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. 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. 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.

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

1Some 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).

1Hoare Logic is sometimes called Floyd-Hoare Logic, due to the important contributions of Floyd to the underlying ideas.
<table>
<thead>
<tr>
<th><strong>Topic</strong></th>
<th><strong>Slides</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1 - 5</td>
</tr>
<tr>
<td>Hoare’s notation</td>
<td>7 - 11</td>
</tr>
<tr>
<td>Formal proofs</td>
<td>12 - 16</td>
</tr>
<tr>
<td>Partial and total correctness</td>
<td>17 - 19</td>
</tr>
<tr>
<td>Example specification</td>
<td>20 - 25</td>
</tr>
<tr>
<td>Review of predicate calculus</td>
<td>26 - 34</td>
</tr>
<tr>
<td>Floyd-Hoare Logic</td>
<td>35 - 59</td>
</tr>
<tr>
<td>How does one find an invariant</td>
<td>60 - 62</td>
</tr>
<tr>
<td>Additional and derived rules</td>
<td>63 - 72</td>
</tr>
<tr>
<td>Forwards and backwards proof</td>
<td>73 - 79</td>
</tr>
<tr>
<td>Annotations</td>
<td>80 - 82, 92 - 93</td>
</tr>
<tr>
<td>Mechanizing verification</td>
<td>83 - 86</td>
</tr>
<tr>
<td>Verification conditions</td>
<td>87 - 112</td>
</tr>
<tr>
<td>Weakest preconditions</td>
<td>113 121</td>
</tr>
<tr>
<td>Strongest postconditions</td>
<td>122 - 133</td>
</tr>
<tr>
<td>Total correctness</td>
<td>134 - 154</td>
</tr>
<tr>
<td>Soundness and completeness</td>
<td>155 - 179</td>
</tr>
<tr>
<td>Overview of additional topics</td>
<td>180 - 181</td>
</tr>
<tr>
<td>Arrays</td>
<td>182 - 190</td>
</tr>
<tr>
<td>Blocks and local variables</td>
<td>191 - 194</td>
</tr>
<tr>
<td><strong>FOR</strong>-commands</td>
<td>196 - 213</td>
</tr>
<tr>
<td>Axiomatic semantics and Clarke’s theorem</td>
<td>214 - 215</td>
</tr>
<tr>
<td>Refinement</td>
<td>216 - 251</td>
</tr>
<tr>
<td>Separation logic</td>
<td>252 - 293</td>
</tr>
<tr>
<td>Conclusion</td>
<td>294</td>
</tr>
</tbody>
</table>
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:

\[ B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots \]

Commands:

\[ C ::= V := E \]
\[ \mid C_1 ; C_2 \]
\[ \mid \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \]
\[ \mid \text{WHILE } B \text{ DO } C \]

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

<table>
<thead>
<tr>
<th>Acceptable Initial State</th>
<th>Acceptable Final State</th>
</tr>
</thead>
<tbody>
<tr>
<td>“X is greater than zero”</td>
<td>“Y is the square root of X”</td>
</tr>
</tbody>
</table>

**Action of the Program**

| 8 |

## Meaning of Hoare’s Notation

- \( \{P\} \ C \ {Q}\) is true if
  - whenever \( C \) is executed in a state satisfying \( P \)
  - and \( \) if \( \) the execution of \( C \) terminates
  - then the state in which \( C \) terminates satisfies \( Q \)

- Example: \( \{X = 1\} \ X:=X+1 \ \{X = 2\}\)
  - \( P \) is the condition that the value of \( X \) is 1
  - \( Q \) is the condition that the value of \( X \) is 2
  - \( C \) is the assignment command \( X:=X+1 \)
    * i.e. ‘\( X \) becomes \( X+1 \)’

- \( \{X = 1\} \ X:=X+1 \ \{X = 2\} \) is true

- \( \{X = 1\} \ X:=X+1 \ \{X = 3\} \) is false

## Formal versus Informal Proof

- Mathematics text books give informal proofs
- English arguments are used
  - proof of \( (X + 1)^2 = X^2 + 2 \times X + 1 \)
    - “follows by the definition of squaring and distributivity laws”

- Formal verification uses formal proof
  - the rules used are described and followed very precisely
  - formal proof has been used to discover errors in published informal ones

- Here is an example formal proof

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

- A proof consists of a sequence of lines
- Each line is an instance of an **axiom**
  - like the definition of \( |p|^2 \)
- or follows from previous lines by a **rule of inference**
  - like the substitution of equals for equals
- The statement occurring on the last line of a proof is the statement **proved** by it
  - thus \( (x+1)^2 = x^2 + 2x + 1 \) 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

Hoare Logic and Verification Conditions

- Hoare Logic is a deductive proof system for **Hoare triples** \( \{P\} \xrightarrow{C} \{Q\} \)
- Can use Hoare Logic directly to verify programs
  - original proposal by Hoare
  - tedious and error prone
  - impractical for large programs
- Can ‘compile’ proving \( \{P\} \xrightarrow{C} \{Q\} \) 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

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.”
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

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

Auxiliary Variables

- \( \{X=x \land Y=y\} R:=X; X:=Y; Y:=R \{X=y \land Y=x\} \)
  - this says that if the execution of
    \[ R:=X; X:=Y; Y:=R \]
  - terminates (which it does)
  - then the values of \( X \) and \( Y \) are exchanged
- The variables \( x \) and \( y \), which don’t occur in the command and are used to name the initial values of program variables \( X \) and \( Y \)
- They are called auxiliary variables or ghost variables
- Informal convention:
  - program variable are upper case
  - auxiliary variable are lower case
More simple examples

- \{x=x \land y=y\} \times := y; y:=x \{x=x \land y=x\}
  - this says that \times := y; y:=x exchanges the values of x and y
  - this is not true

- \{T\} C \{Q\}
  - this says that whenever C halts, Q holds

- \{P\} C \{T\}
  - this specification is true for every condition P and every command C
  - because T is always true

- \{P\} C \{T\}
  - this says that C terminates if initially P holds
  - it says nothing about the final state

- \{T\} C \{P\}
  - this says that C always terminates and ends in a state where P holds

Some Easy Exercises

- When is \{T\} C \{T\} true?

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

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

A More Complicated Example

- \{T\}
  R:=x;
  Q:=0;
  WHILE Y\leq R DO
    (R:=R-Y; Q:=Q+1)
  \{R < Y \land X = R + (Y \times Q)\}

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

Specification can be Tricky

- “The program must set Y to the maximum of X and Y”
  - \{T\} C \{Y = \max(X,Y)\}

  - A suitable program:
    - IF X \geq Y THEN Y := X ELSE X := X

  - Another?
    - IF X \geq 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
  \[ \vdash \{ T \} \subseteq \{ \text{max}(X,Y) \} \]

- The correct formalisation of what was intended is probably
  \[ \vdash \{ X=x \land Y=y \} \subseteq \{ \text{max}(x,y) \} \]

- 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, z, etc

- We use conventional notation, e.g. here are some terms:
  \[ X, y, Z, 1, 2, 325, \]
  \[ -X, -(X+1), (x \times y) + Z, \]
  \[ \sqrt{1+x^2}, \; x!, \; \sin(x), \; \text{rem}(X,Y) \]

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

Atomic Statements

- Examples of atomic statements are
  \[ T, \; F, \; X = 1, \; R < Y, \; X = R+(Y \times Q) \]

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

- Other atomic statements are built from terms using predicates, e.g.
  \[ \text{ODD}(X), \; \text{PRIME}(3), \; X = 1, \; (X+1)^2 \geq x^2 \]

- ODD and PRIME are examples of predicates

- = and ≥ are examples of infixed predicates

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

- Compound statements are built up from atomic statements using:
  - \( \neg \) (not)
  - \( \wedge \) (and)
  - \( \vee \) (or)
  - \( \Rightarrow \) (implies)
  - \( \iff \) (if and only if)
  - The single arrow \( \rightarrow \) is commonly used for implication instead of \( \Rightarrow \)

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

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 \( \iff \)

- For example
  - \( \neg P \wedge Q \) is equivalent to \( (\neg P) \wedge Q \)
  - \( P \wedge Q \Rightarrow R \) is equivalent to \( (P \wedge Q) \Rightarrow R \)
  - \( P \wedge Q \iff \neg R \vee S \) is equivalent to \( (P \wedge Q) \iff ((\neg R) \vee S) \)

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”
- \( P \iff Q \) is equivalent to \( (P \Rightarrow Q) \wedge (Q \Rightarrow P) \)
- Sometimes write \( P = Q \) or \( P \equiv Q \) for \( P \iff Q \)

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
  \[
  \vdash S_1, \ldots, \vdash S_n \\
  \vdash S
  \]
  - 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

\[ \vdash \{ P \} C_1 \{ Q \}, \quad \vdash \{ Q \} C_2 \{ R \} \]
\[ \vdash \{ P \} C_1 ; C_2 \{ R \} \]

- 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
  \[ \vdash \{ x=1 \} x + (x + 1) = 3 \]
  - VCs are handed to a theorem prover
  - “Extended Static Checking” (ESC) is an industrial example

Judgements

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

Reminder of our little programming language

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

Expressions

\[ E ::= \text{N} \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots \]

Boolean expressions

\[ B ::= \text{T} \mid \text{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots \]

Commands

\[ C ::= \text{V} := E \quad \text{Assignments} \]
\[ | \quad C_1 ; C_2 \quad \text{Sequences} \]
\[ | \quad \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \quad \text{Conditionals} \]
\[ | \quad \text{WHILE } B \text{ DO } C \quad \text{WHILE-commands} \]

Syntactic Conventions

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

- $Q[E/V]$ is the result of replacing all occurrences of $V$ in $Q$ by $E$
  - read $Q[E/V]$ as ‘$Q$ with $E$ for $V$’
  - for example: $(x+1) > x)[y+z/x] = ((y+z)+1 > y+z)
  - ignoring issues with bound variables for now (e.g. variable capture)
- Same notation for substituting into terms, e.g. $E_1[E_2/V]$
- Think of this notation as the ‘cancellation law’
  $V[E/V] = E$
  which is analogous to the cancellation property of fractions
  $v \times (e/v) = e$
- Note that $Q[x/V]$ doesn’t contain $V$ (if $V \neq x$)

The Assignment Axiom (Hoare)

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

$$\vdash \{P\} V := E \{∃v. V = E[v/V] \land P[v/V]\}$$
- where $v$ is a new variable (i.e. doesn’t equal $V$ or occur in $P$ or $E$

Example instance
$$\vdash \{X=1\} X := X + 1 \{∃v. X = X + 1[v/X] \land X = 1[v/X]\}$$

- Simplifying the postcondition
  $$\vdash \{X=1\} X := X + 1 \{∃v. X = X + 1[v/X] \land X = 1[v/X]\}$$
  $$\vdash \{X=1\} X := X + 1 \{∃v. X = X + 1 \land v = 1\}$$
  $$\vdash \{X=1\} X := X + 1 \{∃v. X = 1 + 1 \land v = 1\}$$
  $$\vdash \{X=1\} X := X + 1 \{X = 1 + 1 \land ∃v. v = 1\}$$
  $$\vdash \{X=1\} X := X + 1 \{X = 2 \land T\}$$
  $$\vdash \{X=1\} X := X + 1 \{X = 2\}$$
- Forwards Axiom equivalent to standard one but harder to use

