Title:	*Hoare Logic*
Lecturer:	**Mike Gordon**
Class:	**CST Part II**
Duration:	**8 lectures**

The Part II course *Hoare Logic* has evolved from earlier Part II courses (some with different names). The web page for the current (i.e. 2014-2015) course is `http://www.cl.cam.ac.uk/~mjcg/HoareLogic/`. On this page there is a link to a PDF document called "background reading" that provides material that may help deepen your understanding of the topics covered in the lectures. The next page of this handout reproduces the first page of the background reading document.[1]

Some of the slides in this handout are edited versions of ones used in past courses and some are new this year. There is a fair chance that notational inconsistencies, omissions and errors are present. If you discover such defects please send details to `Mike.Gordon@cl.cam.ac.uk`. I will correct errors in the online version of the slides, so the printed copies in this handout might differ from what you see in the lectures.

I do not plan to present all the slides in this handout during the lectures. The reason for including many more slides than will be used is because the extra ones may provide a learning aid for reviewing the course.

The slides presented in the lectures are indicated by a tick ✓ to the right of the title; slides with content that goes beyond what is covered in the 2014/15 course are indicated by a cross ✗. The ticks and crosses in the slides will only become accurate after the lecture is given. I plan to update the online handout and slides on the course web page after each lecture. For lectures not yet given the ticks and crosses are from last year's course. I may present a different subset of the material in this year's course, thus the ticks and crosses may well change. **The examination questions will be based on the material presented in the lectures.**

Acknowledgements.
Many people have helped create these slides and provided feedback on them. Particular thanks to Paul Curzon and John Wickerson.

MJCG September 19, 2014

[1] Some parts of the background reading consist of revised and updated extracts from the book *Programming Language Theory and its Implementation*, Michael J. C. Gordon, Prentice-Hall International Series in Computer Science (edited by Professor C.A.R Hoare), 1988. Although this book is long out of print, Google reveals that it is available online (e.g. `http://bit.ly/1uOm7HA`).

Background reading on *Hoare Logic*

Mike Gordon

Learning Guide for the CST Part II course. This document aims to provide background reading to support the lectures – think of it as a free downloadable textbook. Chapters 1–5 introduce classical ideas of specification and proof of programs due to Floyd and Hoare.[1] Although much of the material is old – see the dates on some of the cited references – it is still a foundation for current research. Chapter 6 is a very brief introduction to program refinement; this provides rules to 'calculate' an implementation from a Hoare-style specification. Chapter 7 is an introduction to the ideas of separation logic, an extension of Hoare logic for specifying and verifying programs that manipulate pointers. Separation logic builds on early ideas of Burstall, but its modern form is due to O'Hearn and Reynolds.

Note that there may be topics presented in the lectures that are not covered in this document and there may be material in this document that is not related to the topics covered in the lectures. For example, the topics of program refinement and separation logic may only be described very superficially, if at all. **The examination questions will be based on the material presented in the lectures.**

The Part II course *Hoare Logic* has evolved from an earlier Part II course, whose web page can be found on my home page (`www.cl.cam.ac.uk/~mjcg`). Some exam questions from that course might be good exercises (but note that some are based on material not covered in this course). A separate document containing exercises for the current course is available from the web page.

Warning. The material here consists of reorganized extracts from lecture notes for past courses, together with new material. There is a fair chance that notational inconsistencies, omissions and errors are present. If you discover such defects please send details to `Mike.Gordon@cl.cam.ac.uk`.

Acknowledgements. Thanks to Martin Vechev and John Wickerson for finding many errors (some serious) in a previous draft of these notes and also for suggestions for improving the text.

MJCG September 19, 2014

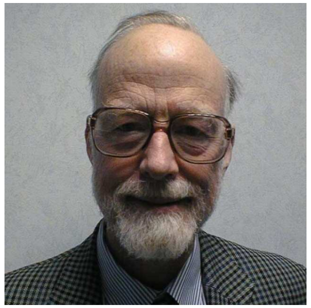
[1] Hoare Logic is sometimes called Floyd-Hoare Logic, due to the important contributions of Floyd to the underlying ideas.

Topics and corresponding slides

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



Program Specification and Verification

- This course is about *formal* ways of specifying and validating software
- This contrasts with *informal* methods:
  - natural language specifications
  - testing
- Formal methods are *not* a panacea
  - formally verified designs may still not work
  - can give a false sense of security
- Assurance versus debugging
  - formal verification (FV) can reveal hard-to-find bugs
  - can also be used for assurance e.g. “proof of correctness”
  - Microsoft use FV for debugging, NSA use FV for assurance
- Goals of course:
  - enable you to understand and criticise formal methods
  - provide a stepping stone to current research

Testing

- Testing can quickly find obvious bugs
  - only trivial programs can be tested exhaustively
  - the cases you do not test can still hide bugs
  - coverage tools can help
- How do you know what the correct test results should be?
- Many industries’ standards specify maximum failure rates
  - e.g. fewer than 10^{-6} failures per second
  - assurance that such rates have been achieved cannot be obtained by testing

Formal Methods

- *Formal Specification* - using mathematical notation to give a precise description of what a program should do
- *Formal Verification* - using precise rules to mathematically prove that a program satisfies a formal specification
- *Formal Development (Refinement)* - developing programs in a way that ensures mathematically they meet their formal specifications
- Formal Methods should be used in conjunction with testing, *not* as a replacement

Should we always use formal methods?

- They can be expensive
  - though can be applied in varying degrees of effort
- There is a trade-off between expense and the need for correctness
- It may be better to have something that works most of the time than nothing at all
- For some applications, correctness is especially important
  - nuclear reactor controllers
  - car braking systems
  - fly-by-wire aircraft
  - software controlled medical equipment
  - voting machines
  - cryptographic code
- Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible

Floyd-Hoare Logic

- This course is concerned with Floyd-Hoare Logic
  - also known just as Hoare Logic
- Floyd-Hoare Logic is a method of reasoning mathematically about *imperative* programs
- It is the basis of mechanized program verification systems
  - the architecture of these will be described later
- Industrial program development methods like SPARK use ideas from Floyd-Hoare Logic to obtain high assurance
- Developments to the logic still under active development
  - e.g. separation logic (reasoning about pointers)
  - 2/3 of 2010 BCS Distinguished Dissertation awards concerned separation logic

A Little Programming Language ✓

Expressions:

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

Boolean expressions:

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

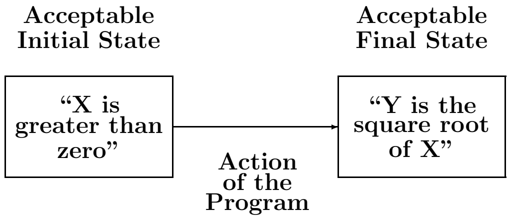
Commands:

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

Some Notation

- Programs are built out of *commands* like assignments, conditionals, while-loops etc
- The terms ‘program’ and ‘command’ are synonymous
  - the former generally used for commands representing complete algorithms
- The term ‘statement’ is used for conditions on program variables that occur in correctness specifications
  - potential for confusion: some people use this word for commands

Specification of Imperative Programs



Hoare's notation

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

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

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

Meaning of Hoare's Notation

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

Formal versus Informal Proof

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

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

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

The Structure of Proofs

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

Formal proof is syntactic 'symbol pushing'

- Formal Systems reduce verification and proof to symbol pushing
- The rules say...
  - if you have a string of characters of this form
  - you can obtain a new string of characters of this other form
- Even if you don't know what the strings are intended to mean, provided the rules are designed properly and you apply them correctly, you will get correct results
  - though not necessarily the desired result
- Thus computers can do formal verification
- Formal verification by hand generally not feasible
  - maybe hand verify high-level design, but not code
- Famous paper that's worth reading:
  - "Social processes and the proofs of theorems and programs". R. A. DeMillo, R. J. Lipton, and A. J. Perlis. CACM, May 1979
- Also see the book "Mechanizing Proof" by Donald MacKenzie

Hoare's Verification Grand Challenge

- Bill Gates, keynote address at WinHec 2002

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

- Hoare has posed a challenge

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

The Deliverables

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

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

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

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

Hoare Logic and Verification Conditions ✓

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

Partial Correctness Specification

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

Total Correctness Specification

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

Total Correctness

- Informally:

  Total correctness = Termination + Partial correctness
- Total correctness is the ultimate goal
  - usually easier to show partial correctness and termination separately
- Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of `X`

```
WHILE X>1 DO
  IF ODD(X) THEN X := (3×X)+1 ELSE X := X DIV 2
```

  - `X DIV 2` evaluates to the result of rounding down `X/2` to a whole number
  - the Collatz conjecture is that this terminates with `X=1`
- Microsoft's TERMINATOR tool proves systems code terminates

Auxiliary Variables

- $\{\mathtt{X=x} \land \mathtt{Y=y}\}$ `R:=X; X:=Y; Y:=R` $\{\mathtt{X=y} \land \mathtt{Y=x}\}$
  - this says that *if* the execution of

    `R:=X; X:=Y; Y:=R`

    terminates (which it does)
  - *then* the values of `X` and `Y` are exchanged
- The variables `x` and `y`, which don't occur in the command and are used to name the initial values of program variables `X` and `Y`
- They are called *auxiliary* variables or *ghost* variables
- Informal convention:
  - program variable are upper case
  - auxiliary variable are lower case

More simple examples

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

A More Complicated Example

- $$\begin{array}{l} \{\mathtt{T}\} \\ \quad \left.\begin{array}{l} \mathtt{R:=X;} \\ \mathtt{Q:=0;} \\ \mathtt{WHILE\ Y\leq R\ DO} \\ \quad \mathtt{(R:=R-Y;\ Q:=Q+1)} \end{array}\right\} \mathtt{C} \\ \{\mathtt{R} < \mathtt{Y}\ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\} \end{array}$$
- This is $\{\mathtt{T}\}\ \mathtt{C}\ \{\mathtt{R} < \mathtt{Y}\ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$
  - where `C` is the command indicated by the braces above
  - the specification is true if whenever the execution of `C` halts, then `Q` is quotient and `R` is the remainder resulting from dividing `Y` into `X`
  - it is true (even if `X` is initially negative!)
  - in this example `Q` is a program variable
  - don't confuse `Q` with the metavariable Q used in previous examples to range over postconditions (Sorry: my bad notation!)

Some Easy Exercises

- When is $[\mathtt{T}]\ C\ [\mathtt{T}]$ true?
- Write a partial correctness specification which is true if and only if the command C has the effect of multiplying the values of `X` and `Y` and storing the result in `X`
- Write a specification which is true if the execution of C always halts when execution is started in a state satisfying P

Specification can be Tricky

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

Specification can be Tricky (ii)

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

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

- The correct formalisation of what was intended is probably

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

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

Review of Predicate Calculus

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

Terms (Expressions)

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

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

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

Atomic Statements

- Examples of atomic statements are

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

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

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

- ODD and PRIME are examples of predicates
- $=$ and $\geq$ are examples of *infixed* predicates
- X, 1, 3, X+1, $\mathtt{(X+1)}^2$, $\mathtt{x}^2$ are terms in above atomic statements

Compound statements

- Compound statements are built up from atomic statements using:

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

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

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

More on Implication

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

Precedence

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

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

Universal quantification

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

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

- The statement

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

  abbreviates

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

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

Existential quantification

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

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

- The statement

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

  abbreviates

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

Summary

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

Floyd-Hoare Logic

✓

- To construct formal proofs of partial correctness specifications, *axioms* and *rules of inference are needed*
- This is what Floyd-Hoare logic provides
  - the formulation of the deductive system is due to Hoare
  - some of the underlying ideas originated with Floyd
- A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an *axiom* of the logic or follows from earlier lines by a *rule of inference* of the logic
  - proofs can also be trees, if you prefer
- A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion

Notation for Axioms and Rules

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

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

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

An example rule

The sequencing rule

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

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

Reminder of our little programming language

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

Expressions

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

Boolean expressions

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

Commands

C ::= V `:=` E	Assignments
| C_1 ; C_2	Sequences
| `IF` B `THEN` C_1 `ELSE` C_2	Conditionals
| `WHILE` B `DO` C	WHILE-commands

Judgements

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

Syntactic Conventions

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

Substitution Notation

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

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

which is analogous to the cancellation property of fractions

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

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

The Assignment Axiom (Hoare)

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

> **The Assignment Axiom**
>
> $$\vdash \{Q[E/V]\}\ V{:=}E\ \{Q\}$$
>
> Where V is any variable, E is any expression, Q is any statement.

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

The Backwards Fallacy

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

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

  - where $P[V/E]$ denotes the result of substituting V for E in P
  - this has the false consequence $\vdash \{\mathtt{X=0}\}\ \mathtt{X:=1}\ \{\mathtt{X=0}\}$
    (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 $\vdash \{\mathtt{X=0}\}\ \mathtt{X:=1}\ \{\mathtt{1=0}\}$
    (which follows by taking P to be `X=0`, V to be `X` and E to be `1`)

A Forwards Assignment Axiom (Floyd)

- This is the original semantics of assignment due to Floyd

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

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

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

- Simplifying the postcondition

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

- Forwards Axiom equivalent to standard one but harder to use

Precondition Strengthening

- Recall that

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

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

- Using this notation, the rule of precondition strengthening is

Precondition strengthening

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

- Note the two hypotheses are different kinds of judgements

Example

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

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

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

Postcondition weakening

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

Postcondition weakening

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

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**

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

  - **since `(Y=0)[E/X] = (Y=0)` (because `X` does not occur in `(Y=0)`)**
  - **this is clearly false; after the assignment `Y` will have the value `1`**

An Example Formal Proof

- **Here is a little formal proof**

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

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

The sequencing rule

- **Syntax:** $C_1;\ \cdots\ ;C_n$

- **Semantics: the commands $C_1, \cdots, C_n$ are executed in that order**

- **Example:** `R:=X; X:=Y; Y:=R`
  - **the values of `X` and `Y` are swapped using `R` as a temporary variable**
  - **note** ***side effect*****: value of `R` changed to the old value of `X`**

The sequencing rule

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

Example Proof

Example: By the assignment axiom:

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

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

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

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

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

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

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

Conditionals

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

The Conditional Rule

The conditional rule

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

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

  $\vdash$ (X≥Y ⇒ X = max(X,Y)) ∧ (¬(X≥Y) ⇒ Y = max(X,Y))

  it follows that

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

  and

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

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

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

WHILE-commands

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

Invariants

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

The `WHILE`-Rule

The `WHILE`-rule

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

- **It is easy to show**

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

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

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


Example

- **From the previous slide**

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

- **It is easy to deduce that**

```
⊢ {T} R:=X; Q:=0 {X=R+(Y×Q)}
```

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

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


Summary

- **We have given:**
  - a notation for specifying what a program does
  - a way of proving that it meets its specification
- **Now we look at ways of finding proofs and organising them:**
  - finding invariants
  - derived rules
  - backwards proofs
  - annotating programs prior to proof
- **Then we see how to automate program verification**
  - the automation mechanises some of these ideas

How does one find an invariant?

