}))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\exists v.\ \mathtt{X} = \mathtt{Y} \wedge (\mathtt{R} = v \wedge v = x \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\exists v.\ \mathtt{X} = \mathtt{Y} \wedge (\mathtt{R} = x \wedge v = x \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\mathtt{X} = \mathtt{Y} \wedge (\mathtt{R} = x \wedge (\exists v.\ v = x) \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\mathtt{X} = \mathtt{Y} \wedge (\mathtt{R} = x \wedge \mathtt{T} \wedge \mathtt{Y} = y)))\\
&= \mathtt{sp}(\mathtt{Y:=R},\ (\mathtt{X} = \mathtt{Y} \wedge \mathtt{R} = x \wedge \mathtt{Y} = y))\\
&= \exists v.\ \mathtt{Y} = \mathtt{R}\mathtt{[}v/\mathtt{Y}\mathtt{]} \wedge (\mathtt{X} = \mathtt{Y} \wedge \mathtt{R} = x \wedge \mathtt{Y} = y)\mathtt{[}v/\mathtt{Y}\mathtt{]}\\
&= \exists v.\ \mathtt{Y} = \mathtt{R} \wedge (\mathtt{X} = v \wedge \mathtt{R} = x \wedge v = y)\\
&= \exists v.\ \mathtt{Y} = \mathtt{R} \wedge (\mathtt{X} = y \wedge \mathtt{R} = x \wedge v = y)\\
&= \mathtt{Y} = \mathtt{R} \wedge (\mathtt{X} = y \wedge \mathtt{R} = x \wedge (\exists v.\ v = y))\\
&= \mathtt{Y} = \mathtt{R} \wedge (\mathtt{X} = y \wedge \mathtt{R} = x \wedge \mathtt{T})\\
&= \mathtt{Y} = \mathtt{R} \wedge \mathtt{X} = y \wedge \mathtt{R} = x\\
&= \mathtt{Y} = x \wedge \mathtt{X} = y \wedge \mathtt{R} = x
\end{aligned}$$

- **So to prove $\{\mathtt{X} = x \wedge \mathtt{Y} = y\}\ \mathtt{R:=X;\ X:=Y;\ Y:=R}\ \{\mathtt{Y} = x \wedge \mathtt{X} = y\}$**
  just prove: $(\mathtt{Y} = x \wedge \mathtt{X} = y \wedge \mathtt{R} = x) \Rightarrow \mathtt{Y} = x \wedge \mathtt{X} = y$

Conditional example

- **Compute sp of the maximum program:**

$$
\begin{array}{rl}
& \mathtt{sp}(\mathtt{IF\ X<Y\ THEN\ MAX:=Y\ ELSE\ MAX:=X},\ (\mathtt{X}=x \wedge \mathtt{Y}=y))\\
= & \mathtt{sp}(\mathtt{MAX:=Y},\ ((\mathtt{X}=x \wedge \mathtt{Y}=y) \wedge \mathtt{X}<\mathtt{Y}))\\
 & \vee\\
 & \mathtt{sp}(\mathtt{MAX:=X},\ ((\mathtt{X}=x \wedge \mathtt{Y}=y) \wedge \neg(\mathtt{X}<\mathtt{Y})))\\
= & \exists v.\ \mathtt{MAX}=\mathtt{Y}[v/\mathtt{MAX}] \wedge ((\mathtt{X}=x \wedge \mathtt{Y}=y) \wedge \mathtt{X}<\mathtt{Y})[v/\mathtt{MAX}]\\
 & \vee\\
 & \exists v.\ \mathtt{MAX}=\mathtt{X}[v/\mathtt{MAX}] \wedge ((\mathtt{X}=x \wedge \mathtt{Y}=y) \wedge \neg(\mathtt{X}<\mathtt{Y}))[v/\mathtt{MAX}]\\
= & \exists v.\ \mathtt{MAX}=\mathtt{Y} \wedge ((\mathtt{X}=x \wedge \mathtt{Y}=y) \wedge \mathtt{X}<\mathtt{Y})\\
 & \vee\\
 & \exists v.\ \mathtt{MAX}=\mathtt{X} \wedge \mathtt{X}=x \wedge \mathtt{Y}=y \wedge \neg(\mathtt{X}<\mathtt{Y}))\\
= & (\mathtt{MAX}=\mathtt{Y} \wedge \mathtt{X}=x \wedge \mathtt{Y}=y \wedge \mathtt{X}<\mathtt{Y}) \vee (\mathtt{MAX}=\mathtt{X} \wedge \mathtt{X}=x \wedge \mathtt{Y}=y \wedge \neg(\mathtt{X}<\mathtt{Y})\\
= & (\mathtt{MAX}=y \wedge \mathtt{X}=x \wedge \mathtt{Y}=y \wedge x<y) \vee (\mathtt{MAX}=x \wedge \mathtt{X}=x \wedge \mathtt{Y}=y \wedge \neg(x<y)\\
= & \mathit{if}\ x<y\ \mathit{then}\ (\mathtt{MAX}=y \wedge \mathtt{X}=x \wedge \mathtt{Y}=y)\ \mathit{else}\ (\mathtt{MAX}=x \wedge \mathtt{X}=x \wedge \mathtt{Y}=y)\\
= & \mathtt{MAX}=(\mathit{if}\ x<y\ \mathit{then}\ y\ \mathit{else}\ x) \wedge \mathtt{X}=x \wedge \mathtt{Y}=y\\
= & \mathtt{MAX}=\mathit{max}(x,y) \wedge \mathtt{X}=x \wedge \mathtt{Y}=y
\end{array}
$$

Computing sp as symbolic execution: assignment (1)

- **Floyd assignment formula makes computing sp messy in general**
- **For a special case it becomes like symbolic execution**
- **Symbolic state:** $\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E_i \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R$
- **Suppose** E_1**, …,** E_n **or** R **doesn't contain** $\mathtt{X}_1$**, …,** $\mathtt{X}_n$ **then**

$$
\begin{array}{rl}
& \mathtt{sp}(\mathtt{X}_i\mathtt{:=}E,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E_i \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R))\\
= & \exists v.\ \mathtt{X}_i=E[v/\mathtt{X}_i] \wedge (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E_i \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)[v/\mathtt{X}_i]\\
= & \exists v.\ \mathtt{X}_i=E[v/\mathtt{X}_i] \wedge (\mathtt{X}_1=E_1 \wedge \cdots \wedge v=E_i \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)\\
= & \mathtt{X}_i=E[E_i/\mathtt{X}_i] \wedge (\mathtt{X}_1=E_1 \wedge \cdots \wedge (\exists v.\ v=E_i) \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)\\
= & \mathtt{X}_i=E[E_i/\mathtt{X}_i] \wedge (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{T} \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)\\
= & \mathtt{X}_i=E[E_i/\mathtt{X}_i] \wedge \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R\\
= & \mathtt{X}_i=E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \wedge \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R\\
= & \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R
\end{array}
$$

- **Note** $E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n]$ **doesn't contain** $\mathtt{X}_1$**, …,** $\mathtt{X}_n$

Computing sp as symbolic execution: assignment (2)

- **Suppose** $\mathtt{X} \neq \mathtt{X}_i$ **and** $\mathtt{X}$ **doesn't occur in** E_i **or** E **for** $1 \leq i \leq n$**, then**

$$
\begin{array}{rl}
& \mathtt{sp}(\mathtt{X:=}E,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R))\\
= & \exists v.\ \mathtt{X}=E[v/\mathtt{X}] \wedge (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)[v/\mathtt{X}]\\
= & \exists v.\ \mathtt{X}=E \wedge \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R\\
= & \mathtt{X}=E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \wedge \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R
\end{array}
$$

- **Summarising: we have two symbolic computation rules:**
  1. **if** E_1**, …,** E_n **or** R **doesn't contain** $\mathtt{X}_1$**, …,** $\mathtt{X}_n$ **then:**

     $\mathtt{sp}(\mathtt{X}_i\mathtt{:=}E,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E_i \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R))$

     $= (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_i=E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)$
  2. **if** $\mathtt{X} \neq \mathtt{X}_i$ **and** $\mathtt{X}$ **doesn't occur in** E_i **or** E **for** $1 \leq i \leq n$**, then**

     $\mathtt{sp}(\mathtt{X:=}E,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R))$

     $= (\mathtt{X}\ =\ E[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \wedge \mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R)$

Computing sp as symbolic execution: conditional (1)

- **Suppose if none of** E_1**, …,**E_n**,** R_1**,** R_2 **contain** $\mathtt{X}_1$**, …,** $\mathtt{X}_n$ **then**

$$
\begin{array}{l}
\mathtt{sp}(C_1,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R_1))\ =\ (\mathtt{X}_1=E_{11} \wedge \cdots \wedge \mathtt{X}_n=E_{1n} \wedge R_1)\\
\mathtt{sp}(C_2,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R_2))\ =\ (\mathtt{X}_1=E_{21} \wedge \cdots \wedge \mathtt{X}_n=E_{2n} \wedge R_2)
\end{array}
$$

- **Conditional notation:** $(B \rightarrow E_1 \mid E_2)\ =\ \mathit{if}\ B\ \mathit{then}\ E_1\ \mathit{else}\ E_2$
- **Then for** E_1**, …,**E_n**,** R **not containing** $\mathtt{X}_1$**, …,** $\mathtt{X}_n$

$$
\begin{array}{rl}
& \mathtt{sp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R))\\
= & \mathtt{sp}(C_1,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R) \wedge S)\\
 & \vee\\
 & \mathtt{sp}(C_2,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge R) \wedge \neg S)\\
= & \mathtt{sp}(C_1,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge (R \wedge S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n])))\\
 & \vee\\
 & \mathtt{sp}(C_2,\ (\mathtt{X}_1=E_1 \wedge \cdots \wedge \mathtt{X}_n=E_n \wedge (R \wedge \neg S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n])))\\
= & (\mathtt{X}_1=E_{11} \wedge \cdots \wedge \mathtt{X}_n=E_{1n} \wedge (R \wedge S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n])) \qquad \textbf{(by assumption)}\\
 & \vee\\
 & (\mathtt{X}_1=E_{21} \wedge \cdots \wedge \mathtt{X}_n=E_{2n} \wedge (R \wedge \neg S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n]))\\
= & (\mathtt{X}_1=(S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \rightarrow E_{11}|E_{21})) \wedge \cdots \wedge (\mathtt{X}_n=(S[E_1/\mathtt{X}_1]\cdots[E_n/\mathtt{X}_n] \rightarrow E_{1n}|E_{2n})) \wedge R
\end{array}
$$

Computing sp as symbolic execution: conditional (2)

- **From last slide if $E_1, \ldots, E_n$, R do not contain $\mathtt{X}_1, \ldots, \mathtt{X}_n$**

  $\mathtt{sp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R))$

  $= (\mathtt{X}_1{=}(S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}{\rightarrow}E_{11}|E_{21}))\wedge\cdots\wedge(\mathtt{X}_n{=}(S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}{\rightarrow}E_{1n}|E_{2n})) \wedge R$

  where

  $\mathtt{sp}(C_1,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R_1))\ =\ (\mathtt{X}_1 = E_{11} \wedge \cdots \wedge \mathtt{X}_n = E_{1n} \wedge R_1)$

  $\mathtt{sp}(C_2,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R_2))\ =\ (\mathtt{X}_1 = E_{21} \wedge \cdots \wedge \mathtt{X}_n = E_{2n} \wedge R_2)$

- **If C_1 or C_2 don't assign to $\mathtt{X}_i$ then $E_i = E_{1i} = E_{2i}$ so**
  $(S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}{\rightarrow}E_{1i}|E_{2i}) = E_i$
  so formula above can be further simplified

- **If R determines the value of $S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}$ then can simplify**
  $(\mathtt{X}_i = (S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]} \rightarrow E_{1i} \mid E_{2i}))$

Summary of sp loop-free code symbolic execution

- **Symbolic state:** $(\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R)$

- $\mathtt{sp}(\mathtt{SKIP},\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R)) = (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R))$

- **If $E_1, \ldots, E_n$ or R doesn't contain $\mathtt{X}_1, \ldots, \mathtt{X}_n$ then:**

  $\mathtt{sp}(\mathtt{X}_i\mathtt{:=}E,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_i = E_i \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R))$

  $= (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_i = E\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]} \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R)$

- **If $\mathtt{X} \neq \mathtt{X}_i$ and $\mathtt{X}$ doesn't occur in E_i or E for $1 \leq i \leq n$, then**

  $\mathtt{sp}(\mathtt{X:=}E,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R))$

  $= (\mathtt{X} = E\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]} \wedge \mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R)$

- **If $E_1, \ldots, E_n$, R does not contain $\mathtt{X}_1, \ldots, \mathtt{X}_n$**

  $\mathtt{sp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R))$

  $= (\mathtt{X}_1{=}(S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}{\rightarrow}E_{11}|E_{21}))\wedge\cdots\wedge(\mathtt{X}_n{=}(S\mathtt{[}E_1/\mathtt{X}_1\mathtt{]}\cdots\mathtt{[}E_n/\mathtt{X}_n\mathtt{]}{\rightarrow}E_{1n}|E_{2n})) \wedge R$

  where

  $\mathtt{sp}(C_1,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R_1))\ =\ (\mathtt{X}_1 = E_{11} \wedge \cdots \wedge \mathtt{X}_n = E_{1n} \wedge R_1)$
  $\mathtt{sp}(C_2,\ (\mathtt{X}_1 = E_1 \wedge \cdots \wedge \mathtt{X}_n = E_n \wedge R_2))\ =\ (\mathtt{X}_1 = E_{21} \wedge \cdots \wedge \mathtt{X}_n = E_{2n} \wedge R_2)$

Example symbolic computation using sp (1)

- $\mathtt{sp}(C_1\ ;\ C_2,\ P)\ =\ \mathtt{sp}(C_2,\ \mathtt{sp}(C_1,\ P))$ **hence**

```
sp(R := 0;
   K := 0;
   IF I < J THEN K := K + 1 ELSE SKIP;
   IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
   (I = i ∧ J = j ∧ i < j))  =

sp(K := 0;
   IF I < J THEN K := K + 1 ELSE SKIP;
   IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
   (R = 0 ∧ I = i ∧ J = i ∧ i < j))  =

sp(IF I < J THEN K := K + 1 ELSE SKIP;
   IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
   (K = 0 ∧ R = 0 ∧ I = i ∧ J = j ∧ i < j))
```

Example symbolic computation using sp (2)

Considering each conditional branch and merging gives something like:

```
sp(IF I < J THEN K := K + 1 ELSE SKIP;
   IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
   (K = 0 ∧ R = 0 ∧ I = i ∧ J = j ∧ i < j))  =
(K=(i<j → E11 | E12) ∧ R=(i<j → E21 | E22) ∧ I=(i<j → E31 | E32) ∧ J=(i<j → E41 | E42) ∧ i<j)
```

Noting '$\wedge\ i < j$' it follows that this simplifies to:

```
sp(IF I < J THEN K := K + 1 ELSE SKIP;
   IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
   (K = 0 ∧ R = 0 ∧ I = i ∧ J = j ∧ i < j))  =
(K = E11 ∧ R = E21 ∧ I = E31 ∧ J = E41 ∧ i < j)
```

so need only consider the `I < J` branch

Example symbolic computation using `sp` (3)

$$\begin{array}{l}
\mathtt{sp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP};\\
\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)) \ =
\end{array}$$

$$\begin{array}{l}
\mathtt{sp}(\mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{K} = (\mathtt{K}+1)[0/\mathtt{K}] \wedge \mathtt{R} = 0 \wedge \mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)) \ =
\end{array}$$

$$\begin{array}{l}
\mathtt{sp}(\mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{K} = 1 \wedge \mathtt{R} = 0 \wedge \mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)) \ = \qquad \textbf{(by a similar argument)}
\end{array}$$

$$(\mathtt{K} = 1 \wedge \mathtt{R} = j{-}i \wedge \mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)$$

