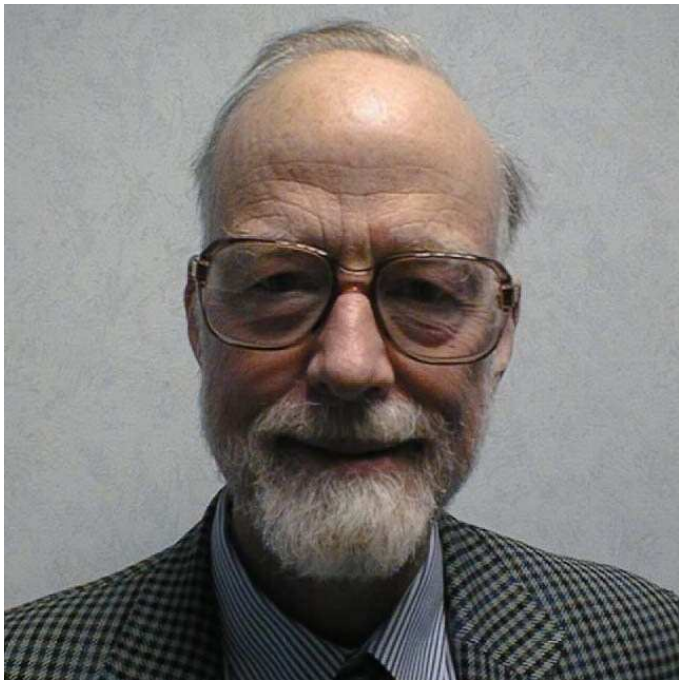


# Hoare Logic

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

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



# A Little Programming Language

## Expressions:

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

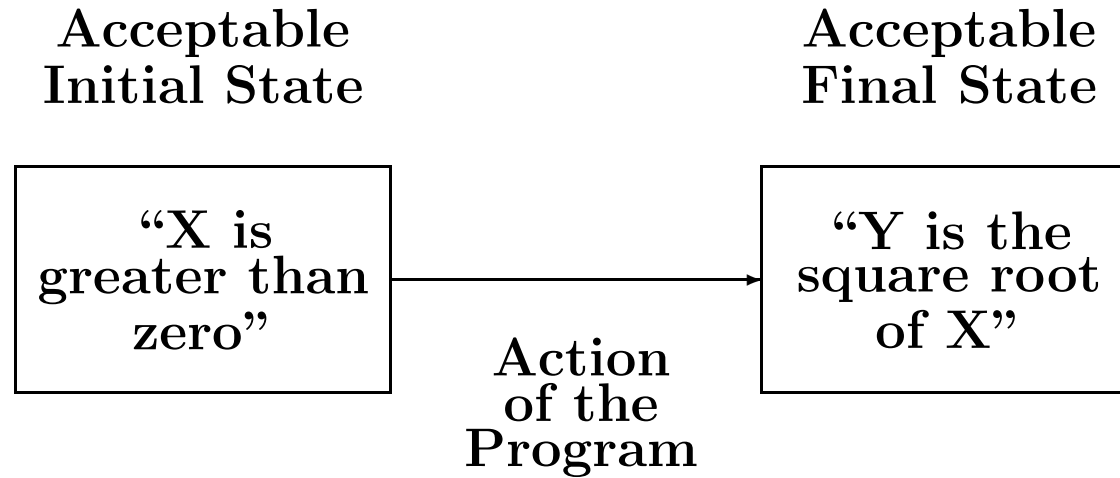
## Boolean expressions:

$B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \dots$

## Commands:

$C ::= V := E$   
|  $C_1 ; C_2$   
| IF  $B$  THEN  $C_1$  ELSE  $C_2$   
| WHILE  $B$  DO  $C$

# Specification of Imperative Programs



## Hoare's notation

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

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

where:

- $C$  is a command
- $P$  and  $Q$  are conditions on the program variables used in  $C$
- Conditions on program variables will be written using standard mathematical notations together with *logical operators* like:
  - $\wedge$  ('and'),  $\vee$  ('or'),  $\neg$  ('not'),  $\Rightarrow$  ('implies')
- Hoare's original notation was  $P \{C\} Q$  not  $\{P\} C \{Q\}$ , but the latter form is now more widely used



# Meaning of Hoare's Notation

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

# Hoare Logic and Verification Conditions

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

# Partial Correctness Specification

- An expression  $\{P\} C \{Q\}$  is called a *partial correctness specification*
  - $P$  is called its *precondition*
  - $Q$  its *postcondition*
- $\{P\} C \{Q\}$  is true if
  - whenever  $C$  is executed in a state satisfying  $P$
  - and *if* the execution of  $C$  terminates
  - then the state in which  $C$ 's execution terminates satisfies  $Q$
- These specifications are ‘partial’ because for  $\{P\} C \{Q\}$  to be true it is *not* necessary for the execution of  $C$  to terminate when started in a state satisfying  $P$
- It is only required that *if* the execution terminates, *then*  $Q$  holds
- $\{X = 1\} \text{ WHILE } T \text{ DO } X := X \{Y = 2\}$  – **this specification is true!**

# Total Correctness Specification

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

# Total Correctness

- Informally:

$$\textit{Total correctness} = \textit{Termination} + \textit{Partial correctness}$$

- 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

- Microsoft's T2 tool proves systems code terminates

# 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

# 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

# Judgements

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



# Reminder of our little programming language ✓

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

## Expressions

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

## Boolean expressions

$$B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \dots$$

## Commands

$$\begin{aligned} C ::= & V := E \\ & \mid C_1 ; C_2 \\ & \mid \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \\ & \mid \text{WHILE } B \text{ DO } C \end{aligned}$$

**Assignments**

**Sequences**

**Conditionals**

**WHILE-commands**

# Substitution Notation

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

- Think of this notation as the ‘cancellation law’

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

which is analogous to the cancellation property of fractions

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

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

# The Assignment Axiom (Hoare)

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

## The Assignment Axiom

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

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

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

# The Backwards Fallacy

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

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

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

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

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

# Validity

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

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

- 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

# A Forwards Assignment Axiom (Floyd)

- This is the original semantics of assignment due to Floyd

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

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

- Example instance

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

- Simplifying the postcondition

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

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

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

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

$$\vdash \{X=1\} X := X+1 \{ X = 2 \wedge \text{T} \}$$

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

- Forwards Axiom equivalent to standard one but harder to use

# The Assignment Axiom (Hoare)

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

## The Assignment Axiom

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

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

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



# 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

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

- Note the two hypotheses are different kinds of judgements

## 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

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

# 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:**  $C_1; \dots ; C_n$
- **Semantics:** the commands  $C_1, \dots, C_n$  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

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

## Example Proof

**Example:** By the assignment axiom:

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

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

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

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

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

# Conditionals

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

# The Conditional Rule

## The conditional rule

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

- From Assignment Axiom + Precondition Strengthening and

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

it follows that

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

and

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

- Then by the conditional rule it follows that

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

## WHILE-commands

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



# Invariants

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

# The WHILE-Rule

## The WHILE-rule

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

- It is easy to show

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

- Hence by the WHILE-rule with  $P = 'X=R+(Y \times Q)'$  and  $S = 'Y \leq R'$

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

## Example

- From the previous slide

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

- It is easy to deduce that

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

- Hence by the sequencing rule and postcondition weakening

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

- 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

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

- Look at the facts:
  - invariant  $P$  must hold initially
  - with the negated test  $\neg S$  the invariant  $P$  must establish the result
  - when the test  $S$  holds, the body must leave the invariant  $P$  unchanged
- Think about how the loop works – the invariant should say that:
  - what **has been done so far** together with what **remains to be done**
  - holds **at each iteration** of the loop
  - and gives **the desired result** when the loop terminates