The `WHILE`-rule

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

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

Example

- Consider a factorial program

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

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

Related example

```
{X=0 ∧ 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 an Y increase: X by 1 and Y by X
- An invariant is Y = X!
- At end need Y = N!, but WHILE-rule only gives ¬(X<N)
- Ah Ha! Invariant needed: $\boxed{Y = X! \land X \leq N}$
- At end $X \leq N \land \neg(X<N) \Rightarrow X=N$
- Often need to strenthen invariants to get them to work
  - typical to add stuff to 'carry along' like X≤N

Conjunction and Disjunction

Specification conjunction

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

Specification disjunction

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

- These rules are useful for splitting a proof into independent bits
  - they enable $\vdash$ {P} C {$Q_1 \land Q_2$} to be proved by proving separately that both $\vdash$ {P} C {Q_1} and also that $\vdash$ {P} C {Q_2}
- Any proof with these rules could be done without using them
  - i.e. they are theoretically redundant (proof omitted)
  - however, useful in practice

Combining Multiple Steps

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

Derived rules for finding proofs

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

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

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

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

Derived Rules

- We will establish derived rules for all commands

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

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

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

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

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

The Derived Assignment Rule

- An example proof

1. $\vdash\ \{\texttt{R=X} \land \texttt{0=0}\}\ \texttt{Q:=0}\ \{\texttt{R=X} \land \texttt{Q=0}\}$ By the assignment axiom.
2. $\vdash\ \texttt{R=X}\ \Rightarrow\ \texttt{R=X} \land \texttt{0=0}$ By pure logic.
3. $\vdash\ \{\texttt{R=X}\}\ \texttt{Q:=0}\ \{\texttt{R=X} \land \texttt{Q=0}\}$ By precondition strengthening.

- Can generalise this proof to a proof schema:

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

- This proof schema justifies:

Derived Assignment Rule

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

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

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

Derived Sequenced Assignment Rule

- The following rule will be useful later

Derived Sequenced Assignment Rule

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

- Intuitively work backwards:
  - push Q 'through' $V\texttt{:=}E$, changing it to $Q[E/V]$
- Example: By the assignment axiom:

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

- Hence by the sequenced assignment rule

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

Backward Hoare & forward Floyd assignment axioms

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

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

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

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

The Derived While Rule

Derived While Rule

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

- **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×Q)`

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

$\vdash$ `X=R+(Y×Q)∧¬(Y≤R)` $\Rightarrow$ `X=R+(Y×Q)∧¬(Y≤R)`

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

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


The Derived Sequencing Rule

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

The Derived Sequencing Rule

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

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

Example

- **By the assignment axiom**

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

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

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

Forwards and backwards proof

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

Example

- Suppose one wants to show

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

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

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

- Similarly this subgoal can be reduced to