Hence by $\{P\}\ C\ \{Q\} \ = \ \mathrm{sp}(C, P) \Rightarrow Q$ it follows that

$$\begin{array}{l}
\{(\mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j\}\\
\quad \mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP};\\
\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J})\\
\{R = j{-}i\}
\end{array}$$

Same example using `wlp`

$$\begin{array}{l}
\mathtt{wlp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP};\\
\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{R} = \mathtt{J}{-}\mathtt{I})) \ =
\end{array}$$

$$\begin{array}{l}
\mathtt{wlp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP},\\
\quad (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \ \mid\ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J}))
\end{array}$$

$$\begin{array}{l}
\mathtt{wlp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad (\mathtt{I} < \mathtt{J} \ \rightarrow\ (\mathtt{K}+1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \ \mid\ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J})\\
\qquad\qquad \mid\ (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \ \mid\ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J})) \ =
\end{array}$$

$$\begin{array}{l}
(\mathtt{I} < \mathtt{J} \ \rightarrow\ (0{+}1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J})\\
\qquad\quad \mid\ (0 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J})) \ =
\end{array}$$

$$(\mathtt{I} < \mathtt{J} \ \rightarrow\ (\neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J}) \ \mid\ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J}) \ =$$

$$(\mathtt{I} < \mathtt{J} \ \rightarrow\ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \ \mid\ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J}) \ =$$

$$(\mathtt{I} < \mathtt{J})$$

Example comparing `sp` with `wlp`

- Going forward (simplifying on-the-fly)

$$\begin{array}{l}
\mathtt{sp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP};\\
\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)) \ =\\
(\mathtt{K} = 1 \wedge \mathtt{R} = j{-}i \wedge \mathtt{I} = i \wedge \mathtt{J} = j \wedge i < j)
\end{array}$$

- Going backwards

$$\begin{array}{l}
\mathtt{wlp}(\mathtt{R} := 0;\\
\quad \mathtt{K} := 0;\\
\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K}+1\ \mathtt{ELSE\ SKIP};\\
\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I}-\mathtt{J},\\
\quad (\mathtt{R} = \mathtt{J}{-}\mathtt{I})) \ =\\
(\mathtt{I} < \mathtt{J} \ \rightarrow\ (0{+}1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J})
\end{array}$$

Computing `sp` versus `wlp`

- Computing `sp` is like execution
  - can simplify as one goes along with the ‘current state’
  - may be able to resolve branches, so can avoid executing them
  - Floyd assignment rule complicated in general
  - `sp` used for symbolically exploring ‘reachable states’
    (see Specification and Verification II, especially model checking)
- Computing `wlp` is like backwards proof
  - don’t have ‘current state’, so can’t simplify using it
  - can’t determine conditional tests, so get big `if-then-else` trees
  - Hoare assignment rule simpler for arbitrary formulae
  - `wlp` used for improved verification conditions
    (see later slides)

Exercises

- **Compute**

```
sp(R := 0;
    K := 0;
    IF I < J THEN K := K + 1 ELSE SKIP;
    IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J,
    (I = i ∧ J = j ∧ j ≤ i))
```

- **Hence show**

```
{(I = i ∧ J = j ∧ j ≤ i}
    R := 0;
    K := 0;
    IF I < J THEN K := K + 1 ELSE SKIP;
    IF K = 1 ∧ ¬(I = J) THEN R := J − I ELSE R := I − J)
{R = i−j}
```

- **Do same example use wlp**

Using sp to generate verification conditions

- **If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $\texttt{sp}(C,\ P) \Rightarrow Q$**
- **Cannot in general compute a finite formula for $\texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P)$**
- **The following holds**

  $\texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P)\ = \texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ \texttt{sp}(C,\ (P \wedge S)))\ \vee\ (P \wedge \neg S)$
- **Above doesn't define $\texttt{sp}(C, P)$ to be a finite statement**
- **As with wlp, we will describe a hybrid VC and sp method**

sp-based verification conditions

- **Define $\texttt{asp}(C, P)$ to be an approximate strongest postcondition**
- **Define $\texttt{svc}(C, P)$ to be a set of verification conditions**
- **Idea is that if $\wedge\texttt{svc}(C, P) \Rightarrow \{P\}\ C\ \{\texttt{asp}(C, P)\}$**
- **If C is loop-free then**
  - $\texttt{asp}(C, P) = \texttt{sp}(C, P)$
  - $\texttt{svc}(C, P) = \{\}$

Definition of asp

- **Define asp recursively by:**

$$\begin{aligned}
&\texttt{asp}(P,\ \texttt{SKIP}) &&= P\\
&\texttt{asp}(P,\ V\ \texttt{:=}\ E) &&= \exists v.\ V = E[v/V] \wedge P[v/V]\\
&\texttt{asp}(P,\ C_1\ ;\ C_2) &&= \texttt{asp}(\texttt{asp}(P, C_1), C_2)\\
&\texttt{asp}(P,\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2) &&= \texttt{asp}(P \wedge S,\ C_1)\ \vee\ \texttt{asp}(P \wedge \neg S,\ C_2)\\
&\texttt{asp}(P,\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C) &&= R \wedge \neg S
\end{aligned}$$

Definition of `svc`

- Define `svc` recursively by:

$$
\begin{array}{ll}
\texttt{svc}(P,\ \texttt{SKIP}) & = \{\} \\
\texttt{svc}(P,\ V \texttt{ := } E) & = \{\} \\
\texttt{svc}(P,\ C_1 \texttt{ ; } C_2) & = \texttt{svc}(P, C_1) \cup \texttt{svc}(\texttt{svc}_1(P, C_1), C_2) \\
\texttt{svc}(P,\ \texttt{IF } S \texttt{ THEN } C_1 \texttt{ ELSE } C_2) & = \texttt{svc}(P \wedge S,\ C_1)\ \cup\ \texttt{svc}(P \wedge \neg S,\ C_2) \\
\texttt{svc}(P,\ \texttt{WHILE } S \texttt{ DO } \{R\}\ C) & = \{P \Rightarrow R,\ \texttt{asp}(R \wedge S,\ C) \Rightarrow R\} \\
 & \quad \cup\ \texttt{svc}(R \wedge S,\ C)
\end{array}
$$

- Theorem: $\wedge\texttt{svc}(P, C) \Rightarrow \{P\}\ C\ \{\texttt{asp}(P, C)\}$

- Proof by induction on C (exercise)

Summary

- Annotate then generate VCs is the classical method
  - practical tools: Gypsy (1970s), SPARK (current)
  - weakest preconditions are alternative explanation of VCs
  - `wlp` needs fewer annotations than VC method described earlier
  - `wlp` also used for refinement

- VCs and `wlp` go backwards, `sp` goes forward
  - `sp` provides verification method based on symbolic simulation
  - widely used for loop-free code
  - current research potential for forwards full proof of correctness
  - probably need mixture of forwards and backwards methods (Hoare's view)

Range of methods for proving $\{P\}C\{Q\}$

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute
  - check states reached satisfy decidable properties

- Full proof of correctness
  - add invariants to loops
  - generate verification conditions
  - prove verification conditions with a theorem prover

- Research goal: unifying framework for a spectrum of methods

Proving Programs Terminate

Total Correctness Specification

- So far our discussion has been concerned with partial correctness
  - what about termination
- A total correctness specification $[P]\ C\ [Q]$ is true if and only if
  - whenever C is executed in a state satisfying P, then the execution of C terminates
  - after C terminates Q holds
- Except for the `WHILE`-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness

Termination of `WHILE`-Commands

- `WHILE`-commands are the only commands that might not terminate
- Consider now the following proof

1. $\vdash$ {T} `SKIP` {T} (`SKIP` axiom)
2. $\vdash$ {T $\wedge$ T} `SKIP` {T} (precondition strengthening)
3. $\vdash$ {T} `WHILE T DO SKIP` {T $\wedge$ $\neg$T} (2 and the `WHILE`-rule)

- If the `WHILE`-rule worked for total correctness, then this would show:

$$\vdash [\mathtt{T}]\ \mathtt{WHILE\ T\ DO\ SKIP}\ [\mathtt{T} \wedge \neg\mathtt{T}]$$

- Thus the `WHILE`-rule is unsound for total correctness

Rules for Non-Looping Commands

- Replace $\{$ and $\}$ by $[$ and $]$, respectively, in:
  - Assignment axiom (see next slide for discussion)
  - Consequence rules
  - Conditional rules
  - Sequencing rule
  - Block rule
- The following is a valid derived rule

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ [P]\ C\ [Q]}$$

if C contains no `WHILE`-commands

Total Correctness Assignment Axiom

- Assignment axiom for total correctness

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

- Note that the assignment axiom for total correctness states that assignment commands *always* terminate
- So all function applications in expressions must terminate
- This might not be the case if functions could be defined recursively
- Consider X `:=` $\mathit{fact}(-1)$, where $\mathit{fact}(n)$ is defined recursively:

$$\mathit{fact}(n)\ =\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathit{fact}(n-1)$$

Error Termination

- We assume erroneous expressions like $1/0$ don't cause problems
- Most programming languages will cause an error stop when division by zero is encountered encountered
- In our logic it follows that

$$\vdash\ [\mathtt{T}]\ \mathtt{X} := 1/0\ [\mathtt{X} = 1/0]$$

- The assignment `X :=` $1/0$ halts in a state in which `X` $= 1/0$ holds
- This assumes that $1/0$ denotes some value that `X` can have

Two Possibilities

- There are two possibilities
  (i) $1/0$ denotes some number;
  (ii) $1/0$ denotes some kind of 'error value'.
- It seems at first sight that adopting (ii) is the most natural choice
  - this makes it tricky to see what arithmetical laws should hold
  - is $(1/0) \times 0$ equal to 0 or to some 'error value'?
  - if the latter, then it is no longer the case that $\forall n.\ n \times 0 = 0$ is valid
- It is possible to make everything work with undefined and/or error values, but the resultant theory is a bit messy

Example

- We assume that arithmetic expressions *always* denote numbers
- In some cases exactly what the number is will be not fully specified
  - for example, we will assume that m/n denotes a number for any m and n
  - only assume: $\neg(n = 0) \quad \Rightarrow \quad (m/n) \times n = m$
  - it is not possible to deduce anything about $m/0$ from this
  - in particular it is not possible to deduce that $(m/0) \times 0 = 0$
  - but $(m/0) \times 0 = 0$ does follow from $\forall n.\ n \times 0 = 0$
- People still argue about this – e.g. advocate "three-valued" logics

`WHILE`-rule for Total Correctness (i)

- `WHILE`-commands are the only commands in our little 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` S `DO` C terminates
  - show that some non-negative quantity decreases on each iteration of C
  - this decreasing quantity is called a **variant**

`WHILE`-Rule for Total Correctness (ii)

- In the rule below, the variant is E, and the fact that it decreases is specified with an auxiliary variable n
- The hypothesis $\vdash\ P \wedge S \Rightarrow E \geq 0$ ensures the variant is non-negative

`WHILE`-rule for total correctness

$$\frac{\vdash\ [P \wedge S \wedge (E = n)]\ C\ [P \wedge (E < n)], \quad \vdash\ P \wedge S \Rightarrow E \geq 0}{\vdash\ [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [P \wedge \neg S]}$$

where E is an integer-valued expression
and n is an identifier not occurring in P, C, S or E.

Example

- We show

  $\vdash\ [\texttt{Y} > 0]$ `WHILE Y≤R DO BEGIN R:=R-Y; Q:=Q+1 END` $[\texttt{T}]$

- Take

  $P\ =\ \texttt{Y} > \texttt{0}$
  $S\ =\ \texttt{Y} \leq \texttt{R}$
  $E\ =\ \texttt{R}$
  $C\ =$ `BEGIN R:=R-Y Q:=Q+1 END`

- We want to show $\vdash\ [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [\texttt{T}]$
- By the `WHILE`-rule for total correctness it is sufficient to show

  (i) $\vdash\ [P \wedge S \wedge (E = \texttt{n})]\ C\ [P \wedge (E < \texttt{n})]$
  (ii) $\vdash\ P \wedge S \Rightarrow E \geq 0$

Example Continued (1)

- From previous slide:

  $P\ =\ \texttt{Y} > \texttt{0}$
  $S\ =\ \texttt{Y} \leq \texttt{R}$
  $E\ =\ \texttt{R}$
  $C\ =$ `BEGIN R:=R-Y Q:=Q+1 END`

- We want to show

  (i) $\vdash\ [P \wedge S \wedge (E = \texttt{n})]\ C\ [P \wedge (E < \texttt{n})]$
  (ii) $\vdash\ P \wedge S \Rightarrow E \geq 0$

- The first of these, (i), can be proved by establishing

  $\vdash\ \{P \wedge S \wedge (E = \texttt{n})\}\ C\ \{P \wedge (E < \texttt{n})\}$

- Then using the total correctness rule for non-looping commands

Example Continued (2)

- From previous slide:

  $P\ =\ \texttt{Y} > \texttt{0}$
  $S\ =\ \texttt{Y} \leq \texttt{R}$
  $E\ =\ \texttt{R}$
  $C\ =$ `BEGIN R:=R-Y Q:=Q+1 END`

- The verification condition for $\{P \wedge S \wedge (E = \texttt{n})\}\ C\ \{P \wedge (E < \texttt{n})\}$ is:

  $\texttt{Y} > 0\ \wedge\ \texttt{Y} \leq \texttt{R}\ \wedge\ \texttt{R} = \texttt{n}\ \ \Rightarrow$
  $(\texttt{Y} > 0\ \wedge\ \texttt{R} < \texttt{n})[\texttt{Q}{+}1/\texttt{Q}][\texttt{R}{-}\texttt{Y}/\texttt{R}]$

  i.e. $\texttt{Y} > 0\ \wedge\ \texttt{Y} \leq \texttt{R}\ \wedge\ \texttt{R} = \texttt{n}\ \ \Rightarrow\ \ \texttt{Y} > 0\ \wedge\ \texttt{R}{-}\texttt{Y} < \texttt{n}$

  which follows from the laws of arithmetic

- The second subgoal, (ii), is just $\vdash\ \texttt{Y} > 0\ \wedge\ \texttt{Y} \leq \texttt{R} \Rightarrow \texttt{R} \geq 0$

Termination Specifications

- **The relation between partial and total correctness is informally given by the equation**

Total correctness = Termination + Partial correctness

- **This informal equation can be represented by the following two rules of inferences**

$$\frac{\vdash\ \{P\}\ C\ \{Q\} \qquad \vdash\ [P]\ C\ [\mathtt{T}]}{\vdash\ [P]\ C\ [Q]}$$

$$\frac{\vdash\ [P]\ C\ [Q]}{\vdash\ \{P\}\ C\ \{Q\} \qquad \vdash\ [P]\ C\ [\mathtt{T}]}$$

Derived Rules

- **Multiple step rules for total correctness can be derived in the same way as for partial correctness**
  - **the rules are the same up to the brackets used**
  - **same derivations with total correctness rules replacing partial correctness ones**

Example: The Derived Skip Rule

Derived Skip Rule

$$\frac{\vdash\ P \Rightarrow Q}{\vdash\ [P]\ \mathtt{SKIP}\ [Q]}$$

- **Proof:**

1. $\vdash\ P \Rightarrow Q$ By assumption.
2. $\vdash\ [Q]\ \mathtt{SKIP}\ [Q]$ By the skip axiom for total correctness.
3. $\vdash\ [P]\ \mathtt{SKIP}\ [Q]$ By precondition strengthening with 1 and 2.