## Example

- Consider a factorial program

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

- Look at the facts
  - initially  $X=n$  and  $Y=1$
  - finally  $X=0$  and  $Y=n!$
  - on each loop  $Y$  is increased and,  $X$  is decreased
- Think how the loop works
  - $Y$  holds the result so far
  - $X!$  is what remains to be computed
  - $n!$  is the desired result
- The invariant is  $X! \times Y = n!$ 
  - ‘stuff to be done’  $\times$  ‘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 (X:=X+1; Y:=Y×X)
{Y=N!}
```

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

# Conjunction and Disjunction

## Specification conjunction

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

## Specification disjunction

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

- 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



## Derived rules for finding proofs

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

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

then we could instantiate

$P \mapsto Precondition, C \mapsto Command, Q \mapsto Postcondition$

to get instances of  $H_1, \dots, H_n$  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

$$\begin{array}{c} \dots \\ \hline \vdash \{P\} V := E \{Q\} \\ \dots \\ \hline \vdash \{P\} C_1; C_2 \{Q\} \\ \dots \\ \hline \vdash \{P\} \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \{Q\} \\ \dots \\ \hline \vdash \{P\} \text{WHILE } S \text{ DO } C \{Q\} \end{array}$$

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

# The Derived Assignment Rule

- An example 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.

- Can generalise this proof to a proof schema:

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

- This proof schema justifies:

## Derived Assignment Rule

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

- Note:  $Q[E/V]$  is the **weakest liberal precondition**  $wlp(V := E, Q)$

- Example proof above can now be done in one less step

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

## Derived Sequenced Assignment Rule

- The following rule will be useful later

### Derived Sequenced Assignment Rule

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

- Intuitively work backwards:
  - push  $Q$  ‘through’  $V := E$ , changing it to  $Q[E/V]$

- Example: By the assignment axiom:

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

- Hence by the sequenced assignment rule

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

# The Derived Sequencing Rule

- The rule below follows from the sequencing and consequence rules

## The Derived Sequencing Rule

$$\frac{\begin{array}{l} \vdash P \Rightarrow P_1 \\ \vdash \{P_1\} C_1 \{Q_1\} \quad \vdash Q_1 \Rightarrow P_2 \\ \vdash \{P_2\} C_2 \{Q_2\} \quad \vdash Q_2 \Rightarrow P_3 \\ \cdot \\ \cdot \\ \cdot \\ \vdash \{P_n\} C_n \{Q_n\} \quad \vdash Q_n \Rightarrow Q \end{array}}{\vdash \{P\} C_1; \dots ; C_n \{Q\}}$$

- Exercise: why no derived conditional rule?

# The Derived While Rule

## Derived While Rule

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

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

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

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

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

- Then, by the derived While rule

$$\begin{array}{l} \vdash \{R=X \wedge Q=0\} \\ \quad \text{WHILE } Y \leq R \text{ DO} \\ \quad \quad (R:=R-Y; Q:=Q+1) \\ \quad \quad \{X=R+(Y \times Q) \wedge \neg(Y \leq R)\} \end{array}$$

## 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

# Backwards versus Forwards Proof

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

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

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



## Annotations

- The sequencing rule introduces a new statement  $R$

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

- 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

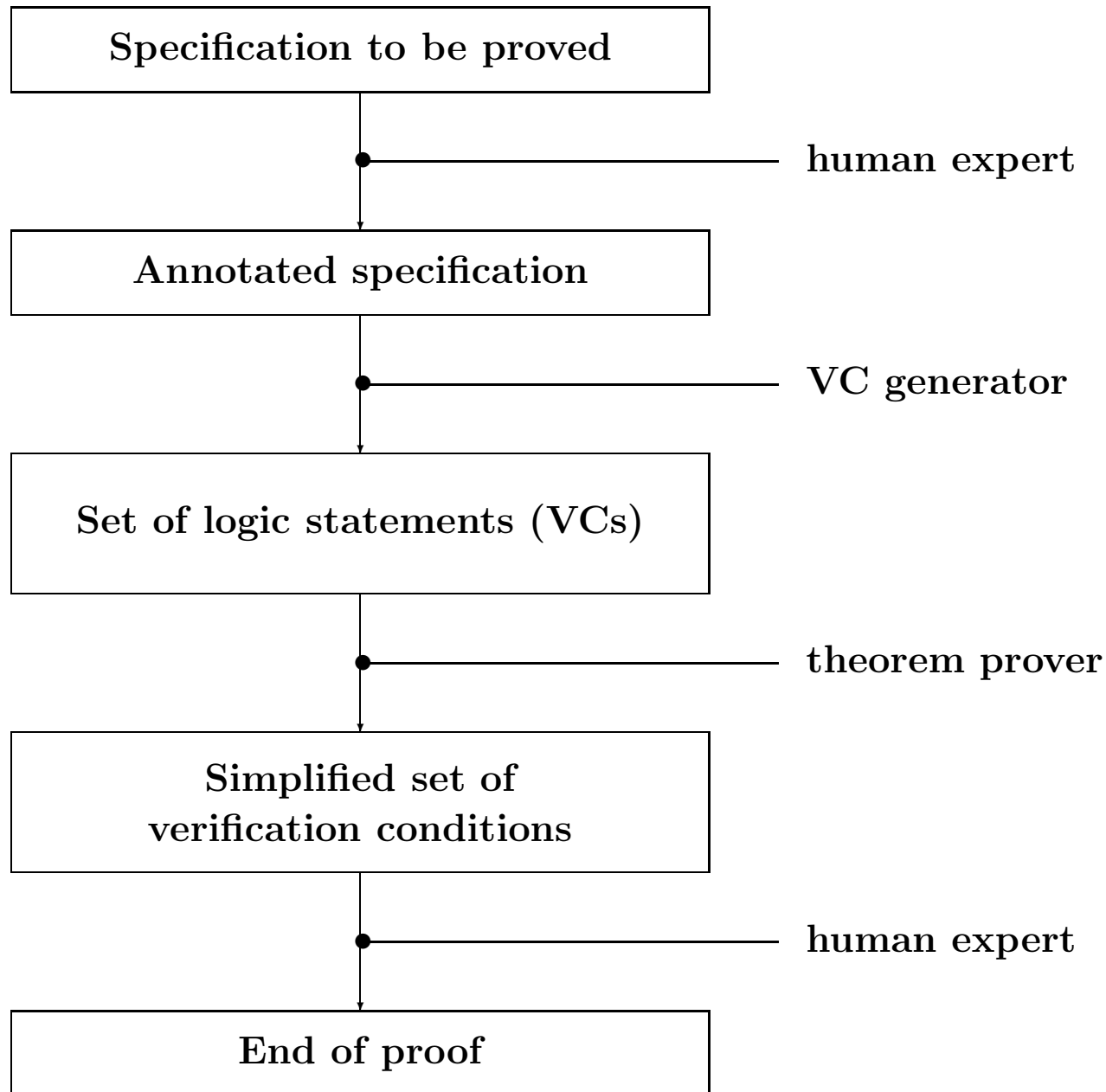
```
{T}
  R:=X;
  Q:=0; {R=X ∧ Q=0} ← P1
  WHILE Y≤R DO {X = R+Y×Q} ← P2
    (R:=R-Y; Q:=Q+1)
  {X = R+Y×Q ∧ R<Y}
```

## 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.

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

# Architecture of a Verifier



## Verification conditions ✓

- The three steps in proving  $\{P\}C\{Q\}$  with a verifier
- **1** The program  $C$  is *annotated* by inserting statements (*assertions*) expressing conditions that are meant to hold at intermediate points
  - tricky: needs intelligence and good understanding of how the program works
  - automating it is an artificial intelligence problem
- **2** A set of logic statements called *verification conditions* (VCs) is then generated from the annotated specification
  - this is purely mechanical and easily done by a program
- **3** The verification conditions are proved
  - needs automated theorem proving (i.e. more artificial intelligence)
- To improve automated verification one can try to
  - reduce the number and complexity of the annotations required
  - increase the power of the theorem prover
  - still a research area

# Validity of Verification Conditions

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

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

## Example

- Step **1** is to insert annotations  $P_1$  and  $P_2$

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

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

## Example Continued

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

- 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  $X=R+Y \times Q$  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 **2** will generate the following four verification conditions

$$(i) T \Rightarrow (X=X \wedge 0=0)$$

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

$$(iii) (X = R+(Y \times Q)) \wedge Y \leq R \Rightarrow (X = (R-Y)+(Y \times (Q+1)))$$

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

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

## Annotation of Commands

- An annotated command is a command with statements (*assertions*) embedded within it
- A command is *properly annotated* if statements have been inserted at the following places
  - (i) before  $C_2$  in  $C_1;C_2$  if  $C_2$  is *not* an assignment command
  - (ii) after the word DO in WHILE commands
- The inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs
- Can reduce number of annotations using weakest preconditions (see later)

# Annotation of Specifications

- A properly annotated specification is a specification  $\{P\}C\{Q\}$  where  $C$  is a properly annotated command
- Example: To be properly annotated, assertions should be at points ① and ② of the specification below

$$\begin{array}{l} \{X=n\} \\ \quad Y:=1; \longleftarrow \textcircled{1} \\ \quad \text{WHILE } X \neq 0 \text{ DO } \longleftarrow \textcircled{2} \\ \quad \quad (Y:=Y \times X; X:=X-1) \\ \{X=0 \wedge Y=n!\} \end{array}$$

- Suitable statements would be

$$\begin{array}{l} \text{at } \textcircled{1}: \quad \{Y = 1 \wedge X = n\} \\ \text{at } \textcircled{2}: \quad \{Y \times X! = n!\} \end{array}$$

## 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
  - $vc_1, \dots, vc_n$
  - together with the VCs for  $C_1$  and those for  $C_2$
- Each VC rule corresponds to either a primitive or derived rule

## 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, i.e. 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

## Assignment commands

The single verification condition generated by

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

is

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

- **Example:** The verification condition for

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

is

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

(which is clearly true)