A Forwards Assignment Axiom (Floyd)

- This is the original semantics of assignment due to Floyd
  $$\vdash \{P\} V := E \{∃v. V = E[v/V] \land P[v/V]\}$$
  - where $v$ is a new variable (i.e. doesn’t equal $V$ or occur in $P$ or $E$

- Example instance
  $$\vdash \{X=1\} X := X + 1 \{∃v. X = X + 1[v/X] \land X = 1[v/X]\}$$

The Backwards Fallacy

- Many people feel the assignment axiom is ‘backwards’
- One common erroneous intuition is that it should be
  $$\vdash \{P\} V := E \{P[V/E]\}$$
  - where $P[V/E]$ denotes the result of substituting $V$ for $E$ in $P$
  - this has the false consequence $\vdash \{X=0\} X := 1 \{X=0\}$
  (since $(X=0)[X/1]$ is equal to $(X=0)$ as $1$ doesn’t occur in $(X=0)$)
- Another erroneous intuition is that it should be
  $$\vdash \{P\} V := E \{P[E/V]\}$$
  - this has the false consequence $\vdash \{X=0\} X := 1 \{X=0\}$
  (which follows by taking $P$ to be $X=0$, $V$ to be $X$ and $E$ to be $1$)
### Precondition Strengthening

- Recall that
  \[ \vdash S_1, \ldots, \vdash S_n \]
  \[ \vdash S \]
  means \( \vdash S \) can be deduced from \( \vdash S_1, \ldots, \vdash S_n \).

- Using this notation, the rule of precondition strengthening is

  \[
  \vdash P \Rightarrow P', \vdash \{ P' \} C \{ Q \}, \vdash \{ Q \}
  \]

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

- Note the two hypotheses are different kinds of judgements.

### Postcondition Weakening

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

  \[
  \vdash \{ P \} C \{ Q \}, \vdash Q' \Rightarrow Q
  \]

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

### Example

- From
  \[
  \vdash X=n \Rightarrow X+1=n+1
  \]
  \[
  \text{trivial arithmetical fact}
  \]

- \[
  \vdash \{ X + 1 = n + 1 \} X := X + 1 \{ X = n + 1 \}
  \]
  \[
  \text{from earlier slide}
  \]

- It follows by precondition strengthening that

  \[
  \vdash \{ X = n \} \ X := X + 1 \{ X = n + 1 \}
  \]

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

### 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

  ⊢ \{ Y=0 \} X:=BEGIN Y:=1; 2 END \{ Y=0 \}

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

  • this is clearly false; after the assignment \( Y \) will have the value 1

The sequencing rule

• Syntax: \( C_1; \ldots; C_n \)

• Semantics: the commands \( C_1, \ldots, C_n \) 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

  \[
  \frac{\vdash \{ P \} C_1 \{ Q \}, \vdash \{ Q \} C_2 \{ R \}}{\vdash \{ P \} C_1;C_2 \{ R \}}
  \]

An Example Formal Proof

• Here is a little formal proof

1. \[ \vdash [R\times X \land 0=0] \quad Q:=0 \quad [R\times X \land Q=0] \] By the assignment axiom
2. \[ R=X \quad \Rightarrow \quad R=X \land Q=0 \] By pure logic
3. \[ \vdash [R\times X \land Q=0] \] By precondition strengthening
4. \[ R=X \land Q=0 \quad \Rightarrow \quad R=X\times(Y \times Q) \] By laws of arithmetic
5. \[ \vdash [R\times X \land Q=0] \quad [R\times X\times(Y \times Q)] \] By postcondition weakening

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

Example Proof

Example: By the assignment axiom:

(i) \[ \vdash \{ X=x \land Y=y \} \quad R:=X \quad \{ R=x \land Y=y \} \]
(ii) \[ \vdash \{ R=x \land Y=y \} \quad X:=Y \quad \{ R=x \land X=Y \} \]
(iii) \[ \vdash \{ R=x \land X=y \} \quad Y:=R \quad \{ Y=x \land X=y \} \]

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

(iv) \[ \vdash \{ X=x \land Y=y \} \quad R:=X; \quad X:=Y \quad \{ R=x \land X=Y \} \]

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

(v) \[ \vdash \{ X=x \land Y=y \} \quad R:=X; \quad X:=Y; \quad Y:=R \quad \{ Y=x \land X=y \} \]
**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 $\text{MAX} := Y$ ELSE $\text{MAX} := X$
  - the value of the variable $\text{MAX}$ it set to the maximum of the values of $X$ and $Y$

**WHILE-commands**

- **Syntax:** WHILE $S$ DO $C$
- **Semantics:**
  - if the statement $S$ is true in the current state, then $C$ is executed and the WHILE-command is repeated
  - if $S$ is false, then nothing is done
  - thus $C$ is repeatedly executed until the value of $S$ becomes false
  - if $S$ never becomes false, then the execution of the command never terminates
- **Example:** WHILE $\neg(X = 0)$ DO $X := X - 2$
  - if the value of $X$ is non-zero, then its value is decreased by 2 and then the process is repeated
- **Invariants**

- **Syntax:** IN $P$ AND SAYS $Q$
- **Semantics:**
  - $P$ is an invariant of $Q$ whenever $S$ holds
- **WHILE-rule:**
  - 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 $P$ and 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 Conditional Rule**

- The conditional rule
  \begin{align*}
  \vdash & \{P \land S\} C_1 \{Q\}, \quad \vdash \{P \land \neg S\} C_2 \{Q\} \\
  \therefore & \vdash \{P\} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \{Q\}
  \end{align*}

- **From Assignment Axiom + Precondition Strengthening and**
  \begin{align*}
  \vdash & \left\{ X \geq Y \Rightarrow X = \max(X, Y) \right\} \land \left\{ \neg(X \geq Y) \Rightarrow Y = \max(X, Y) \right\} \\
  \text{it follows that} \\
  \vdash & \left\{ T \land X \geq Y \right\} \text{ MAX} := X \left\{ \text{MAX=} \max(X, Y) \right\} \\
  \text{and} \\
  \vdash & \left\{ T \land \neg(X \geq Y) \right\} \text{ MAX} := Y \left\{ \text{MAX=} \max(X, Y) \right\}
  \end{align*}

- **Then by the conditional rule it follows that**
  \begin{align*}
  \vdash & \{T\} \text{ IF } X \geq Y \text{ THEN } \text{MAX} := X \text{ ELSE } \text{MAX} := Y \left\{ \text{MAX=} \max(X, Y) \right\}
  \end{align*}
**The WHILE-Rule**

**The WHILE-rule**

\[ \begin{array}{c}
\vdash \{ P \land S \} \rightarrow C \{ P \} \\
\vdash \{ P \} \rightarrow \text{WHILE } S \text{ DO } C \{ P \land \neg S \}
\end{array} \]

- It is easy to show
  \[ \vdash \{ X=R+(Y\times Q) \land Y \leq R \} \rightarrow (R:=R-Y; Q:=Q+1) \{ X=R+(Y\times Q) \} \]
- Hence by the WHILE-rule with \( P = 'X=R+(Y\times Q)' \) and \( S = 'Y\leq R' \)
  \[ \vdash \{ X=R+(Y\times Q) \} \rightarrow \text{WHILE } Y \leq R \text{ DO } (R:=R-Y; Q:=Q+1) \{ X=R+(Y\times Q) \land \neg(Y\leq R) \} \]

**Example**

- From the previous slide
  \[ \vdash \{ X=R+(Y\times Q) \} \rightarrow \text{WHILE } Y \leq R \text{ DO } (R:=R-Y; Q:=Q+1) \{ X=R+(Y\times Q) \land \neg(Y\leq R) \} \]

- It is easy to deduce that
  \[ \vdash \{ T \} \rightarrow (R:=X; Q:=0) \{ X=R+(Y\times Q) \} \]
- Hence by the sequencing rule and postcondition weakening
  \[ \vdash \{ T \} \rightarrow R:=X; Q:=0; \text{WHILE } Y \leq R \text{ DO } (R:=R-Y; Q:=Q+1) \{ R<Y \land X=R+(Y\times Q) \} \]

**Summary**

- We have given:
  - a notation for specifying what a program does
  - a way of proving that it meets its specification

- Now we look at ways of finding proofs and organising them:
  - finding invariants
  - derived rules
  - backwards proofs
  - annotating programs prior to proof

- Then we see how to automate program verification
  - the automation mechanises some of these ideas

**How does one find an invariant?**

- 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

\[
\begin{align*}
\{&X=n \land Y=1\} \\
\text{WHILE } &X\neq 0 \text{ DO} \\
&Y:=Y\times X; \ X:=X-1 \\
\{&X=0 \land Y=n!\}
\end{align*}
\]

- Look at the facts
  - initially \(X=n\) and \(Y=1\)
  - finally \(X=0\) and \(Y=n!\)
  - on each loop \(Y\) is increased and, \(X\) is decreased

- Think how the loop works
  - \(Y\) holds the result so far
  - \(X!\) is what remains to be computed
  - \(n!\) is the desired result

- The invariant is \(X!\times Y = n!\)
  - ‘stuff to be done’ \(\times\) ‘result so far’ = ‘desired result’
  - decrease in \(X\) combines with increase in \(Y\) to make invariant

Related example

- \(\{X=0 \land Y=1\}\)
- \(\text{WHILE } X<N \text{ DO} (X:=X+1; \ Y:=Y\times X)\)
- \(\{Y=N!\}\)

- Look at the Facts
  - initially \(X=0\) and \(Y=1\)
  - finally \(X=N\) and \(Y=N!\)
  - on each iteration both \(X\) an \(Y\) increase: \(X\) by 1 and \(Y\) by \(X\)

- An invariant is \(Y = X!\)

- At end need \(Y = N!\), but WHILE-rule only gives \(\neg(X<N)\)

- Ah Ha! Invariant needed: \(Y = X! \land X\leq N\)

- 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 \(X\leq N\)

Conjunction and Disjunction

<table>
<thead>
<tr>
<th>Specification conjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash {P_1} \land {Q_1} \vdash {P_2} \land {Q_2} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specification disjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash {P_1} \lor {Q_1}, \vdash {P_2} \lor {Q_2} )</td>
</tr>
</tbody>
</table>

- These rules are useful for splitting a proof into independent bits
  - they enable \( \vdash \{P\} \land \{Q, \land Q_2\} \) to be proved by proving separately that both \( \vdash \{P\} \land \{Q_1\} \) and also that \( \vdash \{P\} \land \{Q_2\} \)

- 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
    - \(\vdash \{T\} \Rightarrow X \leftarrow (R \land X)\)
  - Rather than:
    - by the assignment axiom
      - \(\vdash \{X \land X\} \Rightarrow X \leftarrow (R \land X)\)
    - by precondition strengthening with \(\vdash T \Rightarrow X \land X\)
      - \(\vdash \{T\} \Rightarrow X \land X\)
Derived rules for finding proofs

- Suppose the goal is to prove \( \{ \text{Precondition} \} \text{ Command} \{ \text{Postcondition} \} \)
- If there were a rule of the form
  \[
  \vdash H_1, \ldots, H_n
  \]
  \[
  \vdash \{ P \} C \{ Q \}
  \]
  then we could instantiate
  \[
  P \mapsto \text{Precondition}, C \mapsto \text{Command}, Q \mapsto \text{Postcondition}
  \]
to get instances of \( H_1, \ldots, H_n \) 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

The Derived Assignment Rule

- An example proof
  1. \( \vdash \{ R=X \land 0=0 \} \ Q:=0 \ \{ R=X \land Q=0 \} \) By the assignment axiom.
  2. \( \vdash R=X \Rightarrow R=X \land 0=0 \) By pure logic.
  3. \( \vdash \{ R=X \} \ Q:=0 \ \{ R=X \land Q=0 \} \) By derived assignment.