- **Derivation Tree**

$$\dfrac{\vdash\ P \Rightarrow Q \quad \dfrac{}{\vdash\ [Q]\ \mathtt{SKIP}\ [Q]}\,SKP}{\vdash\ [P]\ \mathtt{SKIP}\ [Q]}\,PRE$$

The Derived While Rule

- **Derived `WHILE`-rule needs to handle the variant**

Derived `WHILE`-rule for total correctness

$$\frac{\begin{array}{l}\vdash\ P \Rightarrow R \\ \vdash\ R \wedge S \Rightarrow E \geq 0 \\ \vdash\ R \wedge \neg S\ \Rightarrow Q \\ \vdash\ [R \wedge S \wedge (E = n)]\ C\ [R \wedge (E < n)]\end{array}}{\vdash\ [P]\ \mathtt{WHILE}\ S\ \mathtt{DO}\ C\ [Q]}$$

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
- Variant added directly after the invariant, in square brackets
- No other extra annotations are needed for total correctness
- VCs generation algorithm same as for partial correctness

`WHILE` Annotation

- A correctly annotated total correctness specification of a `WHILE`-command thus has the form

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

  where R is the invariant and E the variant

- Note that the variant is intended to be a non-negative expression that decreases each time around the `WHILE` loop
- The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them (as before)

`WHILE` VCs

- A correctly annotated specification of a `WHILE`-command has the form

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

`WHILE`-commands

The verification conditions generated from

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

are

(i) $P \Rightarrow R$

(ii) $R \wedge \neg S \Rightarrow Q$

(iii) $R \wedge S \Rightarrow E \geq 0$

(iv) the verification conditions generated by

$$[R \wedge S \wedge (E = n)]\ C[R \wedge (E < n)]$$

where n is a variable not occurring in P, R, E, C, S or Q.

Example

- The verification conditions for

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

are:

(i) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

(ii) `X = R+Y×Q ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)`

(iii) `X = R+Y×Q ∧ Y≤R ⇒ R≥0`

together with the verification condition for

```
[X = R+(Y×Q) ∧ (Y≤R) ∧ (R=n)]
 BEGIN R:=R-Y; Q:=Q+1 END
[X=R+(Y×Q) ∧ (R<n)]
```

Example Continued

- The single verification condition for

```
[X = R+(Y×Q) ∧ (Y≤R) ∧ (R=n)]
 BEGIN R:=R-Y; Q:=Q+1 END
[X=R+(Y×Q) ∧ (R<n)]
```

  is

```
(iv) X = R+(Y×Q) ∧ (Y≤R) ∧ (R=n) ⇒
     X = (R-Y)+(Y×(Q+1)) ∧ ((R-Y)<n)
```

- But this isn't true
  - take `Y=0`
- To prove `R-Y<n` we need to know `Y>0`
- *Exercise:* Explain why one would not expect to be able to prove the verification conditions of this last example
- *Hint:* Consider the original specification

Summary

- We have given rules for total correctness
- They are similar to those for partial correctness
- The main difference is in the `WHILE`-rule
  - because `WHILE` commands are the only ones that can fail to terminate
- Must prove a non-negative expression is decreased by the loop body
- Derived rules and VC generation rules for partial correctness easily extended to total correctness
- Interesting stuff on the web
  - http://www.crunchgear.com/2008/12/31/zune-bug-explained-in-detail
  - http://research.microsoft.com/TERMINATOR

Arrays and `FOR`-commands
Soundness and Completeness

Overview

- All the axioms and rules given so far were quite straightforward
  - may have given a false sense of simplicity
- Hard to give rules for anything other than *very* simple constructs
  - an incentive for using simple languages
- We already saw with the assignment axiom that intuition over how to formulate a rule might be wrong
  - the assignment axiom can seem 'backwards'
- We now look at the remaing commands in our little language
  - array assignments
  - `FOR`-commands

Array assignments

- **Syntax:** $V(E_1)$`:=`E_2
- **Semantics: the state is changed by assigning the value of the term E_2 to the E_1-th component of the array variable V**
- **Example:** `A(X+1) := A(X)+2`
  - if the the value of `X` is x
  - and the value of the x-th component of `A` is n
  - then the value stored in the $(x+1)$-th component of `A` becomes $n+2$

Naive Array Assignment Axiom Fails

- **The axiom**

  $\vdash\ \{P\texttt{[}E_2/A(E_1)\texttt{]}\}\ A(E_1) := E_2\ \{P\}$

  doesn't work
- **Take** $P \equiv$ '`X=Y` $\wedge$ `A(Y)=0`', $E_1 \equiv$ '`X`', $E_2 \equiv$ '`1`'
  - since `A(X)` does not occur in P
  - it follows that P`[1/A(X)] =` P
  - hence the axiom yields: $\vdash$ `{X=Y` $\wedge$ `A(Y)=0} A(X):=1 {X=Y` $\wedge$ `A(Y)=0}`
  - which is false since it implies: $\vdash\ \{\cdots\}$ `A(X):=1 {A(X)=0}`
- **Must take into account possibility that changes to `A(X)` may change `A(Y)`, `A(Z)` etc**
  - since `X` might equal `Y`, `Z` etc (i.e. **aliasing**)
- **Related to the *Frame Problem* in AI**

Reasoning About Arrays

- **The naive array assignment axiom**

  $$\vdash\ \{P\texttt{[}E_2/A(E_1)\texttt{]}\}\ A(E_1) := E_2\ \{P\}$$

  does not work: changes to $A(X)$ may also change $A(Y)$, $A(Z)$, ...
- **The solution, due to Hoare, is to treat an array assignment**

  $$A(E_1){:=}E_2$$

  as an ordinary assignment

  $$A := A\{E_1{\leftarrow}E_2\}$$

  where the term $A\{E_1{\leftarrow}E_2\}$ denotes an array identical to A, except that the E_1-th component is changed to have the value E_2

Array Assignment axiom

- **Array assignment is a special case of ordinary assignment**

  $$A\texttt{:=}A\{E_1{\leftarrow}E_2\}$$

- **Array assignment axiom just ordinary assignment axiom**

  $$\vdash\ \{P\texttt{[}A\{E_1{\leftarrow}E_2\}/A\texttt{]}\}\ A\texttt{:=}A\{E_1{\leftarrow}E_2\}\ \{P\}$$

- **Thus:**

The array assignment axiom

$$\vdash\ \{P\texttt{[}A\{E_1{\leftarrow}E_2\}/A\texttt{]}\}\ A(E_1)\texttt{:=}E_2\ \{P\}$$

Where A is an array variable, E_1 is an integer valued expression, P is any statement and the notation $A\{E_1{\leftarrow}E_2\}$ denotes the array identical to A, except that $A(E_1) = E_2$.

Array Axioms

- In order to reason about arrays, the following axioms, which define the meaning of the notation $A\{E_1 \leftarrow E_2\}$, are needed

The array axioms

$$\vdash \ A\{E_1 \leftarrow E_2\}(E_1) \ = \ E_2$$

$$\vdash \ E_1 \neq E_3 \ \Rightarrow \ A\{E_1 \leftarrow E_2\}(E_3) \ = \ A(E_3)$$

- Second of these is a *Frame Axiom*

Example

- We show

```
⊢ {A(X)=x ∧ A(Y)=y}
    R    := A(X);
    A(X) := A(Y);
    A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

- Working backwards using the array assignment axiom

```
⊢ {(A{Y←R})(X)=y ∧ (A{Y←R})(Y)=x}
   A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

- Array assignments are variable assignments of array values, so:

```
⊢ {(A{Y←R})(X)=y ∧ (A{Y←R})(Y)=x}
   A := A{Y←R}
  {A(X)=y ∧ A(Y)=x}
```

Example Continued (1)

- Using

```
⊢ A{Y←R}(Y) = R
```

- It follows that

```
⊢ {(A{Y←R})(X)=y ∧ R=x}
   A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

- Continuing backwards

```
⊢ {((A{X←A(Y)}){Y←R})(X)=y ∧ R=x}
   A(X) := A(Y)
  {(A{Y←R})(X)=y ∧ R=x}
```

- Maybe more intuitive if the assignment is rewritten

```
⊢ {((A{X←A(Y)}){Y←R})(X)=y ∧ R=x}
   A := A{X←A(Y)}
  {(A{Y←R})(X)=y ∧ R=x}
```

Example Continued (2)

- Continuing backwards

```
⊢ {((A{X←A(Y)}){Y←A(X)})(X)=y ∧ A(X)=x}
   R := A(X)
  {((A{X←A(Y)}){Y←R})(X)=y ∧ R=x}
```

- Hence by the derived sequencing rule

```
⊢ {((A{X←A(Y)}){Y←A(X)})(X)=y ∧ A(X)=x}
   R := A(X); A(X) := A(Y); A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

- By the array axioms (considering the cases X=Y and X≠Y separately):

```
⊢ ((A{X←A(Y)}){Y←A(X)})(X)  =  A(Y)
```

- Hence, (as desired)

```
⊢ {A(Y)=y ∧ A(X)=x}
   R := A(X); A(X) := A(Y); A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

FOR-commands

- Syntax: FOR V:=E_1 UNTIL E_2 DO C
  - restriction: V must not occur in E_1 or E_2,
    or be the left hand side of an assignment in C
    (explained later)
- Semantics:
  - if the values of terms E_1 and E_2 are positive numbers e_1 and e_2
  - and if $e_1 \leq e_2$
  - then C is executed $(e_2 - e_1) + 1$ times with the variable V taking on the sequence of values e_1, e_1+1, $\ldots$, e_2 in succession
  - for any other values, the FOR-command has no effect
- Example: `FOR N:=1 UNTIL M DO X:=X+N`
  - if the value of the variable `M` is m and $m \geq 1$, then the command `X:=X+N` is repeatedly executed with `N` taking the sequence of values `1`, $\ldots$, m
  - if $m < 1$ then the FOR-command does nothing

Subtleties of FOR-commands

- There are many subtly different versions of FOR-commands
- For example
  - the expressions E_1 and E_2 could be evaluated at each iteration
  - and the controlled variable V could be treated as global rather than local
- Early languages like Algol 60 failed to notice such subtleties
- Note that with the semantics presented here
  FOR-commands cannot *generate* non termination

More on the semantics of FOR-commands

- The semantics of

  FOR V:=E_1 UNTIL E_2 DO C

  is as follows

(i) E_1 and E_2 are evaluated once to get values e_1 and e_2, respectively.

(ii) If either e_1 or e_2 is not a number, or if $e_1 > e_2$, then nothing is done.

(iii) If $e_1 \leq e_2$ the FOR-command is equivalent to:

BEGIN VAR V; V:=e_1; C; V:=e_1+1; C ; $\ldots$; V:=e_2; C END

i.e. C is executed $(e_2 - e_1) + 1$ times with V taking on the sequence of values e_1, e_1+1, $\ldots$, e_2

- If C doesn't modify V then FOR-command equivalent to:

$$\texttt{BEGIN VAR } V\texttt{; } V\texttt{:=}e_1\texttt{; } \ldots\ \underbrace{C\texttt{ ; } V\texttt{:=}V\texttt{+}1}_{\text{repeated}}\texttt{; } \ldots\ V\texttt{:=}e_2\texttt{; } C\texttt{ END}$$

The Derived Frame Rule (used in motivating FOR Rule)

- The following can be derived

The frame rule

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ \{P \land R\}\ C\ \{Q \land R\}}$$

where no variable assigned to in C occurs in R

- Outline of derivation
  - prove $\{R\}\ C\ \{R\}$ by induction on C
  - then use Specification Conjunction

Towards the FOR-Rule

- **If $e_1 \leq e_2$ the FOR-command is equivalent to:**

  $\texttt{BEGIN VAR } V\texttt{; } V\texttt{:=}e_1\texttt{; } \ldots\ C \texttt{ ; } V\texttt{:=}V\texttt{+1; } \ldots\ \ V\texttt{:=}e_2\texttt{; } C \texttt{ END}$

- **Assume C doesn't modify V and $\vdash\ \{P\}\ C\ \{P\texttt{[}V\texttt{+}1/V\texttt{]}\}$**

- **Hence:**

$$\begin{array}{ll}
\vdash\ \{P\texttt{[}e_1/V\texttt{]}\}\ V\texttt{:=}e_1\ \{P \wedge V{=}e_1\} & \text{(assign. ax + pre. streng.)} \\
\vdash\ \{P \wedge V{=}e_1\}\ C\ \{P\texttt{[}V{+}1/V\texttt{]} \wedge V{=}e_1\} & \text{(assumption + frame rule)} \\
\vdash\ \{P\texttt{[}V{+}1/V\texttt{]} \wedge V{=}e_1\}\ V\texttt{:=}V{+}1\ \{P \wedge V{=}e_1{+}1\} & \text{(assign. ax)} \\
\vdots & \\
\vdash\ \{P \wedge V{=}v\}\ C\ \{P\texttt{[}V{+}1/V\texttt{]} \wedge V{=}v\} & \text{(assumption + frame rule)} \\
\vdash\ \{P\texttt{[}V{+}1/V\texttt{]} \wedge V{=}v\}\ V\texttt{:=}V{+}1\ \{P \wedge V{=}v{+}1\} & \text{(assign. ax + pre. streng.)} \\
\vdots & \\
\vdash\ \ \{P \wedge V{=}e_2\}\ C\ \{P\texttt{[}V{+}1/V\texttt{]} \wedge V{=}e_2\} & \text{(assumption + frame rule)} \\
\vdash\ \ \{P \wedge V{=}e_2\}\ C\ \{P\texttt{[}e_2{+}1/V\texttt{]}\} & \text{(post. weak.)}
\end{array}$$

- **Hence by the derived block rule**

$$\frac{\vdash \{P\}C\{P\texttt{[}V\texttt{+1}/V\texttt{]}\}}{\vdash \{P\texttt{[}e_1/V\texttt{]}\}\texttt{BEGIN VAR }V\texttt{;}V\texttt{:=}e_1\texttt{; }\ldots C\texttt{;}V\texttt{:=}V{+}1\texttt{; }\ldots V\texttt{:=}e_2\texttt{;}C\texttt{ END}\{P\texttt{[}e_2{+}1/V\texttt{]}\}}$$

Problems with the FOR-rule (i)

- **Previous derivation suggests a rule**

$$\frac{\vdash \{P\}\ C\ \{P\texttt{[}V\texttt{+1}/V\texttt{]}\}}{\vdash \{P\texttt{[}E_1/V\texttt{]}\}\ \texttt{FOR } V\texttt{:=}E_1 \texttt{ UNTIL } E_2 \texttt{ DO } C\ \{P\texttt{[}E_2\texttt{+1}/V\texttt{]}\}}$$

- **This is a good start, but needs debugging**

- **Consider:**

$$\vdash\ \{\texttt{X=Y}\}\ \texttt{X:=Y+1}\ \{\texttt{X=Y+1}\}$$

- **Taking P as '`X=Y`' we have**

$$\vdash\ \{P\}\ \texttt{X:=Y+1}\ \{P\texttt{[Y+1/Y]}\}$$

- **By the FOR-rule above, with $V = \texttt{Y}$, $E_1 = \texttt{3}$ and $E_2 = \texttt{1}$**

$$\vdash\ \{\ \underbrace{\texttt{X=3}}_{P\texttt{[3/Y]}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\ \underbrace{\texttt{X=2}}_{P\texttt{[1+1/Y]}}\ \}$$

Problems with the FOR-rule (ii)

- **The conclusion below is clearly undesirable**

$$\vdash\ \{\ \underbrace{\texttt{X=3}}_{P\texttt{[3/Y]}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\ \underbrace{\texttt{X=2}}_{P\texttt{[1+1/Y]}}\ \}$$

- **It was specified that**
  - if the value of E_1 were greater than the value of E_2
  - then the FOR-command should have no effect
  - in this example it changes the value of `X` from `3` to `2`

- **To avoid this, the FOR-rule can be modified to**

$$\frac{\vdash \{P\}\ C\ \{P\texttt{[}V\texttt{+1}/V\texttt{]}\}}{\vdash \{P\texttt{[}E_1/V\texttt{]}\ \wedge\ E_1 \leq E_2\}\ \texttt{FOR } V\texttt{:=}E_1 \texttt{ UNTIL } E_2 \texttt{ DO } C\ \{P\texttt{[}E_2\texttt{+1}/V\texttt{]}\}}$$

Problems with the FOR-rule (iii)

- **FOR-rule so far**

$$\frac{\vdash \{P\}\ C\ \{P\texttt{[}V\texttt{+1}/V\texttt{]}\}}{\vdash \{P\texttt{[}E_1/V\texttt{]}\ \wedge\ E_1 \leq E_2\}\ \texttt{FOR } V\texttt{:=}E_1 \texttt{ UNTIL } E_2 \texttt{ DO } C\ \{P\texttt{[}E_2\texttt{+1}/V\texttt{]}\}}$$

- **On the example just considered this rule results in**

$$\vdash\ \{\texttt{X=3}\ \wedge\ \underbrace{\texttt{3} \leq \texttt{1}}_{\text{never true!}}\}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \ \{\texttt{X=2}\}$$

- **This conclusion is harmless**
  - only asserts `X` changed if FOR-command executed in impossible starting state

Problems with the FOR-rule (iv)

- **Unfortunately, there is a bug in**

$$\frac{\vdash \{P\}\ C\ \{P[V+1/V]\}}{\vdash \{P[E_1/V]\ \wedge\ E_1 \le E_2\}\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[E_2\texttt{+1}/V]\}}$$

- **Take P to be '`Y=1`' and note that**

$$\vdash \{\underbrace{\texttt{Y=1}}_{P}\}\ \texttt{Y:=Y-1}\ \{\underbrace{\texttt{Y+1=1}}_{P[\texttt{Y+1/Y}]}\}$$

- **So by this FOR-rule**

$$\vdash \{\underbrace{\texttt{1=1}}_{P[\texttt{1/Y}]}\ \wedge 1 \le 1\}\ \texttt{FOR Y:=1 UNTIL 1 DO Y:=Y-1}\ \{\underbrace{\texttt{2=1}}_{P[\texttt{1+1/Y}]}\}$$

Problems with the FOR-rule (v)

- **Whatever the command does, it doesn't lead to a state in which `2=1`**
- **The problem is that the body of the FOR-command modifies the controlled variable**
- **This is why it was explicitly assumed that the body didn't modify the controlled variable**

Problems with the FOR-rule (vi)

- **This problem may also arise if any variables in the expressions E_1 and E_2 are modified**
- **For example, taking P to be `Z=Y`, then**

$$\vdash \{\underbrace{\texttt{Z=Y}}_{P}\}\ \texttt{Z:=Z+1}\ \{\underbrace{\texttt{Z=Y+1}}_{P[\texttt{Y+1/Y}]}\}$$

- **Thus the following can be derived**

$$\vdash \{\underbrace{\texttt{Z=1}}_{P[\texttt{1/Y}]}\ \wedge 1 \le \texttt{Z}\}\ \texttt{FOR Y:=1 UNTIL Z DO Z:=Z+1}\ \{\underbrace{\texttt{Z=Z+1}}_{P[\texttt{Z+1/Y}]}\}$$

- **This is clearly wrong**
  - **one can never have `Z=Z+1`**
- **Not a problem because the FOR-command doesn't terminate?**
  - **in some languages this might be the case**
  - **semantics of our language defined so that FOR-commands always terminate**

The FOR-Rule

- **To rule out the problems that arise when the controlled variable or variables in the bounds expressions, are changed by the body, we simply impose a side condition on the rule that stipulates that it cannot be used in these situations**

The FOR-rule

$$\frac{\vdash \{P \wedge (E_1 \le V) \wedge (V \le E_2)\}\ C\ \{P[V\texttt{+1}/V]\}}{\vdash \{P[E_1/V] \wedge (E_1 {\le} E_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[E_2\texttt{+1}/V]\}}$$

where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

- **Note $(E_1 \le V) \wedge (V \le E_2)$ in precondition of rule hypothesis**
  - **added to strengthen rule to allow proofs to use facts about V's range of values**
- **Can be tricky to think up P**

Comment on the FOR-Rule

- **The FOR-rule does not enable anything to be deduced about FOR-commands whose body assigns to variables in the bounds expressions**
- **This precludes such assignments being used if commands are to be reasoned about**
- **Only defining rules of inference for non-tricky uses of constructs motivates writing programs in a perspicuous manner**
- **It is possible to devise a rule that does cope with assignments to variables in bounds expressions**
- **Consider:**

$$\frac{\vdash \{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ C\ \{P[V\texttt{+1}/V]\}}{\vdash \{P[E_1/V] \wedge (E_1 \leq E_2) \wedge (E_2 = e_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[e_2\texttt{+1}/V]\}}$$

The FOR-axiom

- **To cover the case when $E_2 < E_1$, we need the FOR-axiom below**

The FOR-axiom

$$\vdash\ \{P \wedge (E_2 < E_1)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\}$$

- **This says that when E_2 is less than E_1 the FOR-command has no effect**

Exercise: understand the example on this slide

- **By the assignment axiom and precondition strengthening**