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

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



## Conditionals

The verification conditions generated from

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

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

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

are

(i) the VCs for  $\{T \wedge X \geq Y\} \text{MAX} := X \{ \text{MAX} = \max(X, Y) \}$

(ii) the VCs for  $\{T \wedge \neg(X \geq Y)\} \text{MAX} := Y \{ \text{MAX} = \max(X, Y) \}$

## 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
  - Case 1: it is of the form  $C_1;\{R\}C_2$  and  $C_2$  is not an assignment
  - Case 2: it is of the form  $C;V := E$
- And  $C$ ,  $C_1$  and  $C_2$  are properly annotated

## 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

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

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

# Justification of VCs for Sequences (1)

- **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

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

- Hence by the sequencing rule

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

## Justification of VCs for Sequences (2)

- **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 WHILE-Commands

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

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

- The annotation  $R$  is called an invariant

### WHILE-commands

The verification conditions generated from

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

are

- (i)  $P \Rightarrow R$
- (ii)  $R \wedge \neg S \Rightarrow Q$
- (iii) the verification conditions generated by  $\{R \wedge S\} C \{R\}$

## Justification of WHILE VCs

- If the verification conditions for

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

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

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



## 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 Dijkstra'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:
  - $\text{wlp}(C, Q) = \text{'The weakest predicate } P \text{ such that } \{P\} C \{Q\}'$
  - $\text{wp}(C, Q) = \text{'The weakest predicate } P \text{ such that } [P] C [Q]'$
- If  $P$  and  $Q$  are predicates then  $Q \Rightarrow P$  means  $P$  is 'weaker' than  $Q$

# Rules for weakest preconditions

- Relation with Hoare specifications:

$$\{P\} C \{Q\} \Leftrightarrow P \Rightarrow \text{wlp}(C, Q)$$

$$[P] C [Q] \Leftrightarrow P \Rightarrow \text{wp}(C, Q)$$

- Dijkstra gives rules for computing weakest preconditions:

$$\text{wp}(V := E, Q) = Q[E/V]$$

$$\text{wp}(C_1; C_2, Q) = \text{wp}(C_1, \text{wp}(C_2, Q))$$

$$\text{wp}(\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2, Q) = (S \Rightarrow \text{wp}(C_1, Q)) \wedge (\neg S \Rightarrow \text{wp}(C_2, Q))$$

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

## Using wlp to improve verification condition method ✓

- If  $C$  is **loop-free** then VC for  $\{P\} C \{Q\}$  is  $P \Rightarrow \text{wlp}(C, Q)$ 
  - no annotations needed in sequences!
- Cannot in general compute a **finite** formula for  $\text{wlp}(\text{WHILE } S \text{ DO } C, Q)$
- The following holds
$$\text{wlp}(\text{WHILE } S \text{ DO } C, Q) = \textit{if } S \textit{ then } \text{wlp}(C, \text{wlp}(\text{WHILE } S \text{ DO } C, Q)) \textit{ else } Q$$
- Above doesn't define  $\text{wlp}(C, Q)$  as a finite statement
- Could use a hybrid VC and wlp method

# Strongest postconditions

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

## Strongest postconditions for loop-free code ✓

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

- 
- $\text{sp}(V := E, P)$  corresponds to Floyd assignment axiom
  - Can *dynamically prune* conditionals because  $\text{sp}(C, \text{F}) = \text{F}$
  - Computer strongest postconditions is *symbolic execution*

# 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’  
(related to *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

## Using sp to generate verification conditions

- If  $C$  is loop-free then VC for  $\{P\} C \{Q\}$  is  $\text{sp}(C, P) \Rightarrow Q$
- Cannot in general compute a **finite** formula for  $\text{sp}(\text{WHILE } S \text{ DO } C, P)$
- The following holds
$$\text{sp}(\text{WHILE } S \text{ DO } C, P) = \text{sp}(\text{WHILE } S \text{ DO } C, \text{sp}(C, (P \wedge S))) \vee (P \wedge \neg S)$$
- Above doesn't define  $\text{sp}(C, P)$  to be a finite statement
- As with wlp, can use a hybrid VC and sp method



- 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



*decidable checking*

*proof of correctness*

# 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.  $\vdash \{T\} X := X \{T\}$  (assignment axiom)

2.  $\vdash \{T \wedge T\} X := X \{T\}$  (precondition strengthening)

3.  $\vdash \{T\} \text{WHILE } T \text{ DO } X := X \{T \wedge \neg T\}$  (2 and the WHILE-rule)

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

$$\vdash [T] \text{WHILE } T \text{ DO } X := X [T \wedge \neg T]$$

- 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 rule
  - Sequencing 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  $X := fact(-1)$ , where  $fact(n)$  is defined recursively:

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

## Error Termination

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

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

- 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



## 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  $\text{WHILE } S \text{ DO } C$  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

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

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

# The Derived While Rule

- Derived WHILE-rule needs to handle the variant

Derived WHILE-rule for total correctness

$$\vdash P \Rightarrow R$$

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

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

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

---

$$\vdash [P] \text{ WHILE } S \text{ DO } C [Q]$$

## 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 for WHILE-free code same as for partial correctness

## WHILE Annotation

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

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

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)

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

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

## WHILE-commands

The verification conditions generated from

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

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  $Q$ .

## 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/en-us/projects/t2/>

# Soundness and completeness of Hoare logic ✓

- Review of first-order logic
  - syntax: languages, function symbols, predicate symbols, terms, formulae
  - semantics: interpretations, valuations
  - soundness and completeness
- Formal semantics of Hoare triples
  - preconditions and postconditions as terms
  - semantics of commands
  - soundness of Hoare axioms and rules
  - completeness and relative completeness



## Semantics: terms and formulae

- Assume: language  $\mathcal{L}$ , interpretation  $\mathcal{I} = (D, I)$ , valuation  $s \in \text{Var} \rightarrow D$
- Define  $\text{Esem } E \ s \in D$  by:
  - if  $E \in \text{Var}$  then  $\text{Esem } E \ s = s(E)$
  - if  $E = f$ , where  $f$  a function symbol of arity 0, then  $\text{Esem } E \ s = I[f]$
  - if  $E = f(E_1, \dots, E_n)$ , then  $\text{Esem } E \ s = I[f](\text{Esem } E_1 \ s, \dots, \text{Esem } E_n \ s)$
- Define  $\text{Ssem } S \ s \in \text{Bool}$  by:
  - if  $S = p$ , where  $p$  a predicate symbol of arity 0, then  $\text{Ssem } S \ s = I[p]$
  - if  $S = p(E_1, \dots, E_n)$ , then  $\text{Ssem } S \ s = I[p](\text{Esem } E_1 \ s, \dots, \text{Esem } E_n \ s)$