```
{X=x ∧ Y=y} R:=X {R=x ∧ Y=y}
```

- This clearly follows from the assignment axiom

Backwards versus Forwards Proof

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

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

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

Example Backwards Proof

- To prove

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

- By the sequencing rule, it is sufficient to prove

(i) `⊢ {T} R:=X; Q:=0 {R=X ∧ Q=0}`

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

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

Example Continued (1)

- **From previous slide:**

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

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

  (iii) $\vdash$ `{T} R:=X {R=X ∧ 0=0}`

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

  $\vdash$ `T ⇒ X=X ∧ 0=0`

- **This is true by pure logic**

Example continued (2)

- **From an earlier slide:**

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

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

  (iv) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

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

  and

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

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

Example Continued (3)

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

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

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

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

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

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

- **This is true by arithmetic**

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

Annotations

- **The sequencing rule introduces a new statement** R

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

- **To apply this backwards, one needs to find a suitable statement** R

- **If** C_2 **is** V`:=`E **then sequenced assignment gives** Q`[`E/V`]` **for** R

- **If** C_2 **isn't an assignment then need some other way to choose** R

- **Similarly, to use the derived While rule, must invent an invariant**

Annotate First

- It is helpful to think up these statements before you start the proof and then annotate the program with them
  - the information is then available when you need it in the proof
  - this can help avoid you being bogged down in details
  - the annotation should be true whenever control reaches that point
- Example, the following program could be annotated at the points P_1 and P_2 indicated by the arrows

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


Summary

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

NEW TOPIC: Mechanizing Program Verification

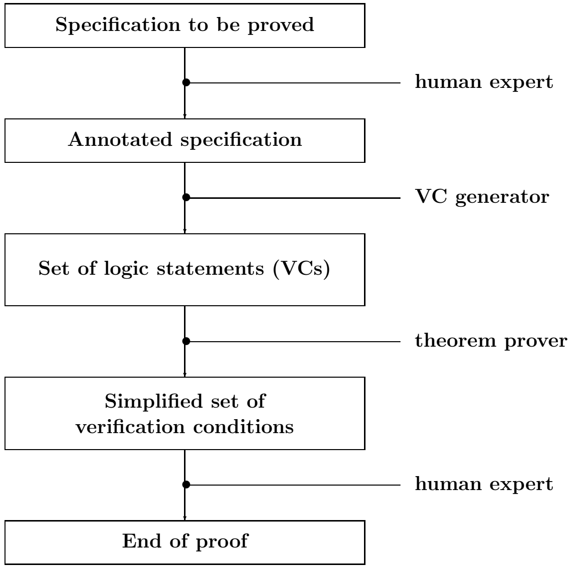
- The architecture of a simple program verifier will be described
- Justified with respect to the rules of Floyd-Hoare logic
- It is clear that
  - proofs are long and boring, even if the program being verified is quite simple
  - lots of fiddly little details to get right, many of which are trivial,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

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

Mechanization

- **Goal:** automate the routine bits of proofs in Floyd-Hoare logic
- Unfortunately, logicians have shown that it is impossible in principle to design a *decision procedure* to decide automatically the truth or falsehood of an arbitrary mathematical statement
- This does not mean that one cannot have procedures that will prove many *useful* theorems
  - the non-existence of a general decision procedure merely shows that one cannot hope to prove *everything* automatically
  - in practice, it is quite possible to build a system that will mechanize the boring and routine aspects of verification
- The standard approach to this will be described in the course
  - ideas very old (JC King's 1969 CMU PhD, Stanford verifier in 1970s)
  - used by program verifiers (e.g. Gypsy and SPARK verifier)
  - provides a verification front end to different provers (see *Why* system)

Architecture of a Verifier



Commentary

- Input: a Hoare triple annotated with mathematical statements
  - these annotations describe relationships between variables
- The system generates a set of purely mathematical statements called *verification conditions* (or VCs)
- If the verification conditions are provable, then the original specification can be deduced from the axioms and rules of Hoare logic
- The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically
  - if it fails, advice is sought from the user

Verification conditions

- The three steps in proving $\{P\}C\{Q\}$ with a verifier
- $\boxed{1}$ 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
- $\boxed{2}$ A set of logic statements called *verification conditions* (VCs) is then generated from the annotated specification
  - this is purely mechanical and easily done by a program
- $\boxed{3}$ The verification conditions are proved
  - needs automated theorem proving (i.e. more artificial intelligence)
- To improve automated verification one can try to
  - reduce the number and complexity of the annotations required
  - increase the power of the theorem prover
  - still a research area

Validity of Verification Conditions

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

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

Example

- Step $\boxed{1}$ 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×Q` holds, even though the values of `R` and `Q` vary
  - P_2 is an *invariant* of the `WHILE`-command

Generating and Proving Verification Conditions

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

  (i) `T ⇒ (X=X ∧ 0=0)`
  (ii) `(R=X ∧ Q=0) ⇒ (X = R+(Y×Q))`
  (iii) `(X = R+(Y×Q)) ∧ Y≤R) ⇒ (X = (R-Y)+(Y×(Q+1)))`
  (iv) `(X = R+(Y×Q)) ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)`

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

Annotation of Commands

- An annotated command is a command with statements (*assertions*) embedded within it
- A command is *properly annotated* if statements have been inserted at the following places

  (i) before C_2 in C_1;C_2 if C_2 is *not* an assignment command
  (ii) after the word `DO` in `WHILE` commands

- The inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs
- Can reduce number of annotations using weakest preconditions (see later)

Annotation of Specifications

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

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

- **Suitable statements would be**

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

Verification Condition Generation

- **The VCs generated from an annotated specification $\{P\}C\{Q\}$ are obtained by considering the various possibilities for C**
- **We will describe it command by command using rules of the form:**
- **The VCs for $C(C_1, C_2)$ are**
  - vc_1, ... , vc_n
  - together with the VCs for $\mathtt{C}_1$ and those for $\mathtt{C}_2$
- **Each VC rule corresponds to either a primitive or derived rule**

A VC Generation Program

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

  VC $(C(C_1, C_2)) = [vc_1, \ldots, vc_n]$ @ (**VC** C_1) @ (**VC** C_2)

- **The rules are chosen so that only one VC rule applies in each case**
  - applying them is then purely mechanical
  - the choice is based on the syntax
  - only one rule applies in each case so VC generation is deterministic

Justification of VCs

- **This process will be justified by showing that $\vdash \{P\}C\{Q\}$ if all the verification conditions can be proved**
- **We will prove that for any `C`**
  - assuming the VCs of $\{P\}C\{Q\}$ are provable
  - then $\vdash \{P\}C\{Q\}$ is a theorem of the logic

Justification of Verification Conditions

- The argument that the verification conditions are sufficient will be by *induction* on the structure of C
- Such inductive arguments have two parts
  - show the result holds for atomic commands, 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

97

VC for Assignments

Assignment commands

The single verification condition generated by

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

is

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

- Example: The verification condition for

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

  is

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

  (which is clearly true)

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

98

Justification of Assignment VC

- We must show that if the VCs of $\{P\}\ V := E\ \{Q\}$ are provable then $\vdash\ \{P\}\ V := E\ \{Q\}$
- Proof:
  - Assume $\vdash\ P \Rightarrow Q\texttt{[}E/V\texttt{]}$ as it is the VC
  - From derived assignment rule it follows that $\vdash\ \{P\}\ V := E\ \{Q\}$

99

VCs for Conditionals

Conditionals

The verification conditions generated from

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

are

(i) the verification conditions generated by

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

(ii) the verifications generated by

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

- Example: The verification conditions for

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

  are

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

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

100

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

VCs for Sequences

Sequences

1. The verification conditions generated by

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

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

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

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

2. The verification conditions generated by

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

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

Example

- The verification conditions generated from

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

- Are those generated by

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

- This simplifies to

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

- The verification conditions generated by this are those generated by

  `{X=x ∧ Y=y} R:=X {(X=y ∧ R=x)[Y/X]}`

- Which simplifies to

  `{X=x ∧ Y=y} R:=X {Y=y ∧ R=x}`

Example Continued

- The only verification condition generated by

  `{X=x ∧ Y=y} R:=X {Y=y ∧ R=x}`

  is

  `X=x ∧ Y=y ⇒ (Y=y ∧ R=x)[X/R]`

- Which simplifies to

  `X=x ∧ Y=y ⇒ Y=y ∧ X=x`

- Thus the single verification condition from

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

  is

  `X=x ∧ Y=y ⇒ Y=y ∧ X=x`

Justification of VCs for Sequences (1)

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

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

  are provable

- Then the verification conditions for

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

  and

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

  must both be provable

- Hence by induction

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

- Hence by the sequencing rule

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

Justification of VCs for Sequences (2)

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

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

  are provable, then the verification conditions for

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

  are also provable

- Hence by induction

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

- Hence by the derived sequenced assignment rule

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

VCs for `WHILE`-Commands

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

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

- The annotation R is called an invariant

`WHILE`-commands

The verification conditions generated from

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

are

(i) $P \Rightarrow R$

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

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

Example

- The verification conditions for

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

are:

(i) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

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

together with the verification condition for

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

which consists of the single condition

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

Example Summarised

- By previous transparency

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

if

`⊢ R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

and

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

and

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

Justification of `WHILE` VCs

- If the verification conditions for

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

  are provable, then

  $\vdash\ P \Rightarrow R$

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

  and the verification conditions for

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

  are provable

- By induction

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

- Hence by the derived `WHILE`-rule

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

Summary

- Have outlined the design of an automated program verifier
- Annotated specifications compiled to mathematical statements
  - if the statements (VCs) can be proved, the program is verified
- Human help is required to give the annotations and prove the VCs
- The algorithm was justified by an inductive proof
  - it appeals to the derived rules
- All the techniques introduced earlier are used
  - backwards proof
  - derived rules
  - annotation

Dijkstra's weakest preconditions

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

Rules for weakest preconditions

- Relation with Hoare specifications:

$$\begin{array}{lcl} \{P\}\ C\ \{Q\} & \Leftrightarrow & P \Rightarrow \texttt{wlp}(C,Q) \\ [P]\ C\ [Q] & \Leftrightarrow & P \Rightarrow \texttt{wp}(C,Q) \end{array}$$

- Dijkstra gives rules for computing weakest preconditions:

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

  for deterministic loop-free code the same equations hold for `wlp`
- Rule for `WHILE`-commands doesn't give a first order result
- Weakest preconditions closely related to verification conditions
- VCs for $\{P\}\ C\ \{Q\}$ are related to $P \Rightarrow \texttt{wlp}(C,Q)$
  - VCs use annotations to ensure first order formulas can be generated

Sequencing example

- Swapping variables:

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

- So since $\{P\}\ C\ \{Q\} \ \Leftrightarrow \ P \Rightarrow \texttt{wlp}(C,Q)$
  to prove
  $\{\texttt{X} = x \wedge \texttt{Y} = y\}\ \texttt{R:=X; X:=Y; Y:=R}\ \{\texttt{Y} = x \wedge \texttt{X} = y\}$
  just need to prove:
  $(\texttt{X} = x \wedge \texttt{Y} = y) \Rightarrow (\texttt{X} = x \wedge \texttt{Y} = y)$
  which is clearly true (instance of $S \Rightarrow S$)

Conditional example

- Compute `wlp` of the maximum program:

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

- So to prove
  $\{\texttt{X} = x \wedge \texttt{Y} = y\}\ \texttt{IF X<Y THEN MAX:=X ELSE MAX:=Y}\ \{\texttt{MAX} = max(x,y)\}$
  just prove:
  $(\texttt{X} = x \wedge \texttt{Y} = y) \Rightarrow (\texttt{X<Y} \Rightarrow \texttt{Y} = max(x,y)) \wedge (\neg(\texttt{X<Y}) \Rightarrow \texttt{X} = max(x,y))$
  which follows from the defining property of max
  $\vdash \ \forall x\ y.\ (x \geq y \Rightarrow x = max(x,y)) \wedge (\neg(x \geq y) \Rightarrow y = max(x,y))$

Using `wlp` to improve verification condition method ✓

- If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $P \Rightarrow \texttt{wlp}(C, Q)$
  - no annotations needed in sequences!
- Cannot in general compute a finite formula for $\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)$
- The following holds

  $\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)\ =\ \mathit{if}\ S\ \mathit{then}\ \texttt{wlp}(C,\ \texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q))\ \mathit{else}\ Q$
- Above doesn't define $\texttt{wlp}(C, Q)$ as a finite statement
- Could use a hybrid VC and `wlp` method

`wlp`-based verification condition method ✗

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

Definition of `awp` ✗

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

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

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

Definition of `wvc` ✗

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

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

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

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

Strongest postconditions

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