```
⊢ {X = ((N-1)×N) DIV 2} X:=X+N {X=(N×(N+1)) DIV 2}
```

- **Strengthening the precondition of this again yields**

```
⊢ {(X = ((N-1)×N) DIV 2)∧(1≤N)∧(N≤M)}
   X:=X+N
  {X = (N×(N+1)) DIV 2}
```

- **Hence by the FOR-rule**

```
⊢ {(X = ((1-1)×1) DIV 2)∧(1≤M)}
   FOR N:=1 UNTIL M DO X:=X+N
 {X = (M×(M+1)) DIV 2}
```

- **Hence**

```
⊢ {(X=0)∧(1≤M)}
   FOR N:=1 UNTIL M DO X:=X+N
  {X = (M×(M+1)) DIV 2}
```

Note on using the FOR-Rule

- **Note that if any of the following hold**

(i) $\vdash\ \{P\}\ C\ \{P[V\texttt{+1}/V]\}$
(ii) $\vdash\ \{P\ \wedge\ (E_1 \leq V)\}\ C\ \{P[V\texttt{+1}/V]\}$
(iii) $\vdash\ \{P\ \wedge\ (V \leq E_2)\}\ C\ \{P[V\texttt{+1}/V]\}$

- **Then by precondition strengthening:**

$\vdash\ \{P\ \wedge\ (E_1 \leq V)\ \wedge\ (V \leq E_2)\}\ C\ \{P[V\texttt{+1}/V]\}$

- **So any of (i), (ii) or (iii) above is a sufficient hypothesis for FOR Rule**

Deriving the For Rule

- The following is a command equivalent to `FOR I:=E`$_1$ `UNTIL E`$_2$ `DO C`

```
BEGIN
 VAR I;
 VAR UpperBound;
 I := E₁;
 UpperBound := E₂;
 WHILE I≤UpperBound DO (C; I := I+1)
END
```

  - `UpperBound` is assumed to be a 'new' variable
  - and `I` is not assigned to inside `C`
- Thus we could derive a rule from the implementation
  - we must be sure the implementation is correct
- Exercise: try deriving the `FOR`-rule from the `WHILE`-rule

Exercise: think about Wickerson's `FOR`-Rule (see below)

The FOR rule as presented in the notes had always seemed quite unsatisfactory to me, because it couldn't deal with the case when the lower and upper bounds on the looping variable were the wrong way around (hence the need for the FOR-axiom).

I have derived a new rule, which removes the need for the FOR-axiom completely. This rule doesn't suffer from the problems that the early incarnations of the FOR-rule suffered from in the lecture notes, and I believe the rule to be equally powerful.

It is derived, very easily, by noting that: FOR V:=E1 UNTIL E2 DO C is equivalent to:

```
 BEGIN VAR V; V:=E1; WHILE V<=E2 DO (C; V:=V+1) END
```

(where the standard syntactic constraints still apply, i.e. neither V nor any variable in E1 or E2 may be assigned to in C). Then we simply apply the Floyd-Hoare rules of blocks, sequencing and while-commands to derive the following rule:

```
   |- P ==> R[E1/V]   |- R & V>E2 ==> Q   |- {R & V<=E2} C {R[V+1/V]}
   ------------------------------------------------------------------
              |- {P} FOR V:=E1 UNTIL E2 DO C {Q}
```

This rule is similar to, but subtly different from, the FOR-rule derived in the notes. I've tried my rule on various examples in the notes, and I reckon it works fine. I'll just give one quick example here; suppose we want to prove:

```
 {X = 2} FOR V := 10 UNTIL 0 DO X:=1 {X=2}
```

Then we set R = (V=10 & X=2). The three antecedents of the (new) rule are instantiated to

```
 (1) X=2 ==> 10=10 & X=2
 (2) V=10 & X=2 & V>0 ==> X=2
 (3) |- {V=10 & X=2 & V<=0} X:=1 {V+1=10 & X=2}
```

Note that (1) and (2) are trivially true, and (3) holds because the precondition is unsatisfiable (V cannot be both equal to 10 and no greater than 0).

Soundness and Completeness

- It is clear from the discussion of the `FOR`-rule that it is not always straightforward to devise correct rules of inference
- It is important that the axioms and rules be sound. There are two approaches to ensure this

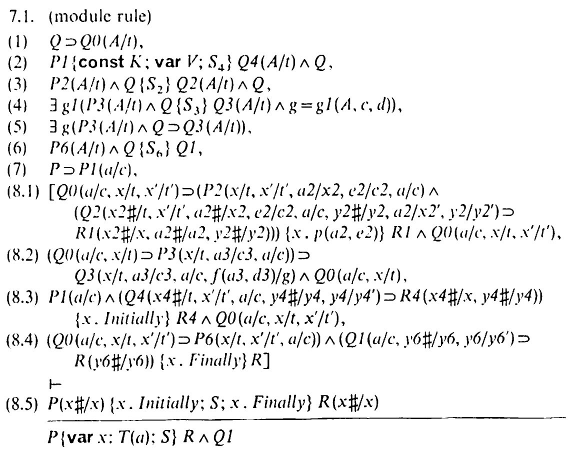
  (i) define the language by the axioms and rules of the logic
  (ii) prove that the logic fits the language

- Approach (i) is called *axiomatic semantics*
  - the idea is to *define* the semantics of the language by requiring that it make the axioms and rules of inference true
  - it is then up to implementers to ensure that the logic matches the language

Axiomatic Semantics

- One snag with axiomatic semantics is that most existing languages have already been defined in some other way
  - usually by informal and ambiguous natural language statements
- The other snag with axiomatic semantics is that it is known to be impossible to devise adequate Floyd-Hoare logics for certain constructs (Clarke's Theorem)
  - it could be argued that this is not a snag at all but an advantage, because it forces programming languages to be made logically tractable
- An example of a language defined axiomatically is Euclid

From *Proof rules for the programming language Euclid*



7.1. (module rule)

(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) $[Q0(a/c, x/t, x'/t') \supset (P2(x/t, x'/t', a2/x2, e2/c2, a/c) \wedge$
$(Q2(x2\sharp/t, x'/t', a2\sharp/x2, e2/c2, a/c, y2\sharp/y2, a2/x2', y2/y2') \supset$
$R1(x2\sharp/x, a2\sharp/a2, y2\sharp/y2)))\ \{x\,.\,p(a2, e2)\}\ R1 \wedge Q0(a/c, x/t, x'/t')$,

(8.2) $(Q0(a/c, x/t) \supset P3(x/t, a3/c3, a/c)) \supset$
$Q3(x/t, a3/c3, a/c, f(a3, d3)/g) \wedge Q0(a/c, x/t)$,

(8.3) $P1(a/c) \wedge (Q4(x4\sharp/t, x'/t', a/c, y4\sharp/y4, y4/y4') \supset R4(x4\sharp/x, y4\sharp/y4))$
$\{x\,.\,\mathit{Initially}\}\ R4 \wedge Q0(a/c, x/t, x'/t')$,

(8.4) $(Q0(a/c, x/t, x'/t') \supset P6(x/t, x'/t', a/c)) \wedge (Q1(a/c, y6\sharp/y6, y6/y6') \supset$
$R(y6\sharp/y6))\ \{x\,.\,\mathit{Finally}\}\ R]$

$\vdash$

$$\frac{(8.5)\quad P(x\sharp/x)\ \{x\,.\,\mathit{Initially};\ S;\ x\,.\,\mathit{Finally}\}\ R(x\sharp/x)}{P\{\textbf{var}\ x\colon T(a);\ S\}\ R \wedge Q1}$$

Proving the Logic Sound

- **The other approach to ensuring the logic is sound is to prove that the axioms and rules are valid**
- **To do this, a mathematical model of states is constructed**
  - **a function, $\mathit{Meaning}$ say, is defined which takes an arbitrary command C to a function $\mathit{Meaning}(C)$ from states to states**
  - **thus $\mathit{Meaning}(C)(s)$ denotes the state resulting from executing command C in state s**
- **The specification $\{P\}C\{Q\}$ is then defined to be true if**
  - **whenever P is true in a state s**
  - **and $\mathit{Meaning}(C)(s) = s'$**
  - **then Q is true in state s'**
- **It is then possible to attempt to prove rigorously that all the axioms are true and that the rules of inference lead from true premisses to true conclusions**

Completeness

- **Even if we are sure that our logic is sound, how can we be sure that every true specification can be proved?**
- **Maybe for some particular `P`, `Q` and `C` the specification `{P}C{Q}` is true, but the rules of our logic were too weak to prove it. Consider:**
  - `{X=1} BEGIN VAR X; X:=0 END {X=1}`
  - `{T} FOR X:=1 UNTIL 1 DO Y:=X {Y=1}`
- **A logic is said to be *complete* if every true statement in it is provable**

Completeness of Hoare Logic

- **Consider `{T} IF` S `THEN X := 1 ELSE X := 0 {X = 1}`**
  - **S could be an arbitrary statement of arithmetic**
  - **arithmetic is known to be undecidable**
- **There are various subtle technical problems in formulating precisely what it means for a Floyd-Hoare logic to be complete**
  - **it is necessary to distinguish incompleteness arising due to incompleteness in the assertion language (e.g. arithmetic)**
  - **from incompleteness due to inadequate axioms and rules for programming language constructs**
  - **the completeness of a Floyd-Hoare logic must thus be defined independently of that of its assertion language**
- **The correct notion of completeness is called ‘relative completeness’**
  - ***roughly*: complete assuming an oracle for logic and mathematical theories**

Clarke's Results

- Ed Clarke (of Carnegie Mellon University) has proved the impossibility of giving complete inference systems for certain combinations of programming language constructs
- He shows it is *impossible* to give a sound and relatively complete system for any language combining:
  - procedures as parameters of procedure calls
  - recursion
  - static scopes
  - global variables
  - internal procedures as parameters of procedure calls
- These features are found in Algol 60, which thus cannot have an adequate Floyd-Hoare logic

Summary

- Devising rules for programming constructs can be tricky
  - for safety critical applications logically tractable languages should be used
- Two desirable properties of the logic are:
  - soundness: only true things can be proved
  - completeness: everything that is true can be proved
- Fully automatic program verification is impossible
  - arithmetic is undecidable
  - in practice a lot of success verifying weak properties automatically (e.g. `http://research.microsoft.com/en-us/um/redmond/groups/rise/staticanalysis.aspx`)
- User-assisted program verification is active research area

Program Refinement

Refinement

- So far we have focused on proving programs meet specifications
- An alternative is to ensure a program is correct by construction
- The proof is performed in conjunction with the development
  - errors are spotted earlier in the design process
  - the reasons for design decisions are available
- Programming becomes less of a black art
  and more like an engineering discipline
- Rigorous development methods such as the B-Method, SPARK and the Vienna Development Method (VDM) are based on this idea
- The approach here is based on "Programming From Specifications"
  - a book by Carroll Morgan
  - simplified and with a more concrete semantics

Refinement Laws

- Laws of Programming refine a specification to a program
- As each law is applied, proof obligations are generated
- The laws are derived from the Hoare logic rules
- Several laws will be applicable at a given time
  - corresponding to different design decisions
  - and thus different implementations
- The "Art" of Refinement is in choosing appropriate laws to give an efficient implementation
- For example, given a specification that an array should be sorted:
  - one sequence of laws will lead to Bubble Sort
  - a different sequence will lead to Insertion Sort
  - see Morgan's book for an example of this

Refinement Specifications

- A *refinement specification* has the form $[P,\ Q]$
  - P is the precondition
  - Q is the postcondition
- Unlike a partial or total correctness specification, a refinement specification does not include a command
- Goal: derive a command that satisfies the specification
- P and Q correspond to the pre and post condition of a total correctness specification
- A command is required which if started in a state satisfying P, will terminate in a state satisfying Q

Example

- `[T, X=1]`
  - this specifies that the code provided should terminate in a state where `X` has value `1` whatever state it is started in
- [`X>0, Y=X`^{2}]
  - from a state where `X` is greater than zero, the program should terminate with `Y` the square of `X`

A Little Wide Spectrum Programming Language

- Let P, Q range over statements (predicate calculus formulae)
- Add specifications to commands

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

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