- This proof schema justifies:

```
\[ \vdash \{ P \} V:=E \ {Q} \]
```

- Note: \( Q[E/V] \) is the **weakest liberal precondition** \( wlp[V:=E, Q] \)
- Example proof above can now be done in one less step
  1. \( \vdash R=X \Rightarrow R=X \land 0=0 \) By pure logic.
  2. \( \vdash \{ R=X \} \ Q:=0 \ \{ R=X \land 0=0 \} \) By derived assignment.

Derived Rules

- We will establish derived rules for all commands

```
\[ \vdash \{ P \} V:=E \ {Q} \]
\[ \vdash \{ P \} C_1;C_2 \ {Q} \]
\[ \vdash \{ P \} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \ {Q} \]
\[ \vdash \{ P \} \text{ WHILE } S \text{ DO } C \ {Q} \]
```

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

Derived Sequenced Assignment Rule

- The following rule will be useful later

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

- Intuitively work backwards:
  - push \( Q \) 'through' \( V:=E \), changing it to \( Q[E/V] \)
- Example: By the assignment axiom:
  \[
  \vdash \{ X:=x \land Y=y \} \ R:=X \ \{ R=x \land Y=y \}
  \]
- Hence by the sequenced assignment rule
  \[
  \vdash \{ X:=x \land Y=y \} \ R:=X; \ X:=Y \ \{ R=x \land X=y \}
  \]  

The Derived Sequencing Rule

- The rule below follows from the sequencing and consequence rules

\[
\begin{align*}
\text{The Derived Sequencing Rule} \\
\vdash P & \Rightarrow P_1 \\
\vdash \{P_1\} C_1 \{Q_1\} & \Rightarrow Q_1 \Rightarrow P_2 \\
\vdash \{P_2\} C_2 \{Q_2\} & \Rightarrow Q_2 \Rightarrow P_3 \\
\vdots & \vdots \\
\vdash \{P_n\} C_n \{Q_n\} & \Rightarrow Q_n \Rightarrow Q \\
\vdash \{P\} C_1; \ldots ; C_n \{Q\}
\end{align*}
\]

- Exercise: why no derived conditional rule?

Backward Hoare & forward Floyd assignment axioms

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

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

  Floyd axiom: \(\vdash \{P\} V := E \{E[V/V] \land P[v/V]\}\)

- Exercise 1 (easy): derive forward axiom from Hoare axiom
  
  - hint: \(P \Rightarrow \exists v. E = E[v/V] \land P[v/V]\)

- Exercise 2 (a bit harder): derive Hoare axiom from forward axiom
  
  - hint: if \(v\) is a new variable then \(P[E/V] = P[E[v/V]/V]\)

- Exercise 3: devise and justify a derived assignment rule based on the Floyd assignment axiom

The Derived While Rule

- This follows from the While Rule and the rules of consequence

\[
\begin{align*}
\text{Derived While Rule} \\
\vdash P \Rightarrow R & \vdash \{R \land S\} C \{R\} \vdash R \land \neg S \Rightarrow Q \\
\vdash \{P\} \text{WHILE } S \text{ DO } C \{Q\}
\end{align*}
\]

- Example: it is easy to show

  \[
  \begin{align*}
  &\vdash \{X=x \land Y=y\} R:=X \{R=x \land Y=y\} \\
  &\vdash \{R=x \land Y=y\} X:=Y \{R=x \land X=y\} \\
  &\vdash \{R=x \land X=y\} Y:=R \{Y=x \land X=y\}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \vdash \{X=x \land Y=y\} R:=X; X:=Y; Y:=R \{Y=x \land X=y\}
  \end{align*}
  \]

Example

- By the assignment axiom

  \[
  \begin{align*}
  (i) &\vdash \{X=x \land Y=y\} R:=X \{R=x \land Y=y\} \\
  (ii) &\vdash \{R=x \land Y=y\} X:=Y \{R=x \land X=y\} \\
  (iii) &\vdash \{R=x \land X=y\} Y:=R \{Y=x \land X=y\}
  \end{align*}
  \]
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
  \(\{X=x \land Y=y\} \ R:=X; \ X:=Y; \ Y:=R \ \{Y=x \land X=y\}\)
- By the assignment axiom and derived sequenced assignment rule it is sufficient to show the subgoal
  \(\{X=x \land Y=y\} \ R:=X; \ X:=Y \ \{R=x \land X=y\}\)
- Similarly this subgoal can be reduced to
  \(\{X=x \land Y=y\} \ R:=X \ \{R=x \land Y=y\}\)
- This clearly follows from the assignment axiom

Example Backwards Proof

- To prove
  \(\vdash \{T\}\)
  \(R:=X;\)
  \(Q:=0;\)
  \(WHILE \ Y \leq R \ DO\)
  \(\ (R:=R-Y; \ Q:=Q+1)\)
  \(\{X=R+(Y \times Q) \land R<Y\}\)
- By the sequencing rule, it is sufficient to prove
  \[(i) \ \vdash \{T\} \ R:=X; \ Q:=0 \ \{R=X \land Q=0\}\)
  \[(ii) \ \vdash \{R=X \land Q=0\}\)
  \(WHILE \ Y \leq R \ DO\)
  \(\ (R:=R-Y; \ Q:=Q+1)\)
  \(\{X=R+(Y \times Q) \land R<Y\}\)
- Where does \{R=X \land Q=0\} come from? (Answer later)
Example Continued (1)

- From previous slide:
  (i) ⊢ {T} R:=X; Q:=0 {R=X ∧ Q=0}

- To prove (i), by the sequenced assignment axiom, we must prove:
  (iii) ⊢ {T} R:=X {R=X ∧ 0=0}

- To prove (iii), by the derived assignment rule, we must prove:
  ⊢ T ⇒ X=X ∧ 0=0

  This is true by pure logic

Example Continued (2)

- From an earlier slide:
  (ii) ⊢ {R=X ∧ Q=0}
  WHILE Y≤R DO
  (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) (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+(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 $R$

  ⊢ {P} $C_1$ {R}
  ⊢ {R} $C_2$ {Q}

  ⊢ {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

\[
\{ T \} \\
R := X; \\
Q := 0; \{ R = X \land Q = 0 \} \leftarrow P_1 \\
\text{WHILE } Y \leq R \text{ DO} \{ X = R + Y \times Q \} \leftarrow P_2 \\
(R := R - Y; Q := Q + 1) \\
\{ X = R + Y \times Q \land R < Y \} \\
\]

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.
  \[
  \vdash (R = X \land Q = 0) \Rightarrow (X = R + Y \times Q) \\
  \]

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

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

Verification conditions

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

Validity of Verification Conditions

- It will be shown that
  - if one can prove all the verification conditions generated from \( \{P\}C\{Q\} \)
    - then \( \vdash \{P\}C\{Q\} \)
  - Step 2 converts a verification problem into a conventional mathematical problem
- The process will be illustrated with:
  \[
  \{T\}
  R:=X;
  Q:=0;
  \text{WHILE } Y\leq R \text{ DO}
  \begin{align*}
  & (R:=R-Y; Q:=Q+1) \\
  \end{align*}
  \{X = R+Y\times Q \land R\leq Y\}

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
Example

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

\[
\begin{align*}
\{ T \} \\
R &:= X; \\
Q &:= 0; \{ R=X \land Q=0 \} \leftarrow P_1 \\
\text{WHILE } Y \leq R \text{ DO } \{ X = R+Y \times Q \} \leftarrow P_2 \\
(R &:= R-Y; Q &:= Q+1) \\
\{ X = R+Y \times Q \land R<Y \}
\end{align*}
\]

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

Example Continued

\[
\begin{align*}
\{ T \} \\
R &:= X; \\
Q &:= 0; \{ R=X \land Q=0 \} \leftarrow P_1 \\
\text{WHILE } Y \leq R \text{ DO } \{ X = R+Y \times Q \} \leftarrow P_2 \\
(R &:= R-Y; Q &:= Q+1) \\
\{ X = R+Y \times Q \land R<Y \}
\end{align*}
\]

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

Generating and Proving Verification Conditions

- Step 2 will generate the following four verification conditions

\[
\begin{align*}
(i) & \quad T \Rightarrow (X=X \land 0=0) \\
(ii) & \quad (R=X \land Q=0) \Rightarrow (X = R+(Y \times Q)) \\
(iii) & \quad (X = R+(Y \times Q)) \land Y \leq R \Rightarrow (X = (R-Y)+(Y \times (Q+1))) \\
(iv) & \quad (X = R+(Y \times Q)) \land \neg(Y \leq R) \Rightarrow (X = R+(Y \times Q) \land R<Y)
\end{align*}
\]

- Notice that these are statements of arithmetic
  - the constructs of our programming language have been ‘compiled away’

- Step 3 consists in proving the four verification conditions
  - easy with modern automatic theorem provers

Annotation of Commands

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

\[
\begin{align*}
(i) & \quad \text{before } C_2 \text{ in } C_1; C_2 \text{ if } C_2 \text{ is not an assignment command} \\
(ii) & \quad \text{after the word } \text{DO } \text{in WHILE commands}
\end{align*}
\]

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

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

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

\begin{verbatim}
{X=n}
Y:=1; ←− 1
WHILE X \neq 0 DO ←− 2
    (Y:=Y\times X; X:=X-1)
{X=0 \land Y=n!}
\end{verbatim}

- Suitable statements would be

  at 1: \{Y = 1 \land X = n\}
  at 2: \{Y \times 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
    \[ vc_1, \ldots, vc_n \]
    together with the VCs for C_1 and those for C_2.
  - Each VC rule corresponds to either a primitive or derived rule.

A VC Generation Program

- The algorithm for generating verification conditions is recursive on the structure of commands.