  - $\text{Ssem } (\neg S) \ s = \neg(\text{Ssem } S \ s)$
  - $\text{Ssem } (S_1 \wedge S_2) \ s = (\text{Ssem } S_1 \ s) \wedge (\text{Ssem } S_2 \ s)$
  - $\text{Ssem } (S_1 \vee S_2) \ s = (\text{Ssem } S_1 \ s) \vee (\text{Ssem } S_2 \ s)$
  - $\text{Ssem } (S_1 \Rightarrow S_2) \ s = (\text{Ssem } S_1 \ s) \Rightarrow (\text{Ssem } S_2 \ s)$
  - $\text{Ssem } (\forall v. S) \ s = \text{if (for all } d \in D : \text{Ssem } S \ (s[d/v]) = \text{true}) then true else false}$
  - $\text{Ssem } (\exists v. S) \ s = \text{if (for some } d \in D : \text{Ssem } S \ (s[d/v]) = \text{true}) then true else false}$
- Note: will just say “ $\text{Ssem } S \ s$ ” to mean that “ $\text{Ssem } S \ s = \text{true}$ ”

## Satisfiability, validity and completeness

- Recall that a language  $\mathcal{L}$  specifies predicate and function symbols
- $S$  is *satisfiable* iff for some interpretation of  $\mathcal{L}$  and  $s$ :  $S \text{ sem } s = \text{true}$
- $S$  is *valid* iff for all interpretations of  $\mathcal{L}$  and all  $s$ :  $S \text{ sem } s = \text{true}$
- Notation:  $\models S$  means  $S$  is valid
- Deductive system for first-order logic specifies  $\vdash S$  – i.e.  $S$  is provable
- Soundness: if  $\vdash S$  then  $\models S$  (easy induction on length of proof)
- Completeness: if  $\models S$  then  $\vdash S$  (Gödel 1929)

# Sentences, Theories

- A *sentence* is a statement with *no free variables*
  - truth or falsity of sentences solely determined by interpretation
  - if  $S$  is a sentence then  $S \text{ sem } s_1 = S \text{ sem } s_2$  for all  $s_1, s_2$
- A *theory* is a set of sentences
  - $\Gamma$  will range over sets of sentences
- $\Gamma \vdash S$  means  $S$  can be deduced from  $\Gamma$  using first-order logic
- $\Gamma$  is *consistent* iff there is no  $S$  such that  $\Gamma \vdash S$  and  $\Gamma \vdash \neg S$
- $\Gamma \models_{\mathcal{I}} S$  means  $S$  true if  $\mathcal{I}$  makes all of  $\Gamma$  true
- $\Gamma \models S$  means  $\Gamma \models_{\mathcal{I}} S$  true for all  $\mathcal{I}$
- **Soundness and Completeness:**  $\Gamma \models S$  iff  $\Gamma \vdash S$

# Gödel's incompleteness theorem

- $\mathcal{L}_{\text{PA}}$  is the language of Peano Arithmetic
- $\mathcal{I}_{\text{PA}}$  is the *standard interpretation* of arithmetic
- $\models_{\mathcal{I}_{\text{PA}}} S$  means  $S$  is true in  $\mathcal{I}_{\text{PA}}$
- PA is the first-order theory of Peano Arithmetic
- There exists a sentence  $G$  of  $\mathcal{L}_{\text{PA}}$  and neither  $\text{PA} \vdash G$  nor  $\text{PA} \vdash \neg G$ 
  - Gödel's first incompleteness theorem (1930)
  - as  $G$  is a sentence either  $\models_{\mathcal{I}_{\text{PA}}} G$  or  $\models_{\mathcal{I}_{\text{PA}}} \neg G$
  - so there is a sentence,  $G_T$  say, true in  $\mathcal{I}_{\text{PA}}$  but can't be proved from PA
  - i.e.  $\models_{\mathcal{I}_{\text{PA}}} G_T$  but not  $\text{PA} \vdash G_T$

# Semantics of Hoare triples

- Recall that  $\{P\} C \{Q\}$  is true if
  - whenever  $C$  is executed in a state satisfying  $P$
  - and *if* the execution of  $C$  terminates
  - then  $C$  terminates in a state satisfying  $Q$
- $P$  and  $Q$  are first-order statements
- Will formalise semantics of  $\{P\} C \{Q\}$  to express:
  - whenever  $C$  is executed in a state  $s_1$  such that  $\text{Ssem } P \ s_1$
  - and *if* the execution of  $C$  starting in  $s_1$  terminates
  - then  $C$  terminates in a state  $s_2$  such that  $\text{Ssem } Q \ s_2 = \text{true}$
- Need to define “ $C$  starts in  $s_1$  and terminates in  $s_2$ ”
  - this is the semantics of commands
  - will define  $\text{Csem } C \ s_1 \ s_2$  to mean if  $C$  starts in  $s_1$  then it can terminate in  $s_2$
- Semantics of  $\{P\} C \{Q\}$  is  $\text{Hsem } P \ C \ Q$  where:  
$$\text{Hsem } P \ C \ Q = \forall s_1 \ s_2. \text{Ssem } P \ s_1 \wedge \text{Csem } C \ s_1 \ s_2 \Rightarrow \text{Ssem } Q \ s_2$$
- Sometimes write  $\models \{P\} C \{Q\}$  to mean  $\text{Hsem } P \ C \ Q$

# Semantics of commands

- **Assignments**

$$\text{Csem } (V := E) s_1 s_2 = (s_2 = s_1 [(\text{Esem } E s_1)/V])$$

- **Sequences**

$$\text{Csem } (C_1; C_2) s_1 s_2 = \exists s. \text{Csem } C_1 s_1 s \wedge \text{Csem } C_2 s s_2$$

- **Conditional**

$$\begin{aligned} \text{Csem } (\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2) s_1 s_2 \\ &= (\text{Ssem } S s_1 \wedge \text{Csem } C_1 s_1 s_2) \vee (\neg \text{Ssem } S s_1 \wedge \text{Csem } C_2 s_1 s_2) \\ &= \textit{if } \text{Ssem } S s_1 \textit{ then } \text{Csem } C_1 s_1 s_2 \textit{ else } \text{Csem } C_2 s_1 s_2 \end{aligned}$$

- **While-commands**

$$\text{Csem } (\text{WHILE } S \text{ DO } C) s_1 s_2 = \exists n. \text{Iter } n (\text{Ssem } S) (\text{Csem } C) s_1 s_2$$

where the function `Iter` is defined by recursion on  $n$  as follows:

$$\text{Iter } 0 p c s_1 s_2 = \neg(p s_1) \wedge (s_1 = s_2)$$

$$\text{Iter } (n+1) p c s_1 s_2 = p s_1 \wedge \exists s. c s_1 s \wedge \text{Iter } n p c s s_2$$

- argument  $n$  of `Iter` is the number of iterations
- argument  $p$  is a predicate on states (e.g. `Ssem S`)
- argument  $c$  is a semantic function (e.g. `Csem C`)
- arguments  $s_1$  and  $s_2$  are the initial and final states, respectively

# Soundness of Hoare Logic: summary

- **Assignment axiom:**

$$\forall s_1 s_2. \text{Ssem } (Q[E/V]) s_1 \wedge \text{Csem } (V := E) s_1 s_2 \Rightarrow \text{Ssem } Q s_2 \\ \models \{Q[E/V]\}V := E\{Q\}$$

- **Precondition strengthening:**

$$(\forall s. \text{Ssem } P s \Rightarrow \text{Ssem } P' s) \wedge \text{Hsem } P' C Q \Rightarrow \text{Hsem } P C Q \\ (\models P \Rightarrow P') \wedge \models \{P'\}C\{Q\} \Rightarrow \models \{P\}C\{Q\}$$

- **Postcondition weakening:**

$$\text{Hsem } P C Q' \wedge (\forall s. \text{Ssem } Q' s \Rightarrow \text{Ssem } Q s) \Rightarrow \text{Hsem } P C Q \\ \models \{P\}C\{Q'\} \wedge (\models Q' \Rightarrow Q) \Rightarrow \models \{P\}C\{Q\}$$

- **Sequencing rule:**

$$\text{Hsem } P C_1 Q \wedge \text{Hsem } Q C_2 R \Rightarrow \text{Hsem } P (C_1; C_2) R \\ \models \{P\}C_1\{Q\} \wedge \models \{Q\}C_2\{R\} \Rightarrow \models \{P\}C_1; C_2\{R\}$$

- **Conditional rule:**

$$\text{Hsem } (P \wedge S) C_1 Q \wedge \text{Hsem } (P \wedge \neg S) C_2 Q \Rightarrow \text{Hsem } P (\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2) Q \\ \models \{P \wedge S\}C_1\{Q\} \wedge \models \{P \wedge \neg S\}C_2\{Q\} \Rightarrow \models \{P\}\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2\{Q\}$$

- **WHILE rule:**

$$\text{Hsem } (P \wedge S) C P \Rightarrow \text{Hsem } P (\text{WHILE } S \text{ DO } C) (P \wedge \neg S) \\ \models \{P \wedge S\}C\{P\} \Rightarrow \models \{P\}\text{WHILE } S \text{ DO } C$$

# Completeness and decidability of Hoare Logic

- Soundness:  $\vdash \{P\}C\{Q\} \Rightarrow \models \{P\}C\{Q\}$
- Decidability:  $\{T\}C\{F\} \Leftrightarrow C$  doesn't halt
  - the Halting Problem is undecidable
- Completeness: really want  $\models_{\mathcal{I}_{PA}} \{P\}C\{Q\} \Rightarrow PA \vdash \{P\}C\{Q\}$ 
  - to show this not possible, first observe that for any  $P$
  - $\models_{\mathcal{I}_{PA}} \{T\}X:=X\{P\}$  if and only if  $\models_{\mathcal{I}_{PA}} P$
  - $PA \vdash \{T\}X:=X\{P\}$  if and only if  $PA \vdash P$
- If Hoare logic were complete, then taking  $P$  above to be  $G_T$ :  
 $\models_{\mathcal{I}_{PA}} G_T \Rightarrow \models_{\mathcal{I}_{PA}} \{T\}X:=X\{G_T\} \Rightarrow PA \vdash \{T\}X:=X\{G_T\} \Rightarrow PA \vdash G_T$   
contradicting Gödel's theorem
- Must separate completeness of programming and specification logics



## Relative completeness (Cook 1978) – basic idea ✓

- $\models_{\mathcal{I}_{\text{PA}}} \{P\}C\{Q\}$  entails  $\Gamma_{\text{PA}} \vdash \{P\}C\{Q\}$ , where  $\Gamma_{\text{PA}} = \{S \mid \models_{\mathcal{I}_{\text{PA}}} S\}$