```
C ::= SKIP
    | V := E
    | C₁ ; C₂
    | IF B THEN C₁ ELSE C₂
    | BEGIN VAR V₁ ; .. VAR V₁ ; C END
    | WHILE B DO C
    | [P, Q]
```

Specifications as Sets of Commands

- Refinement specifications can be mixed with other commands but are not in general executable
- Example

```
R:=X;
Q:=0;
[R=X ∧ Y>0 ∧ Q=0,  X=R+Y×Q]
```

- Think of a specification as defining the set of implementations

$$[P,\ Q] \ = \ \{\ C \ \mid \ \ \vdash \ [P]\ C\ [Q]\ \}$$

- For example

$$[\mathtt{T},\ \mathtt{X=1}] \ = \ \{\texttt{"X:=1"},\ \texttt{"IF }\neg\texttt{(X=1) THEN X:=1"},\ \texttt{"X:=2;X:=X-1"},\ \cdots\ \}$$

Notation for combining sets of commands

- Wide spectrum language commands are sets of ordinary commands
- Let c, c_1, c_2 etc. denote sets of commands, then define:

$$\begin{array}{ll}
c_1;\ \cdots\ ;c_n & = \{\ C \ \mid \ \exists C_1 \cdots C_n.\ C \ = \ C_1;\ \cdots\ ;C_n \ \wedge \\
 & \qquad\qquad C_1 \in c_1 \ \wedge\ \cdots\ \wedge\ C_n \in c_n\ \} \\
 & = \{\ C_1;\ \cdots\ ;C_n \ \mid\ C_1 \in c_1 \ \wedge\ \cdots\ \wedge\ C_n \in c_n\ \} \\
\mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ c\ \mathtt{END} & = \{\ \mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ C\ \mathtt{END} \ \mid\ C \in c\ \} \\
\mathtt{IF}\ S\ \mathtt{THEN}\ c_1\ \mathtt{ELSE}\ c_2 & = \{\ \mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2 \ \mid\ C_1 \in c_1 \ \wedge\ C_2 \in c_2\ \} \\
\mathtt{WHILE}\ S\ \mathtt{DO}\ c & = \{\ \mathtt{WHILE}\ S\ \mathtt{DO}\ C \ \mid\ C \in c\ \}
\end{array}$$

Refinement based program development

- The client provides a non-executable program (the specification)
- The programmer's job is to transform it into an executable program
- It will pass through a series of stages in which some parts are executable, but others are not
- Specifications give lots of freedom about how a result is obtained
  - executable code has no freedom
  - mixed programs have some freedom
- We use the notation $p_1 \supseteq p_2$ to mean program p_2 is more refined (i.e. has less freedom) than program p_1
- N.B. The standard notation is $p_1 \sqsubseteq\ p_2$ (see the notes for a discussion)
- A program development takes us from the specification, through a series of mixed programs to (we hope) executable code

$$spec \supseteq\ mixed_1 \supseteq\ ... \supseteq\ mixed_n \supseteq code$$

Skip Law

The Skip Law

$$[P,\ \ P] \supseteq \{\mathtt{SKIP}\}$$

- Derivation:

$$\begin{array}{l}
C \in \{\mathtt{SKIP}\} \\
\quad \Leftrightarrow\ C \ \ = \ \ \mathtt{SKIP} \\
\quad \Rightarrow\ \ \vdash\ [P]\ C\ [P] \ \ \text{(Skip Axiom)} \\
\quad \Leftrightarrow\ C \in [P,\ P] \quad \text{(Definition of } [P,\ P])
\end{array}$$

- Examples

$$[\mathtt{X=1},\ \ \mathtt{X=1}] \ \supseteq\ \{\mathtt{SKIP}\}$$

$$[\mathtt{T},\ \ \mathtt{T}] \ \supseteq\ \{\mathtt{SKIP}\}$$

$$[\mathtt{X=R+Y}\times\mathtt{Q},\ \ \mathtt{X=R+Y}\times\mathtt{Q}] \ \supseteq\ \{\mathtt{SKIP}\}$$

Notational Convention

- **Omit { and } around individual commands**
- **Skip law becomes:**

$$[P,\ \ P] \supseteq \texttt{SKIP}$$

- **Examples become:**

```
[X=1,  X=1] ⊇ SKIP
[T,  T] ⊇ SKIP
[X=R+Y×Q,  X=R+Y×Q] ⊇ SKIP
```

Assignment Law

The Assignment Law

$$[P[E/V],\ \ P] \supseteq \{V \texttt{ := } E\}$$

- **Derivation**

$$\begin{array}{l} C \in \{V \texttt{ := } E\} \\ \quad \Leftrightarrow C \ = \ V \texttt{ := } E \\ \quad \Rightarrow\ \vdash\ [P\texttt{[}E/V\texttt{]}]\ C\ [P]\ \ \text{(Assignment Axiom)} \\ \quad \Leftrightarrow C \in [P\texttt{[}E/V\texttt{]},\ P] \quad \text{(Definition of } [P\texttt{[}E/V\texttt{]},\ P]\text{)} \end{array}$$

- **Examples (using bracket-omitting convention)**

```
[Y=1,  X=1] ⊇ X:=Y
[X+1=n+1,  X=n+1] ⊇ X:=X+1
```

Laws of Consequence

Precondition Weakening

$$[P,\ \ Q] \supseteq [R,\ \ Q]$$
provided $\vdash\ \ P\ \Rightarrow\ R$

Postcondition Strengthening

$$[P,\ \ Q] \supseteq [P,\ \ R]$$
provided $\vdash\ \ R\ \Rightarrow\ Q$

- **We are now "weakening the precondition" and "strengthening the post condition"**
  - **this is the opposite terminology to the Hoare rules**
  - **refinement consequence rules are 'backwards'**

Derivation of Consequence Laws

- **Derivation of Precondition Weakening**

$$\begin{array}{l} C \in [R,\ Q] \\ \quad \Leftrightarrow\ \vdash\ [R]\ C\ [Q]\ \ \text{(Definition of } [R,\ Q]\text{)} \\ \quad \Rightarrow\ \vdash\ [P]\ C\ [Q]\ \ \text{(Precondition Strengthening } \vdash P{\Rightarrow}R\text{)} \\ \quad \Leftrightarrow C \in [P,\ Q] \quad \text{(Definition of } [P,\ Q]\text{)} \end{array}$$

- **Derivation of Postcondition Strengthening**

$$\begin{array}{l} C \in [P,\ R] \\ \quad \Leftrightarrow\ \vdash\ [P]\ C\ [R]\ \ \text{(Definition of } [R,\ Q]\text{)} \\ \quad \Rightarrow\ \vdash\ [P]\ C\ [Q]\ \ \text{(Postcondition Weakening } \vdash R{\Rightarrow}Q\text{)} \\ \quad \Leftrightarrow C \in [P,\ Q] \quad \text{(Definition of } [P,\ Q]\text{)} \end{array}$$

Examples (illustrates refinement notation)

- **A previous example:**

  `[X=1,  X=1]`
  $\supseteq$ (`Skip`)
  `SKIP`

- **An alternative refinement:**

  `[Y=1,  X=1]`
  $\supseteq$ (`Precondition Weakening` $\vdash$ `Y=1` $\Rightarrow$ `1=1`)
  `[1=1,  X=1]`
  $\supseteq$ (`Assignment`)
  `X := 1`

- **Another example**

  `[T,  R=X]`
  $\supseteq$ (`Precondition Weakening` $\vdash$ `T` $\Rightarrow$ `X=X`)
  `[X=X,  R=X]`
  $\supseteq$ (`Assignment`)
  `R := X`

Derived Assignment Law

Derived Assignment Law

$$[P,\ \ Q] \supseteq \{V\texttt{:=}E\}$$

provided $\vdash\ P\ \Rightarrow\ Q[E/V]$

- **Derivation**

  `[`P`,  `Q`]`
  $\supseteq$ (`Precondition Weakening` $\vdash\ P \Rightarrow Q[E/V]$)
  `[`$Q[E/V]$`,  `Q`]`
  $\supseteq$ (`Assignment`)
  V`:=`E

- **Example**

  `[T,  R=X]`
  $\supseteq$ (`Derived Assignment` $\vdash$ `T` $\Rightarrow$ `X=X`)
  `R := X`

Technical interlude: Monotonicity

- **Sets of commands are** *monotonic* **w.r.t.** $\supseteq$
  - **if** $c\supseteq c'$, $c_1\supseteq c_1'$, … , $c_n\supseteq c_n'$
  - **then:**

    $c_1;\ \cdots\ ;c_n \supseteq c_1';\ \cdots\ ;c_n'$

    `BEGIN VAR` V_1`;` $\cdots$ `VAR` V_n`;` c `END` $\supseteq$ `BEGIN VAR` V_1`;` $\cdots$ `VAR` V_n`;` c' `END`

    `IF` S `THEN` c_1 `ELSE` c_2 $\supseteq$ `IF` S `THEN` c_1' `ELSE` c_2'

    `WHILE` S `DO` c $\supseteq$ `WHILE` S `DO` c'

- **Monotonicity shows that a command can be refined by separately refining its constituents**

- **Laws of refinement now follow**

Sequencing

The Sequencing Law

$$[P,\ \ Q] \supseteq [P,\ \ R];\ [R,\ \ Q]$$

- **Derivation of Sequencing Law**

$$\begin{array}{ll}
C \in [P,\ R]\ ;\ [R,\ Q] & \\
\quad \Leftrightarrow\ C \in \{\ C_1\ ;\ C_2\ \mid\ C_1 \in [P,\ R]\ \wedge\ C_2 \in [R,\ Q]\} & \text{(Definition of } c_1\ ;\ c_2\text{)} \\
\quad \Leftrightarrow\ C \in \{\ C_1\ ;\ C_2\ \mid\ \ \vdash\ [P]\ C_1\ [R]\ \wedge\ \ \vdash\ [R]\ C_2\ [Q]\} & \text{(Definition of } [P,\ R] \text{ and } [R,\ Q]\text{)} \\
\quad \Rightarrow\ C \in \{\ C_1\ ;\ C_2\ \mid\ \ \vdash\ [P]\ C_1\ ;\ C_2\ [Q]\} & \text{(Sequencing Rule)} \\
\quad \Rightarrow\ \ \vdash\ [P]\ C\ [Q] & \\
\quad \Leftrightarrow\ C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

- **Example**

  `[T,  R=X∧Q=0]`
  $\supseteq$ (`Sequencing`)
  `[T,  R=X] ; [R=X,  R=X∧Q=0]`
  $\supseteq$ (`Derived Assignment` $\vdash$ `T` $\Rightarrow$ `X=X`)
  `R:=X; [R=X,  R=X∧Q=0]`
  $\supseteq$ (`Derived Assignment` $\vdash$ `R=X` $\Rightarrow$ `R=X` $\wedge$ `0=0`)
  `R:=X; Q:=0`

Creating different Programs

- By applying the laws in a different way, we obtain different programs
- Consider previous example: using a different assertion with the sequencing law creates a program with the assignments reversed

```
[T,  R=X∧Q=0]
⊒ (Sequencing)
[T,  Q=0] ; [Q=0,  R=X∧Q=0]
⊒ (Derived Assignment  ⊢  T ⇒ 0=0)
Q:=0; [Q=0,  R=X∧Q=0]
⊒ (Derived Assignment  ⊢  Q=0 ⇒ X=X ∧ Q=0)
Q:=0; R:=X
```

Inefficient Programs

- Refinement does not prevent you making silly coding decisions
- It does prevent you from producing incorrect executable code
- Example

```
[T,  R=X∧Q=0]
⊒ (Sequencing)
[T,  R=X∧Q=0] ; [R=X∧Q=0,  R=X∧Q=0]
⊒ (as previous example)
Q:=0; R:=X; [R=X∧Q=0,  R=X∧Q=0]
⊒ (Skip)
Q:=0; R:=X; SKIP
```

Blind Alleys

- The refinement rules give the freedom to wander down blind alleys
- We may end up with an unrefinable step
  - since it will not be executable, this is safe
  - we will not get an incorrect executable program
- Example

```
[X=x∧Y=y,  X=y∧Y=x]
⊒ (Sequencing)
[X=x∧Y=y,  X=x∧Y=x] ; [X=x∧Y=x,  X=y∧Y=x]
⊒ (Derived Assignment  ⊢ X=x∧Y=y⇒X=x∧X=x)
Y:=X; [X=x∧Y=x,  X=y∧Y=x]
⊒ (Sequencing)
Y:=X;
[X=x∧Y=x,  Y=y∧Y=x];
[Y=y∧Y=x,  X=y∧Y=x]
⊒ (Assignment)
Y:=X;
[X=x∧Y=x,  Y=y∧Y=x];
X:=Y
```

Blocks

The Block Law

$[P,\ \ Q] \sqsupseteq$ `BEGIN VAR` V_1`;` $\ldots$ `; VAR` V_n`;` $[P,\ \ Q]$ `END`

where $V_1, \ldots, V_n$ **do not occur in** P **or** Q

- Derivation: exercise (see notes)
- Example

```
[X=x∧Y=y,  X=y∧Y=x]
⊒ (Block)
BEGIN VAR R; [X=x∧Y=y,  X=y∧Y=x] END
⊒ (Sequencing and Derived Assignment)
BEGIN VAR R; R:=X; X:=Y; Y:=R END
```

Conditional

The Conditional Law

$$[P,\quad Q] \supseteq \texttt{IF}\ S\ \texttt{THEN}\ [P \wedge S,\quad Q]\ \texttt{ELSE}\ [P \wedge \neg S,\quad Q]$$

- **The Conditional Law can be used to refine *any* specification and *any* test can be introduced**
- **You may not make any progress by applying the law however**
  - **you may need the same program on each branch!**

Derivation of the Conditional Law

$C \in \texttt{IF}\ \ S\ \texttt{THEN}\ [P{\wedge}S,\ Q]\ \texttt{ELSE}\ [P{\wedge}\neg S,\ Q]$

$\Leftrightarrow\ C \in \{\texttt{IF}\ \ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \mid$
$\quad C_1 \in [P{\wedge}S,\ Q]\ \&\ C_2 \in [P{\wedge}\neg S,\ Q]\}$ (Definition of $\texttt{IF}\ \ S\ \texttt{THEN}\ \{\cdots\}\ \texttt{ELSE}\ \{\cdots\}$)

$\Leftrightarrow\ C \in \{\texttt{IF}\ \ S\ \texttt{THEN}\ C_1\ \texttt{THEN}\ C_2 \mid$
$\quad \vdash\ [P{\wedge}S]\ C_1\ [Q]\ \&\ \vdash\ [P{\wedge}\neg S]\ C_2\ [Q]\}$ (Definition of $[P{\wedge}S,\ Q]\ \&\ [P{\wedge}\neg S,\ Q]$)

$\Rightarrow\ C \in \{\texttt{IF}\ \ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \mid$
$\quad \vdash\ [P]\ \texttt{IF}\ \ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]\}$ (Two-armed Conditional Rule)

$\Rightarrow\ \vdash\ [P]\ C\ [Q]$

$\Leftrightarrow\ C \in [P,\ Q]$ (Definition of $[P,\ Q]$)

Example