- The rule just given corresponds to the recursive program clause:

\[ VC (C(C_1, C_2)) = [vc_1, \ldots, vc_n] \oplus (VC C_1) \oplus (VC C_2) \]

- 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

VC for Assignments

Assignment commands
The single verification condition generated by
\[
\{P\} V := E \{Q\}
\]
is
\[
P \Rightarrow Q[E/V]
\]

- Example: The verification condition for
\[
\{X=0\} X := X + 1 \{X=1\}
\]
is
\[
X = 0 \Rightarrow (X + 1) = 1
\]
(which is clearly true)

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

Justification of Assignment VC

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

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

VCs for Conditionals

Conditionals
The verification conditions generated from
\[
\{P\} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \{Q\}
\]
are
(i) the verification conditions generated by
\[
\{P \land S\} \; C_1 \{Q\}
\]
(ii) the verifications generated by
\[
\{P \land \neg S\} \; C_2 \{Q\}
\]

- Example: The verification conditions for
\[
\{T\} \text{ IF } X \geq Y \text{ THEN } MAX := X \text{ ELSE } MAX := Y \{\text{MAX} = \text{max}(X, Y)\}
\]
are
(i) the VCs for $\{T \land X \geq Y\} \; \text{MAX} := X \; \{\text{MAX} = \text{max}(X, Y)\}$
(ii) the VCs for $\{T \land \neg(X \geq Y)\} \; \text{MAX} := Y \; \{\text{MAX} = \text{max}(X, Y)\}$
Justification for the Conditional VCs (1)

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

Proof:

- Assume the VCs \{P \land S\} C_1 \{Q\} and \{P \land \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 \land S\} C_1 \{Q\} and \vdash \{P \land \neg S\} C_2 \{Q\}

- Hence by the conditional rule \vdash \{P\} \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \{Q\}

VCs for 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]\}

Review of Annotated Sequences

- If C_1;C_2 is properly annotated, then either
  1. it is of the form C_1;\{R\}C_2 and C_2 is not an assignment
  2. it is of the form C;V := E

- And C, C_1 and C_2 are properly annotated

Example

- The verification conditions generated from
  \{X=x \land Y=y\} R:=X; X:=Y; Y:=R \{X=y \land Y=x\}

- Are those generated by
  \{X=x \land Y=y\} R:=X; X:=Y \{(X=y \land Y=x) [R/Y]\}

- This simplifies to
  \{X=x \land Y=y\} R:=X; X:=Y \{X=y \land R=x\}

- The verification conditions generated by this are those generated by
  \{X=x \land Y=y\} R:=X \{(X=y \land R=x) [Y/X]\}

- Which simplifies to
  \{X=x \land Y=y\} R:=X \{Y=y \land R=x\}
Example Continued

- The only verification condition generated by
  \( \{ X=x \land Y=y \} R := X \{ Y=y \land R = x \} \)
  is
  \( X = x \land Y = y \Rightarrow (Y = y \land R = x) [X/R] \)

- Which simplifies to
  \( X = x \land Y = y \Rightarrow Y = y \land X = x \)

- Thus the single verification condition from
  \( \{ X=x \land Y=y \} R := X; X := Y; Y := R \{ X=y \land Y = x \} \)
  is
  \( X = x \land Y = y \Rightarrow Y = y \land X = x \)

Justification of VCs for Sequences (1)

- Case 1: If the verification conditions for
  \( \{ P \} C_1 ; \{ R \} C_2 \{ Q \} \)
  are provable
  Then the verification conditions for
  \( \{ P \} C_1 \{ R \} \)
  and
  \( \{ R \} C_2 \{ Q \} \)
  must both be provable
  Hence by induction
  \( \vdash \{ P \} C_1 \{ R \} \) and \( \vdash \{ R \} C_2 \{ Q \} \)
- Hence by the sequencing rule
  \( \vdash \{ P \} C_1 ; C_2 \{ Q \} \)

Justification of VCs for Sequences (2)

- Case 2: If the verification conditions for
  \( \{ P \} C ; V := E \{ Q \} \)
  are provable, then the verification conditions for
  \( \{ P \} C \{ Q(E/V) \} \)
  are also provable
  Hence by induction
  \( \vdash \{ P \} C \{ Q(E/V) \} \)
  Hence by the derived sequenced assignment rule
  \( \vdash \{ P \} C ; V := E \{ Q \} \)

VCs for WHILE-Commands

- A correctly annotated specification of a WHILE-command has the form
  \( \{ P \} \text{WHILE } S \text{ DO } \{ R \} C \{ Q \} \)
- The annotation \( R \) is called an invariant
  \( \textbf{WHILE-commands} \)
  The verification conditions generated from
  \( \{ P \} \text{WHILE } S \text{ DO } \{ R \} C \{ Q \} \)
  are
  (i) \( P \Rightarrow R \)
  (ii) \( R \land \neg S \Rightarrow Q \)
  (iii) the verification conditions generated by \( \{ R \land S \} C(R) \)
The verification conditions for
\{R=X \land Q=0\}
WHILE \ Y \leq R DO \{X=R+Y \times Q\}
\{X = R+(Y \times Q) \land R<Y\}
are:

(i) \ R=X \land Q=0 \Rightarrow (X = R+(Y \times Q))

(ii) \ X = R+Y \times Q \land \neg(Y \leq R) \Rightarrow (X = R+(Y \times Q) \land R<Y)

together with the verification condition for
\{X = R+(Y \times Q) \land (Y \leq R)\}
\{R:=R-Y; \ Q:=Q+1\}
\{X=R+(Y \times Q)\}
which consists of the single condition

(iii) \ X = R+(Y \times Q) \land (Y \leq R) \Rightarrow X = (R-Y)+(Y \times (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 \land \neg S) \Rightarrow Q
  and the verification conditions for
  \{R \land S\} C \{R\}
  are provable

- By induction
  \vdash \{R \land 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 Dijkstra's presentation
  - start with a non-deterministic high level pseudo-code
  - refine to deterministic and efficient code
- Weakest preconditions (wp) are for total correctness
- Weakest liberal preconditions (wlp) for partial correctness

Rules for weakest preconditions

- Relation with Hoare specifications:
  \{ P \} \ C \ { Q \} \iff P \Rightarrow \text{wp}(C, Q)
  \[ P \] \ C \ { Q \} \iff P \Rightarrow \text{wp}(C, Q)

- Dijkstra gives rules for computing weakest preconditions:
  \text{wp}(V := E, Q) = Q[E/V]
  \text{wp}(C_1; C_2, Q) = \text{wp}(C_1, \text{wp}(C_2, Q))
  \text{wp}(\text{if } C \text{ then } E \text{ else } F, Q) = (S \Rightarrow \text{wp}(C, Q)) \land (\neg S \Rightarrow \text{wp}(F, Q))

- Rule for \texttt{while}-commands doesn't give a first order result
- Weakest preconditions closely related to verification conditions
- VCs for \{ P \} \ C \ { Q \} are related to \text{wp}(C, Q)
  - VCs use annotations to ensure first order formulas can be generated

Sequencing example

- Swapping variables:
  \text{wlp}(R := X; X := Y; Y := R, (Y = x \land X = y))
  = \text{wlp}(R := X, \text{wlp}(X := Y, \text{wlp}(Y := R, \ldots \{Y = x \land X = y\}))

  \text{just need to prove:}
  (X = x \land Y = y) \Rightarrow (X = x \land Y = y)
  which is clearly true (instance of S \Rightarrow S)

Conditional example

- Compute wlp of the maximum program:
  \text{wlp}(\text{if } X < Y \text{ then } \text{MAX} := Y \text{ else } \text{MAX} := X, \{\text{MAX} = \text{max}(x, y)\})
  = (X < Y \Rightarrow \text{wlp}(\text{MAX} := Y, \{\text{MAX} = \text{max}(x, y)\}))
  \land
  (\neg (X < Y) \Rightarrow \text{wlp}(\text{MAX} := X, \{\text{MAX} = \text{max}(x, y)\}))
  = (X < Y \Rightarrow Y = \text{max}(x, y)) \land (\neg (X < Y) \Rightarrow X = \text{max}(x, y))
  \text{if } X < Y \text{ then } Y = \text{max}(x, y) \text{ else } X = \text{max}(x, y)

- So to prove:
  \{ X = x \land Y = y \} \text{ if } X < Y \text{ then } \text{MAX} := X \text{ else } \text{MAX} := Y \{ \text{MAX} = \text{max}(x, y) \}

  just prove:
  (X = x \land Y = y) \Rightarrow (X < Y \Rightarrow Y = \text{max}(x, y)) \land (\neg (X < y) \Rightarrow X = \text{max}(x, y))
  which follows from the defining property of max
  \vdash \forall x y. (x \geq y \Rightarrow x = \text{max}(x, y)) \land (\neg (x \geq y) \Rightarrow y = \text{max}(x, y))
Using vlp to improve verification condition method

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

Definition of awp

- Assume all WHILE-commands are annotated: $WHILE \ S \ DO\ \{R\} \ C$
- Define awp recursively by:
  $$awp(V := E, Q) = Q[E/V]$$
  $$awp(C_1 ; C_2, Q) = awp(C_1, awp(C_2, Q))$$
  $$awp(IF \ S \ THEN \ C_1 \ ELSE \ C_2, Q) = (S \ \land \ awp(C_1, Q)) \lor (\neg S \ \land \ awp(C_2, Q))$$
  $$awp(WHILE \ S \ DO\ \{R\} \ C, Q) = R$$
- Note:
  $$\ (S \ \land \ awp(C_1, Q)) \lor (\neg S \ \land \ awp(C_2, Q)) = if \ S \ then \ awp(C_1, Q) \ else \ awp(C_2, Q)$$

vlp-based verification condition method

- We define $awp(C, Q)$ and $wvc(C, Q)$
  - $awp(C, Q)$ is a statement sort of approximating $vlp(C, Q)$
  - $wvc(C, Q)$ is a set of verification conditions
- If $C$ is loop-free then
  - $awp(C, Q) = vlp(C, Q)$
  - $wvc(C, Q) = \{\}$
- Denote by $\bigwedge S$ the conjunction of all the statements in $S$
  - $\bigwedge\{\} = T$
  - $\bigwedge(S_1 \cup S_2) = (\bigwedge S_1) \land (\bigwedge S_2)$
- It will follow that $\bigwedge wvc(C, Q) \Rightarrow \{awp(C, Q)\} C \{Q\}$
- Hence to prove $\{P\} C \{Q\}$ it is sufficient to prove all the statements in $wvc(C, Q)$ and $P \Rightarrow awp(C, Q)$

Definition of wvc

- Assume all WHILE-commands are annotated: $WHILE \ S \ DO\ \{R\} \ C$
- Define wvc recursively by:
  $$wvc(V := E, Q) = \{\}$$
  $$wvc(C_1 ; C_2, Q) = wvc(C_1, wvc(C_2, Q)) \cup wvc(C_2, Q)$$
  $$wvc(IF \ S \ THEN \ C_1 \ ELSE \ C_2, Q) = wvc(C_1, Q) \cup wvc(C_2, Q)$$
  $$wvc(WHILE \ S \ DO\ \{R\} \ C, Q) = \{R \land \neg S \Rightarrow Q, R \land S \Rightarrow awp(C, R)\} \cup wvc(C, R)$$
Correctness of \( \text{wlp-based verification conditions} \)

- **Theorem:** \( \forall \text{wvc}(C, Q) \rightarrow \{ \text{awp}(C, Q) \} C \{Q\} \). Proof by Induction on \( C \)
  
  - \( \forall \text{wvc}(V := E, Q) \rightarrow \{ \text{awp}(C, Q) \} C \{Q\} \) is \( (V := E \rightarrow \{Q\}) \)
  
  - \( \forall \text{wvc}(C_1; C_2, Q) \rightarrow \{ \text{awp}(C_1; C_2, Q) \} C_1; C_2 \{Q\} \)
    By induction \( \forall \text{wvc}(C_2, Q) \rightarrow \{ \text{awp}(C_2, Q) \} C_1 \{Q\} \)
    and \( \forall \text{wvc}(C_1; C_2, Q) \rightarrow \{ \text{awp}(C_1; C_2, Q) \} C_2 \{\text{awp}(C_1, Q)\} \)
    hence result by the Sequencing Rule.