- **Proof outline:**
  - define  $\text{wlp}(C, Q)$  in  $\mathcal{L}_{\text{PA}}$
  - straight line code easy - see earlier slides
  - $\text{wlp}(\text{WHILE } S \text{ DO } C, Q)$  needs tricky encoding using Gödel's  $\beta$  function  
(see Winskel's book *The formal semantics of programming languages: an introduction*)
  - $\models_{\mathcal{I}_{\text{PA}}} \{P\}C\{Q\}$  implies  $\models_{\mathcal{I}_{\text{PA}}} P \Rightarrow \text{wlp}(C, Q)$  by induction on  $C$  and semantics
  - $\Gamma_{\text{PA}} \vdash \{\text{wlp}(C, Q)\}C\{Q\}$  by induction on  $C$  and Hoare logic
  - hence  $\models_{\mathcal{I}_{\text{PA}}} \{P\}C\{Q\}$  implies  $\Gamma_{\text{PA}} \vdash \{P\}C\{Q\}$  by precondition strengthening
- Cook's theorem is for any *expressive* assertion language
  - i.e. any language in which  $\text{wlp}(C, Q)$  is definable

## Summary: soundness, decidability, completeness ✓

- Hoare logic is sound
- Hoare logic is undecidable
  - deciding  $\{T\}C\{F\}$  is halting problem
- Hoare logic for our simple language is complete relative to an oracle
  - oracle must be able to prove  $P \Rightarrow \text{wlp}(C, Q)$
  - relative completeness
  - requires expressibility:  $\text{wlp}(C, Q)$  expressible in assertion language

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

- Clarke showed relative completeness fails for complex languages

## Additional topics



Note: only a fragment of these additional topics will be covered!

- Blocks and local variables
- FOR-commands
- Arrays
- Correct-by-Construction (program refinement)
- Separation Logic

## Blocks and local variables

- **Syntax:** BEGIN VAR  $V_1$ ;  $\dots$  VAR  $V_n$ ;  $C$  END
- **Semantics:** command  $C$  is executed, then the values of  $V_1, \dots, V_n$  are restored to the values they had before the block was entered
  - the initial values of  $V_1, \dots, V_n$  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

$$\frac{\vdash \{P\} C \{Q\}}{\vdash \{P\} \text{ BEGIN VAR } V_1; \dots; \text{ VAR } V_n; C \text{ END } \{Q\}}$$

where none of the variables  $V_1, \dots, V_n$  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  $V_1, \dots, V_n$  occur in  $P$  or  $Q$  is an example of a *side condition*

- From

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

it follows by the block rule that

$\vdash \{X=x \wedge Y=y\} \text{ BEGIN VAR } R; R:=X; X:=Y; Y:=R \text{ END } \{Y=x \wedge X=y\}$

since  $R$  does not occur in  $X=x \wedge Y=y$  or  $X=y \wedge Y=x$

- However from

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

one *cannot* deduce

$\vdash \{X=x \wedge Y=y\} \text{ BEGIN VAR } R; R:=X; X:=Y \text{ END } \{R=x \wedge X=y\}$

since  $R$  occurs in  $R=x \wedge X=y$

# 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  $e_1, e_1 + 1, \dots, e_2$  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  $1, \dots, m$
  - 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  $V$ ;  $V := e_1$ ;  $C$ ;  $V := e_1 + 1$ ;  $C$ ; ... ;  $V := e_2$ ;  $C$  END

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

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

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

# The Rule of Constancy (Derived Frame Rule)

- The following derived rule is used on the next slide

The rule of constancy

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

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
- Assume  $C$  doesn't modify  $V$  and  $\vdash \{P\} C \{P[V+1/V]\}$  then:
  - $\vdash \{P \wedge V=v\} C \{P[V+1/V] \wedge V=v\}$  (assumption + constancy rule)
  - $\vdash \{P[V+1/V] \wedge V=v\} V:=V+1 \{P \wedge V=v+1\}$  (assign. ax + pre. streng.)
  - $\vdash \{P \wedge V=v\} C; V:=V+1 \{P \wedge V=v+1\}$  (sequencing)
- So  $C; V:=V+1$  has  $P$  as an invariant and increments  $V$

# Towards the FOR-Rule

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

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

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

- Hence:

$\vdash \{P[e_1/V]\} V := e_1 \{P \wedge V=e_1\}$  (assign. ax + pre. streng.)

⋮

$\vdash \{P \wedge V=v\} C; V := V+1 \{P \wedge V=v+1\}$  (last slide;  $V = e_1, e_1+1, \dots, e_2-1$ )

⋮

$\vdash \{P \wedge V=v\} C; V := V+1 \{P \wedge V=e_2+1\}$

$\vdash \{P \wedge V=e_2\} C \{P[V+1/V] \wedge V=e_2\}$  (assign. ax + assumption + constancy)

$\vdash \{P \wedge V=e_2\} C \{P[e_2+1/V]\}$  (post. weak.)

- Hence by the sequencing and block rules

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

## 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

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

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

- Note  $(E_1 \leq V) \wedge (V \leq E_2)$  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 the rule below ( $e_1, e_2$  are fresh auxiliary variables):

$$\frac{\vdash \{P \wedge (e_1 \leq V) \wedge (V \leq e_2)\} C \{P[V+1/V]\}}{\vdash \{P[E_1/V] \wedge (E_1 \leq E_2) \wedge (E_1 = e_1) \wedge (E_2 = e_2)\} \text{ FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C \{P[e_2+1/V]\}}$$

## The FOR-axiom

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

### The FOR-axiom

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

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

## Ensuring Soundness

- 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 is sound for 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
- Approach (ii) is proving soundness of the logic

# 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 by Clarke's Theorem it is known to be impossible to devise relatively complete Floyd-Hoare logics for languages with certain constructs
  - 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



7.1. (module rule)

$$(1) \quad Q \supset Q0(A/t),$$

$$(2) \quad P1\{\mathbf{const} \ K; \mathbf{var} \ V; S_4\} Q4(A/t) \wedge Q,$$

$$(3) \quad P2(A/t) \wedge Q\{S_2\} Q2(A/t) \wedge Q,$$

$$(4) \quad \exists g1(P3(A/t) \wedge Q\{S_3\} Q3(A/t) \wedge g = g1(A, c, d)),$$

$$(5) \quad \exists g(P3(A/t) \wedge Q \supset Q3(A/t)),$$

$$(6) \quad P6(A/t) \wedge Q\{S_6\} Q1,$$

$$(7) \quad P \supset P1(a/c),$$

$$(8.1) \quad [Q0(a/c, x/t, x'/t') \supset (P2(x/t, x'/t', a2/x2, e2/c2, a/c) \wedge \\ (Q2(x2\#/t, x'/t', a2\#/x2, e2/c2, a/c, y2\#/y2, a2/x2', y2/y2') \supset \\ R1(x2\#/x, a2\#/a2, y2\#/y2))) \{x. p(a2, e2)\} R1 \wedge Q0(a/c, x/t, x'/t'),$$

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

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

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

⊢

$$(8.5) \quad P(x\#/x) \{x. \text{Initially}; S; x. \text{Finally}\} R(x\#/x)$$

---


$$P\{\mathbf{var} \ x: T(a); S\} R \wedge Q1$$

# 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

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

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:  $\vdash \{X=Y \wedge A(Y)=0\} A(X) := 1 \{X=Y \wedge A(Y)=0\}$
- 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

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

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

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

- Array assignment axiom just ordinary assignment axiom

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

- Thus:

### The array assignment axiom

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

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*
  - don't confuse with Frame Rule of Separation Logic (later)
  - “frame” is a rather overloaded word!

## New Topic: Separation logic

- One of several competing methods for reasoning about pointers
- Details took 30 years to evolve
- Shape predicates due to Rod Burstall in the 1970s
- Separation logic: by O'Hearn, Reynolds and Yang around 2000
- Several partially successful attempts before separation logic