```
[T,  M=max(X,Y)]
⊇ (Conditional)
IF X≥Y
THEN [T∧ X≥Y,  M=max(X,Y)]
ELSE [T∧¬(X≥Y),  M=max(X,Y)]
⊇ (Derived Assignment  ⊢ T∧X≥Y ⇒ X=max(X,Y))
IF X≥Y
THEN M:=X
ELSE [T∧¬X≥Y,  M=max(X,Y)]
⊇ ( Derived Assignment ⊢ T∧¬X≥Y ⇒ Y=max(X,Y))
IF X≥Y THEN M:=X ELSE M:=Y
```

While

The While Law

$$[R,\quad R \wedge \neg S] \supseteq \texttt{WHILE}\ S\ \texttt{DO}\ [R \wedge S \wedge (E{=}n),\quad R \wedge (E{<}n)]$$

provided $\vdash\quad R \wedge S \Rightarrow\ \ E \geq 0$

and where E is an integer-valued expression and n is an identifier not occurring in P, S, E or C.

- **Example**

```
[X=R+Y×Q ∧ Y>0,  X=R+Y×Q ∧ Y>0 ∧ ¬ Y≤R]
⊇ (While  ⊢  X=R+Y×Q ∧ Y>0 ∧ Y≤R ⇒ R≥0)
WHILE Y≤R DO
   [X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n,
    X=R+Y×Q ∧ Y>0 ∧ R<n]
```

Derivation of the While Law

$C \in \texttt{WHILE}\ S\ \texttt{DO}\ [P \wedge S \wedge (E = n),\ P \wedge (E < n)]$

$\Leftrightarrow\ C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid C' \in [P \wedge S \wedge (E = n),\ P \wedge (E < n)]\}$ (Definition of $\texttt{WHILE}\ S\ \texttt{DO}\ \{\cdots\}$)

$\Leftrightarrow\ C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid\ \vdash [P \wedge S \wedge (E = n)]\ C'\ [P \wedge (E < n)]\}$ (Definition of $[P \wedge S \wedge (E = n),\ P \wedge (E < n)]$)

$\Rightarrow\ C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid\ \vdash [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C'\ [P \wedge \neg S]\}$ (While Rule & $\vdash\ P \wedge S \Rightarrow E \geq 0$)

$\Rightarrow\ \vdash [P]\ C\ [P \wedge \neg S]$

$\Leftrightarrow\ C \in [P,\ P \wedge \neg S]$ (Definition of $[P,\ P \wedge \neg S]$)

Example (i)

```
[Y>0,  X=R+Y×Q ∧ R ≤ Y]
⊇ (Block)
BEGIN [Y>0,  X=R+Y×Q ∧ R ≤ Y] END
⊇ (Sequencing)
BEGIN
[Y>0,  R=X ∧ Y>0] ;
[R=X ∧ Y>0,  X=R+Y×Q ∧ R ≤ Y]
END
⊇ (Derived Assignment  ⊢  Y>0 ⇒ X=X ∧ Y>0)
BEGIN
R:=X ;
[R=X ∧ Y>0,  X=R+Y×Q ∧ R ≤ Y]
END
```

Example (ii)

```
BEGIN
R:=X ;
[R=X ∧ Y>0,  X=R+Y×Q ∧ R ≤ Y]
END ⊇ (Sequencing)
BEGIN
R:=X ;
[R=X ∧ Y>0,  R=X ∧ Y>0 ∧ Q=0] ;
[R=X ∧ Y>0 ∧ Q=0,  X=R+Y×Q ∧ R ≤ Y]
END
⊇ (Derived Assignment  ⊢  R=X ∧ Y>0 ⇒ R=X ∧ Y>0 ∧ 0=0)
BEGIN
R:=X ;
Q:=0 ;
[R=X ∧ Y>0 ∧ Q=0,  X=R+Y×Q ∧ R ≤ Y]
END
```

- **Exercise: complete the refinement (see next few slides)**

Example (iii)

```
⊇ ( Precondition Weakening
        ⊢ R=X ∧ Y>0 ∧ Q=0 ⇒ X=R+Y×Q ∧ Y>0 )
BEGIN
R:=X; Q:=0;
[X=R+Y×Q ∧ Y>0,  X=R+Y×Q ∧ R ≤ Y]
END
⊇ ( Postcondition Strengthening
        ⊢ X=R+Y×Q ∧ Y>0 ∧ ¬(Y≤R)
              ⇒ X=R+Y×Q ∧ R≤Y )
BEGIN
R:=X; Q:=0;
[X=R+Y×Q ∧ Y>0,  X=R+Y×Q ∧ Y>0 ∧ ¬(Y≤R)]
END
⊇ (While  ⊢ X=R+Y×Q ∧ Y>0 ∧ Y≤R ⇒ R≥0)
BEGIN
R:=X; Q:=0;
WHILE Y ≤ R DO
   [X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n,
    X=R+Y×Q ∧ Y>0 ∧ R<n]
END
```

Example (iv)

```
⊇ (Block)
BEGIN
R:=X; Q:=0;
WHILE Y ≤ R DO
 BEGIN
  [X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n,
   X=R+Y×Q ∧ Y>0 ∧ R<n]
 END
END
⊇ (Sequence)
BEGIN
R:=X; Q:=0;
WHILE Y ≤ R DO
 BEGIN
  [X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n,
   X=(R-Y)+Y×Q ∧ Y>0 ∧ (R-Y)<n] ;
  [X=(R-Y)+Y×Q ∧ Y>0 ∧ (R-Y)<n,
   X=R+Y×Q ∧ Y>0 ∧ R<n]
 END
END
```

Example (v)