- \( \forall \text{wvc}(\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2, Q) \rightarrow \{ \text{awp}(S, Q) \} C_1 \{Q\} \)
  By induction \( \forall \text{wvc}(S, Q) \rightarrow \{ \text{awp}(S, Q) \} C_1 \{Q\} \)
  and \( \forall \text{wvc}(C, Q) \rightarrow \{ \text{awp}(C, Q) \} C_2 \{Q\} \).
  Strengthening preconditions gives \( \forall \text{wvc}(C, Q) \rightarrow \{ \text{awp}(C, Q) \} C_1 \{Q\} \)
  and \( \forall \text{wvc}(C, Q) \rightarrow \{ \text{awp}(C, Q) \} \neg S \} C_2 \{Q\} \), hence result by the Conditional Rule.

- \( \forall \text{wvc}(\text{WHILE } S \text{ DO } R, C, Q) \rightarrow \{ \text{awp}(S, Q) \} \text{WHILE } S \text{ DO } R \rightarrow C, Q \) \)
  By induction \( \forall \text{wvc}(S, Q) \rightarrow \{ \text{awp}(S, Q) \} \text{WHILE } S \text{ DO } R \rightarrow C, Q \)
  hence result by WHILE-Rule.

Strongest postconditions

- Define \( \text{sp}(C, P) \) to be ‘strongest’ \( Q \) such that \( \{P\} C \{Q\} \)
  - partial correctness: \( \{P\} \{\text{sp}(C, P)\} \)
  - strongest means if \( \{P\} C \{Q\} \) then \( \text{sp}(C, P) \rightarrow Q \)

- Note that \( \text{wlp} \) goes ‘backwards’, but \( \text{sp} \) goes ‘forwards’
  - verification condition for \( \{P\} C \{Q\} \) is \( \text{sp}(C, P) \rightarrow Q \)

By ‘strongest’ and Hoare logic postcondition weakening
- \( \{P\} C \{Q\} \) if and only if \( \text{sp}(C, P) \rightarrow Q \)

Sequencing example

- \( \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)
  - \( \forall \text{sp}(S := X; \ Y := R; \ Z := Y; \ Y := R; \ X = x \land Y = y) \)

So to prove \( \{X = x \land Y = y\} R := X; \ X := Y; \ Y := R \{\{x \land X = y\} \}
  just proves: \( \forall \{X = x \land Y = y\} R := X; \ X := Y; \ Y := R \{\{x \land X = y\} \}

• Only consider loop-free code
  - \( \text{sp}(V := E, P) = \forall \exists v, V = E[v/V] \land P[v/V] \)
  - \( \text{sp}(C_1 \land C_2, P) = \text{sp}(C_2, \text{sp}(C_1, P)) \)
  - \( \text{sp}(\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2, P) = \text{sp}(C_1, P \land S) \lor \text{sp}(C_2, P \land \lnot S) \)

• \( \text{sp}(V := E, P) \) corresponds to Floyd assignment axiom

• Can dynamically prune conditionals because \( \text{sp}(C, F) = F \)

• Computer strongest postconditions is symbolic execution
Conditional example

- Compute $sp$ of the maximum program:

  $sp(\text{IF } X < Y \text{ THEN } \text{MAX} := Y \text{ ELSE } \text{MAX} := X, (X = x \land Y = y))$
  
  $\forall$
  
  $sp(\text{MAX} := Y, ((X = x \land Y = y) \land X < Y))$

  $\exists v. \text{ MAX} = Y[v/\text{MAX}] \land ((X = x \land Y = y) \land X < Y)[v/\text{MAX}]$

  $\forall$
  
  $sp(\text{MAX} := X, ((X = x \land Y = y) \land X < Y))$

  $\exists v. \text{ MAX} = X[v/\text{MAX}] \land ((X = x \land Y = y) \land X < Y)[v/\text{MAX}]$

  $\forall$
  

  $\exists v. \text{ MAX} = Y \land ((X = x \land Y = y) \land X < Y)$

  $\forall$
  
  $\exists v. \text{ MAX} = X \land X = x \land Y = y \land \neg (X < Y)$

  $\forall$
  
  $(\text{MAX} = Y \land X = x \land Y = y \land X < Y) \lor (\text{MAX} = X \land X = x \land Y = y \land \neg (X < Y))$

  $\forall$
  
  $(\text{MAX} = y \land X = x \land Y = y \land x < y) \lor (\text{MAX} = x \land X = x \land Y = y \land \neg (x < y))$

  $\forall$
  
  $(\text{MAX} = (if\ x < y\ then\ y\ else\ x) \land X = x \land Y = y)$

  $\forall$
  
  $(\text{MAX} = \text{max}(x, y) \land X = x \land Y = y)$

Computing $sp$ versus $wlp$

- Computing $sp$ is like execution
  - can simplify as one goes along with the ‘current state’
  - may be able to resolve branches, so can avoid executing them
  - Floyd assignment rule complicated in general
  - $sp$ used for symbolically exploring ‘reachable states’ (related to model checking)

- Computing $wlp$ is like backwards proof
  - don’t have ‘current state’, so can’t simplify using it
  - can’t determine conditional tests, so get big if-then-else trees
  - Hoare assignment rule simpler for arbitrary formulae
  - $wlp$ used for improved verification conditions

Using $sp$ to generate verification conditions

- If $C$ is loop-free then VC for $\{P\} C \{Q\}$ is $sp(C, P) \Rightarrow Q$

- Cannot in general compute a finite formula for $sp(\text{WHILE } S \text{ DO } C, P)$

- The following holds
  
  $sp(\text{WHILE } S \text{ DO } C, P) = sp(\text{WHILE } S \text{ DO } C, sp(C, (P \land S))) \lor (P \land \neg S)$

- Above doesn’t define $sp(C, P)$ to be a finite statement

- As with $wlp$, can use a hybrid VC and $sp$ method

Exercises

- Compute

  
  $sp(R := 0; K := 0; \text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE } K := K; \text{IF } K = 1 \land \neg(I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J; \text{IF } I = i \land J = j \land j \leq i)

- Hence show

  
  $(\{I = i \land J = j \land j \leq i\})$

  $R := 0; K := 0; \text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE } K := K; \text{IF } K = 1 \land \neg(I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J; \{R = i - j\}$

- Do same example use $wlp$
Definition of svc

- Define svc recursively by:
  \[ \text{svc}(P, V := E) = \{\} \]
  \[ \text{svc}(P, C_1 ; C_2) = \text{svc}(\text{svc}(P, C_1), C_2) \]
  \[ \text{svc}(P, \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2) = \text{svc}(P \land S, C_1) \cup \text{svc}(P \land \neg S, C_2) \]
  \[ \text{svc}(P, \text{WHILE } S \text{ DO } \{R\} C) = R \land \neg S \]

- Theorem: \( \land \text{svc}(P, C) \Rightarrow \{P\} C \{\text{asp}(P, C)\} \)

- Proof by induction on C (exercise)

Definition of asp

- Define asp recursively by:
  \[ \text{asp}(P, V := E) = \exists v. V = E[v/V] \land P[v/V] \]
  \[ \text{asp}(P, C_1 ; C_2) = \text{asp}(\text{asp}(P, C_1), C_2) \]
  \[ \text{asp}(P, \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2) = \text{asp}(P \land S, C_1) \lor \text{asp}(P \land \neg S, C_2) \]
  \[ \text{asp}(P, \text{WHILE } S \text{ DO } \{R\} C) = R \land \neg S \]

Summary

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

- VCs and wp 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

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 \textbf{WHILE}-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness

Termination of \textbf{WHILE}-Commands

- \textbf{WHILE}-commands are the only commands that might not terminate
- Consider now the following proof
  1. \(\vdash \{T\} X := X \{T\}\) (assignment axiom)
  2. \(\vdash \{T \land T\} X := X \{T\}\) (precondition strengthening)
  3. \(\vdash \{T\} \text{WHILE } T \text{ DO } X := X \{T \land \neg T\}\) (2 and the \textbf{WHILE}-rule)
- If the \textbf{WHILE}-rule worked for total correctness, then this would show:
  \(\vdash \{T\} \text{WHILE } T \text{ DO } X := X \{T \land \neg T\}\)
- Thus the \textbf{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

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

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

\(\text{if } C \text{ contains no } \textbf{WHILE}-\text{commands}\)
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:

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{fact}(n-1) & \text{else}
\end{cases}
\]

Error Termination

- We assume erroneous expressions like \( 1/0 \) don’t cause problems

- Most programming languages will raise an error on division by zero

- In our logic it follows that

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

- The assignment \( X := 1/0 \) halts in a state in which \( X = 1/0 \) holds

- This assumes that \( 1/0 \) denotes some value that \( X \) can have

Two Possibilities

- There are two possibilities

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

  (ii) \( 1/0 \) denotes some kind of ‘error value’.

- It seems at first sight that adopting (ii) is the most natural choice

  - this makes it tricky to see what arithmetical laws should hold

  - is \( (1/0) \times 0 \) equal to 0 or to some ‘error value’?