- Very active research area
  - QMUL, UCL, Cambridge, Harvard, Princeton, Yale
  - Microsoft
  - startup Monoidics bought by Facebook

# Pointers and the state

- So far the state just determined the values of variables
  - values assumed to be numbers
  - preconditions and postconditions are first-order logic statements
  - state same as a valuation  $s : Var \rightarrow Val$
- To model pointers – e.g. as in C – add *heap* to state
  - heap maps *locations* (pointers) to their contents
  - *store* maps variables to values (previously called state)
  - contents of locations can be locations or values

$$\begin{array}{ccccccc} X & \mapsto & l_1 & \mapsto & l_2 & \mapsto & v \\ & \textit{store} & & \textit{heap} & & \textit{heap} & \end{array}$$

## Heap semantics

$$\textit{Store} = Var \rightarrow Val$$

(assume  $Num \subseteq Val$ ,  $nil \in Val$  and  $nil \notin Num$ )

$$\textit{Heap} = Num \rightarrow_{fin} Val$$

$$\textit{State} = \textit{Store} \times \textit{Heap}$$

- **Note:** store also called *stack* or *environment*; heap also called *store*



# Adding pointer operations to our language

## Expressions:

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

## Boolean expressions:

$B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \dots$

## commands:

$C ::= V := E$	value assignments
$V := [E]$	fetch assignments
$[E_1] := E_2$	heap assignments (heap mutation)
$V := \text{cons}(E_1, \dots, E_n)$	allocation assignments
$\text{dispose}(E)$	pointer disposal
$C_1 ; C_2$	sequences
$\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2$	conditionals
$\text{WHILE } B \text{ DO } C$	while commands

# Pointer manipulation constructs and faulting

- Commands executed in a state  $(s, h)$
- Reading, writing or disposing pointers might *fault*
- Fetch assignments:  $V := [E]$ 
  - evaluate  $E$  to get a location  $l$
  - fault if  $l$  is not in the heap
  - otherwise assign contents of  $l$  in heap to the variable  $V$
- Heap assignments:  $[E_1] := E_2$ 
  - evaluate  $E_1$  to get a location  $l$
  - fault if the  $l$  is not in the heap
  - otherwise store the value of  $E_2$  as the new contents of  $l$  in the heap
- Pointer disposal:  $\text{dispose}(E)$ 
  - evaluate  $E$  to get a pointer  $l$  (a number)
  - fault if  $l$  is not in the heap
  - otherwise remove  $l$  from the heap

# Allocation assignments

- Allocation assignments:  $V := \text{cons}(E_1, \dots, E_n)$ 
  - choose  $n$  consecutive locations that are not in the heap, say  $l, l+1, \dots$
  - extend the heap by adding  $l, l+1, \dots$  to it
  - assign  $l$  to the variable  $V$  in the store
  - make the values of  $E_1, E_2, \dots$  be the new contents of  $l, l+1, \dots$  in the heap
- Allocation assignments never fault
- Allocation assignments are *non-deterministic*
  - any suitable  $l, l+1, \dots$  not in the heap can be chosen
  - always exists because the heap is finite

## Example (different from the background reading)

$X := \text{cons}(0, 1, 2); [X] := Y+1; [X+1] := Z; Y := [Y+Z]$

- $X := \text{cons}(0, 1, 2)$  allocates three new pointers, say  $l, l+1, l+2$ 
  - $l$  initialised with contents 0,  $l+1$  with 1 and  $l+2$  with 2
  - variable  $X$  is assigned  $l$  as its value in store
- $[X] := Y+1$  changes the contents of  $l$ 
  - $l$  gets value of  $Y+1$  as new contents in heap
- $[X+1] := Z$  changes the contents of  $l+1$ 
  - $l+1$  gets the value of  $Z$  as new contents in heap
- $Y := [Y+Z]$  changes the value of  $Y$  in the store
  - $Y$  assigned in the store the contents of  $Y+Z$  in the heap
  - faults if the value of  $Y+Z$  is not in the heap

# Local Reasoning and Separation Logic

- Want to just reason about just those locations being modified
  - assume all other locations unchanged
- Solution: separation logic
  - **small** and **forward** assignment axioms + **separating conjunction**
  - **small** means just applies to fragment of heap (*footprint*)
  - **forward** means Floyd-style forward rules that support symbolic execution
  - **non-faulting semantics** of Hoare triples
  - symbolic execution used by tools like smallfoot
  - **separating conjunction** solves frame problem - like rule of constancy for heap
- Need new kinds of assertions to state separation logic axioms

## Sneak preview of the Frame Rule

### The frame rule

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

where no variable modified by  $C$  occurs free in  $R$ .

- Separating conjunction  $P \star Q$ 
  - heap can be split into two disjoint components
  - $P$  is true of one component and  $Q$  of the other
  - $\star$  is commutative and associative

# Local Reasoning and Separation Logic

- Want to just reason about just those locations being modified
  - assume all other locations unchanged
- Solution: separation logic
  - **small** and **forward** assignment axioms + **separating conjunction**
  - **small** means just applies to fragment of heap (*footprint*)
  - **forward** means Floyd-style forward rules that support symbolic execution
  - **non-faulting semantics** of Hoare triples
  - symbolic execution used by tools like smallfoot
  - **separating conjunction** solves frame problem - like rule of constancy for heap
- Need new kinds of assertions to state separation logic axioms

## Separation logic assertions: emp

- emp is an atomic statement of separation logic
- emp is true iff the heap is empty
- The semantics of emp is:  
$$\text{emp}(s, h) \Leftrightarrow h = \{\}$$
(where  $\{\}$  is the empty heap)
- Abbreviation:  $E_1 \dot{=} E_2 =_{def} (E_1 = E_2) \wedge \text{emp}$
- From the semantics:  $(E_1 \dot{=} E_2)(s, h) \Leftrightarrow E_1(s) = E_2(s) \wedge h = \{\}$
- $E_1 = E_2$  is independent of the heap and only depends on the store
- Semantics of  $E_1 = E_2$  is:  
$$(E_1 = E_2)(s, h) \Leftrightarrow E_1(s) = E_2(s)$$
  
no constraint on the heap – any  $h$  will do



## Separation logic: small axioms and faulting

- One might expect a heap assignment axiom to entail:

$$\vdash \{\text{T}\} [0] := 0 \{0 \mapsto 0\}$$

i.e. after executing  $[0] := 0$  the contents of location 0 in the heap is 0

- Recall the sneak preview of the frame rule:

The frame rule

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

where no variable modified by  $C$  occurs free in  $R$ .

- Taking  $R$  to be the points-to statement  $0 \mapsto 1$  yields:

$$\vdash \{\text{T} \star 0 \mapsto 1\} [0] := 0 \{0 \mapsto 0 \star 0 \mapsto 1\}$$

something is wrong with the conclusion!

- Solution: define Hoare triple so  $\vdash \{\text{T}\} [0] := 0 \{0 \mapsto 0\}$  is not true

# Non-faulting interpretation of Hoare triples

- The *non-faulting semantics* of Hoare triples  $\{P\} C \{Q\}$  is:

if  $P$  holds then

- (i) executing  $C$  doesn't fault **and**
- (ii) if  $C$  terminates then  $Q$  holds

$$\models \{P\} C \{Q\} = \forall s \ h. P(s, h) \Rightarrow \neg(C(s, h)\text{fault}) \wedge \forall s' \ h'. C(s, h)(s', h') \Rightarrow Q(s', h')$$

- Now  $\vdash \{T\} [0] := 0 \{0 \mapsto 0\}$  is not true as  $([0] := 0)(s, \{\})\text{fault}$
- Recall the sneak preview of the frame rule:

The frame rule

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

where no variable modified by  $C$  occurs free in  $R$ .

- So can't use frame rule to get  $\vdash \{T \star 0 \mapsto 1\} [0] := 0 \{0 \mapsto 0 \star 0 \mapsto 1\}$

# Store assignment axiom

## Store assignment axiom

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

where  $v$  is an auxiliary variable not occurring in  $E$ .

- $E_1 \doteq E_2$  means value of  $E_1$  and  $E_2$  equal in the store **and** heap is empty
- In Hoare logic (no heap) this is equivalent to the assignment axiom

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

store assign. ax.

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