Strongest postconditions for loop-free code

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

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

Sequencing example

- $\texttt{sp}(\texttt{R:=X;\ X:=Y;\ Y:=R},\ \texttt{X} = x \wedge \texttt{Y} = y)$
$$
\begin{aligned}
&= \texttt{sp}(\texttt{Y:=R},\ \texttt{sp}(\texttt{X:=Y},\ \texttt{sp}(\texttt{R:=X},\ \texttt{X} = x \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ \texttt{sp}(\texttt{X:=Y},\ (\exists v.\ \texttt{R} = \texttt{X}[v/\texttt{R}] \wedge (\texttt{X} = x \wedge \texttt{Y} = y)[v/\texttt{R}])))\\
&= \texttt{sp}(\texttt{Y:=R},\ \texttt{sp}(\texttt{X:=Y},\ (\exists v.\ \texttt{R} = \texttt{X} \wedge (\texttt{X} = x \wedge \texttt{Y} = y))))\\
&= \texttt{sp}(\texttt{Y:=R},\ \texttt{sp}(\texttt{X:=Y},\ (\texttt{R} = \texttt{X} \wedge \texttt{X} = x \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\exists v.\ \texttt{X} = \texttt{Y}[v/\texttt{X}] \wedge (\texttt{R} = \texttt{X} \wedge \texttt{X} = x \wedge \texttt{Y} = y)[v/\texttt{X}]))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\exists v.\ \texttt{X} = \texttt{Y} \wedge (\texttt{R} = v \wedge v = x \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\exists v.\ \texttt{X} = \texttt{Y} \wedge (\texttt{R} = x \wedge v = x \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\texttt{X} = \texttt{Y} \wedge (\texttt{R} = x \wedge (\exists v.\ v = x) \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\texttt{X} = \texttt{Y} \wedge (\texttt{R} = x \wedge \texttt{T} \wedge \texttt{Y} = y)))\\
&= \texttt{sp}(\texttt{Y:=R},\ (\texttt{X} = \texttt{Y} \wedge \texttt{R} = x \wedge \texttt{Y} = y))\\
&= \exists v.\ \texttt{Y} = \texttt{R}[v/\texttt{Y}] \wedge (\texttt{X} = \texttt{Y} \wedge \texttt{R} = x \wedge \texttt{Y} = y)[v/\texttt{Y}]\\
&= \exists v.\ \texttt{Y} = \texttt{R} \wedge (\texttt{X} = v \wedge \texttt{R} = x \wedge v = y)\\
&= \exists v.\ \texttt{Y} = \texttt{R} \wedge (\texttt{X} = y \wedge \texttt{R} = x \wedge v = y)\\
&= \texttt{Y} = \texttt{R} \wedge (\texttt{X} = y \wedge \texttt{R} = x \wedge (\exists v.\ v = y))\\
&= \texttt{Y} = \texttt{R} \wedge (\texttt{X} = y \wedge \texttt{R} = x \wedge \texttt{T})\\
&= \texttt{Y} = \texttt{R} \wedge \texttt{X} = y \wedge \texttt{R} = x\\
&= \texttt{Y} = x \wedge \texttt{X} = y \wedge \texttt{R} = x
\end{aligned}
$$
- **So to prove** $\{\texttt{X} = x \wedge \texttt{Y} = y\}\ \texttt{R:=X;\ X:=Y;\ Y:=R}\ \{\texttt{Y} = x \wedge \texttt{X} = y\}$
  just prove: $(\texttt{Y} = x \wedge \texttt{X} = y \wedge \texttt{R} = x) \Rightarrow \texttt{Y} = x \wedge \texttt{X} = y$

Conditional example

- Compute sp of the maximum program:

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

$$
\begin{aligned}
&= \texttt{sp(MAX:=Y}, ((\texttt{X} = x \wedge \texttt{Y} = y) \wedge \texttt{X} < \texttt{Y})) \\
&\quad \vee \\
&\quad \texttt{sp(MAX:=X}, ((\texttt{X} = x \wedge \texttt{Y} = y) \wedge \neg(\texttt{X} < \texttt{Y}))) \\
&= \exists v.\ \texttt{MAX} = \texttt{Y[}v\texttt{/MAX]} \wedge ((\texttt{X} = x \wedge \texttt{Y} = y) \wedge \texttt{X} < \texttt{Y})\texttt{[}v\texttt{/MAX]} \\
&\quad \vee \\
&\quad \exists v.\ \texttt{MAX} = \texttt{X[}v\texttt{/MAX]} \wedge ((\texttt{X} = x \wedge \texttt{Y} = y) \wedge \neg(\texttt{X} < \texttt{Y}))\texttt{[}v\texttt{/MAX]} \\
&= \exists v.\ \texttt{MAX} = \texttt{Y} \wedge ((\texttt{X} = x \wedge \texttt{Y} = y) \wedge \texttt{X} < \texttt{Y}) \\
&\quad \vee \\
&\quad \exists v.\ \texttt{MAX} = \texttt{X} \wedge \texttt{X} = x \wedge \texttt{Y} = y \wedge \neg(\texttt{X} < \texttt{Y})) \\
&= (\texttt{MAX} = \texttt{Y} \wedge \texttt{X} = x \wedge \texttt{Y} = y \wedge \texttt{X} < \texttt{Y}) \vee (\texttt{MAX} = \texttt{X} \wedge \texttt{X} = x \wedge \texttt{Y} = y \wedge \neg(\texttt{X} < \texttt{Y}) \\
&= (\texttt{MAX} = y \wedge \texttt{X} = x \wedge \texttt{Y} = y \wedge x < y) \vee (\texttt{MAX} = x \wedge \texttt{X} = x \wedge \texttt{Y} = y \wedge \neg(x < y) \\
&= \mathit{if}\ x < y\ \mathit{then}\ (\texttt{MAX} = y \wedge \texttt{X} = x \wedge \texttt{Y} = y)\ \mathit{else}\ (\texttt{MAX} = x \wedge \texttt{X} = x \wedge \texttt{Y} = y) \\
&= \texttt{MAX} = (\mathit{if}\ x < y\ \mathit{then}\ y\ \mathit{else}\ x) \wedge \texttt{X} = x \wedge \texttt{Y} = y \\
&= \texttt{MAX} = \mathit{max}(x, y) \wedge \texttt{X} = x \wedge \texttt{Y} = y
\end{aligned}
$$

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

Exercises

- Compute

$$
\begin{aligned}
&\texttt{sp}(\texttt{R} := 0; \\
&\quad \texttt{K} := 0; \\
&\quad \texttt{IF I} < \texttt{J THEN K} := \texttt{K} + 1 \texttt{ ELSE K} := \texttt{K}; \\
&\quad \texttt{IF K} = 1 \wedge \neg(\texttt{I} = \texttt{J}) \texttt{ THEN R} := \texttt{J} - \texttt{I} \texttt{ ELSE R} := \texttt{I} - \texttt{J}, \\
&\quad (\texttt{I} = i \wedge \texttt{J} = j \wedge j \leq i))
\end{aligned}
$$

- Hence show

$$
\begin{aligned}
&\{(\texttt{I} = i \wedge \texttt{J} = j \wedge j \leq i\} \\
&\quad \texttt{R} := 0; \\
&\quad \texttt{K} := 0; \\
&\quad \texttt{IF I} < \texttt{J THEN K} := \texttt{K} + 1 \texttt{ ELSE K} := \texttt{K}; \\
&\quad \texttt{IF K} = 1 \wedge \neg(\texttt{I} = \texttt{J}) \texttt{ THEN R} := \texttt{J} - \texttt{I} \texttt{ ELSE R} := \texttt{I} - \texttt{J}) \\
&\{R = i - j\}
\end{aligned}
$$

- Do same example use wlp

Using sp to generate verification conditions

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

$\texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P) = \texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ \texttt{sp}(C,\ (P \wedge S)))\ \vee\ (P \wedge \neg S)$

- Above doesn't define $\texttt{sp}(C, P)$ to be a finite statement
- As with wlp, can use a hybrid VC and sp method

sp-based verification conditions

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

Definition of asp

- **Define** asp **recursively by:**

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

Definition of svc

- **Define** svc **recursively by:**

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

- **Theorem:** $\bigwedge\texttt{svc}(P, C) \Rightarrow \{P\}\ C\ \{\texttt{asp}(P, C)\}$
- **Proof by induction on** C **(exercise)**

Summary

- **Annotate then generate VCs is the classical method**
  - **practical tools: Gypsy (1970s), SPARK (current)**
  - **weakest preconditions are alternative explanation of VCs**
  - wlp **needs fewer annotations than VC method described earlier**
  - wlp **also used for refinement**
- **VCs and** wlp **go backwards,** sp **goes forward**
  - sp **provides verification method based on symbolic simulation**
  - **widely used for loop-free code**
  - **current research potential for forwards full proof of correctness**
  - **probably need mixture of forwards and backwards methods (Hoare's view)**

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

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute
  - check states reached satisfy decidable properties
- Full proof of correctness
  - add invariants to loops
  - generate verification conditions
  - prove verification conditions with a theorem prover
- Research goal: unifying framework for a spectrum of methods

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} `WHILE T DO X := X` {T $\wedge$ ¬T} (2 and the `WHILE`-rule)

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

$\vdash$ [T] `WHILE T DO X := X` [T $\wedge$ ¬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:

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

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

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

- The assignment `X :=` $1/0$ halts in a state in which `X` $= 1/0$ holds

- This assumes that $1/0$ denotes some value that `X` can have

Two Possibilities

- There are two possibilities

  (i) $1/0$ denotes some number;

  (ii) $1/0$ denotes some kind of 'error value'.

- It seems at first sight that adopting (ii) is the most natural choice
  - this makes it tricky to see what arithmetical laws should hold
  - is $(1/0) \times 0$ equal to 0 or to some 'error value'?
  - if the latter, then it is no longer the case that $\forall n.\ n \times 0 = 0$ is valid

- It is possible to make everything work with undefined and/or error values, but the resultant theory is a bit messy

Example

- We assume that arithmetic expressions *always* denote numbers

- In some cases exactly what the number is will be not fully specified
  - for example, we will assume that m/n denotes a number for any m and n
  - only assume: $\neg(n = 0) \ \Rightarrow \ (m/n) \times n = m$
  - it is not possible to deduce anything about $m/0$ from this
  - in particular it is not possible to deduce that $(m/0) \times 0 = 0$
  - but $(m/0) \times 0 = 0$ does follow from $\forall n.\ n \times 0 = 0$

- People still argue about this – e.g. advocate "three-valued" logics

WHILE-rule for Total Correctness (i)

- WHILE-commands are the only commands in our little language that can cause non-termination
  - they are thus the only kind of command with a non-trivial termination rule
- The idea behind the WHILE-rule for total correctness is
  - to prove WHILE S DO C terminates
  - show that some non-negative quantity decreases on each iteration of C
  - this decreasing quantity is called a variant

WHILE-Rule for Total Correctness (ii)

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

WHILE-rule for total correctness

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

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

Example

- We show

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

- Take

$$\begin{array}{lcl} P & = & \texttt{Y} > \texttt{0} \\ S & = & \texttt{Y} \leq \texttt{R} \\ E & = & \texttt{R} \\ C & = & \texttt{(R:=R-Y; Q:=Q+1)} \end{array}$$

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

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

Example Continued (1)

- From previous slide:

$$\begin{array}{lcl} P & = & \texttt{Y} > \texttt{0} \\ S & = & \texttt{Y} \leq \texttt{R} \\ E & = & \texttt{R} \\ C & = & \texttt{(R:=R-Y; Q:=Q+1)} \end{array}$$

- We want to show

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

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

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

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

Example Continued (2)

- **From previous slide:**

$$
\begin{aligned}
P &= \mathtt{Y > 0}\\
S &= \mathtt{Y} \leq \mathtt{R}\\
E &= \mathtt{R}\\
C &= \mathtt{R{:=}R{-}Y;\ Q{:=}Q{+}1)}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathtt{Y} > 0 \ \wedge\ \mathtt{Y} \leq \mathtt{R} \ \wedge\ \mathtt{R} = \mathtt{n} \quad \Rightarrow\\
&\quad (\mathtt{Y} > 0 \ \wedge\ \mathtt{R} < \mathtt{n})[\mathtt{Q}{+}1/\mathtt{Q}][\mathtt{R}{-}\mathtt{Y}/\mathtt{R}]
\end{aligned}
$$

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

which follows from the laws of arithmetic

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

Termination Specifications

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

Total correctness = Termination + Partial correctness

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

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

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

Derived Rules

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

The Derived While Rule

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

Derived `WHILE`-rule for total correctness

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

VCs for Termination

- Verification conditions are easily extended to total correctness
- To generate total correctness verification conditions for `WHILE`-commands, it is necessary to add a variant as an annotation in addition to an invariant
- Variant added directly after the invariant, in square brackets
- No other extra annotations are needed for total correctness
- VCs generation algorithm same as for partial correctness

`WHILE` Annotation

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

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

where R is the invariant and E the variant

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

`WHILE` VCs

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

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

`WHILE`-commands

The verification conditions generated from

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

are

(i) $P \Rightarrow R$

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

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

(iv) the verification conditions generated by

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

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

Example

- The verification conditions for

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

are:

(i) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

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

(iii) `X = R+Y×Q ∧ Y≤R ⇒ R≥0`

together with the verification condition for

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

Example Continued

- The single verification condition for

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

  is

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

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

Summary

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

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

Terminology

- **First-order logic, as described in logic books, has** *terms* **and** *formulae*
- **For consistency with earlier stuff we use** *expressions* **and** *statements*
- **Will define sets** *Exp* **of expressions and** *Sta* **of statements**
- **Sets** *Exp* **and** *Sta* **depend on a language** $\mathcal{L}$ **(see next slide)**
  - will write $Exp_{\mathcal{L}}$ and $Sta_{\mathcal{L}}$ to make this clear
  - if language is clear from context may omit language subscript
- **Assume an infinite set** *Var* **of variables**
  - doesn't depend on a language

First-order languages

- A first-order language $\mathcal{L}$ contains
  - zero or more predicate symbols, p_1, p_2, ... each with an arity ≥ 0
  - zero or more function symbols, f_1, f_2, ... each with an arity ≥ 0
  - $\mathcal{L} = (\{p_1, p_2, \ldots\}, \{f_1, f_2, \ldots\})$
- $Exp_{\mathcal{L}}$ is the smallest set such that:
  - $Var \subseteq Exp_{\mathcal{L}}$
  - f a function symbols of $\mathcal{L}$ of arity 0, then $f \in Exp\mathcal{L}$
  - f a function symbols of $\mathcal{L}$ of arity $n > 0$ and $E_i \in Exp_{\mathcal{L}}$, then $f(E_1, \ldots, E_n) \in Exp_{\mathcal{L}}$
- $Sta_{\mathcal{L}}$ is the smallest set such that:
  - p a predicate symbols of $\mathcal{L}$ of arity 0, then $p \in Sta\mathcal{L}$
  - p a predicate symbols of $\mathcal{L}$ of arity $n > 0$ and $E_i \in Exp_{\mathcal{L}}$, then $p(E_1, \ldots, E_n) \in Sta_{\mathcal{L}}$
  - S, S_1, S_2 in $Sta_{\mathcal{L}}$, then $\neg S$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$ are in $Sta_{\mathcal{L}}$
  - $v \in Var$ and S in $Sta_{\mathcal{L}}$, then $\forall v.\ S$ and $\exists v.\ S$ are in $Sta_{\mathcal{L}}$

Semantics: interpretations

- An interpretation $\mathcal{I}$ of language $\mathcal{L}$ provides:
  - *domain* D of values, also called a *universe*
  - meaning $I[p]$ for predicate symbols p and $I[f]$ for function symbols f
- Sets, functions and relations
  - $Bool = \{true, false\}$
  - if $n > 0$, then $A^n = \{(a_1, \ldots, a_n) \mid a_i \in A\}$
  - $A \to B = \{u \mid u : A \to B\}$ (alternative notation: B^A)
- If $\mathcal{I} = (D, I)$ then:
  - if p is a predicate symbol of arity 0, then $I[p] \in Bool$
  - if p is a predicate symbol of arity $n > 0$, then $I[p] \in D^n \to Bool$
  - if f is a function symbol of arity 0, then $I[f] \in D$
  - if f is a function symbol of arity $n > 0$, then $I[f] \in D^n \to D$

Semantics: valuations

- Interpretation provide meaning for predicate and function symbols
- A valuation s for $\mathcal{I} = (D, I)$ determines the values of variables in D
  - $s \in Var \to D$
- Often 'V' not 's' used for valuations – reasons for using 's' here are:
  - valuations are *states* in the semantics of Hoare triples
  - avoid confusion with earlier use of 'V' to range over variables
- Define $s[a/x]$ to be identical to s except that x is mapped to a:

$$(s[a/x])(y) \;=\; \textit{\textbf{if}}\ y = x\ \textit{\textbf{then}}\ a\ \textit{\textbf{else}}\ s(y)$$

- Also use $[\cdots/\cdots]$ notation for syntactic substitution
  - e.g. in assignment axiom $\{Q[E/V]\}V:=E\{Q\}$
  - will relate syntactic and semantic uses of $[\cdots/\cdots]$ soon

Semantics: terms and formulae

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

Satisfiability, validity and completeness

- Assume a language $\mathcal{L}$