  - if the latter, then it is no longer the case that \( \forall n. n \times 0 = 0 \) is valid

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

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 $\vdash P \land S \Rightarrow E \geq 0$ ensures the variant is non-negative

\[ \begin{align*}
\vdash [P \land S \land (E = n)] & \quad C \quad [P \land (E < n)] \quad \vdash P \land S \Rightarrow E \geq 0 \\
\vdash [P] \quad \text{WHILE} \quad S \quad \text{DO} \quad C \quad [P \land \neg S]
\end{align*} \]

Example

- We show

  \[ \vdash [Y > 0] \quad \text{WHILE} \quad Y \leq R \quad \text{DO} \quad (R := R - Y; \; Q := Q + 1) \quad [T] \]

- Take

  \[ \begin{align*}
P & = Y > 0 \\
S & = Y \leq R \\
E & = R \\
C & = (R := R - Y; \; Q := Q + 1)
\end{align*} \]

- We want to show $\vdash [P] \quad \text{WHILE} \quad S \quad \text{DO} \quad C \quad [T]$

- By the WHILE-rule for total correctness it is sufficient to show

  \[ \begin{align*}
  (i) & \vdash [P \land S \land (E = n)] \quad C \quad [P \land (E < n)] \\
  (ii) & \vdash P \land S \Rightarrow E \geq 0
  \end{align*} \]

Example Continued (1)

- From previous slide:

  \[ \begin{align*}
P & = Y > 0 \\
S & = Y \leq R \\
E & = R \\
C & = (R := R - Y; \; Q := Q + 1)
\end{align*} \]

- We want to show

  \[ \begin{align*}
  (i) & \vdash [P \land S \land (E = n)] \quad C \quad [P \land (E < n)] \\
  (ii) & \vdash P \land S \Rightarrow E \geq 0
  \end{align*} \]

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

  \[ \vdash \{P \land S \land (E = n)\} \quad C \quad \{P \land (E < n)\} \]

- Then using the total correctness rule for non-looping commands
Example Continued (2)

- From previous slide:
  \[ P = Y > 0 \]
  \[ S = Y \leq R \]
  \[ E = R \]
  \[ C = R:=R-Y; Q:=Q+1 \]

- The verification condition for \( \{ P \land S \land (E = n) \} \ C \{ P \land (E < n) \} \) is:
  \[ Y > 0 \land Y \leq R \land R = n \Rightarrow (Y > 0 \land R < n) [Q+1/Q] [R−Y/R] \]
  i.e. \( Y > 0 \land Y \leq R \land R = n \Rightarrow Y > 0 \land R−Y < n \)
  which follows from the laws of arithmetic

- The second subgoal, (ii), is just \( \vdash Y > 0 \land Y \leq R \Rightarrow R \geq 0 \)

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
  \[ \vdash \{ P \} C \{ Q \} \vdash \{ P \} [T] \]
  \[ \vdash \{ P \} C \{ Q \} \vdash \{ P \} C \{ T \} \]

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 \texttt{WHILE}-rule needs to handle the variant

\[ \vdash P \Rightarrow R \]
\[ \vdash R \land S \Rightarrow E \geq 0 \]
\[ \vdash R \land \neg S \Rightarrow Q \]
\[ \vdash (R \land S \land (E = n)) \ C (R \land (E < n)) \]
\[ \vdash [P] \text{WHILE } S \text{ DO } C \{ Q \} \]
VCs for Termination

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

WHILE Annotation

- A correctly annotated total correctness specification of a WHILE-command thus has the form
  \[ [P] \text{WHILE } S \text{ DO } \{R\} [E] C [Q] \]
  where \( R \) is the invariant and \( E \) the variant
- Note that the variant is intended to be a non-negative expression that decreases each time around the WHILE loop
- The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them (as before)

WHILE VCs

- A correctly annotated specification of a WHILE-command has the form
  \[ [P] \text{WHILE } S \text{ DO } \{R\} [E] C [Q] \]

The verification conditions generated from \[ [P] \text{WHILE } S \text{ DO } \{R\} [E] C [Q] \]
are:

(i) \( P \Rightarrow R \)
(ii) \( R \land \neg S \Rightarrow Q \)
(iii) \( R \land S \Rightarrow E \geq 0 \)
(iv) the verification conditions generated by
  \[ [R \land S \land (E = n)] \land C[R \land (E < n)] \]
  where \( n \) is a variable not occurring in \( P, R, E, C, S \) or \( Q \).

Example

- The verification conditions for
  \[ [R=X \land Q=0] \land \text{WHILE } Y \leq R \text{ DO } \{X=R+Y\times Q\} [R] \]
  \( (R:=R-Y; \ Q:=Q+1) \)
  \[ X = R+(Y\times Q) \land R<Y \]
are:
  (i) \( R=X \land Q=0 \Rightarrow (X = R+(Y\times Q)) \)
  (ii) \( X = R+Y\times Q \land \neg(Y \leq R) \Rightarrow (X = R+(Y\times Q) \land R<Y) \)
  (iii) \( X = R+Y\times Q \land Y \leq R \Rightarrow R \geq 0 \)

together with the verification condition for
  \[ [X = R+(Y\times Q) \land (Y \leq R) \land (R=n)] \land (R:=R-Y; \ Q:=Q+1) \]
  \[ X=R+(Y\times Q) \land (R<n) \]
Example Continued

- The single verification condition for
  \[ X = R + (Y \times Q) \land (Y \leq R) \land (R = n) \]  
  \( (R := R - Y; Q := Q + 1) \)  
  \[ X = R + (Y \times Q) \land (R < n) \]
  is
  \( (iv) \)  
  \[ X = R + (Y \times Q) \land (Y \leq R) \land (R = n) \Rightarrow \]  
  \[ X = (R - Y) + (Y \times (Q + 1)) \land ((R - Y) < n) \]

- 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 \( \texttt{WHILE} \)-rule
  - because \( \texttt{WHILE} \) 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://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 \textit{terms} and \textit{formulae}
- For consistency with earlier stuff we use \textit{expressions} and \textit{statements}
- Will define sets \( \textit{Exp} \) of expressions and \( \textit{Sta} \) of statements
- Sets \( \textit{Exp} \) and \( \textit{Sta} \) depend on a language \( L \) (see next slide)
  - will write \( \textit{Exp}_L \) and \( \textit{Sta}_L \) to make this clear
  - if language is clear from context may omit language subscript
- Assume an infinite set \( \textit{Var} \) of variables
  - doesn’t depend on a language
First-order languages

A first-order language $\mathcal{L}$ contains

- zero or more predicate symbols, $p_1, p_2, \ldots$ each with an arity $\geq 0$
- zero or more function symbols, $f_1, f_2, \ldots$ each with an arity $\geq 0$
- $\mathcal{L} = \{(p_1, p_2, \ldots); (f_1, f_2, \ldots)\}$

$\text{Exp}_\mathcal{L}$ is the smallest set such that:

- $\text{Var} \subseteq \text{Exp}_\mathcal{L}$
- $f$ a function symbols of $\mathcal{L}$ of arity 0, then $f \in \text{Exp}_\mathcal{L}$
- $f$ a function symbols of $\mathcal{L}$ of arity $n > 0$ and $E_1 \in \text{Exp}_\mathcal{L}$, then $f(E_1, \ldots, E_n) \in \text{Exp}_\mathcal{L}$

$\text{Sta}_\mathcal{L}$ is the smallest set such that:

- $p$ a predicate symbols of $\mathcal{L}$ of arity 0, then $p \in \text{Sta}_\mathcal{L}$
- $p$ a predicate symbols of $\mathcal{L}$ of arity $n > 0$ and $E_1 \in \text{Exp}_\mathcal{L}$, then $p(E_1, \ldots, E_n) \in \text{Sta}_\mathcal{L}$
- $S_1, S_2, S_3$ in $\text{Sta}_\mathcal{L}$, then $\neg S_1, S_1 \land S_2, S_1 \lor S_2, S_1 \Rightarrow S_2$ are in $\text{Sta}_\mathcal{L}$
- $v \in \text{Var}$ and $S$ in $\text{Sta}_\mathcal{L}$, then $\forall v. S$ and $\exists v. S$ are in $\text{Sta}_\mathcal{L}$

Semantics: interpretations

An interpretation $\mathcal{I} = (D, I)$ provides:

- $\text{domain } D$ of values, also called a universe
- meaning $I[p]$ for predicate symbols $p$ and $I[f]$ for function symbols $f$

Sets, functions and relations

- $\text{Bool} = \{\text{true}, \text{false}\}$
- if $n > 0$, then $A^n = \{(a_1, \ldots, a_n) | a_i \in A\}$
- $A \rightarrow B = \{ u | u : A \rightarrow B \}$ (alternative notation: $B^A$)

If $\mathcal{I} = (D, I)$ then:

- if $p$ is a predicate symbol of arity 0, then $I[p] \in \text{Bool}$
- if $p$ is a predicate symbol of arity $n > 0$, then $I[p] \in D^n \rightarrow \text{Bool}$
- 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 $I[f] \in D^n \rightarrow D$

Semantics: terms and formulae

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

Define $E_{\text{sem}} E \in D$ by:

- if $E \in \text{Var}$ then $E_{\text{sem}} E = s(E)$
- if $E = f$, where $f$ a function symbol of arity 0, then $E_{\text{sem}} E = I[f]$
- if $E = f(E_1, \ldots, E_n)$, then $E_{\text{sem}} E = I[f](E_{\text{sem}} E_1, \ldots, E_{\text{sem}} E_n)$

Define $S_{\text{sem}} S \in \text{Bool}$ by:

- if $S = p$, where $p$ a predicate symbol of arity 0, then $S_{\text{sem}} S = I[p]$
- if $S = p(E_1, \ldots, E_n)$, then $S_{\text{sem}} S = I[p](E_{\text{sem}} E_1, \ldots, E_{\text{sem}} E_n)$
- $S_{\text{sem}} (\neg S) = \neg (S_{\text{sem}} S)$
- $S_{\text{sem}} (S_1 \land S_2) = (S_{\text{sem}} S_1) \land (S_{\text{sem}} S_2)$
- $S_{\text{sem}} (S_1 \lor S_2) = (S_{\text{sem}} S_1) \lor (S_{\text{sem}} S_2)$
- $S_{\text{sem}} (S_1 \Rightarrow S_2) = (S_{\text{sem}} S_1) \Rightarrow (S_{\text{sem}} S_2)$
- $S_{\text{sem}} (\forall v. S) = \text{if for all } d \in D : S_{\text{sem}} (s[dy]) = \text{true then true else false}$
- $S_{\text{sem}} (\exists v. S) = \text{if for some } d \in D : S_{\text{sem}} (s[dy]) = \text{true then true else false}$

Note: will just say “$S_{\text{sem}} S = \text{true}””
### Satisfiability, validity and completeness

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

### Gödel’s incompleteness theorem

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

### 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 \( \mathbb{S}_{\mathcal{I}} s_{1} = \mathbb{S}_{\mathcal{I}} s_{2} \) for all \( s_{1}, s_{2} \)
- A **theory** is a set of sentences
  - \( \Gamma \vdash S \) means \( S \) can be deduced from \( \Gamma \) using first-order logic
  - \( \Gamma \) is **consistent** iff there is no \( S \) such that \( \Gamma \vdash S \) and \( \Gamma \vdash \neg S \)
  - \( \Gamma \models_{\mathcal{I}} S \) means \( S \) true if \( \mathcal{I} \) makes all of \( \Gamma \) true
  - \( \Gamma \models S \) means \( \Gamma \models_{\mathcal{I}} S \) true for all \( \mathcal{I} \)
- Soundness and Completeness: \( \Gamma \models S \) iff \( \Gamma \vdash S \)

### 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 \( \mathbb{S}_{\mathcal{I}} s_{1} \)
  - and if the execution of \( C \) starting in \( s_{1} \) terminates
  - then \( C \) terminates in a state \( s_{2} \) such that \( \mathbb{S}_{\mathcal{I}} Q s_{2} = \text{true} \)
- Need to define “\( C \) starts in \( s_{1} \) and terminates in \( s_{2} \)”
  - this is the semantics of commands
  - will define \( \mathbb{C}_{\mathcal{I}} s_{1} s_{2} \) to mean if \( C \) starts in \( s_{1} \) then it can terminate in \( s_{2} \)
- Semantics of \( \{ P \} C \{ Q \} \) is \( \mathbb{H}_{\mathcal{I}} P C Q \) where:
  - \( \mathbb{H}_{\mathcal{I}} P C Q = \forall s_{1} s_{2}. \mathbb{S}_{\mathcal{I}} P s_{1} \land \mathbb{C}_{\mathcal{I}} C s_{1} s_{2} \Rightarrow \mathbb{S}_{\mathcal{I}} Q s_{2} \)
- Sometimes write \( \models \{ P \} C \{ Q \} \) to mean \( \mathbb{H}_{\mathcal{I}} P C Q \)
Semantics of commands