```
⊇ (Assignment)
BEGIN
R:=X; Q:=0;
WHILE Y ≤ R DO
 BEGIN
  [X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n,
   X=(R-Y)+Y×Q ∧ Y>0 ∧ (R-Y)<n] ;
  R := R-Y
 END
END
  ⎛ Derived Assignment                       ⎞
⊇ ⎜     ⊢ X=R+Y×Q ∧ Y>0 ∧ Y≤R ∧ R=n ⇒         ⎟
  ⎝       X=(R-Y)+Y×(Q+1) ∧ Y>0 ∧ (R-Y)<n    ⎠
BEGIN
R:=X; Q:=0;
WHILE Y ≤ R DO
 BEGIN
  Q:= Q+1;
  R:= R-Y
 END
END
```

More Notation

- The notation

$$[P_1,\ P_2,\ P_3,\ \cdots,P_{n-1},\ P_n]$$

  is used to abbreviate:

$$[P_1,\ P_2]\ ;\ [P_2,\ P_3]\ ;\ \cdots\ ;\ [P_{n-1},\ P_n]$$

- Brackets around specifications $\{C\}$ omitted

- If $\mathcal{C}$ is a set of commands, then

$$R := X\ ;\ \mathcal{C}$$

  abbreviates

$$\{R := X\}\ ;\ \mathcal{C}$$

Exercise: check the refinement on this slide

- Let $\mathcal{I}$ stand for $X = R + (Y \times Q)$, then:

$$\begin{array}{l}
[Y > 0,\ \ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Sequencing)} \\
[Y > 0,\ R = X \wedge Y > 0,\ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Assignment)} \\
R := X\ ;\ [R = X \wedge Y > 0,\ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Sequencing)} \\
R := X\ ;\ [R = X \wedge Y > 0,\ R = X \wedge Y > 0 \wedge Q = 0,\ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Assignment)} \\
R := X\ ;\ Q := 0\ ;\ [R = X \wedge Y > 0 \wedge Q = 0,\ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Precondition Weakening)} \\
R := X\ ;\ Q := 0\ ;\ [\mathcal{I} \wedge Y > 0,\ \mathcal{I} \wedge R \leq Y] \\
\supseteq \text{ (Postcondition Strengthening)} \\
R := X\ ;\ Q := 0\ ;\ [\mathcal{I} \wedge Y > 0,\ \mathcal{I} \wedge Y > 0 \wedge \neg(Y \leq R)] \\
\supseteq \text{ (While)} \\
R := X\ ;\ Q := 0\ ; \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I} \wedge Y > 0 \wedge Y \leq R \wedge R = n, \\
\qquad\qquad\qquad \mathcal{I} \wedge Y > 0 \wedge R < n] \\
\supseteq \text{ (Sequencing)} \\
R := X\ ;\ Q := 0\ ; \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I} \wedge Y > 0 \wedge Y \leq R \wedge R = n, \\
\qquad\qquad\qquad X = (R - Y) + (Y \times Q) \wedge Y > 0 \wedge (R - Y) < n, \\
\qquad\qquad\qquad \mathcal{I} \wedge Y > 0 \wedge R < n] \\
\supseteq \text{ (Derived Assignment)} \\
R := X\ ;\ Q := 0\ ; \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I} \wedge Y > 0 \wedge Y \leq R \wedge R = n, \\
\qquad\qquad\qquad X = (R - Y) + (Y \times Q) \wedge Y > 0 \wedge (R - Y) < n]; \\
\qquad\qquad\qquad R := R - Y \\
\supseteq \text{ (Derived Assignment)} \\
R := X\ ;\ Q := 0\ ; \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ Q := Q + 1\ ;\ R := R - Y
\end{array}$$

Derived Laws

- Above development could be shortened by deriving appropriate laws
- For example, a derived `WHILE` law could be derived
- Exercise: Develop a factorial program from the specification:

$$[X = n, \quad Y = n!]$$

- Exercise: devise refinement laws for arrays, one-armed conditionals, and `FOR`-commands

Data Refinement

- So far we have given laws to refine commands
- This is termed *Operation Refinement*
- It is also useful to be able to refine the representation of data
  - replacing an abstract data representation by a more concrete one
  - e.g. replacing numbers by binary representations
- This is termed *Data Refinement*
- Data Refinement Laws allow us to make refinements of this form
- The details are beyond the scope of this course
  - they can be found in Morgan's book

Summary

- Refinement 'laws' based on the Hoare logic can be used to develop programs formally
- A program is gradually converted from an unexecutable specification to executable code
- By applying different laws, different programs are obtained
  - may reach unrefinable specifications (blind alleys)
  - but will never get incorrect code
- A program developed in this way will meet its formal specification
  - one approach to 'Correct by Construction' (CbC) software engineering

New Topic!

Higher-Order Logic

Introduction

- **Specifications in first order logic can be clumsy**
- **Higher-Order Logic is widely used since**
  - it is more expressive than first-order logic
  - specifications are often more natural
  - they are less likely to contain mistakes and are easier to understand
- **The main differences between first and higher-order logic are**
  - no distinction is made between terms and statements
  - functions and predicates are treated as first class objects
- **Higher-order logic is a foundational system like set theory**
  - can develop mathematics inside higher-order logic

Overview

- **Basic notation**
- **Types**
- **Special syntax to make notation user-friendly**
- **Definitions**

Higher-Order Logic

- **Familiar notation:**

Term	*Meaning*
$P(x)$	x has property P
$\neg t$	not t
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \vee t_2$	t_1 or t_2
$t_1 \Rightarrow t_2$	t_1 implies t_2
$\forall x.\ t[x]$	for all x it is the case that $t[x]$
$\exists x.\ t[x]$	for some x it is the case that $t[x]$

- **Higher-order logic is also called:**
  - *higher-order predicate calculus*
  - *simple type theory*

Formulae Become Terms

- **In higher-order logic statements (formulae) are boolean terms**
  - i.e. terms whose value is one of the two truth values (or booleans) `T` or `F`
- **There are only four kinds of primitive terms in higher-order logic**
  - **Variables** x, y, P etc
  - **Constants** stand for fixed values (e.g. numerals 0, 1 etc)
  - **Function applications** have the general form $t_1\ t_2$ where t_1 and t_2 are terms, an example is $P\ 0$
  - **Lambda-terms** denote functions and have the form $\lambda x.\ t$ (where x is a variable and t a term)
- **Notation on previous slide is 'syntactic sugar' (explained later)**

Statements

- Statements are just terms in higher-order logic
  - $\neg t$ is the application of constant $\neg$ to term t
  - $t_1 \Rightarrow t_2$ is the infixed application of constant $\Rightarrow$ to terms t_1 and t_2
- Only a single variable binding mechanism: λ-abstraction
  - quantifiers $\forall$ and $\exists$ are regarded as constants
  - quantification syntax $\forall x.\ t$ and $\exists x.\ t$ abbreviates $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$
- Example: the term

  $\forall n.\ P(n) \Rightarrow P(n+1)$

  is written instead of

  $\forall(\lambda n.\ \Rightarrow(P(n))(P(+\ n\ 1)))$

Higher-Order Variables

- Higher-order logic generalises first order logic by allowing **higher-order** variables
  - i.e. variables ranging over functions and predicates
- The induction axiom for natural numbers can be written as:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

- The legitimacy of simple recursive definitions can be expressed by:

$$\forall n_0.\ \forall f.\ \exists s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

- Sentences like these are not allowed in first order logic
  - in the first example above P ranges over predicates
  - in the second example f and s range over functions

Floyd-Hoare Logic

- Correctness specifications can be expressed in higher-order logic
- A predicate `Spec` is defined by

$$\texttt{Spec}(p, c, q) \ = \ \forall s_1\ s_2.\ p\ s_1 \ \wedge\ c(s_1, s_2) \Rightarrow q\ s_2$$

- `Spec` is a predicate on triples (p, c, q) where p and q are unary predicates and c is a binary predicate
- `Spec` can't be defined in first order logic

Constants

Examples of constants are:

- the equals sign ($=$), the equivalence symbol ($\equiv$),
- the negation symbol ($\neg$),
- the conjunction symbol ($\wedge$), the disjunction symbol ($\vee$), the implication symbol ($\Rightarrow$),
- the universal quantifier ($\forall$), the existential quantifier ($\exists$),
- the pairing symbol (comma: $,$),
- the numerals 0, 1, 2 etc.,
- the arithmetic functions $+$, $-$, $\times$ and $/$, and
- the arithmetic relations $<$, $>$, $\leq$ and $\geq$.

Function applications

- Terms of the form $t_1\ t_2$ are called **applications** or *combinations*
  - can write $f(x)$ instead of $f\ x$
- The subterm t_1 is called the **operator** (or **rator**)
- The term t_2 is called the **operand** (or **rand** or **argument**)
- The result of such a function application can itself be a function and thus terms like $(t_1(t_2))(t_3)$ are allowed
- Functions that take functions as arguments or return functions as results are called **higher-order**
- Function applications can be written as $f\ x$ instead of $f(x)$
  - $t_1\ t_2\ t_3\ \cdots\ t_n$ abbreviates $(\ \cdots\ ((t_1\ t_2)\ t_3)\ \cdots\ t_n)$
  - i.e. application associates to the left

Lambda-terms

- An abstraction is a term of the form $\lambda x.\ t$ (where t is a term)
- $\lambda x.\ t$ denotes the function f defined by:

$$f(x)\ =\ t$$

- **Example:** $\lambda n.\ \texttt{factorial}(\texttt{square}(n))$ denotes function f where:

$$f(n)\ =\ \texttt{factorial}(\texttt{square}(n))$$

- The variable x and term t are called respectively the **bound variable** and **body** of the λ-expression $\lambda x.\ t$

Types

- Consider the predicate $\texttt{P}$ defined by:

$$\texttt{P}\ x\ =\ \neg(x\ x)$$

- From this definition it follows that:

$$\texttt{P}\ \texttt{P}\ =\ \neg(\texttt{P}\ \texttt{P})$$

- This is a version of Russell's paradox
- Russell invented **types** to prevent such inconsistencies

The Structure of Types

- Types denote sets of values: they are **atomic** or **compound**
- Two kinds of atomic types:
  - type constants, e.g. num or bool
  - type variables: α, β, γ etc
- Compound types are built using **type operators**
  - $\rightarrow$ and $\times$ are infixed binary type operators
- If σ, σ_1 and σ_2 are types then so are $\sigma_1{\rightarrow}\sigma_2$ and $\sigma_1{\times}\sigma_2$
  - $\sigma_1{\rightarrow}\sigma_2$ denotes the set of functions with domain σ_1 and range σ_2
  - $f : \sigma_1{\rightarrow}\sigma_2$ means f a function from σ_1 to σ_2
  - $\sigma_1{\times}\sigma_2$ denotes the Cartesian product type
  - $p : \sigma_1{\times}\sigma_2$ means p a pair (x, y) where $x : \sigma_1$ and $y : \sigma_2$

Well-Typed Terms

- Terms must be **well-typed**
- Typing rules are essentially those of ML
- Must be possible to assign a type to each subterm so 1 and 2 hold:

  1. For every subterm $t_1\ t_2$ there are types σ and σ' such that:
     (a) t_1 is assigned $\sigma'{\to}\sigma$
     (b) t_2 is assigned σ'
     (c) $t_1\ t_2$ is assigned the type σ.
  2. Every subterm of the form $\lambda x.\ t$ is assigned a type $\sigma_1{\to}\sigma_2$ where:
     (a) x is assigned σ_1
     (b) t is assigned σ_2.

Notation

- Writing $t{:}\sigma$ indicates that a term t has type σ
- $\sigma_1{\to}\sigma_2{\to}\sigma_3{\to}\ \cdots\ \sigma_n{\to}\sigma$ abbreviates $\sigma_1{\to}(\sigma_2{\to}(\sigma_3{\to}\ \cdots\ (\sigma_n{\to}\sigma)\ \cdots\ ))$
  - i.e. $\to$ associates to the right
- $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda x_1.\ \lambda x_2.\ \ \cdots\ \lambda x_n.\ t$
- The scope of the "." after a λ extends as far to the right as possible
  - for example, $\lambda b.\ b = \lambda x.\ \mathtt{T}$ means $\lambda b.\ (b = (\lambda x.\ \mathtt{T}))$ not $(\lambda b.\ b) = (\lambda x.\ \mathtt{T})$

Types of Constants

- All constants are assumed to have a fixed type
- If this type is polymorphic (i.e. contains type variables) then for the purposes of type checking the constant behaves as though it is assigned every instance of the type
- For example, if `twice` had type $(\alpha{\to}\alpha){\to}(\alpha{\to}\alpha)$, then the terms `twice(sqrt)` and `twice(not)` would be well-typed

Special Syntactic Forms

- Certain applications are conventionally written in special ways, for example:
  - $+\ t_1\ t_2$ is written $t_1 + t_2$
  - $,\ t_1\ t_2$ is written (t_1, t_2)
  - $\forall(\lambda x.\ t)$ is written $\forall x.\ t$
- $+$ and $,$ are examples of **infixes**
- $\forall$ is an example of a **binder**
- Some other *ad hoc* syntactic forms are also used (e.g. conditionals)

Infixes

- If `f` is an infixed constant then applications are written as t_1 `f` t_2 rather than as `f` t_1 t_2
- Standard examples of infixes are the arithmetic functions $+$, $\times$ etc.
- Whether a constant is an infix or not has no logical significance, it is merely syntactic
- Examples of infixes are the following constants:

  $\wedge :\ bool \to bool \to bool$ **(Conjunction)**
  $\vee :\ bool \to bool \to bool$ **(Disjunction)**
  $\Rightarrow :\ bool \to bool \to bool$ **(Implication)**
  $\equiv :\ bool \to bool \to bool$ **(Equivalence)**

  Equality is also an infixed constant; it is polymorphic:

  $= :\ \alpha \to \alpha \to bool$

  Equivalence ($\equiv$) is equality ($=$) restricted to booleans

Binders

- It is sometimes more readable to write `f` $x.\ t$ instead of `f`$(\lambda x.\ t)$
- For example, the quantifiers $\forall$ and $\exists$ are polymorphic constants:

  $\forall :\ (\alpha \to bool) \to bool$
  $\exists :\ (\alpha \to bool) \to bool$
- The idea is that if $P : \sigma \to bool$, then $\forall(P)$ is true if $P(x)$ is true for all x and $\exists(P)$ is true if $P(x)$ is true for some x
- Instead of writing $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$ it is nice to be able to use the more conventional forms $\forall x.\ t$ and $\exists x.\ t$
- **Example:** recall the statement of mathematical induction:

  $$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

  This is a term of type $bool$
- Without infix and binder notation terms are less readable

  $$\forall(\lambda P.\ \Rightarrow(\wedge(P\ 0)(\forall(\lambda n.\ \Rightarrow(P\ n)(P(+\ n\ 1)))))(\forall(\lambda n.\ Pn)))$$

Pairs and Tuples

- A function of n arguments can be represented as a higher-order function of 1 argument that returns a function of $n-1$ arguments
- Thus $\lambda m.\ \lambda n.\ m^2+n^2$ represents the 2 argument function that sums the squares of its arguments
- Functions of this form are called **curried**
- An alternative way of representing multiple argument functions is as single argument functions taking **tuples** as arguments
- If $t_1{:}\sigma_1$ and $t_2{:}\sigma_2$ then (t_1, t_2) has type $\sigma_1 \times \sigma_2$
  - (t_1, t_2) denotes a pair

Paired Functions

- Another representation of the sum-squares function would be as a constant, `sumsq` say, of type $(num \times num) \to num$ defined by:

  $$\texttt{sumsq}(m, n) \ = \ m^2 + n^2$$
- A term of the form (t_1, t_2) is equivalent to the term $,\ t_1\ t_2$ where "$,$" is a polymorphic infixed constant of type $\alpha \to \beta \to (\alpha \times \beta)$
- Instead of having tuples as primitive, they will be treated as iterated pairs. Thus the term:

  $$(t_1, t_2,\ \ldots\ , t_{n-1}, t_n)$$

  is an abbreviation for:

  $$(t_1, (t_2,\ \ldots\ , (t_{n-1}, t_n)\ \ldots\ ))$$

  i.e. "$,$" associates to the right
- To match this, the infixed type operator $\times$ also associates to the right so that if $t_1{:}\sigma_1$, $\ldots$, $t_n{:}\sigma_n$ then:

  $$(t_1,\ \ldots\ , t_n) : \sigma_1 \times\ \cdots\ \times \sigma_n$$

Definitions

- Definitions have the form $\vdash\ \mathtt{c} = t$ where `c` is a new constant and t is a closed term that doesn't contain `c`
- Such a definition is a particular kind of axiom that introduces the constant `c` as an abbreviation for the term t
- The requirement that `c` may not occur in t prevents definitions from being recursive, this is to rule out inconsistent 'definitions' like

$$\vdash\ \mathtt{c} = \mathtt{c} + 1$$

- **Example:** a constant `Cond` can be defined (see the notes) so that `Cond` $t\ t_1\ t_2$ means "if t then t_1 else t_2"
  - the special syntax $(t \rightarrow t_1 \mid t_2)$ is used for such terms

Function Definitions

- A function definition:

$$\vdash\ \mathtt{f}\ =\ \lambda x_1 \cdots x_n.\ t$$

  can be written as:

$$\vdash\ \mathtt{f}\ x_1 \cdots x_n\ =\ t$$

- Definitions have the property that adding a new definition to the set of existing ones cannot introduce any inconsistencies
- As was shown by Russell and Whitehead, with suitable definitions, all of classical mathematics can be constructed from logic
- See notes for how recursive definitions are handled

Type Definitions

- New types can be defined
- For example, can define a new type `three` with values `ONE`, `TWO` and `THREE`
- Can also add recursive type definitions
- For example, could define a new type α `list` with constructors `NIL` and `CONS`
- Such definitions correspond closely to datatype definitions in ML
- Recursion theorems that allow ML style recursive definitions over such recursive types can then be proved
- Details of type definitions not given here

Summary

- Primitive notions
  - variables, constants, function applications, λ-abstractions
- Types
  - statements (formulae) are just boolean terms
- Special syntax to make notation user-friendly
  - infixes, binders, conditional notation
- Definitions
  - introduces new constants consistently

Semantic embedding in a general logic

- **Program verification involves the proof of mathematical statements**
- **A program verifier will need a theorem prover**
- **One approach: verification condition generation**
- **Another approach: embed program logic in logic of a general prover**
  - programming logic rules can then be derived
  - the full power of the theorem prover will be available for proofs
  - the implementation of the rules can be guaranteed to be sound
  - easy to soundly extend the logic
- **Embedding Hoare Logic and other programming logics outlined**
- **First a trivial pedagogical example to illustrate concepts**

Deep Semantic Embedding – a preliminary example

- **Consider the propositional language:**