- S is *satisfiable* iff for some interpretation of $\mathcal{L}$ and s: $\texttt{Ssem}\ S\ s = \mathit{true}$
- S is *valid* iff for all interpretations of $\mathcal{L}$ and all s: $\texttt{Ssem}\ S\ s = \mathit{true}$
- 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 $\texttt{Ssem}\ S\ s_1 = \texttt{Ssem}\ S\ s_2$ for all s_1, s_2
- A *theory* is a set of sentences
- $\Gamma \vdash S$ means S can be deduced from Γ using first-order logic
- Γ 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 Γ 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

- $\mathcal{L}_{\mathrm{PA}}$ is the language of Peano Arithmetic
- $\mathcal{I}_{\mathrm{PA}}$ is the *standard interpretation* of arithmetic
- $\models_{\mathcal{I}_{\mathrm{PA}}} S$ means S is true in $\mathcal{I}_{\mathrm{PA}}$
- PA is the first-order theory of Peano Arithmetic
- There exists a sentence G of $\mathcal{L}_{\mathrm{PA}}$ and neither PA $\vdash G$ nor PA $\vdash \neg G$
  - Gödel's first incompleteness theorem (1930)
  - as G is a sentence either $\models_{\mathcal{I}_{\mathrm{PA}}} G$ or $\models_{\mathcal{I}_{\mathrm{PA}}} \neg G$
  - so there are sentences true in $\mathcal{I}_{\mathrm{PA}}$ that can't be proved from PA
- $\models_{\mathcal{I}_{\mathrm{PA}}} S$ does not imply PA $\vdash S$
  - if it did, then by completeness PA $\vdash G$ or PA $\vdash \neg G$, contradicting Gödel
  - have a higher order theory HPA whose only model is $\mathcal{I}_{\mathrm{PA}}$: HPA $\models S$ iff $\models_{\mathcal{I}_{\mathrm{PA}}} S$
  - but there is no completeness theorem for higher-order logic
  - the problem is axiomatizing induction

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 $\texttt{Ssem}\ P\ s_1$
  - and *if* the execution of C starting in s_1 terminates
  - then C terminates in a state s_2 such that $\texttt{Ssem}\ Q\ s_2 = \mathit{true}$
- Need to define "C starts in s_1 and terminates in s_2"
  - this is the semantics of commands
  - will define $\texttt{Csem}\ C\ s_1\ s_2$ to mean if C starts in s_1 then it can terminate in s_2
- Semantics of $\{P\}\ C\ \{Q\}$ is $\texttt{Hsem}\ P\ C\ Q$ where:

  $\texttt{Hsem}\ P\ C\ Q = \forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2$
- Sometimes write $\models \{P\}\ C\ \{Q\}$ to mean $\texttt{Hsem}\ P\ C\ Q$

Semantics of commands

- Assignments

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

- Sequences

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

- Conditional

  $\texttt{Csem}\ (\texttt{IF}\,S\,\texttt{THEN}\,C_1\,\texttt{ELSE}\,C_2)\ s_1\ s_2$
  $=\ (\texttt{Ssem}\ S\ s_1\ \wedge\ \texttt{Csem}\ C_1\ s_1\ s_2)\ \vee\ (\neg\texttt{Ssem}\ S\ s_1\ \wedge\ \texttt{Csem}\ C_2\ s_1\ s_2)$
  $=\ \textit{if}\ \texttt{Ssem}\ S\ s_1\ \textit{then}\ \texttt{Csem}\ C_1\ s_1\ s_2\ \textit{else}\ \texttt{Csem}\ C_2\ s_1\ s_2$

- While-commands

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

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

  $\texttt{Iter}\ 0\ p\ c\ s_1\ s_2\ = \neg(p\ s_1) \wedge (s_1{=}s_2)$
  $\texttt{Iter}\ (n{+}1)\ p\ c\ s_1\ s_2 = p\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \texttt{Iter}\ n\ p\ c\ s\ s_2$

  - 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

- Semantics of $\{P\}\ C\ \{Q\}$:

  $\forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2$

- Assignment axiom:

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

- Must show:

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

- Unfolding the definition of Csem converts this to:

  $\forall s_1\ s_2.\ \texttt{Ssem}\ (Q\texttt{[}E/V\texttt{]})\ s_1 \wedge (s_2 = s_1\texttt{[}(\texttt{Esem}\ E\ s_1)/V\texttt{]}) \Rightarrow \texttt{Ssem}\ Q\ s_2$

- This simplifies to:

  $\forall s_1.\ \texttt{Ssem}\ (Q\texttt{[}E/V\texttt{]})\ s_1 \Rightarrow \texttt{Ssem}\ Q\ (s_1\texttt{[}(\texttt{Esem}\ E\ s_1)/V\texttt{]})$

  - $\texttt{[}\cdots/\cdots\texttt{]}$ has different meanings in antecedent and consequent
  - in antecedent $Q\texttt{[}E/V\texttt{]}$ is substituting E for V in Q
  - in consequent $s_1\texttt{[}(\texttt{Esem}\ E\ s_1)/V\texttt{]}$ is updating s_1 so value of V is value of E in s_1

- Will prove for all S that: $\texttt{Ssem}\ (S\texttt{[}E/V\texttt{]})\ s = \texttt{Ssem}\ S\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})$

Substitution lemma for expressions: variables

- Assume: language $\mathcal{L}$, interpretation $\mathcal{I} = (I, D)$, valuation $s \in \mathit{Var} \to D$

- $\forall s.\ \texttt{Esem}\ (E\texttt{[}E'/V\texttt{]})\ s\ =\ \texttt{Esem}\ E\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]})$ by induction on E

- If $E = V$ then must show

  $\texttt{Esem}\ (V\texttt{[}E/V\texttt{]})\ s\ =\ \texttt{Esem}\ V\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})$

  $\texttt{Esem}\ E\ s\ =\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})(V)$

  $\texttt{Esem}\ E\ s\ =\ \texttt{Esem}\ E\ s$

- If $E = V'$, where $V \neq V'$, then must show

  $\texttt{Esem}\ (V'\texttt{[}E/V\texttt{]})\ s\ =\ \texttt{Esem}\ V'\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})$

  $\texttt{Esem}\ V'\ s\ =\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})(V')$

  $s(V')\ =\ s(V')$

Substitution lemma for expressions: applications

- Assume: language $\mathcal{L}$, interpretation $\mathcal{I} = (I, D)$, valuation $s \in \mathit{Var} \to D$

- $\forall s.\ \texttt{Esem}\ (E\texttt{[}E'/V\texttt{]})\ s\ =\ \texttt{Esem}\ E\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]})$ by induction on E

- Assume $\texttt{Esem}\ (E_i\texttt{[}E'/V\texttt{]})\ s\ =\ \texttt{Esem}\ E_i\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]})$ for $1 \leq i \leq n$

- If $E = f$, where f has arity 0, then must show

  $\texttt{Esem}\ (f\texttt{[}E'/V\texttt{]})\ s\ =\ \texttt{Esem}\ f\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]})$

  $I[f]\ =\ I[f]$

- If $E = f(E_1, \ldots, E_n)$ then must show

  $\texttt{Esem}\ (f(E_1,\ldots,E_n)\texttt{[}E'/V\texttt{]})\ s\ =\ \texttt{Esem}\ (f(E_1,\ldots,E_n))\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]})$

  $\texttt{Esem}\ (f(E_1\texttt{[}E'/V\texttt{]},\ldots,E_n\texttt{[}E'/V\texttt{]}))\ s\ =$
  $I[f](\texttt{Esem}\ E_1\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]}),\ldots,\texttt{Esem}\ E_n\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]}))$

  $I[f](\texttt{Esem}\ (E_1\texttt{[}E'/V\texttt{]})\ s,\ldots,\texttt{Esem}\ (E_n\texttt{[}E'/V\texttt{]})\ s)\ =$
  $I[f](\texttt{Esem}\ E_1\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]}),\ldots,\texttt{Esem}\ E_n\ (s\texttt{[}(\texttt{Esem}\ E'\ s)/V\texttt{]}))$

  Equation true by induction

Substitution lemma for statements

- Assume: language $\mathcal{L}$, interpretation $\mathcal{I} = (I, D)$, valuation $s \in \mathit{Var} \to D$
- $\forall s.\ \texttt{Ssem}\ (S\texttt{[}E/V\texttt{]})\ s\ =\ \texttt{Ssem}\ S\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})$ by induction on S
- Proof similar to expressions *except* care needed with bound variables
- Assume bound variables renamed to avoid clashes, then:

  $(\forall v.\ S)\texttt{[}E/V\texttt{]} = \forall v.\ S\texttt{[}E/V\texttt{]}$
  $(\exists v.\ S)\texttt{[}E/V\texttt{]} = \exists v.\ S\texttt{[}E/V\texttt{]}$
- Need lemma for expressions when S is $p(E_1, \ldots, E_n)$

  $\texttt{Ssem}\ (p(E_1, \ldots, E_n)\texttt{[}E/V\texttt{]})\ s\ =\ \texttt{Ssem}\ (p(E_1, \ldots, E_n))\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]})$

  $\texttt{Ssem}\ (p(E_1\texttt{[}E/V\texttt{]}, \ldots, E_n\texttt{[}E/V\texttt{]}))\ s\ =$
  $I[p](\texttt{Esem}\ E_1\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]}), \ldots, \texttt{Esem}\ E_n\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]}))$

  $I[p](\texttt{Esem}\ (E_1\texttt{[}E/V\texttt{]})\ s, \ldots, \texttt{Esem}\ (E_n\texttt{[}E/V\texttt{]})\ s)\ =$
  $I[p](\texttt{Esem}\ E_1\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]}), \ldots, \texttt{Esem}\ E_n\ (s\texttt{[}(\texttt{Esem}\ E\ s)/V\texttt{]}))$

  Equation true by induction and lemma for expressions

Soundness of Assignment Axiom

- Semantics of $\{P\}\ C\ \{Q\}$:

  $\forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2$
- Assignment axiom:

  $\vdash\ \{Q\texttt{[}E/V\texttt{]}\}\ V\texttt{:=}E\ \{Q\}$
- Must show:

  $\forall s_1\ s_2.\ \texttt{Ssem}\ (Q\texttt{[}E/V\texttt{]})\ s_1 \wedge \texttt{Csem}\ (V\texttt{:=}E)\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2$
- Showed earlier that this simplifies to:

  $\forall s_1.\ \texttt{Ssem}\ (Q\texttt{[}E/V\texttt{]})\ s_1 \Rightarrow \texttt{Ssem}\ Q\ (s_1\texttt{[}(\texttt{Esem}\ E\ s_1)/V\texttt{]})$
- Follows from substitution lemma for statements

Soundness of Precondition Strengthening

- Precondition strengthening:

  $$\frac{\vdash P \Rightarrow P', \qquad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$
- Sound if for all P, P', C and Q:

  $(\forall s.\ \texttt{Ssem}\ P\ s \Rightarrow \texttt{Ssem}\ P'\ s) \wedge \texttt{Hsem}\ P'\ C\ Q \Rightarrow \texttt{Hsem}\ P\ C\ Q$
- After expanding the definition of `Hsem`:

  $(\forall s.\ \texttt{Ssem}\ P\ s \Rightarrow \texttt{Ssem}\ P'\ s)\ \wedge$
  $(\forall s_1\ s_2.\ \texttt{Ssem}\ P'\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2)$
  $\Rightarrow$
  $\forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2$
- An instance of the clearly true:

  $(\forall s.\ p\ s \Rightarrow p'\ s)\ \wedge\ (\forall s_1\ s_2.\ p'\ s_1 \wedge c\ s_1\ s_2 \Rightarrow q\ s_2)$
  $\Rightarrow$
  $\forall s_1\ s_2.\ p\ s_1 \wedge c\ s_1\ s_2 \Rightarrow q\ s_2$
- Soundness of postcondition weakening similar

Soundness of Sequencing Rule

- Conditional rule:

  $$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1\texttt{;}C_2\ \{R\}}$$
- Sound if:

  $\texttt{Hsem}\ P\ C_1\ Q \wedge \texttt{Hsem}\ Q\ C_2\ R \Rightarrow \texttt{Hsem}\ P\ (C_1\texttt{;}C_2)\ R$
- After expanding the definition of `Hsem`:

  $(\forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ Q\ s_2)\ \wedge$
  $(\forall s_1\ s_2.\ \texttt{Ssem}\ Q\ s_1 \wedge \texttt{Csem}\ C\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ R\ s_2)$
  $\Rightarrow$
  $\forall s_1\ s_2.\ \texttt{Ssem}\ P\ s_1 \wedge \texttt{Csem}\ (C_1\texttt{;}C_2)\ s_1\ s_2 \Rightarrow \texttt{Ssem}\ R\ s_2$
- An instance of the clearly true:

  $(\forall s_1\ s_2.\ p\ s_1 \wedge c_1\ s_1\ s_2 \Rightarrow q\ s_2)\ \wedge\ (\forall s_1\ s_2.\ q\ s_1 \wedge c_2\ s_1\ s_2 \Rightarrow r\ s_2)$
  $\Rightarrow$
  $\forall s_1\ s_2.\ p\ s_1 \wedge (\exists s.\ c_1\ s_1\ s \wedge c_2\ s\ s_2) \Rightarrow r\ s_2$
- Soundness of conditional rule similar

Soundness of WHILE Rule

- **WHILE-Rule:**

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

- **Sound if:**

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

- **Expands to:**

$(\forall s_1\ s_2.\ \mathtt{Ssem}\ (P \wedge S)\ s_1 \wedge \mathtt{Csem}\ C\ s_1\ s_2 \Rightarrow \mathtt{Ssem}\ P\ s_2)$
$\Rightarrow \forall s_1\ s_2.\ \mathtt{Ssem}\ P\ s_1 \wedge \mathtt{Csem}\ (\mathtt{WHILE}\ S\ \mathtt{DO}\ C)\ s_1\ s_2 \Rightarrow \mathtt{Ssem}\ (P \wedge \neg S)\ s_2$

- **Expanding the definition of** $\mathtt{Hsem}\ (\mathtt{WHILE}\ S\ \mathtt{DO}\ C)$ **and simplifying:**

$(\forall s_1\ s_2.\ \mathtt{Ssem}\ P\ s_1 \wedge \mathtt{Ssem}\ S\ s_1 \wedge \mathtt{Csem}\ C\ s_1\ s_2 \Rightarrow \mathtt{Ssem}\ P\ s_1$
$\Rightarrow \forall s_1\ s_2.\ \mathtt{Ssem}\ P\ s_1 \wedge (\exists n.\ \mathtt{Iter}\ n\ (\mathtt{Ssem}\ S)\ (\mathtt{Csem}\ C)\ s_1\ s_2)$
$\qquad \Rightarrow \mathtt{Ssem}\ P\ s_2 \wedge \neg(\mathtt{Ssem}\ S\ s_2)$

- **An instance of:**

$(\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1)$
$\Rightarrow \forall s_1\ s_2.\ p\ s_1 \wedge (\exists n.\ \mathtt{Iter}\ n\ b\ c\ s_1\ s_2) \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$

Soundness of WHILE Rule (continued)

- **From last slide need to prove:**

$(\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1)$
$\Rightarrow \forall s_1\ s_2.\ p\ s_1 \wedge (\exists n.\ \mathtt{Iter}\ n\ b\ c\ s_1\ s_2) \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$

- **This is equivalent to:**

$(\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1)$
$\Rightarrow$
$\forall n\ s_1\ s_2.\ p\ s_1 \wedge \mathtt{Iter}\ n\ b\ c\ s_1\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$

- **Assume** $\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1$**, then prove:**

$\forall n\ s_1\ s_2.\ p\ s_1 \wedge \mathtt{Iter}\ n\ b\ c\ s_1\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$

by mathematical induction of n

- **Routine using definition of** `Iter`**:**

$\mathtt{Iter}\ 0\ p\ c\ s_1\ s_2 \qquad = \neg(p\ s_1) \wedge (s_1{=}s_2)$
$\mathtt{Iter}\ (n{+}1)\ p\ c\ s_1\ s_2 = p\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \mathtt{Iter}\ n\ p\ c\ s\ s_2$

details in background reading

Soundness of Hoare Logic: summary

- **Assignment axiom:**
$\forall s_1\ s_2.\ \mathtt{Ssem}\ (Q[E/V])\ s_1 \wedge \mathtt{Csem}\ (V{:=}E)\ s_1\ s_2 \Rightarrow \mathtt{Ssem}\ Q\ s_2$
$\models \{Q[E/V]\}V{:=}E\{Q\}$
- **Precondition strengthening:**
$(\forall s.\ \mathtt{Ssem}\ P\ s \Rightarrow \mathtt{Ssem}\ P'\ s) \wedge \mathtt{Hsem}\ P'\ C\ Q \Rightarrow \mathtt{Hsem}\ P\ C\ Q$
$(\models P \Rightarrow P')\ \wedge\ \models \{P'\}C\{Q\}\ \Rightarrow\ \models \{P\}C\{Q\}$
- **Postcondition weakening:**
$\mathtt{Hsem}\ P\ C\ Q' \wedge (\forall s.\ \mathtt{Ssem}\ Q'\ s \Rightarrow \mathtt{Ssem}\ Q\ s) \Rightarrow \mathtt{Hsem}\ P\ C\ Q$
$\models \{P\}C\{Q'\}\ \wedge (\models Q' \Rightarrow Q)\ \Rightarrow\ \models \{P\}C\{Q\}$
- **Sequencing rule:**
$\mathtt{Hsem}\ P\ C_1\ Q \wedge \mathtt{Hsem}\ Q\ C_2\ R \Rightarrow \mathtt{Hsem}\ P\ (C_1;C_2)\ R$
$\models \{P\}C_1\{Q\}\ \wedge\ \models \{Q\}C_2\{R\} \Rightarrow\ \models\ \{P\}C_1;C_2\{R\}$
- **Conditional rule:**
$\mathtt{Hsem}\ (P{\wedge}S)\ C_1\ Q{\wedge}\mathtt{Hsem}\ (P{\wedge}\neg Q)\ C_2\ Q \Rightarrow \mathtt{Hsem}\ P\ (\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q$
$\models \{P{\wedge}S\}C_1\{Q\} \wedge\ \models \{P{\wedge}\neg S\}C_2\{Q\}\ \Rightarrow\ \models \{P\}\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\{Q\}$
- **WHILE rule:**
$\mathtt{Hsem}\ (P \wedge S)\ C\ P \Rightarrow \mathtt{Hsem}\ P\ (\mathtt{WHILE}\ S\ \mathtt{DO}\ C)\ (P \wedge \neg S))$
$\models \{P \wedge S\}C\{P\}\ \Rightarrow\ \models \{P\}\mathtt{WHILE}\ S\ \mathtt{DO}\ C$

Completeness and decidability of Hoare Logic

- **Soundness:** $\vdash \{P\}C\{Q\}\ \Rightarrow\ \models \{P\}C\{Q\}$

- **Decidability:** $\{\mathtt{T}\}C\{\mathtt{F}\}\ \Leftrightarrow\ C$ **<u>doesn't</u> halt**
  - **the Halting Problem is undecidable**

- **Completeness: really want** $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}\ \Rightarrow\ \mathbf{PA} \vdash \{P\}C\{Q\}$
  - **not possible**
  - $\models_{\mathcal{I}_{\mathbf{PA}}} \{\mathtt{T}\}\mathtt{X{:=}X}\{P\}$ **if and only if** $\models_{\mathcal{I}_{\mathbf{PA}}} P$
  - $\mathbf{PA} \vdash \{\mathtt{T}\}\mathtt{X{:=}X}\{P\}$ **if and only if** $\mathbf{PA} \vdash P$

- **If complete, as above, then for any statement** P**:**
$\models_{\mathcal{I}_{\mathbf{PA}}} P\ \Rightarrow\ \models_{\mathcal{I}_{\mathbf{PA}}} \{\mathtt{T}\}\mathtt{X{:=}X}\{P\}\ \Rightarrow\ \mathbf{PA} \vdash \{\mathtt{T}\}\mathtt{X{:=}X}\{P\}\ \Rightarrow\ \mathbf{PA} \vdash P$
which can't be by Gödel's theorem

- **Must separate completeness of programming and specification logics**

Relative completeness (Cook 1978) – basic idea

- **Assume** $\texttt{wlp}(C, Q)$ **expressible in** $\mathcal{L}$ **and** $\Gamma \vdash \{\texttt{wlp}(C, Q)\}C\{Q\}$**, some** Γ
- **For simplicity take** $\mathcal{L} = \mathcal{L}_{\mathbf{PA}}$ **and** $\Gamma = \{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\}$
- **Show** $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}$ **entails** $\models_{\mathcal{I}_{\mathbf{PA}}} P \Rightarrow \texttt{wlp}(C, Q)$
- **Hence** $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}$ **entails** $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\} \vdash \{P\}C\{Q\}$
  - **assume** $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}$
  - **then** $P \Rightarrow \texttt{wlp}(C, Q) \in \{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\}$
  - **by expressibility:** $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\} \vdash \{\texttt{wlp}(C, Q)\}C\{Q\}$
  - **hence by precondition strenthening:** $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\} \vdash \{P\}C\{Q\}$
- **Hoare logic is complete:**
  - **relative to** $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\}$
  - **assuming** $\texttt{wlp}(C, Q)$ **is expressible**

Discussion of proof of relative completeness

- **Expressing** $\texttt{wlp}(C, Q)$ **easy for assignments, sequences, conditionals**