- Assignments
  \[ \text{Csem} (V := E) s_1 s_2 = (s_2 = s_1[E_{s_1}/V]) \]

- Sequences
  \[ \text{Csem} (C_1; C_2) s_1 s_2 = \exists s. \text{Csem} C_1 s_1 s \land \text{Csem} C_2 s s_2 \]

- Conditional
  \[ \text{Csem} (\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2) s_1 s_2 = (\text{Ssem } S s_1 \land \text{Csem } C_1 s_1 s_2) \lor (\neg \text{Ssem } S s_1 \land \text{Csem } C_2 s_1 s_2) \]

  \[ = \text{if } S s_1 \text{ then } \text{Csem } C_1 s_1 s_2 \text{ else } \text{Csem } C_2 s_1 s_2 \]

- While-commands
  \[ \text{Csem} (\text{WHILE } S \text{ DO } C) s_1 s_2 = \exists n. \text{Iter } n (\text{Ssem } S) (\text{Csem } C) s_1 s_2 \]

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

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

  \( \text{Iter } (n+1) p c s_1 s_2 = p s_1 \land \exists s_3. c s_1 s_3 \land \text{Iter } n p c s s_2 \)

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

Semantics of \{P\} C \{Q\}

\[ \forall s_1 s_2. \text{Ssem } P s_1 \land \text{Csem } C s_1 s_2 \Rightarrow \text{Ssem } Q s_2 \]

Assignment axiom:

\[ \vdash \{Q[E/V]\} \vdash \{E \} \]

Must show:

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

Unfolding the definition of \( \text{Csem } \) converts this to:

\[ \forall s_1 s_2. \text{Ssem } (Q[E/V]) s_1 \land (s_2 = s_1[E_{s_1}/V]) \Rightarrow \text{Ssem } Q s_2 \]

This simplifies to:

\[ \forall s_1. \text{Ssem } (Q[E/V]) s_1 \Rightarrow \text{Ssem } Q (s_1[E_{s_1}/V]) \]

- [...] \] has different meanings in antecedent and consequent
- in antecedent \( Q[E/V] \) is substituting \( E \) for \( V \) in \( Q \)
- in consequent \( s_1[E_{s_1}/V] \) is updating \( s_1 \), so value of \( V \) is value of \( E \) in \( s_1 \)

Will prove for all \( S \) that: \( \text{Ssem } (S[E/V]) s = \text{Ssem } S (s[E_{s}/V]) \)

Substitution lemma for expressions: variables

- Assume: language \( L \), interpretation \( I = (I, D) \), valuation \( s \in \text{Var} \rightarrow D \)

\[ \forall s. \text{Ssem } (E[E'/V]) s = \text{Ssem } (s[E_{s}/V]) \text{ by induction on } E \]

- If \( E = V \) then must show
  \[ \text{Ssem } (V[E/V]) s = \text{Ssem } (s[E_{s}/V]) \]
  \[ \text{Ssem } E s = (s[E_{s}/V]) s \]

- If \( E = V' \), where \( V \neq V' \), then must show
  \[ \text{Ssem } (V'[E/V]) s = \text{Ssem } (s[E_{s}/V]) \]
  \[ \text{Ssem } V' s = (s[E_{s}/V]) (s[V']) \]
  \[ s(V') = s(V') \]

Substitution lemma for expressions: applications

- Assume: language \( L \), interpretation \( I = (I, D) \), valuation \( s \in \text{Var} \rightarrow D \)

\[ \forall s. \text{Ssem } (E[E'/V]) s = \text{Ssem } (s[E_{s}/V]) \text{ by induction on } E \]

- Assume \( \text{Ssem } (E_i[E'/V]) s = \text{Ssem } E_i (s[E_{s}/V]) \) for \( 1 \leq i \leq n \)

- If \( E = f \), where \( f \) has arity 0, then must show
  \[ \text{Ssem } (f[E'/V]) s = \text{Ssem } f (s[E_{s}/V]) \]
  \[ I[f] = I[f] \]

- If \( E = E_1 \ldots E_n \) then must show
  \[ \text{Ssem } (E_1 \ldots E_n[E'/V]) s = \text{Ssem } (E_1 \ldots E_n) (s[E_{s}/V]) \]
  \[ \text{Ssem } (f(E_1[E'/V], \ldots, E_n[E'/V])) s = I[f] I[E_1 (s[E_{s}/V]), \ldots, E_n (s[E_{s}/V])] \]
  \[ I[f] I[E_1 (s[E_{s}/V]), \ldots, E_n (s[E_{s}/V])] = \]

Equation true by induction
Soundness of Precondition Strengthening

- Assume: language $L$, interpretation $I = (I, D)$, valuation $s \in \text{Var} \rightarrow D$
- $\forall s. \text{Ssem} (S[E/V]) s = \text{Ssem} S (s[\text{Esem} E s/V])$ by induction on $S$
- Proof similar to expressions except care needed with bound variables
- Assume bound variables renamed to avoid clashes, then:
  \[(\forall s. S[E/V] = \forall v. S[E/V]) \quad (\forall s. S[E/V] = \exists v. S[E/V])\]
- Need lemma for expressions when $S$ is $p(E_1, \ldots, E_n)\$
  \[\text{Ssem}(p(E_1, \ldots, E_n)[E/V]) s = \text{Ssem}(p(E_1, \ldots, E_n)) (s[\text{Esem} E s/V])\]
- Equation true by induction and lemma for expressions
- Soundness of postcondition weakening similar

Precondition strengthening:

\[\vdash P \Rightarrow P', \quad \vdash \{P'\} C \{Q\}\]
\[\vdash \{P\} C \{Q\}\]

Soundness of Sequencing Rule

- Conditional rule:
  \[\vdash \{P\} C_1 \{Q\}, \quad \vdash \{Q\} C_2 \{R\}\]
  \[\vdash \{P\} C_1;C_2 \{R\}\]
- Sound if:
  \[\text{Hsem} P \land \text{Csem} C \Leftrightarrow \text{Qsem} Q\]
  \[\Rightarrow \forall s_1 s_2. \text{Ssem} P s_1 \land \exists s. \text{Csem} C s_1 s_2 \Rightarrow \text{Qsem} Q s_2\]
- An instance of the clearly true:
  \[\forall s_1 s_2. p s_1 \land c s_1 s_2 \Rightarrow q s_2\]
  \[\Rightarrow \forall s_1 s_2. p s_1 \land c s_1 s_2 \Rightarrow q s_2\]
- Soundness of conditional rule similar

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

\[\forall s_1 s_2. \text{Ssem} P s_1 \land \text{Csem} C s_1 s_2 \Rightarrow \text{Qsem} Q s_2\]

Assignment axiom:

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

Must show:

\[\forall s_1 s_2. \text{Ssem} (Q[E/V]) s_1 \land \text{Csem} (V = E) s_1 s_2 \Rightarrow \text{Qsem} Q s_2\]

Follows from substitution lemma for statements
Soundness of Hoare Logic: summary

- Assignment axiom:
  \[ \forall s_1 s_2. \text{Ssem} (Q[E/V]) s_1 \land \text{Csem} (V := E) s_1 s_2 \Rightarrow \text{Ssem} Q \{s_2\} \]

- Sound if:
  \[ \text{Hsem} (P \land S) C \Rightarrow \text{Hsem} (P \land S) \text{DO} C \{P \land \neg S\} \]

- Expands to:
  \[ (\forall s_1 s_2. \text{Ssem} (P \land S) s_1 \land \text{Csem} C s_1 s_2 \Rightarrow \text{Ssem} P s_1) \]
  \[ \Rightarrow \forall s_1 s_2. \text{Ssem} P s_1 \land \text{Csem} \text{WHILE} S \text{DO} C s_1 s_2 \Rightarrow \text{Ssem} (P \land \neg S) s_2 \]

- Expanding the definition of \( \text{Hsem} (\text{WHILE} S \text{DO} C) \) and simplifying:
  \[ (\forall s_1 s_2. \text{Ssem} P s_1 \land \text{Ssem} S s_1 \land \text{Csem} C s_1 s_2 \Rightarrow \text{Ssem} P s_1) \]
  \[ \Rightarrow \forall s_1 s_2. \text{Ssem} P s_1 \land (\exists n. \text{Iter} n \{S s_1\} \{C s_1 s_2\}) \]
  \[ \Rightarrow \text{Ssem} P s_2 \land \neg \{S s_2\} \]

- An instance of:
  \[ (\forall s_1 s_2. p s_1 \land b s_1 \land c s_1 s_2 \Rightarrow p s_1) \]
  \[ \Rightarrow \forall s_1 s_2. p s_1 \land (\exists n. \text{Iter} n b c s_1 s_2) \Rightarrow p s_2 \land \neg (b s_2) \]

Soundness of WHILE Rule

- From last slide need to prove:
  \[ (\forall s_1 s_2. p s_1 \land b s_1 \land c s_1 s_2 \Rightarrow p s_1) \]
  \[ \Rightarrow \forall s_1 s_2. p s_1 \land (\exists n. \text{Iter} n b c s_1 s_2) \Rightarrow p s_2 \land \neg (b s_2) \]

- This is equivalent to:
  \[ (\forall s_1 s_2. p s_1 \land b s_1 \land c s_1 s_2 \Rightarrow p s_1) \]
  \[ \Rightarrow \forall n s_1 s_2. p s_1 \land \text{Iter} n b c s_1 s_2 \Rightarrow p s_2 \land \neg (b s_2) \]

- Assume \( \forall s_1 s_2. p s_1 \land b s_1 \land c s_1 s_2 \Rightarrow p s_1 \), then prove:
  \[ \forall n s_1 s_2. p s_1 \land \text{Iter} n b c s_1 s_2 \Rightarrow p s_2 \land \neg (b s_2) \]
  by mathematical induction of \( n \)

- Routine using definition of \( \text{Iter} \):
  \[ \text{Iter} 0 p c s_1 s_2 \Rightarrow \neg (p s_1) \land (s_1 = s_2) \]
  \[ \text{Iter} (n+1) p c s_1 s_2 \Rightarrow p s_1 \land \exists s. c s_1 s \land \text{Iter} n p c s s_2 \]
  details in background reading

Completeness and decidability of Hoare Logic

- Soundness: \( \vdash \{P\}C\{Q\} \Rightarrow \vdash \{P\}C\{Q\} \)

- Decidability: \( \{T\}C\{F\} \Rightarrow C \text{ doesn't halt} \)
  - the Halting Problem is undecidable