$$\mathit{wff} \ ::= \ \mathtt{True} \ \mid \ \mathtt{N}\ \mathit{wff} \ \mid \ \mathtt{C}\ \mathit{wff}\ \mathit{wff} \ \mid \ \mathtt{D}\ \mathit{wff}\ \mathit{wff}$$

- **Can represent wffs inside higher order logic**
  - by values of some type, wff say
  - we haven't described type definitions
- **Can define a semantic function, $\mathcal{M}$ say, by recursion:**

$$\begin{aligned}
\mathcal{M}(\mathtt{True}) &= T \\
\mathcal{M}(\mathtt{N}\ w) &= \neg\mathcal{M}(w) \\
\mathcal{M}(\mathtt{C}\ w_1\ w_2) &= \mathcal{M}(w_1) \wedge \mathcal{M}(w_2) \\
\mathcal{M}(\mathtt{D}\ w_1\ w_2) &= \mathcal{M}(w_1) \vee \mathcal{M}(w_2)
\end{aligned}$$

  - $\mathcal{M}$ is a constant of higher order logic of type $\mathit{wff}{\to}\mathit{bool}$
  - defined via a recursion theorem – details omitted

Shallow Embedding

- $\mathit{wff} \ ::= \ \mathtt{True} \ \mid \ \mathtt{N}\ \mathit{wff} \ \mid \ \mathtt{C}\ \mathit{wff}\ \mathit{wff} \ \mid \ \mathtt{D}\ \mathit{wff}\ \mathit{wff}$
- **Translate wffs in higher order logic terms**
- **Suppose $[\![w]\!]$ is the translation of w into higher order logic**
- **Translation mapping $w \mapsto [\![w]\!]$ not defined inside higher order logic**
  - corresponds to informal 'parsing and pretty-printing' conventions
- **Let $\leadsto$ mean "is translated to"**
- **A shallow embedding: is a set of translation rules:**

$$\begin{aligned}
[\![\mathtt{True}]\!] &\leadsto \texttt{"}T\texttt{"} \\
[\![\mathtt{N}\ w]\!] &\leadsto \texttt{"}\neg\texttt{"} \frown [\![w]\!] \\
[\![\mathtt{C}\ w_1\ w_2]\!] &\leadsto [\![w_1]\!] \frown \texttt{"}\wedge\texttt{"} \frown [\![w_2]\!] \\
[\![\mathtt{D}\ w_1\ w_2]\!] &\leadsto [\![w_1]\!] \frown \texttt{"}\vee\texttt{"} \frown [\![w_2]\!]
\end{aligned}$$

Deep vs Shallow Embedding

- **Deep and shallow embedding are two ends of a spectrum**
  - some aspects of the semantics formalized inside higher order logic
  - others specified via informal notational conventions
- **Deep embedding: theorems about embedded language provable**
  - for example: $\forall w_1\ w_2 \in \mathit{wff}.\ \mathcal{M}(\mathtt{C}w_1w_2) = \mathcal{M}(\mathtt{NDN}w_1\mathtt{N}w_2)$
  - formalizes more of the embedding, but requires expressive host logic
- **Shallow embedding: only theorems in embedded language provable**
  - quantification over wffs is not expressible
  - less in the logic, and hence the embedding is less demanding
  - so is often easier to support complex notations
- **A shallow embeddings of Hoare specifications will be described**
  - outline how the axioms and rules of Floyd-Hoare logic can be derived
- **Homeier's Sunrise system is a deep embedding of Hoare logic**

Embedding command semantics into higher order logic

- **Commands translate to predicates on pairs of states**
- $C \leadsto c$ **where the meaning of** C **is given by** c

$$c(s_1, s_2) = \begin{cases} \mathtt{T} & \text{if executing } C \text{ in state } s_1 \text{ results in state } s_2 \\ \mathtt{F} & \text{otherwise} \end{cases}$$

- **Example: if** `c` **is the predicate denoted by** `WHILE T DO SKIP` **then:**

$$\forall s_1\ s_2.\ \mathtt{c}(s_1, s_2) = \mathtt{F}$$

States

- **Program variables will be represented by strings**
- **The type** *state* **of states that we use is defined by:**

$$\mathit{state} \ = \ \mathit{string}{\rightarrow}\mathit{num}$$

- **State** s **has** ‘X‘, ‘Y‘, ‘Z‘ **bound to** 1, 2, 3; **all other strings bound to** 0
  - $s \ = \ \lambda x.\ \mathit{if}\ x = \text{‘}X\text{‘}\ \mathit{then}\ 1\ \mathit{else}\ \mathit{if}\ x = \text{‘}Y\text{‘}\ \mathit{then}\ 2\ \mathit{else}\ \mathit{if}\ x = \text{‘}Z\text{‘}\ \mathit{then}\ 3\ \mathit{else}\ 0$
- **A more complex type of states is needed for arrays**

Translations represent meaning (denotational semantics)

- E, B, C **translate to** e, b **and** c, **respectively, where:**

$$\begin{array}{rcl} e & : & \mathit{state}{\rightarrow}\mathit{num} \\ b & : & \mathit{state}{\rightarrow}\mathit{bool} \\ c & : & \mathit{state} \times \mathit{state}{\rightarrow}\mathit{bool} \end{array}$$

  represent meanings of expressions and commands
- **The embedding translation of** $X+1$ **is** $\lambda s.\ s\text{‘}X\text{‘} + 1$
- **The embedding translation of** $(X+Y) > 10$ **is**

$$\lambda s.\ (s\text{‘}X\text{‘} + s\text{‘}Y\text{‘}) > 10$$

- $[\![E]\!]$ **and** $[\![B]\!]$ **denote the translations (meanings) of** E **and** B
- **For example:**

$$\begin{array}{rcl} [\![X+1]\!] & = & \lambda s.\ s\text{‘}X\text{‘} + 1 \\ [\![(X+Y) > 10]\!] & = & \lambda s.\ (s\text{‘}X\text{‘} + s\text{‘}Y\text{‘}) > 10 \end{array}$$

Semantics of Commands

- **A command** C **translates to a term** $[\![C]\!]$ **of type** $\mathit{state} \times \mathit{state}{\rightarrow}\mathit{bool}$
- **Define constants as ‘constructors’ for each kind of command**
- **For example:**

$$[\![\mathtt{SKIP}]\!] \ = \ \mathtt{Skip}$$

  where the constant `Skip` **is defined by:**

$$\mathtt{Skip}(s_1, s_2) \ = \ (s_1 = s_2)$$

Semantics of other programming constructs

- Define constants $\texttt{Assign}$, $\texttt{Seq}$, $\texttt{If}$, $\texttt{While}$
  - just formalise semantics of language (denotational or operational)
  - details of a denotational semantics in the notes
- $\llbracket V \texttt{ := } E\rrbracket \ = \ \texttt{Assign}(`V`, \llbracket E\rrbracket)$
  - $\texttt{Assign} : (\mathit{string} \times (\mathit{state}{\rightarrow}\mathit{num})) \rightarrow (\mathit{state} \times \mathit{state}) \rightarrow \mathit{bool}$
- $\llbracket C_1;\ C_2\rrbracket \ = \ \texttt{Seq}(\llbracket C_1\rrbracket, \llbracket C_2\rrbracket)$
  - $\texttt{Seq} : ((\mathit{state} \times \mathit{state}{\rightarrow}\mathit{bool}) \times (\mathit{state} \times \mathit{state}{\rightarrow}\mathit{bool})) \rightarrow (\mathit{state} \times \mathit{state}) \rightarrow \mathit{bool}$
- $\llbracket \texttt{IF } B \texttt{ THEN } C_1 \texttt{ ELSE } C_2\rrbracket \ = \ \texttt{If}(\llbracket B\rrbracket, \llbracket C_1\rrbracket, \llbracket C_2\rrbracket)$
  - $\texttt{If} : ((\mathit{state}{\rightarrow}\mathit{bool}) {\times} (\mathit{state}{\times}\mathit{state}{\rightarrow}\mathit{bool}) {\times} (\mathit{state}{\times}\mathit{state}{\rightarrow}\mathit{bool})) \rightarrow (\mathit{state}{\times}\mathit{state}) \rightarrow \mathit{bool}$
- $\llbracket \texttt{WHILE } B \texttt{ DO } C\rrbracket \ = \ \texttt{While}(\llbracket B\rrbracket, \llbracket C\rrbracket)$
  - $\texttt{While} : ((\mathit{state}{\rightarrow}\mathit{bool}) \times (\mathit{state} \times \mathit{state}{\rightarrow}\mathit{bool})) \rightarrow (\mathit{state} \times \mathit{state}) \rightarrow \mathit{bool}$

Example

- The command

$$\begin{array}{l} R \texttt{ := } X\texttt{;} \\ Q \texttt{ := } 0\texttt{;} \\ \texttt{WHILE } Y \leq R \texttt{ DO } (R \texttt{ := } R - Y\texttt{; } Q \texttt{ := } Q+1) \end{array}$$

- translates to (means)

$$\begin{array}{l} \texttt{Seq} \\ \ (\texttt{Assign}(`R`, \llbracket X\rrbracket), \\ \ \ \texttt{Seq} \\ \ \ \ (\texttt{Assign}(`Q`, \llbracket 0\rrbracket), \\ \ \ \ \ \texttt{While} \\ \ \ \ \ \ (\llbracket Y \leq R\rrbracket, \\ \ \ \ \ \ \ \texttt{Seq}(\texttt{Assign}(`R`, \llbracket R-Y\rrbracket,\ \texttt{Assign}(`Q`, \llbracket Q+1\rrbracket)))) \end{array}$$

- Expanding $\llbracket\cdots\rrbracket$ notation results in:

$$\begin{array}{l} \texttt{Seq} \\ \ (\texttt{Assign}(`R`, \lambda s.\ s`X`), \\ \ \ \texttt{Seq} \\ \ \ \ (\texttt{Assign}(`Q`, \lambda s.\ 0), \\ \ \ \ \ \texttt{While} \\ \ \ \ \ \ ((\lambda s.\ s`Y` \leq s`R`), \\ \ \ \ \ \ \ \texttt{Seq}(\texttt{Assign}(`R`, \lambda s.\ s`R` - s`Y`),\ \texttt{Assign}(`Q`, \lambda s.\ s`Q` + 1))))) \end{array}$$

Semantics of Partial Correctness

- A partial correctness specification $\{P\}\ C\ \{Q\}$ translates to:

$$\forall s_1\ s_2.\ \llbracket P\rrbracket\ s_1\ \wedge\ \llbracket C\rrbracket(s_1,s_2) \Rightarrow \llbracket Q\rrbracket\ s_2$$

- To abbreviate this formula, define a constant $\texttt{Spec}$ by:

$$\texttt{Spec}(p,c,q) \ = \ \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1,s_2) \Rightarrow q\ s_2$$

- The pre and postconditions P and Q are not just the logical formulae themselves, but are the translations $\llbracket P\rrbracket$ and $\llbracket Q\rrbracket$
- Thus:

$$\{X=1\}\ X \texttt{ := } X+1\ \{X=2\}$$

  translates to

$$\texttt{Spec}(\llbracket X=1\rrbracket,\ \texttt{Assign}(`X`, \llbracket X+1\rrbracket),\ \llbracket X=2\rrbracket)$$

  which is

$$\texttt{Spec}((\lambda s.\ s`X` = 1),\ \texttt{Assign}(`X`, (\lambda s.\ s`X` + 1),\ (\lambda s.\ s`X` = 2))$$

Floyd-Hoare Logic

- Floyd-Hoare logic can be embedded in higher order logic by regarding the concrete syntax of partial correctness specifications as an abbreviation for the corresponding semantic statements
- For example:

$$\{X=x\}\ X \texttt{ := } X+1\ \{X=x+1\}$$

  can be interpreted as abbreviating:

$$\texttt{Spec}(\llbracket X=x\rrbracket,\ \texttt{Assign}(`X`, \llbracket X+1\rrbracket),\ \llbracket X=x+1\rrbracket)$$

- The translation between the syntactic 'surface structure' and the semantic 'deep structure' is straightforward; it can easily be mechanized with a simple parser and pretty-printer
- If partial correctness specifications are interpreted this way then the axioms and rules of Hoare logic can be derived in higher order logic

Example: SKIP-axiom

- To derive the SKIP-axiom it must be shown for arbitrary P that:

$$\vdash \{P\}\ \texttt{SKIP}\ \{P\}$$

which abbreviates:

$$\vdash \texttt{Spec}([\![P]\!], \texttt{Skip}, [\![P]\!])$$

- Expanding the definition of Spec, we must show,

$$\vdash \forall s_1\ s_2.\ [\![P]\!]\ s_1\ \wedge\ \texttt{Skip}(s_1, s_2) \Rightarrow [\![P]\!]\ s_2$$

- Expanding the definition of Skip, we must show,

$$\vdash \forall s_1\ s_2.\ [\![P]\!]\ s_1\ \wedge\ (s_1 = s_2) \Rightarrow [\![P]\!]\ s_2$$

- Using properties of equality:

$$\vdash \forall s_1\ s_2.\ [\![P]\!]\ s_1 \Rightarrow [\![P]\!]\ s_1$$

Rules proved as implications

- Lift $\neg$ and $\wedge$ to predicates: $\neg p$, $p \wedge q$ mean $\lambda s.\ \neg p(s)$, $\lambda s.\ p(s) \wedge q(s)$
- To derive the sequencing rule prove:

$$\vdash \forall p\ q\ r\ c_1\ c_2.\\ \texttt{Spec}(p, c_1, q)\ \wedge\ \texttt{Spec}(q, c_2, r)\ \Rightarrow\ \texttt{Spec}(p, \texttt{Seq}(c_1, c_2), r)$$

- To derive the IF-rule prove:

$$\vdash p\ q\ b\ c_1\ c_2.\\ \texttt{Spec}(p \wedge b, c_1, q)\ \wedge\ \texttt{Spec}(p \wedge \neg b, c_2, q)\ \Rightarrow\ \texttt{Spec}(p, \texttt{If}(b, c_1, c_2), q)$$

- To derive the WHILE-rule prove:

$$\vdash \forall p\ b\ c.\ \texttt{Spec}(p \wedge b, c, p)\ \Rightarrow\ \texttt{Spec}(p, \texttt{While}(b, c), p \wedge \neg b)$$

- By deriving the Hoare logic rules, we guarantee their soundness

Other programming logic constructs

- Variants on Hoare logic:
  - VDM-style specifications
  - strongest and weakest preconditions
  - Dynamic logic
- Originally separate programming logics
- We will define them in higher order logic
- Illustrates benefits of using a general host logic

VDM-style specifications

- Used in the Vienna Development Method (VDM)
  - introduced a variation on Hoare-style specifications
- VDM notation reduces the need for auxiliary logical variables by providing a way of refering to the initial values of variables in post-conditions using 'hooked' variables like $\overleftarrow{X}$
- Here's a Hoare-style example:

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

- VDM-style version is:

$$\{\texttt{T}\}\ R := X\ ;\ X := Y\ ;\ Y := R\ \{Y = \overleftarrow{X}\ \wedge\ X = \overleftarrow{Y}\}$$

- $\overleftarrow{X}$, $\overleftarrow{Y}$ denote values of X, Y in precondition state

Aczel's semantics of VDM

- **Postcondition is a binary relation on initial and final states**
- **Regard**

$$\{P[X_1,\ldots,X_n]\}\ C\ \{Q[X_1,\ldots,X_n,\overleftarrow{X_1},\ldots,\overleftarrow{X_n}]\}$$

as an abbreviation for

$$\texttt{VDM\_Spec}([\![P[X_1,\ldots,X_n]]\!],\ [\![C]\!],\ [\![Q[X_1,\ldots,X_n,\overleftarrow{X_1},\ldots,\overleftarrow{X_n}]]\!]_2)$$

where `VDM_Spec` **is defined by:**

$$\texttt{VDM\_Spec}(p,c,r) = \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1,s_2) \Rightarrow r(s_1,s_2)$$

and the notation $[\![\ \cdots\ ]\!]_2$ **is defined by:**

$$[\![Q[X_1,\ldots,X_n,\overleftarrow{X_1},\ldots,\overleftarrow{X_n}]]\!]_2\ =\ \lambda(s_1,s_2).\ Q[s_2`X_1`,\ldots,s_2`X_n`,s_1`X_1`,\ldots,s_1`X_n`]$$

Aczel's WHILE-rule for total correctness

- **Following rule due to Peter Aczel:**

$$\frac{\vdash\ [P\ \wedge\ B]\ C\ [P\ \wedge\ R] \qquad \vdash\ \texttt{Transitive}\ R \qquad \vdash\ \texttt{Well\_Founded}\ R}{\vdash\ [P]\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ [P\ \wedge\ \neg B\ \wedge\ R^*]}$$

where:

$$\begin{array}{lcl}\texttt{Transitive}\ r & = & \forall s_1\ s_2\ s_3.\ r(s_1,s_2)\ \wedge\ r(s_2,s_3) \Rightarrow r(s_1,s_3)\\ \texttt{Well\_Founded}\ r & = & \neg\exists f : \mathit{num}\rightarrow\mathit{state}.\ \forall n.\ r(f(n),f(n+1))\\ & & \textbf{(no infinite chain: } r(s_0,s_1)\wedge r(s_1,s_2)\wedge\cdots)\end{array}$$

- `Transitive` **and** `Well_Founded` **easily defined in higher order logic**
  - **cannot be defined in first order logic**

Dijkstra's weakest preconditions

- **Idea is to calculate a program to achieve a postcondition**
  - **not a theory of post hoc verification**
- **Non-determinism a key idea in Dijksta's presentation**
  - **start with a non-deterministic high level pseudo-code**
  - **refine to deterministic and efficient code**
- **Weakest preconditions (wp) are for total correctness**
- **Weakest** *liberal* **preconditions (wlp) for partial correctness**
- **If** C **is a command and** Q **a predicate, then informally:**
  - $\texttt{wlp}(C,Q)\ =\ $ *'The weakest predicate* P *such that* $\{P\}\ C\ \{Q\}$*'*
  - $\texttt{wp}(C,Q)\ =\ $ *'The weakest predicate* P *such that* $[P]\ C\ [Q]$*'*
- **If** P **and** Q **are predicates then** $Q\ \Rightarrow\ P$ **means** P **is 'weaker' than** Q**, i.e. everything satisfying** Q **also satisfies** P

Meaning of weakest preconditions

- **If** C **is a command and** Q **a predicate, then informally:**

$$\texttt{wlp}(C,Q)\ =\ \text{'}\textit{The weakest predicate } P \textit{ such that } \{P\}\ Q\ \{Q\}\text{'}$$

- **What is the 'weakest predicate' satisfying a condition**
- **Easy definition in higher order logic:**

$$\vdash\ \texttt{wlp}(c,q)\ =\ \lambda s.\ \forall s'.\ c(s,s') \Rightarrow q\ s'$$

- $\texttt{wlp}(c,q)(s)$ **holds if** s **always leads via** c **to a state satisfying** q
- **The relationship with Hoare's notation is given by:**

$$\vdash\ \texttt{Spec}(p,c,q)\ =\ \forall s.\ p\ s \Rightarrow \texttt{wlp}(c,q)\ s$$

Strongest postconditions

- Define $\mathtt{sp}(C,P)$ to be 'strongest' Q such that $\{P\}\ C\ \{Q\}$
  - partial correctness: $\{P\}\ C\ \{\mathtt{sp}(C,P)\}$
  - $\{P\}\ C\ \{Q\}$ equivalent to $\mathtt{sp}(C,P) \Rightarrow Q$
- Note that `wlp` goes 'backwards', but `sp` goes 'forwards'
  - backwards: $\{P\}\ C\ \{Q\}$ if and only if $P \Rightarrow \mathtt{wlp}(C,Q)$
  - forwards: $\{P\}\ C\ \{Q\}$ if and only if $\mathtt{sp}(C,P) \Rightarrow Q$
- Strongest postcondition calculation related to symbolic execution

Meaning of strongest postconditions

- If C is a command and P a predicate, then informally:

$$\mathtt{sp}(C,P) \;=\; \text{'}\textit{The strongest predicate } Q \textit{ such that } \{P\}\ Q\ \{Q\}\text{'}$$

- Easy definition in higher order logic:

$$\vdash\ \mathtt{sp}(c,p) \;=\; \lambda s'.\ \exists s.\ p(s) \wedge c(s,s')$$

- The relationship with Hoare's notation is given by:

$$\vdash\ \mathtt{Spec}(p,c,q) \;=\; \forall s'.\ \mathtt{sp}(c,p)(s') \Rightarrow q(s')$$

- Exercise: prove this from the definitions of `Spec` and `sp`

Dynamic logic

- Dynamic logic relates Hoare logic to modal logic
- Invented by V.R. Pratt based on an idea of R.C. Moore (MIT)
- States of computation are thought of as *possible worlds*
  - if a command C transforms an initial state s to a final state s'
  - then s' is thought of as *accessible* from s (a concept from modal logic)
- Modal logic has *modal formulae* $\Box q$ and $\Diamond q$ where:
  - $\Box q$ is true in s if q is true in *all states* accessible from s
  - $\Diamond q$ is true in s if if q is true in *some state* accessible from s
- Exercise: convince yourself that $\Diamond q \;\Leftrightarrow\; \neg\Box\neg q$ and $\Box q \;\Leftrightarrow\; \neg\Diamond\neg q$

Modal operators of Dynamic Logic

- Dynamic logic has operators $\mathtt{[}C\mathtt{]}$ and $\mathtt{<}C\mathtt{>}$ for each command C
  - $\mathtt{[}C\mathtt{]}$ is analogous to $\Box$
  - $\mathtt{<}C\mathtt{>}$ is analogous to $\Diamond$
- These can be defined on the relation c denoted by C as follows:

$$\begin{aligned}\mathtt{[}c\mathtt{]}q &= \lambda s.\ \forall s'.\ c(s,s') \Rightarrow q\ s' \\ \mathtt{<}c\mathtt{>}q &= \neg(\mathtt{[}c\mathtt{]}(\neg q))\end{aligned}$$

  where $\neg$ is negation lifted pointwise to predicates
- Relation to weakest preconditions

$$\vdash\ \mathtt{wlp}(c,q) \;=\; \mathtt{[}c\mathtt{]}q$$

Conclusion

- Hoare, VDM, Dijskstra, Dynamic Logic, ... ?
- From a semantic embedding perspective its just packaging!
- Don't need separate logics, just definitions in a general host logic
- Current research in the Lab (not exhaustive!)
  - derive Hoare Logic from processor models (Fox, Myreen)
  - extend Hoare logic to pointers, concurrency and more (Parkinson et al. ...)
  - semantically embed separation logic in higher order logic (Tuerk)
- **Wednesday:** guest lecture by Joe Hurd of Galois (`www.galois.com`)