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

- **Expressing** $\texttt{wlp}((\texttt{WHILE}\, S\, \texttt{DO}\, C), Q)$ **is harder**
  - **tricky encoding in first-order arithmetic using Gödel's** β **function (see Winskel's book** *The formal semantics of programming languages: an introduction***)**
- **In the background reading**
  - $\texttt{wlp}(\texttt{WHILE}\, S\, \texttt{DO}\, C, Q)$ **defined using** *infinite conjunctions* **(expressibility)**
  - $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}$ **implies** $\models_{\mathcal{I}_{\mathbf{PA}}} P \Rightarrow \texttt{wlp}(C, Q)$ **by induction on** C **and semantics**
  - $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\} \vdash \{\texttt{wlp}(C, Q)\}C\{Q\}$ **by induction on** C **and Hoare logic**
  - **hence** $\models_{\mathcal{I}_{\mathbf{PA}}} \{P\}C\{Q\}$ **implies** $\{S \mid \models_{\mathcal{I}_{\mathbf{PA}}} S\} \vdash \{P\}C\{Q\}$

Summary: soundness, decidability, completeness

- **Hoare logic is sound**
- **Hoare logic is undecidable**
  - **deciding** $\{\texttt{T}\}C\{\texttt{F}\}$ **is halting problem**
- **Hoare logic for our simple language is complete relative to an oracle**
  - **oracle must be able to prove** $P \Rightarrow \texttt{wlp}(C, Q)$
  - **relative completeness**
  - **requires expressibility:** $\texttt{wlp}(C, Q)$ **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**
- $\texttt{FOR}$**-commands**
- **Arrays**
- **Correct-by-Construction (program refinement)**
- **Separation Logic**

Overview

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

Array assignments

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

Naive Array Assignment Axiom Fails

- The axiom

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

  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

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

  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

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

  as an ordinary assignment

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

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

Array Assignment axiom

- Array assignment is a special case of ordinary assignment

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

- Array assignment axiom just ordinary assignment axiom

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

- Thus:

The array assignment axiom

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

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

Array Axioms

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

The array axioms

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

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

- Second of these is a *Frame Axiom*
  - don't confuse with Frame Rule of Separation Logic (later)
  - "frame" is a rather overloaded word!

For more rigour define a first order theory ARRAY

- $\mathcal{L}_{\mathrm{ARRAY}}\ =\ \{\{isarray\}, \{lookup, update\}\}$
  - $isarray$ has arity 1, $lookup$ has arity 2, $update$ has arity 3
- $\mathcal{I}_{\mathrm{ARRAY}}$
  - domain is $\mathbb{V} \cup \{\phi \mid \phi : \mathbb{N} \to \mathbb{V}\}$ for some set of values $\mathbb{V}$
  - $I_{\mathrm{ARRAY}}[isarray](a)$ is true iff a is a function ϕ
  - $I_{\mathrm{ARRAY}}[lookup](a, i) =$ ***if*** a is a function ϕ ***then*** $\phi(i)$ ***else*** 0
  - $I_{\mathrm{ARRAY}}[update](a, i, v) =$ ***if*** a is a function ϕ ***then*** $\phi[v/i]$ ***else*** a
- ARRAY contains the following axioms
  - $\forall a\ i\ v.\ isarray(a) \Rightarrow (lookup(update(a, i, v), i) = v)$
  - $\forall a\ i\ j\ v.\ isarray(a) \wedge \neg(i = j) \Rightarrow (lookup(update(a, i, v), j) = lookup(a, j))$
  - $\forall a_1\ a_2.\ isarray(a_1) \wedge isarray(a_2) \wedge (\forall i.\ lookup(a_1, i) = lookup(a_2, i)) \Rightarrow (a_1 = a_2)$
- "$a(i)$" means "$lookup(a, i)$" and "$a\{i{\leftarrow}v\}$" means "$update(a, i, v)$"
- Assuming a is an array ($isarray(a)$ is true) then from array axioms:
  - $a\{i{\leftarrow}v\}(i) = v$
  - $\neg(i = j) \Rightarrow (a\{i{\leftarrow}v\}(j) = a(j))$

Example

- We show

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

- Working backwards using the array assignment axiom

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

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

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

Example Continued (1)

- Using

$\vdash$ `A{Y←R}(Y) = R`

- It follows that

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

- Continuing backwards

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

- Maybe more intuitive if the assignment is rewritten

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


Example Continued (2)

- Continuing backwards

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

- Hence by the derived sequencing rule

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

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

$\vdash$ `((A{X←A(Y)}){Y←A(X)})(X)  =  A(Y)`

- Hence (as desired)

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


Blocks and local variables

- Syntax: `BEGIN VAR` V_1`;` $\cdots$ `VAR` V_n`;` C `END`

- Semantics: command C is executed, then the values of $V_1, \cdots, V_n$ are restored to the values they had before the block was entered
  - the initial values of $V_1, \cdots, V_n$ inside the block are unspecified

- Example: `BEGIN VAR R; R:=X; X:=Y; Y:=R END`
  - the values of `X` and `Y` are swapped using `R` as a temporary variable
  - this command does *not* have a side effect on the variable `R`

The Block Rule

- The block rule takes care of local variables

The block rule

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

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

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

The Side Condition

- The syntactic condition that none of the variables $V_1, \ldots, V_n$ occur in P or Q is an example of a *side condition*

- From
  $\vdash$ {X=x ∧ Y=y} R:=X; X:=Y; Y:=R {Y=x ∧ X=y}
  it follows by the block rule that
  $\vdash$ {X=x ∧ Y=y} BEGIN VAR R; R:=X; X:=Y; Y:=R END {Y=x ∧ X=y}
  since R does not occur in X=x ∧ Y=y or X=y ∧ Y=x

- However from
  $\vdash$ {X=x ∧ Y=y} R:=X; X:=Y {R=x ∧ X=y}
  one *cannot* deduce
  $\vdash$ {X=x ∧ Y=y} BEGIN VAR R; R:=X; X:=Y END {R=x ∧ X=y}
  since R occurs in R=x ∧ X=y

Exercises

- Consider the specification

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

  Can this be deduced from the rules given so far?

  (i) if so, give a proof of it

  (ii) if not, explain why not and suggest additional rules and/or axioms to enable it to be deduced

- Is the following true?

  $\vdash$ {X=x ∧ Y=y} X:=X+Y; Y:=X-Y; X:=X-Y {Y=x ∧ X=y}

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

- Show

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

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, … , 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, … , 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

$$\texttt{FOR } V\texttt{:=}E_1 \texttt{ UNTIL } E_2 \texttt{ DO } C$$

  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:

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

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

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

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

The 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 $\vdash\ \{P\}\ C\ \{P\texttt{[}V\texttt{+}1/V\texttt{]}\}$ then:

$\vdash\ \{P \wedge V{=}v\}\ C\ \{P\texttt{[}V\texttt{+}1/V\texttt{]} \wedge V{=}v\}$ (assumption + constancy rule)

$\vdash\ \{P\texttt{[}V\texttt{+}1/V\texttt{]} \wedge V{=}v\}\ V\texttt{:=}V\texttt{+}1\ \{P \wedge V{=}v{+}1\}$ (assign. ax + pre. streng.)

$\vdash\ \{P \wedge V{=}v\}\ C\texttt{;}\ V\texttt{:=}V\texttt{+}1\ \{P \wedge V{=}v{+}1\}$ (sequencing)

- So $C\texttt{;}\ V\texttt{:=}V\texttt{+}1$ has P as an invariant and increments V

Towards the FOR-Rule

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

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

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

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

$\vdots$

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

$\vdots$

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

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

- Hence by the sequencing and block rules

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

Problems with the FOR-rule (i)

- Previous derivation suggests a rule

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

- This is a good start, but needs debugging
- Consider:

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

- Taking P as 'X=Y' this is:

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

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

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

Problems with the FOR-rule (ii)

- **The conclusion below is clearly undesirable**

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

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

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

Problems with the FOR-rule (iii)

- **FOR-rule so far**

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

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

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

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

Problems with the FOR-rule (iv)

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

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

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

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

- **So by this FOR-rule**

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

Problems with the FOR-rule (v)

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

Problems with the FOR-rule (vi)

- Problem may also arise if variables in expressions E_1 or E_2 modified
- For example, taking P to be Z=Y, then

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

- Thus the following can be derived

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

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

The FOR-Rule

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

The FOR-rule

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

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):

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

The FOR-axiom

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

The FOR-axiom

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

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

Exercise: understand the example on this slide

- By the assignment axiom and precondition strengthening

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

- Strengthening the precondition of this again yields

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

- Hence by the `FOR`-rule

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

- Hence

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


Note on using the `FOR`-Rule

- Note that if any of the following hold

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

- Then by precondition strengthening:

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

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

Deriving the `FOR` Rule

- The following is a command equivalent to `FOR I:=`E_1 `UNTIL` E_2 `DO C`

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

  - `UpperBound` is assumed to be a 'new' variable
  - and `I` is not assigned to inside `C`

- Thus we could derive a rule from the implementation
  - we must be sure the implementation is correct

- Exercise: try deriving the `FOR`-rule from the `WHILE`-rule

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

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

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

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

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

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

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

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

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

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

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

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

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

From *Proof rules for the programming language Euclid*

7.1. (module rule)

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

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

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

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

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

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

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

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

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

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

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

$\vdash$

(8.5) $P(x\sharp/x)\ \{x\,.\,Initially;\ S;\ x\,.\,Finally\}\ R(x\sharp/x)$

$$\frac{}{P\{\textbf{var } x\colon T(a);\ S\}\ R \wedge Q1}$$

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

217

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

218

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
- $[\mathtt{X>0},\ \mathtt{Y=X}^2]$
  - from a state where `X` is greater than zero, the program should terminate with `Y` the square of `X`

219

A Little Wide Spectrum Programming Language

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

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

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

$$\begin{array}{lll} C & ::= & \mathtt{SKIP} \qquad\qquad \text{(does nothing, SKIP-Axiom is } \vdash [P]\ \mathtt{SKIP}\ [P]) \\ & \mid & V \ \mathtt{:=}\ E \\ & \mid & C_1\ ;\ C_2 \\ & \mid & \mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2 \\ & \mid & \mathtt{BEGIN\ VAR}\ V_1\ ;\ ..\ \mathtt{VAR}\ V_1\ ;\ C\ \mathtt{END} \\ & \mid & \mathtt{WHILE}\ B\ \mathtt{DO}\ C \\ & \mid & [P,\ Q] \end{array}$$

220

Specifications as Sets of Commands

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

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

- Think of a specification as defining the set of implementations

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

- For example

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

- Don't confuse use of $\{\cdots\}$ 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:

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

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

$$\mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ c\ \mathtt{END} = \{\ \mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ C\ \mathtt{END}\ \mid\ C \in c\ \}$$

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

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

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

$$spec \supseteq mixed_1 \supseteq \ldots \supseteq mixed_n \supseteq code$$

Skip Law

The Skip Law

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

- Derivation:

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

- Examples

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

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

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

Notational Convention

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

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

- **Examples become:**

`[X=1,  X=1]` $\supseteq$ `SKIP`

`[T,  T]` $\supseteq$ `SKIP`

`[X=R+Y`$\times$`Q,  X=R+Y`$\times$`Q]` $\supseteq$ `SKIP`

Assignment Law

The Assignment Law

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

- **Derivation**

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

- **Examples**

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

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

Laws of Consequence

Precondition Weakening

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

provided $\vdash\quad P \;\Rightarrow\; R$

Postcondition Strengthening

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

provided $\vdash\quad R \;\Rightarrow\; Q$

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

Derivation of Consequence Laws

- **Derivation of Precondition Weakening**

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

- **Derivation of Postcondition Strengthening**

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

Examples (illustrates refinement notation)

- **A previous example:**