rule of constancy

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

exists introduction

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

predicate logic

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

$[V/V]$  is identity

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

predicate logic:  $v$  not in  $E$

$$\vdash \{\top \wedge Q[E/V]\} V := E \{(\exists v. V=E[v/V]) \wedge Q\}$$

predicate logic

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

rules of consequence

- Separation logic: exists introduction valid, rule of constancy invalid

## Fetch assignment axiom

### Fetch assignment axiom

$$\vdash \{(V = v_1) \wedge E \mapsto v_2\} V := [E] \{(V = v_2) \wedge E[v_1/V] \mapsto v_2\}$$

where  $v_1, v_2$  are auxiliary variables not occurring in  $E$ .

- Precondition guarantees the assignment doesn't fault
- $V$  is assigned the contents of  $E$  in the heap
- Small axiom: precondition and postcondition specify singleton heap
- If neither  $V$  nor  $v$  occur in  $E$  then the following holds:

$$\vdash \{E \mapsto v\} V := [E] \{(V = v) \wedge E \mapsto v\}$$

(proof: instantiate  $v_1$  to  $V$  and  $v_2$  to  $v$  and then simplify)

## Heap assignment axiom

Heap assignment axiom (heap mutation)

$$\vdash \{E \mapsto \_ \} [E] := F \{E \mapsto F \}$$

- Precondition guarantees the assignment doesn't fault
- Contents of  $E$  in heap is updated to be value of  $F$
- Small axiom: precondition and postcondition specify singleton heap

# Pointer allocation

## Allocation assignment axiom

$$\vdash \{V \doteq v\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1[v/V], \dots, E_n[v/V]\}$$

where  $v$  is an auxiliary variable not equal to  $V$  or occurring in  $E_1, \dots, E_n$

- Never faults

- If  $V$  doesn't occur in  $E_1, \dots, E_n$  then:

$$\vdash \{V \doteq v\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1[v/V], \dots, E_n[v/V]\} \quad \text{alloc. assign. ax}$$

$$\vdash \{V \doteq v\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1, \dots, E_n\} \quad V \text{ not in } E_i \text{ assump.}$$

$$\vdash \{\exists v. V \doteq v\} V := \text{cons}(E_1, \dots, E_n) \{\exists v. V \mapsto E_1, \dots, E_n\} \quad \text{exists intro.}$$

$$\vdash \{\exists v. V = v \wedge \text{emp}\} V := \text{cons}(E_1, \dots, E_n) \{\exists v. V \mapsto E_1, \dots, E_n\} \quad \text{definition of } \doteq$$

$$\vdash \{\text{emp}\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1, \dots, E_n\} \quad \text{predicate logic}$$

- Which is a derivation of:

## Derived allocation assignment axiom

$$\vdash \{\text{emp}\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1, \dots, E_n\}$$

where  $V$  doesn't occur in  $E_1, \dots, E_n$ .

# Pointer deallocation

## Dispose axiom

$$\vdash \{E \mapsto -\} \text{dispose}(E) \{\text{emp}\}$$

- Attempting to deallocate a pointer not in the heap faults
- Small axiom: singleton precondition heap, empty postcondition heap
- Sanity checking example proof:

$$\vdash \{E_1 \mapsto -\} \text{dispose}(E_1) \{\text{emp}\}$$

dispose axiom

$$\vdash \{\text{emp}\} V := \text{cons}(E_2) \{V \mapsto E_2\}$$

derived allocation assignment axiom

$$\vdash \{E_1 \mapsto -\} \text{dispose}(E_1); V := \text{cons}(E_2) \{V \mapsto E_2\}$$

sequencing rule

## Compound command rules

- Following rules apply to both Hoare logic and separation logic

The sequencing rule

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

The conditional rule

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

The WHILE-rule

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

- For separation logic, need to think about faulting



# Local Reasoning and Separation Logic

- Want to just reason about just those locations being modified
  - assume all other locations unchanged
- Solution: separation logic
  - **small** and **forward** assignment axioms + **separating conjunction**
  - **small** means just applies to fragment of heap (*footprint*)
  - **forward** means Floyd-style forward rules that support symbolic execution
  - **non-faulting semantics** of Hoare triples
  - symbolic execution used by tools like smallfoot
  - **separating conjunction** solves frame problem - like rule of constancy for heap
- Need new kinds of assertions to state separation logic axioms

## Summary of pointer manipulating axioms

### Store assignment axiom

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

where  $v$  is an auxiliary variable not occurring in  $E$ .

### Fetch assignment axiom

$$\vdash \{(V = v_1) \wedge E \mapsto v_2\} V := [E] \{(V = v_2) \wedge E[v_1/V] \mapsto v_2\}$$

where  $v_1, v_2$  are auxiliary variables not occurring in  $E$ .

### Heap assignment axiom

$$\vdash \{E \mapsto \_ \} [E] := F \{E \mapsto F\}$$

### Allocation assignment axiom

$$\vdash \{V \doteq v\} V := \text{cons}(E_1, \dots, E_n) \{V \mapsto E_1[v/V], \dots, E_n[v/V]\}$$

where  $v$  is an auxiliary variable not equal to  $V$  or occurring in  $E_1, \dots, E_n$

### Dispose axiom

$$\vdash \{E \mapsto \_ \} \text{dispose}(E) \{\text{emp}\}$$

## The frame rule

### The rule of constancy

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

where no variable modified by  $C$  occurs free in  $R$ .

- Rule of constancy is not valid for heap assignments

$$\vdash \{X \mapsto \_ \} [X] := 0 \{X \mapsto 0\}$$

but not (c.f. arrays)

$$\{X \mapsto \_ \wedge Y \mapsto 1\} [X] := 0 \{X \mapsto 0 \wedge Y \mapsto 1\}$$

as  $X$  and  $Y$  could initially both be bound to the same location

- Using  $\star$  instead of  $\wedge$  forces  $X$  and  $Y$  to point to different locations

### The frame rule

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

where no variable modified by  $C$  occurs free in  $R$ .

- Soundness a little tricky due to faulting

## Compound command rules

- Following rules apply to both Hoare logic and separation logic

The sequencing rule

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

The conditional rule

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

The WHILE-rule

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

- For separation logic, need to think about faulting

{contents of pointers X and Y are equal} X := [X] ; Y := [Y] {X = Y} ✓

● **Proof:**

- $\vdash \{(\mathbf{X} = x) \wedge \mathbf{X} \mapsto v\} \mathbf{X} := [\mathbf{X}] \{(\mathbf{X} = v) \wedge x \mapsto v\}$       fetch assignment axiom
- $\vdash \{(\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v\} \mathbf{Y} := [\mathbf{Y}] \{(\mathbf{Y} = v) \wedge y \mapsto v\}$       fetch assignment axiom
- $\vdash \{((\mathbf{X} = x) \wedge \mathbf{X} \mapsto v) \star ((\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v)\}$       frame rule  
 $\quad \mathbf{X} := [\mathbf{X}]$   
 $\quad \{((\mathbf{X} = v) \wedge x \mapsto v) \star (((\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v))\}$
- $\vdash \{((\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v) \star ((\mathbf{X} = v) \wedge x \mapsto v)\}$       frame rule  
 $\quad \mathbf{Y} := [\mathbf{Y}]$   
 $\quad \{((\mathbf{Y} = v) \wedge y \mapsto v) \star ((\mathbf{X} = v) \wedge x \mapsto v)\}$
- $\vdash \{((\mathbf{X} = x) \wedge \mathbf{X} \mapsto v) \star ((\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v)\}$       sequencing rule and commutativity of  $\star$   
 $\quad \mathbf{X} := [\mathbf{X}] ; \mathbf{Y} := [\mathbf{Y}]$   
 $\quad \{((\mathbf{X} = v) \wedge x \mapsto v) \star ((\mathbf{Y} = v) \wedge y \mapsto v)\}$
- $\vdash \{\exists v x y. ((\mathbf{X} = x) \wedge \mathbf{X} \mapsto v) \star ((\mathbf{Y} = y) \wedge \mathbf{Y} \mapsto v)\}$       exists-introduction (3 times)  
 $\quad \mathbf{X} := [\mathbf{X}] ; \mathbf{Y} := [\mathbf{Y}]$   
 $\quad \{\exists v x y. ((\mathbf{X} = v) \wedge x \mapsto v) \star ((\mathbf{Y} = v) \wedge y \mapsto v)\}$