```
[X=1,  X=1]
⊒ (Skip)
SKIP
```

- **An alternative refinement:**

```
[Y=1,  X=1]
⊒ (Precondition Weakening  ⊢  Y=1  ⇒  1=1)
[1=1,  X=1]
⊒ (Assignment)
X := 1
```

- **Another example**

```
[T,  R=X]
⊒ (Precondition Weakening  ⊢  T  ⇒  X=X)
[X=X,  R=X]
⊒ (Assignment)
R := X
```


Derived Assignment Law

Derived Assignment Law

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

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

- **Derivation**

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

- **Example**

```
[T,  R=X]
⊒ (Derived Assignment  ⊢  T ⇒ X=X)
R := X
```


One Slide Technical Interlude: Monotonicity

- **A command can be refined by separately refining its constituents**
- **This is because sets of commands are** *monotonic* **w.r.t.** $\sqsupseteq$
  - **if** $c{\sqsupseteq}c'$, $c_1{\sqsupseteq}c'_1$, $\dots$, $c_n{\sqsupseteq}c'_n$
  - **then:**

$$\begin{array}{ll}
c_1;\ \cdots\ ;c_n & \sqsupseteq\ c'_1;\ \cdots\ ;c'_n \\
\mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ c\ \mathtt{END} & \sqsupseteq\ \mathtt{BEGIN\ VAR}\ V_1;\ \cdots\ \mathtt{VAR}\ V_n;\ c'\ \mathtt{END} \\
\mathtt{IF}\ S\ \mathtt{THEN}\ c_1\ \mathtt{ELSE}\ c_2 & \sqsupseteq\ \mathtt{IF}\ S\ \mathtt{THEN}\ c'_1\ \mathtt{ELSE}\ c'_2 \\
\mathtt{WHILE}\ S\ \mathtt{DO}\ c & \sqsupseteq\ \mathtt{WHILE}\ S\ \mathtt{DO}\ c'
\end{array}$$

- **Laws of refinement for non-atomic commands now follow**

Sequencing

The Sequencing Law

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

- **Derivation of Sequencing Law**

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

- **Example**

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

Creating different Programs

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

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


Inefficient Programs

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

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


Blind Alleys

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

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


Blocks

The Block Law

$[P,\ \ Q] \supseteq$ BEGIN VAR V_1; ... ; VAR V_n; $[P,\ \ Q]$ END

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

- Derivation: exercise
- Example

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

Conditional

The Conditional Law

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

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

Derivation of the Conditional Law

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

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

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

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

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

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

Example

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


While

The While Law

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

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

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

- **Example**

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

Derivation of the While Law

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

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

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

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

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

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

Example (i)

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


Example (ii)

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

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

Example (iii)

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

Example (iv)

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


Example (v)

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


More Notation

- The notation

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

  is used to abbreviate:

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

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

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

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

  abbreviates

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

Exercise: check the refinement on this slide

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

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

Derived Laws

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

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

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

Data Refinement

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

Summary

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

New Topic: 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 $s : Var \to Val$
- 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

$$\mathtt{X} \underset{store}{\mapsto} l_1 \underset{heap}{\mapsto} l_2 \underset{heap}{\mapsto} v$$

Heap semantics

$Store = Var \to Val$ (assume $Num \subseteq Val$, $\mathtt{nil} \in Val$ and $\mathtt{nil} \notin Num$)
$Heap = Num \rightharpoonup_{fin} Val$
$State = Store \times Heap$

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

Adding pointer operations to our language

Expressions:

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

Boolean expressions:

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

commands:

$C ::= V$ `:=` E	value assignments
$\mid$ V`:=[`E`]`	fetch assignments
$\mid$ `[`E_1`]:=`E_2	heap assignments (heap mutation)
$\mid$ V`:=cons(`$E_1, \ldots, E_n$`)`	allocation assignments
$\mid$ `dispose(`E`)`	pointer disposal
$\mid$ C_1 `;` C_2	sequences
$\mid$ `IF` B `THEN` C_1 `ELSE` C_2	conditionals
$\mid$ `WHILE` B `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: `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`:=cons(`$E_1, \ldots, E_n$`)`
  - choose n consecutive locations that are not in the heap, say $l, l{+}1, \ldots$
  - extend the heap by adding $l, l{+}1, \ldots$ to it
  - assign l to the variable V in the store
  - make the values of $E_1, E_2, \ldots$ be the new contents of $l, l{+}1, \ldots$ in the heap
- Allocation assignments never fault
- Allocation assignments are *non-deterministic*
  - any suitable $l, l{+}1, \ldots$ not in the heap can be chosen
  - always exists because the heap is finite

Example (different from the background reading)

`X:=cons(0,1,2); [X]:=Y+1; [X+1]:=Z; Y:=[Y+Z]`

- `X:=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**

Separating assertions

- **Another example:** `X:=cons(0); Z:=1; Y:=X; [Y]:=Z; Y:=[X]`
  - **assigns** `X` **to a new pointer,** l **say, and then updates contents of** l **to** 0
  - **assigns** `Z` **to** 1 **and** `Y` **to** l
  - **updates the contents of the value of** `Y`**, i.e.** l**, to be the value of** `Z`**, i.e.** 1
  - **assigns** `Y` **to contents of value of** `X`**, i.e. contents of** l**, i.e.** 1
- **Want to prove:** $\{\mathtt{T}\}$ `X:=cons(0);Z:=1; Y:=X; [Y]:=Z; Y:=[X]` $\{\mathtt{Y}=1\}$
  - **need additional axioms for fetch, store and allocation assignments**
  - **need assertions in specification language to describe contents of heap**
- **Intuitively**

  $\{\mathtt{T}\}$ `X:=cons(0)` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0\}$ *where* l *is a new location and* $\mathbf{H}$ *is the heap*

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0\}$ `Z:=1` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1\}$

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1\}$ `Y:=X` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$ `[Y]:=Z` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}1 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}1 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$ `Y:=[X]` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}1 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}1\}$
- **How can this be formalised? The tricky bit is the heap mutation:**

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$ `[Y]:=Z` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}1 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$

Heap assignment (mutation)

- **A plausible Floyd-style forward heap assignment axiom:**

  $\{E_1 = l \wedge E_2 = v\}\ [E_1]\mathtt{:=}E_2\ \{\mathbf{H}(l) = v\}$
- **How can we get from this to:**

  $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$ `[Y]:=Z` $\{\mathtt{X}{=}l \wedge \mathbf{H}(l){=}1 \wedge \mathtt{Z}{=}1 \wedge \mathtt{Y}{=}l\}$
- **Appropriate instance of plausible heap assignment axiom:**

  $\{\mathtt{Y} = l \wedge \mathtt{Z} = 1\}$ `[Y]:=Z` $\{\mathbf{H}(l) = 1\}$

The rule of constancy (derived rule of Hoare logic)

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

where no variable modified by C occurs free in R.

- **Hence:**

  $\{\mathtt{Y}{=}l \wedge \mathtt{Z}{=}1 \wedge \underbrace{(\mathtt{X}{=}l \wedge \mathbf{H}(l){=}0)}_{R}\}$ `[Y]:=Z` $\{\underbrace{\mathbf{H}(l){=}1} \wedge (\mathtt{X}{=}l \wedge \underbrace{\mathbf{H}(l){=}0})\}$

  Fail!

Rule of constancy (Reynolds' name)

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.

- **Derived rule of basic Hoare logic (proof: structural induction on** C**)**
  - **useful for strengthening invariants**
  - **also useful for decomposing proofs – an example use case**
    - **suppose** $\vdash \{P_1\}C_1\{Q_1\}$ **and** $\vdash \{P_2\}C_2\{Q_2\}$
    - **suppose no variable modified by** C_1 **occurs in** P_2
    - **then by rule of constancy:** $\vdash \{P_1 \wedge P_2\}C_1\{Q_1 \wedge P_2\}$
    - **suppose no variable modified by** C_2 **occurs in** Q_1
    - **then by rule of constancy:** $\vdash \{P_2 \wedge Q_1\}C_2\{Q_2 \wedge Q_1\}$
    - **hence by commutativity of** $\wedge$ **and sequencing rule:** $\vdash \{P_1 \wedge P_2\}C_1;C_2\{Q_1 \wedge Q_2\}$
- **Rule of constancy not valid for heap assignments:**

  $\vdash\ \{\mathtt{T}\}\,[\mathtt{X}]\mathtt{:=}0\,\{\mathbf{H}(\mathtt{X}) = 0\}$

  but not

  $\vdash \{\mathtt{T}\ \wedge\ \mathbf{H}(\mathtt{Y}) = 1\}\,[\mathtt{X}]\mathtt{:=}0\,\{\mathbf{H}(\mathtt{X}) = 0 \wedge \mathbf{H}(\mathtt{Y}) = 1\}$

  because $\mathtt{X} = \mathtt{Y}$ **possible**

Reasoning about the heap

- Could explicitly model locations and the heap directly in assertions
  - can be made to work – indeed still used, e.g. by Rockwell Collins
- Have a distinguished variable, say H, and then translate:

$V\texttt{:=[}E\texttt{]}$	$\leadsto$	$V\texttt{:=}\mathbf{H}(E)$	(assign value of E in H to V)
$\texttt{[}E_1\texttt{]:=}E_2$	$\leadsto$	$\mathbf{H}\texttt{:=}\mathbf{H}\texttt{[}E_2\texttt{/}E_1\texttt{]}$	(change H at E_1 to be E_2)
$V\texttt{:=cons}(E_1,\ldots,E_n)$	$\leadsto$	$\cdots$	(not sure about this case)
$\texttt{dispose}(E)$	$\leadsto$	$\mathbf{H}\texttt{:=}\mathbf{H}\texttt{-}E$	(delete E from domain of H)

- If $\texttt{[}E_1\texttt{]:=}E_2$ is $\mathbf{H}\texttt{:=}\mathbf{H}\texttt{[}E_2\texttt{/}E_1\texttt{]}$ then $\texttt{[}E_1\texttt{]:=}E_2$ modifies variable H
  - rule of constancy now valid, but less useful
  - adjoined variable R cannot mention H
  - need stronger notion of separation involving disjoint heaps

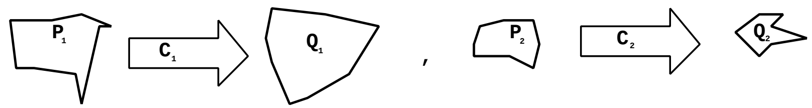
Heap assigment axiom again

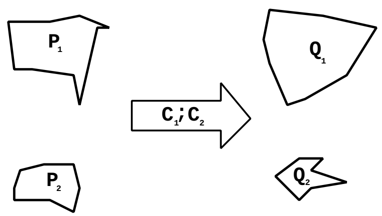
- Translating $\texttt{[}E_1\texttt{]:=}E_2$ to $\mathbf{H}\texttt{:=}\mathbf{H}\texttt{[}E_2\texttt{/}E_1\texttt{]}$ yields by assignment axiom:
  $\{Q\texttt{[}\mathbf{H}\texttt{[}E_2\texttt{/}E_1\texttt{]/}\mathbf{H}\texttt{]}\}\texttt{[}E_1\texttt{]:=}E_2\{Q\}$
- An instance is:
  $\{(\underbrace{\texttt{X}=l \wedge \mathbf{H}(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l}_{Q})\texttt{[}\mathbf{H}\texttt{[Z/Y]/}\mathbf{H}\texttt{]}\}\ \texttt{[Y]:=Z}\ \{\underbrace{\texttt{X}=l \wedge \mathbf{H}(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l}_{Q}\}$
  performing the substitution $Q\texttt{[}\mathbf{H}\texttt{[Z/Y]/}\mathbf{H}\texttt{]}$ gives:
  $\{(\texttt{X}=l \wedge (\mathbf{H}\texttt{[Z/Y]})(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l)\}\ \texttt{[Y]:=Z}\ \{\texttt{X}=l \wedge \mathbf{H}(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l\}$
  the conjunct $(\mathbf{H}\texttt{[Z/Y]})(l)=1$ is true, hence:
  $\{(\texttt{X}=l \wedge \texttt{Z}=1 \wedge \texttt{Y}=l)\}\ \texttt{[Y]:=Z}\ \{\texttt{X}=l \wedge \mathbf{H}(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l\}$
  then by precondition strengthening:
  $\{\texttt{X}=l \wedge \mathbf{H}(l)=0 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l\}\ \texttt{[Y]:=Z}\ \{\texttt{X}=l \wedge \mathbf{H}(l)=1 \wedge \texttt{Z}=1 \wedge \texttt{Y}=l\}$
  as wanted!

Rule of constancy again

- If $\texttt{[}E_1\texttt{]:=}E_2$ is translated to $\mathbf{H}\texttt{:=}\mathbf{H}\texttt{[}E_2\texttt{/}E_1\texttt{]}$
  then any command C containing a heap assignment will modify H
- If C_1, C_2 both contain heap assignments
  and either Q_1 or P_2 contains H, then can't do:
  - suppose $\vdash \{P_1\}C_1\{Q_1\}$ and $\vdash \{P_2\}C_2\{Q_2\}$
  - suppose no variable modified by C_1 occurs in P_2
  - then by rule of constancy: $\vdash \{P_1 \wedge P_2\}C_1\{Q_1 \wedge P_2\}$
  - suppose no variable modified by C_2 occurs in Q_1
  - then by rule of constancy: $\vdash \{P_2 \wedge Q_1\}C_2\{Q_2 \wedge Q_1\}$
  - hence by commutativity of $\wedge$ and sequencing rule: $\vdash \{P_1 \wedge P_2\}C_1;C_2\{Q_1 \wedge Q_2\}$
- Would like:
  - C_1 and C_2 modify disjoint parts of the heap and $\vdash \{P_1\}C_1\{Q_1\}$ and $\vdash \{P_2\}C_2\{Q_2\}$
  - P_1 only refers to locations modified by C_1 and P_2 only refers to locations modified by C_2
  - suppose no variable modified by C_1 occurs in P_2
  - then by *some rule*: $\vdash \{P_1 \wedge P_2\}C_1\{Q_1 \wedge P_2\}$
  - suppose no variable modified by C_2 occurs in Q_1
  - then by *some rule*: $\vdash \{P_2 \wedge Q_1\}C_2\{Q_2 \wedge Q_1\}$
  - hence by commutativity of $\wedge$ and sequencing rule: $\vdash \{P_1 \wedge P_2\}C_1;C_2\{Q_1 \wedge Q_2\}$

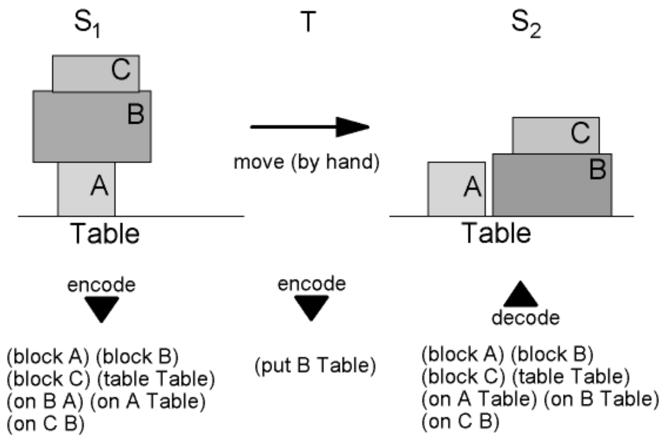
Diagram





The Frame Problem

- Treating $[E_1]$:=E_2 as H:=H$\{E_1 \leftarrow E_2\}$ works
  - forces one to use array frame axiom to prove locations $\neq E_1$ unchanged
  - clumsy ... but used successfully by some (e.g. ACL2 + Rockwell Collins)
  - need to always reason about the whole heap
- Analogy from AI



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
  - 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

Notation for separation assertions

- Expressions E evaluated in the store s, just like before
  - $E(s)$ means E true in s – i.e. `Esem` E s
- In general an assertion depends of store s and heap h
  - $P(s, h)$ means P true in state (s, h) (`SSsem` P (s, h) in background reading)
  - semantics of first-order logic statement S (doesn't depend on heap) is `Ssem` S s
- Notation and terminology for finite functions
  - $\mathtt{dom}(f)$ is domain of finite function f, so if $f : A \rightharpoonup_{fin} B$ then $\mathtt{dom}(f) = A$
  - $f[b/a]$ is same as f except a maps to b, $\mathtt{dom}(f[b/a]) = \mathtt{dom}(f) \cup \{a\}$
  - $f\text{-}a$ is the result of deleting a from $\mathtt{dom}(f)$, so $\mathtt{dom}(f\text{-}a) = \mathtt{dom}(f) \setminus \{a\}$
  - $\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ finite function with domain $\{l_1, \ldots, l_n\}$ and maps l_i to v_i
- Notation and terminology for operations on the heap
  - l is in heap h means $l \in \mathtt{dom}(h)$
  - $\mathtt{dom}(h_1 \cup h_2) = \mathtt{dom}(h_1) \cup \mathtt{dom}(h_2)$ and $(h_1 \cup h_2)(l) = \textbf{\textit{if}}\ l \in \mathtt{dom}(h_1)\ \textbf{\textit{then}}\ h_1(l)\ \textbf{\textit{else}}\ h_2(l)$
  - $h_1 \star h_2$ only defined if $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$, then $h_1 \star h_2 = h_1 \cup h_2$
    (there are two operators called "$\star$": joining heaps and separating conjunction)

Separation logic assertions: points-to

- $E_1 \mapsto E_2$ is the *points-to* relation where E_1, E_2 are expressions
- $E_1 \mapsto E_2$ means:
  - heap consists of one location: the value of E_1
  - the contents of the location (the value of E_1) is the value of E_2
- Semantics of $E_1 \mapsto E_2$ is defined by:
  $(E_1 \mapsto E_2)(s,h) \;\Leftrightarrow\; \mathtt{dom}(h) = \{E_1(s)\} \;\wedge\; h(E_1(s)) = E_2(s)$
- Example: $(\mathtt{X} \mapsto \mathtt{Y}{+}1)(s, \{20 \mapsto 43\})$ is true iff $s(\mathtt{X}) = 20$ and $s(\mathtt{Y}) = 42$
- Abbreviation: $E \mapsto \_ \;=_{def}\; \exists X.\ E \mapsto X$ (where X does not occur in E)
- By semantics: $(E \mapsto \_)\ (s,h) \;\Leftrightarrow\; \mathtt{dom}(h) = \{E(s)\}$

269

Separation logic assertions: separating conjunction

- $P_1 \star P_2$ is the *separating conjunction* of statements P_1 and P_2
- $P_1 \star P_2$ means:
  - the heap h can be split into two disjoint sub-heaps h_1, h_2 so that: $h = h_1 \star h_2$ (Note: "$\star$" in $h_1 \star h_2$ is the disjoint union of finite functions)
  - P_1 is true for h_1 and P_2 is true for h_2 (same store used for both P_1 and P_2)
- The semantics of the separating conjuction $P \star Q$ is defined by:
  $(P \star Q)(s,h) \;\Leftrightarrow\; \exists h_1\ h_2.\ h = h_1 \star h_2 \;\wedge\; P(s,h_1) \;\wedge\; Q(s,h_2)$
- Example: $(\mathtt{X} \mapsto 0 \star \mathtt{X}{+}1 \mapsto 0)\ (s, \{20 \mapsto 0, 21 \mapsto 0\})$ is true iff $s(\mathtt{X}) = 20$
- Abbreviation: $E \mapsto E_0, \ldots, E_n \;=_{def}\; (E \mapsto E_0) \star \cdots \star (E{+}n \mapsto E_n)$
  - specifies contents of $n{+}1$ contiguous locations starting at E
  - for $0 \le i \le n$ the contents of location $E{+}i$ is value of E_i
- Example: $(\mathtt{X} \mapsto \mathtt{Y}, \mathtt{Z})(s, \{x \mapsto y, x{+}1 \mapsto z\})$ is true iff $s(\mathtt{X}){=}x \wedge s(\mathtt{Y}){=}y \wedge s(\mathtt{Z}){=}z$

270

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:
  $\mathtt{emp}\ (s,h) \;\Leftrightarrow\; h = \{\}$ (where $\{\}$ is the empty heap)
- Abbreviation: $E_1 \doteq E_2 \;=_{def}\; (E_1 = E_2) \;\wedge\; \mathtt{emp}$
- From the semantics: $(E_1 \doteq E_2)\ (s,h) \;\Leftrightarrow\; E_1(s) = E_2(s) \;\wedge\; h = \{\}$
- $E_1 = E_2$ 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

271

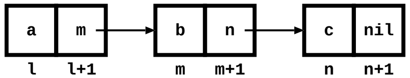
Summary of separation logic assertions (there are more)

- Points-to $E_1 \mapsto E_2$
  $E_1 \mapsto E_2 \;\Leftrightarrow\; \mathtt{dom}(h) = \{E_1(s)\} \;\wedge\; h(E_1(s)) = E_2(s)$
- Separating conjuction $P \star Q$
  $(P \star Q)(s,h) \;\Leftrightarrow\; \exists h_1\ h_2.\ h = h_1 \star h_2 \;\wedge\; P(s,h_1) \;\wedge\; Q(s,h_2)$
- Empty heap emp
  $\mathtt{emp}\ (s,h) \;\Leftrightarrow\; h = \{\}$ (where $\{\}$ is the empty heap)
- Abbreviation: $E \mapsto \_ \;=_{def}\; \exists X.\ E \mapsto X$ (where X does not occur in E)
- Abbreviation: $E \mapsto F_0, \ldots, F_n \;=_{def}\; (E \mapsto F_0) \star \cdots \star (E{+}n \mapsto F_n)$
- Abbreviation: $E_1 \doteq E_2 \;=_{def}\; (E_1 = E_2) \;\wedge\; \mathtt{emp}$

272

Example: reversing a linked list

- Diagram of list [a, b, c] stored in a linked-list data-structure



  - a is the contents of location l, m is the contents of location l+1
  - b is the contents of location m, n is the contents of location m+1
  - c then contents of location n, nil is the contents of location n+1

- Would like to specify

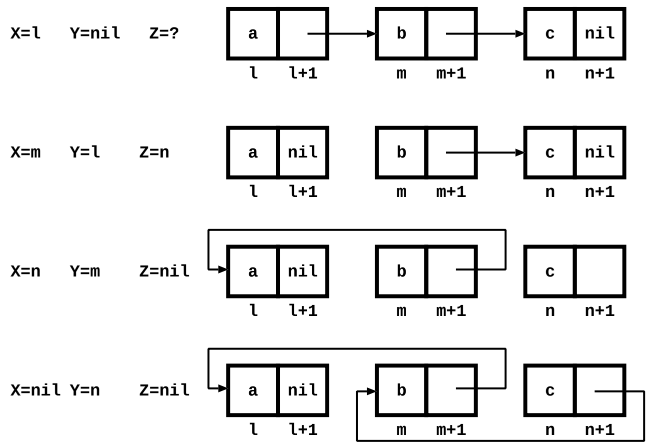
  {X *points to a linked list holding* x}
  Y:=nil;
  WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
  {Y *points to a linked list holding* rev(x)}

- Need to formalize "X points to a linked list holding x"

  - rev([$a_0, a_1, \ldots, a_{n-1}, a_n$]) = [$a_n, a_{n-1}, \ldots, a_1, a_0$]

Diagram illustrating linked list reverse operation

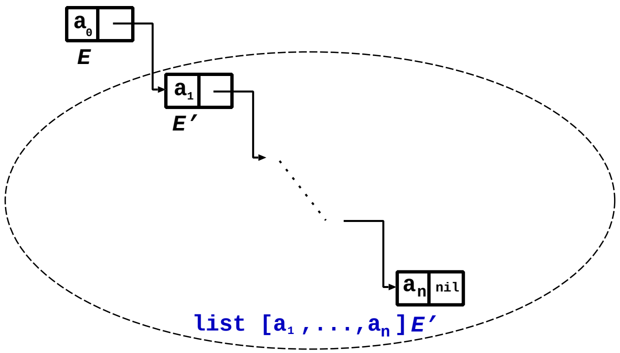
{X *points to a linked list holding* [a, b, c]}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{Y *points to a linked list holding* [c, b, a]}



Lists

- Assume nil $\in$ *Val* and [$a_1, \ldots, a_n$] $\in$ *Val* for $a_i \in$ *Val*
- Define list x E to mean x is stored as a linked list at location E:

  list [] $E \;\Leftrightarrow\; (E \doteq$ nil$)$
  list $([a_0, a_1, \ldots, a_n])$ $E \;\Leftrightarrow\; \exists E'.\ (E \mapsto a_0, E') \star$ list $[a_1, \ldots, a_n]$ E'



- Can then specify:

  {list x X}
  Y:=nil;
  WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
  {list (rev(x)) Y}

Separation logic: small axioms and faulting

- One might expect a heap assignment axiom to entail:

  $\vdash$ {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:

  $\vdash$ {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 $\vdash$ {T}[0]:=0{$0 \mapsto 0$} is not true

Semantics of commands (i)

- $C(s,h)(s',h')$ means executing C in state (s,h) can terminate in (s',h')
  - in the background reading: `Csem` C (s,h) (s',h')
- $C(s,h)$`fault` means executing C in state (s,h) can fault
  - in the background reading: `Csem` C (s,h) `fault`
- Sometimes $C(s,h)r$ where r (for "result") is a state or `fault`
- Semantics of store assignments (only store changed):
  $(V\texttt{:=}E)(s,h)r \;=\; (r = (s[(E(s))/V], h))$
- Semantics of fetch assignments (only store changed):
  $(V\texttt{:=[}E\texttt{]})(s,h)r \;=\; (r = \textbf{\textit{if}}\ E(s){\in}\texttt{dom}(h)\ \textbf{\textit{then}}(s[h(E(s))/V], h)\ \textbf{\textit{else}}\ \texttt{fault})$
- Semantics of heap assignments (only heap changed):
  $(\texttt{[}E_1\texttt{]:=}E_2)(s,h)r \;=\; (r = \textbf{\textit{if}}\ E_1(s){\in}\texttt{dom}(h)\ \textbf{\textit{then}}(s, h[E_2(s)/E_1(s)])\ \textbf{\textit{else}}\ \texttt{fault})$
- Semantics of pointer disposal (only heap changed):
  $(\texttt{dispose}(E))(s,h)r \;=\; (r = \textbf{\textit{if}}\ E(s) \in \texttt{dom}(h)\ \textbf{\textit{then}}(s, h\text{-}(E(s)))\ \textbf{\textit{else}}\ \texttt{fault})$

Semantics of commands (ii)

- Semantics of allocation assignments (store and heap changed):
  $$(V\texttt{:=cons}(E_1,\ldots,E_n))(s,h)r =$$
  $$\exists l.\ l \notin \texttt{dom}(h) \ \wedge\cdots\wedge\ l{+}(n{-}1) \notin \texttt{dom}(h)$$
  $$\wedge$$
  $$(r = (s[l/V],\ \ h[E_1(s)/l]\cdots[E_n(s)/l{+}(n{-}1)]))$$
- Non-deterministic:
  $(V\texttt{:=cons}(E_1,\ldots,E_n))(s,h)r$ is true for any result r
  for which the right hand side of the equation above holds.
- As the heap is finite, there will be infinitely many such results
- Never faults

Semantics of commands (iii)

- Semantics of sequences:
  $$(C_1;C_2)(s,h)r =$$
  $$\textbf{\textit{if}}\ (\exists s'\ h'.\ r = (s',h'))$$
  $$\textbf{\textit{then}}\ (\exists s'\ h'.\ C_1(s,h)(s',h') \wedge C_2(s',h')r)$$
  $$\textbf{\textit{else}}\ ((C_1(s,h)r \wedge (r = \texttt{fault}))$$
  $$\vee$$
  $$\exists s'\ h'.\ C_1(s,h)(s',h') \wedge C_2(s',h')r \wedge (r = \texttt{fault}))$$
- As in simple language, but propagate faults
  - if $C_1(s,h)$`fault` then $(C_1;C_2)(s,h)$`fault`
- Semantics of conditionals:
  $(\texttt{IF}\,S\,\texttt{THEN}\,C_1\,\texttt{ELSE}\,C_2)(s,h)r \;=\; \textbf{\textit{if}}\ \texttt{Ssem}\ S\ s\ \ \textbf{\textit{then}}\ \ C_1(s,h)r\ \ \textbf{\textit{else}}\ \ C_2(s,h)r$
  - S is a first-order logic statement (doesn't depend on heap), hence `Ssem` S s

Semantics of commands (iv)

- Semantics of while-commands:
  $(\texttt{WHILE}\,S\,\texttt{DO}\,C)(s,h)r \;=\; \exists n.\ \texttt{Iter}\ n\ (\texttt{Ssem}\ S)\ (\texttt{Csem}\ C)\ (s,h)\ r$
  where the recursive function `Iter` is redefined to handle faulting:
  $$\texttt{Iter}\ 0\ p\ c\ (s,h)\ r \;=\; \neg(p\ s) \wedge (r = (s,h))$$
  $$\texttt{Iter}\ (n{+}1)\ p\ c\ (s,h)\ r \;=$$
  $$p\ s \wedge (\textbf{\textit{if}}\ (\exists s'\ h'.\ r = (s'h'))$$
  $$\textbf{\textit{then}}\ (\exists s'\ h'.\ c(s,h)(s'h') \wedge \texttt{Iter}\ n\ p\ c\ (s',h')\ r)$$
  $$\textbf{\textit{else}}\ ((c(s,h)\,r \wedge (r = \texttt{fault}))$$
  $$\vee$$
  $$\exists s'\ h'.\ c\,(s,h)\,(s',h') \wedge \texttt{Iter}\ n\ p\ c\ (s',h')\ r \wedge (r = \texttt{fault})))$$
- Looks horrible ... but is just the obvious fault-propagating semantics
  - $\texttt{Iter} : \mathit{Num}{\to}(\mathit{Store}{\to}\mathit{Bool}){\to}(\mathit{State}{\to}\mathit{Result}{\to}\mathit{Bool}){\to}\mathit{State}{\to}\mathit{Result}{\to}\mathit{Bool}$

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

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

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

The frame rule

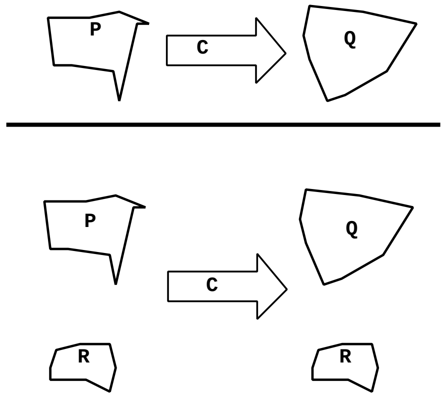
$$\frac{\vdash\ \{P\}\,C\,\{Q\}}{\vdash\ \{P\star R\}\,C\,\{Q\star R\}}$$

where no variable modified by C occurs free in R.

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

Small axioms

- A key idea of separation logic is to make the axioms *small*
- Precondition of $\{P\}C\{Q\}$ specifies smallest heap ensuring no fault
- Effects on bigger heaps derived from frame rule



Purely logical rules

- Following rules apply to both Hoare logic and separation logic

Rules of consequence

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

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

Exists introduction

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ \{\exists x.\ P\}\ C\ \{\exists x.\ Q\}}$$

where x does not occur in C

- For separation logic, need to think about faulting

Store assignment axiom

Store assignment axiom

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

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

$\vdash\ \{V{=}v\}\,V\texttt{:=}E\,\{V{=}E\texttt{[}v/V\texttt{]}\}$	store assign. ax.
$\vdash\ \{V{=}v\wedge Q\texttt{[}E\texttt{[}v/V\texttt{]}/V\texttt{]}\}\,V\texttt{:=}E\,\{V{=}E\texttt{[}v/V\texttt{]}\wedge Q\texttt{[}E\texttt{[}v/V\texttt{]}/V\texttt{]}\}$	rule of constancy
$\vdash\ \{\exists v.\ V{=}v\wedge Q\texttt{[}E\texttt{[}v/V\texttt{]}/V\texttt{]}\}\,V\texttt{:=}E\,\{\exists v.\ V{=}E\texttt{[}v/V\texttt{]}\wedge Q\texttt{[}E\texttt{[}v/V\texttt{]}/V\texttt{]}\}$	exists introduction
$\vdash\ \{\exists v.\ V{=}v\wedge Q\texttt{[}E\texttt{[}V/V\texttt{]}/V\texttt{]}\}\,V\texttt{:=}E\,\{\exists v.\ V{=}E\texttt{[}v/V\texttt{]}\wedge Q\texttt{[}V/V\texttt{]}\}$	predicate logic
$\vdash\ \{\exists v.\ V{=}v\wedge Q\texttt{[}E/V\texttt{]}\}\,V\texttt{:=}E\,\{\exists v.\ V{=}E\texttt{[}v/V\texttt{]}\wedge Q\}$	$\texttt{[}V/V\texttt{]}$ is identity
$\vdash\ \{(\exists v.\ V{=}v)\wedge Q\texttt{[}E/V\texttt{]}\}\,V\texttt{:=}E\,\{(\exists v.\ V{=}E\texttt{[}v/V\texttt{]})\wedge Q\}$	predicate logic: v not in E
$\vdash\ \{\texttt{T}\wedge Q\texttt{[}E/V\texttt{]}\}\,V\texttt{:=}E\,\{(\exists v.\ V{=}E\texttt{[}v/V\texttt{]})\wedge Q\}$	predicate logic
$\vdash\ \{Q\texttt{[}E/V\texttt{]}\}\,V\texttt{:=}E\,\{Q\}$	rules of consequence

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

Fetch assignment axiom

Fetch assignment axiom

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

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:**
  $\vdash \{E \mapsto v\}\, V\texttt{:=[}E\texttt{]}\, \{(V = v) \wedge E \mapsto v\}$
  (proof: instantiate v_1 **to** V **and** v_2 **to** v **and then simplify)**

Heap assignment axiom

Heap assignment axiom (heap mutation)

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

- **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

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

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

- **Never faults**
- **If** V **doesn't occur in** $E_1,\ldots,E_n$ **then:**

$\vdash \{V \doteq v\}\, V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}\, \{V \mapsto E_1[v/V], \ldots, E_n[v/V]\}$	alloc. assign. ax
$\vdash \{V \doteq v\}\, V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}\, \{V \mapsto E_1, \ldots, E_n\}$	V not in E_i assump.
$\vdash \{\exists v.\ V \doteq v\}\, V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}\, \{\exists v.\ V \mapsto E_1, \ldots, E_n\}$	exists intro.
$\vdash \{\exists v.\ V{=}v \wedge \texttt{emp}\}\, V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}\, \{\exists v.\ V \mapsto E_1, \ldots, E_n\}$	definition of $\doteq$
$\vdash \{\texttt{emp}\}\, V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}\, \{V \mapsto E_1, \ldots, E_n\}$	predicate logic

- **Which is a derivation of:**

Derived allocation assignment axiom

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

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

Pointer deallocation

Dispose axiom

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

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

$\vdash \{E_1 \mapsto \_\}\, \texttt{dispose(}E_1\texttt{)}\, \{\texttt{emp}\}$	dispose axiom
$\vdash \{\texttt{emp}\}\, V\texttt{:=cons(}E_2\texttt{)}\, \{V \mapsto E_2\}$	derived allocation assignment axiom
$\vdash \{E_1 \mapsto \_\}\, \texttt{dispose(}E_1\texttt{);}V\texttt{:=cons(}E_2\texttt{)}\, \{V \mapsto E_2\}$	sequencing rule

Summary of pointer manipulating axioms

Store assignment axiom

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

where v is an auxiliary variable not occurring in E.

Fetch assignment axiom

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

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

Heap assignment axiom

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

Allocation assignment axiom

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

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

Dispose axiom

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

Compound command rules

- **Following rules apply to both Hoare logic and separation logic**

The sequencing rule

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

The conditional rule

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

The WHILE-rule

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

- **For separation logic, need to think about faulting**

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.

- **Not valid for heap assignments**

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

  but not

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

  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**

$\{$*contents of pointers* X *and* Y *are equal*$\}$ X:=[X]; Y:=[Y] $\{\texttt{X} = \texttt{Y}\}$

- **Proof:**

$\vdash \{(\texttt{X} = x) \wedge \texttt{X} \mapsto v\}\ \texttt{X:=[X]}\ \{(\texttt{X} = v) \wedge x \mapsto v\}$ — fetch assignment axiom

$\vdash \{(\texttt{Y} = y) \wedge \texttt{Y} \mapsto v\}\ \texttt{Y:=[Y]}\ \{(\texttt{Y} = v) \wedge y \mapsto v\}$ — fetch assignment axiom

$\vdash \{((\texttt{X} = x) \wedge \texttt{X} \mapsto v) \star ((\texttt{Y} = y) \wedge \texttt{Y} \mapsto v)\}$ — frame rule
$\quad \texttt{X:=[X]}$
$\quad \{((\texttt{X} = v) \wedge x \mapsto v) \star (((\texttt{Y} = y) \wedge \texttt{Y} \mapsto v))\}$

$\vdash \{((\texttt{Y} = y) \wedge \texttt{Y} \mapsto v) \star ((\texttt{X} = v) \wedge x \mapsto v)\}$ — frame rule
$\quad \texttt{Y:=[Y]}$
$\quad \{((\texttt{Y} = v) \wedge y \mapsto v) \star ((\texttt{X} = v) \wedge x \mapsto v)\}$

$\vdash \{((\texttt{X} = x) \wedge \texttt{X} \mapsto v) \star ((\texttt{Y} = y) \wedge \texttt{Y} \mapsto v)\}$ — sequencing rule and commutativity of $\star$
$\quad \texttt{X:=[X];Y:=[Y]}$
$\quad \{((\texttt{X} = v) \wedge x \mapsto v) \star ((\texttt{Y} = v) \wedge y \mapsto v)\}$

$\vdash \{\exists v\ x\ y.\ ((\texttt{X} = x) \wedge \texttt{X} \mapsto v) \star ((\texttt{Y} = y) \wedge \texttt{Y} \mapsto v)\}$ — exists-introduction (3 times)
$\quad \texttt{X:=[X];Y:=[Y]}$
$\quad \{\exists v\ x\ y.\ ((\texttt{X} = v) \wedge x \mapsto v) \star ((\texttt{Y} = v) \wedge y \mapsto v)\}$

$\vdash\ \{\exists v.\ \texttt{X} \mapsto v \star \texttt{Y} \mapsto v\}\ \texttt{X:=[X]; Y:=[Y]}\ \{\texttt{X} = \texttt{Y}\}$ — rules of consequence (see next slide)

Logic of separating assertions, soundness, completeness

- To use separation logic various properties of $\star$, $\mapsto$ etc. are needed
- For rule of consequence in proof on preceding slide need:

  $(\exists v.\ \mathtt{X}\mapsto v \star \mathtt{Y}\mapsto v) \ \Rightarrow\ \exists v\ x\ y.\ ((\mathtt{X}=x)\wedge \mathtt{X}\mapsto v)\star((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)$

  $(\exists v\ x\ y.\ ((\mathtt{X}=v)\wedge x\mapsto v)\star((\mathtt{Y}=v)\wedge y\mapsto v)) \ \Rightarrow\ (\mathtt{X}=\mathtt{Y})$

- No complete deductive system exists – not a problem in practice
- Using separation logic like ordinary Hoare logic, but more fiddly
- Proof of linked list example given in John Wickerson's slides:

```
{list x X}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{list (rev(x)) Y}
```

- Separation logic is sound and relatively complete
  - similar proof using appropriate generalisation of `wlp`
  - faulting adds complications

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.:

```
{∃x. list x X}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{∃x. list x Y}
```

    automatically finds memory usage errors

- Finally, something to think about:
  should we be verifying code in old fashioned languages (pramatism)
  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]