- $\vdash \{\exists v. \mathbf{X} \mapsto v \star \mathbf{Y} \mapsto v\} \mathbf{X} := [\mathbf{X}] ; \mathbf{Y} := [\mathbf{Y}] \{\mathbf{X} = \mathbf{Y}\}$       rules of consequence (see next slide)

## Current research and the future

- Extending separation logic to cover practical language features
  - various concurrency idioms
  - objects
- Building tools to mechanise separation logic
  - much work on *shape analysis*, e.g.:  
 $\{\exists x. \text{list } x \ X\}$   
 $Y := \text{nil};$   
 $\text{WHILE } \neg(X = \text{nil}) \ \text{DO } (Z := [X+1]; [X+1] := Y; Y := X; X := Z)$   
 $\{\exists x. \text{list } x \ Y\}$   
automatically finds memory usage errors
- Finally, something to think about:  
*should we be verifying code in old fashioned languages (pragmatism)*  
*or creating new methods to create correct software (idealism)?*

“The tension between idealism and pragmatism is as profound (almost) as that between good and evil (and just as pervasive).”

[Tony Hoare]

## New Topic: 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

$$E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \dots$$
$$B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \dots$$
$$C ::= \text{SKIP} \quad (\text{does nothing, SKIP-Axiom is } \vdash [P] \text{ SKIP } [P])$$

|  $V := E$

|  $C_1 ; C_2$

| IF  $B$  THEN  $C_1$  ELSE  $C_2$

| BEGIN VAR  $V_1 ; \dots \text{ VAR } V_n ; C$  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  $\wedge$  Y > 0  $\wedge$  Q=0, X=R+Y $\times$ Q]

- Think of a specification as defining the set of implementations

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

- For example

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

- Don't confuse use of  $\{\dots\}$  as set brackets and in Hoare triples

## 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:

$$c_1; \dots ; c_n = \{ C \mid \exists C_1 \dots C_n. C = C_1; \dots ; C_n \wedge C_1 \in c_1 \wedge \dots \wedge C_n \in c_n \}$$

$$= \{ C_1; \dots ; C_n \mid C_1 \in c_1 \wedge \dots \wedge C_n \in c_n \}$$

$$\text{BEGIN VAR } V_1; \dots \text{ VAR } V_n; c \text{ END} = \{ \text{BEGIN VAR } V_1; \dots \text{ VAR } V_n; C \text{ END} \mid C \in c \}$$

$$\text{IF } S \text{ THEN } c_1 \text{ ELSE } c_2 = \{ \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \mid C_1 \in c_1 \wedge C_2 \in c_2 \}$$

$$\text{WHILE } S \text{ DO } c = \{ \text{WHILE } S \text{ DO } C \mid C \in c \}$$

# 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  $p_1 \sqsubseteq p_2$
- A program development takes us from the specification, through a series of mixed programs to (we hope) executable code

$$spec \supseteq mixed_1 \supseteq \dots \supseteq mixed_n \supseteq code$$

## The Skip Law

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

- **Derivation:**

$$C \in \{\text{SKIP}\}$$

$$\Leftrightarrow C = \text{SKIP}$$

$$\Rightarrow \vdash [P] C [P] \quad (\text{Skip Axiom})$$

$$\Leftrightarrow C \in [P, P] \quad (\text{Definition of } [P, P])$$

- **Examples**

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

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

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

# Notational Convention

- Omit { and } around individual commands

- Skip law becomes:

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

- Examples become:

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

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

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



# Assignment Law

## The Assignment Law

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

- Derivation

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

- Examples

$$[Y=1, X=1] \supseteq X:=Y$$

$$[X+1=n+1, X=n+1] \supseteq X:=X+1$$

## Precondition Weakening

$$\begin{array}{l} [P, Q] \supseteq [R, Q] \\ \text{provided } \vdash P \Rightarrow R \end{array}$$

## Postcondition Strengthening

$$\begin{array}{l} [P, Q] \supseteq [P, R] \\ \text{provided } \vdash R \Rightarrow Q \end{array}$$

- 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’

# Derived Assignment Law

## Derived Assignment Law

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

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

- **Derivation**

$$[P, Q]$$

$$\supseteq (\text{Precondition Weakening } \vdash P \Rightarrow Q[E/V])$$

$$[Q[E/V], Q]$$

$$\supseteq (\text{Assignment})$$

$$V := E$$

- **Example**

$$[T, R=X]$$

$$\supseteq (\text{Derived Assignment } \vdash T \Rightarrow X=X)$$

$$R := X$$

## The Sequencing Law

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

- Derivation of Sequencing Law

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

- Example

$$\begin{aligned} [T, R=X \wedge Q=0] & \\ \supseteq (\text{Sequencing}) & \\ [T, R=X]; [R=X, R=X \wedge Q=0] & \\ \supseteq (\text{Derived Assignment } \vdash T \Rightarrow X=X) & \\ R:=X; [R=X, R=X \wedge Q=0] & \\ \supseteq (\text{Derived Assignment } \vdash R=X \Rightarrow R=X \wedge 0=0) & \\ R:=X; Q:=0 & \end{aligned}$$

## 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

$[T, R=X \wedge Q=0]$

$\supseteq$  (Sequencing)

$[T, Q=0] ; [Q=0, R=X \wedge Q=0]$

$\supseteq$  (Derived Assignment  $\vdash T \Rightarrow 0=0$ )

$Q:=0; [Q=0, R=X \wedge Q=0]$

$\supseteq$  (Derived Assignment  $\vdash Q=0 \Rightarrow X=X \wedge Q=0$ )

$Q:=0; R:=X$

# Inefficient Programs

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

$[T, R=X \wedge Q=0]$

$\supseteq$  (Sequencing)

$[T, R=X \wedge Q=0] ; [R=X \wedge Q=0, R=X \wedge Q=0]$

$\supseteq$  (as previous example)

$Q:=0; R:=X; [R=X \wedge Q=0, R=X \wedge Q=0]$

$\supseteq$  (Skip)

$Q:=0; R:=X; \text{SKIP}$

# 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**

$[X=x \wedge Y=y, \quad X=y \wedge Y=x]$

$\supseteq$  (Sequencing)

$[X=x \wedge Y=y, \quad X=x \wedge Y=x] ; [X=x \wedge Y=x, \quad X=y \wedge Y=x]$

$\supseteq$  (Derived Assignment  $\vdash X=x \wedge Y=y \Rightarrow X=x \wedge X=x$ )

$Y:=X; [X=x \wedge Y=x, \quad X=y \wedge Y=x]$

$\supseteq$  (Sequencing)

$Y:=X;$

$[X=x \wedge Y=x, \quad Y=y \wedge Y=x];$

$[Y=y \wedge Y=x, \quad X=y \wedge Y=x]$

$\supseteq$  (Assignment)

$Y:=X;$

$[X=x \wedge Y=x, \quad Y=y \wedge Y=x];$

$X:=Y$

(no way to refine this!)

## The Block Law

$$[P, Q] \supseteq \text{BEGIN VAR } V_1; \dots; \text{VAR } V_n; [P, Q] \text{ END}$$

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

- Derivation: exercise
- Example

$$[X=x \wedge Y=y, X=y \wedge Y=x]$$
$$\supseteq (\text{Block})$$
$$\text{BEGIN VAR } R; [X=x \wedge Y=y, X=y \wedge Y=x] \text{ END}$$
$$\supseteq (\text{Sequencing and Derived Assignment})$$
$$\text{BEGIN VAR } R; R:=X; X:=Y; Y:=R \text{ END}$$



## The Conditional Law

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

- 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!

## The While Law

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

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

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

- Example

$$[X=R+Y \times Q \wedge Y > 0, X=R+Y \times Q \wedge Y > 0 \wedge \neg Y \leq R]$$
$$\supseteq (\text{While } \vdash X=R+Y \times Q \wedge Y > 0 \wedge Y \leq R \Rightarrow R \geq 0)$$

WHILE  $Y \leq R$  DO

$$[X=R+Y \times Q \wedge Y > 0 \wedge Y \leq R \wedge R=n,$$
$$X=R+Y \times Q \wedge Y > 0 \wedge R < n]$$

## More Notation

- The notation

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

is used to abbreviate:

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

- 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:

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

# 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

## Exciting mystery guest lecture!

- The last lecture will be on some current research
  - hot area where Computer Lab is world leader
- Shows that Hoare logic still relevant
  - possible PhD area if you like this course