- Completeness: really want \( \models_{\text{PA}} \{P\}C\{Q\} \Rightarrow \text{PA} \vdash \{P\}C\{Q\} \)
  - not possible
  - \( \models_{\text{PA}} \{T\}x := x\{P\} \) if and only if \( \models_{\text{PA}} P \)
  - \( \text{PA} \vdash \{T\}x := x\{P\} \) if and only if \( \text{PA} \vdash P \)

- If complete, as above, then for any statement \( P \):
  \[ \models_{\text{PA}} P \Rightarrow \models_{\text{PA}} \{T\}x := x\{P\} \Rightarrow \text{PA} \vdash \{T\}x := x\{P\} \Rightarrow \text{PA} \vdash P \]
  which can’t be by Gödel’s theorem

- Must separate completeness of programming and specification logics
Relative completeness (Cook 1978) – basic idea

- Assume \( \wp(C, Q) \) expressible in \( \mathcal{L} \) and \( \Gamma \models \{ \wp(C, Q) \} C \{ Q \} \), some \( \Gamma \)
- For simplicity take \( \mathcal{L} = \mathcal{L}_{PA} \) and \( \Gamma = \{ S \models \wp_{PA} S \} \)
- Show \( \models_{PA} \{ P \} C \{ Q \} \) entails \( \models_{PA} P \Rightarrow \wp(C, Q) \)
- Hence \( \models_{PA} \{ P \} C \{ Q \} \) entails \( \{ S \models \wp_{PA} S \} \vdash \{ P \} C \{ Q \} \)
  - assume \( \models_{PA} \{ P \} C \{ Q \} \)
  - then \( P \Rightarrow \wp(C, Q) \in \{ S \models \wp_{PA} S \} \)
  - by expressibility: \( \{ S \models \wp_{PA} S \} \vdash \{ \wp(C, Q) \} C \{ Q \} \)
  - hence by precondition strengthening: \( \{ S \models \wp_{PA} S \} \vdash \{ P \} C \{ Q \} \)
- Hoare logic is complete:
  - relative to \( \{ S \models \wp_{PA} S \} \)
  - assuming \( \wp(C, Q) \) is expressible

Discussion of proof of relative completeness

- Expressing \( \wp(C, Q) \) easy for assignments, sequences, conditionals
  \( \wp(V := E, Q) = Q[E/V] \)
  \( \wp(C_1 \land C_2, Q) = \wp(C_1, \wp(C_2, Q)) \)
  \( \wp(\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2, Q) = (S \land \wp(C_1, Q)) \lor (\neg S \land \wp(C_2, Q)) \)
- Expressing \( \wp(\text{WHILE } S \text{ DO } C), Q) \) is harder
  - tricky encoding in first-order arithmetic using Gödel’s \( \beta \) function
    (see Winskel's book *The formal semantics of programming languages: an introduction*)
- In the background reading
  - \( \wp(\text{WHILE } S \text{ DO } C), Q) \) defined using infinite conjunctions (expressibility)
  - \( \models_{PA} \{ P \} C \{ Q \} \) implies \( \models_{PA} P \Rightarrow \wp(C, Q) \) by induction on \( C \) and semantics
  - \( \{ S \models \wp_{PA} S \} \vdash \{ \wp(C, Q) \} C \{ Q \} \) by induction on \( C \) and Hoare logic
  - hence \( \models_{PA} \{ P \} C \{ Q \} \) implies \( \{ S \models \wp_{PA} S \} \vdash \{ P \} C \{ Q \} \)

Summary: soundness, decidability, completeness

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

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

- Clarke showed relative completeness fails for complex languages

Additional topics

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

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

Naive Array Assignment Axiom Fails

- The axiom
  \[
  \vdash \{P[A(E_1) ← E_2]\} \ A(E_1) := E_2 \ \{P\}
  \]
doesn’t work

- Take \( P \equiv \{X=Y \land A(X)=0\} \), \( E_1 \equiv \{X\} \), \( E_2 \equiv \{Y\} \)
  - since \( A(X) \) does not occur in \( P \)
  - it follows that \( P[1/A(X)] \equiv P \)
  - hence the axiom yields: \( \vdash \{X=Y \land A(Y)=0\} \ A(X) := 1 \ \{X=Y \land A(Y)=0\} \)

- 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

Array assignments

- Syntax: \( V(E_1) := E_2 \)

- Semantics: the state is changed by assigning the value of the term \( E_2 \) to the \( E_1 \)-th component of the array variable \( V \)

- Example: \( A(x+1) := A(x)+2 \)
  - if the value of \( x \) is \( x \)
  - and the value of the \( y \)-th component of \( A \) is \( n \)
  - then the value stored in the \((x+1)\)-th component of \( A \) becomes \( n+2 \)

Reasoning About Arrays

- The naive array assignment axiom
  \[
  \vdash \{P[A(E_1) ← E_2]\} \ A(E_1) := E_2 \ \{P\}
  \]
does not work: changes to \( A(X) \) may also change \( A(Y), A(Z), \ldots \)

- The solution, due to Hoare, is to treat an array assignment

\[
A(E_1) := E_2
\]
as an ordinary assignment

\[
A := A[E_1 ← E_2]
\]
where the term \( A[E_1 ← E_2] \) denotes an array identical to \( A \) except that the \( E_1 \)-th component is changed to have the value \( E_2 \)
Array Assignment axiom

- Array assignment is a special case of ordinary assignment
  \[ A := A{E_1 ← E_2} \]
- Array assignment axiom just ordinary assignment axiom
  \[ ⊢ \{ P[A{E_1 ← E_2}/A] \} A := A{E_1 ← E_2} \{ P \} \]
- Thus:

The array assignment axiom

\[ ⊢ \{ P[A{E_1 ← E_2}/A] \} A(E_i) := E_2 \{ P \} \]

Where \( A \) is an array variable, \( E_i \) is an integer valued expression, \( P \) is any statement and the notation \( A{E_1 ← E_2} \) 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 \( A{E_1 ← E_2} \), are needed

  \[ ⊢ A{E_1 ← E_2}(E_1) = E_2 \]
  \[ ⊢ E_1 ≠ E_2 \implies A{E_1 ← E_2}(E_3) = A(E_3) \]

- Second of these is a Frame Axiom
  - don’t confuse with Frame Rule of Separation Logic (later)
  - “frame” is a rather overloaded word!

Example

- We show
  \[ ⊢ \{ A(X)=x \land A(Y)=y \} \]
  \[ R := A(X); \]
  \[ A(X) := A(Y); \]
  \[ A(Y) := R \]
  \[ \{ A(X)=y \land A(Y)=x \} \]

  Working backwards using the array assignment axiom

- Second of these is a Frame Axiom

For more rigour define a first order theory ARRAY

- \( L_{ARRAY} = \{ \{ isarray \}, \{ lookup, update \} \} \)
  - \( isarray \) has arity 1, \( lookup \) has arity 2, \( update \) has arity 3
- \( I_{ARRAY} \)
  - domain is \( V ∪ \{ φ \mid φ : N → V \} \) for some set of values \( V \)
  - \( I_{ARRAY}[isarray](a) \) is true iff \( a \) is a function \( φ \)
  - \( I_{ARRAY}[lookup](a, i) = if a is a function φ then φ(i) else 0 \)
  - \( I_{ARRAY}[update](a, i, v) = if a is a function φ then φ[v/i] else a \)
- \( ARRAY \) contains the following axioms
  - \( ∀ a i v, isarray(a) ⇒ (lookup(update(a, i, v), i) = v) \)
  - \( ∀ a i j v, isarray(a) ⇒ (lookup(update(a, i, v), j) = lookup(a, j)) \)
  - \( ∀ a_i, a_j, isarray(a_1) \land ¬(i = j) ⇒ (lookup(update(a, i, v), i) = lookup(a, i)) \)
  - “\( a(i) \)” means “\( lookup(a, i) \)” and “\( a{[i ← v]} \)” means “\( update(a, i, v) \)”
- Assuming \( a \) is an array \( isarray(a) \) is true then from array axioms:
  - \( a{[i ← v]}(i) = v \)
  - \( ¬(i = j) ⇒ (a{[i ← v]}(j) = a(j)) \)
Example Continued (1)

- Using
  \[ \vdash A(Y \leftarrow R)(Y) = R \]
- It follows that
  \[ \vdash \{ (A(Y \leftarrow R))(X) = y \land R = x \} \]
  \[ A(Y) := R \]
  \[ \{ A(X) = y \land \ldots \text{intuitive if the assignment is rewritten} \} \]
- Continuing backwards
  \[ \vdash \{ ((A(X \leftarrow A(Y))(Y \leftarrow R))(X) = y \land R = x \} \]
  \[ A := A(X \leftarrow A(Y)) \]
  \[ \{ (A(Y \leftarrow R))(X) = y \land R = x \} \]

Example Continued (2)

- Continuing backwards
  \[ \vdash \{ ((A(X \leftarrow A(Y))(Y \leftarrow A(X))(X) = y \land A(X) = x \} \]
  \[ R := \ldots = A(Y) \]
- Hence (as desired)
  \[ \vdash \{ A(Y) = y \land A(X) = x \} \]
  \[ R := A(X); A(X) := A(Y); A(Y) := R \]
  \[ \{ A(X) = y \land A(Y) = x \} \]

Blocks and local variables

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

\[ \vdash \{ P \} \begin{array}{l} C \end{array} \{ Q \} \]

\[ \vdash \{ P \} \text{BEGIN VAR } V_1; \ldots; V_n; C \text{ END } \{ Q \} \]

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

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

  it follows by the block rule that

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

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

- However from

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

  one cannot deduce

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

  since $R$ occurs in $R=x \land X=y$.

FOR-commands

- Syntax: $\text{FOR } V := E_1 \ \text{UNTIL } E_2 \ \text{DO } C$
  - restriction: $V$ must not occur in $E_1$ or $E_2$,
    or be the left hand side of an assignment in $C$
    (explained later)

- Semantics:
  - if the values of terms $E_1$ and $E_2$ are positive numbers $e_1$ and $e_2$
    - and if $e_1 \leq e_2$
    - then $C$ is executed $(e_2-e_1)+1$ times with the variable $V$ taking on the sequence of values $e_1, e_1+1, \ldots, e_2$ in succession
    - for any other values, the FOR-command has no effect

- Example: $\text{FOR } N:=1 \ \text{UNTIL } M \ \text{DO } X:=X+N$
  - if the value of the variable $N$ is $m$ and $m \geq 1$, then the command $X:=X+N$ is repeatedly executed with $N$ taking the sequence of values $1, \ldots, m$
  - if $m < 1$ then the FOR-command does nothing

Exercises

- Consider the specification

  $\{X=x\} \ \text{BEGIN VAR} \ X; \ X:=1 \ \text{END} \ \{X=x\}$

  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?

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

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

- Show

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

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 \texttt{FOR}-commands

- The semantics of
  \[\texttt{FOR } V := E_1 \texttt{ UNTIL } E_2 \texttt{ DO } C\]

  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 \texttt{FOR}-command is equivalent to:

  \[
  \begin{align*}
  \text{BEGIN } & \text{VAR } V; \quad V := e_1; \quad \ldots \quad C; \quad V := e_2; \quad \text{END}
  \end{align*}
  \]

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

- If \(C\) doesn't modify \(V\) then \texttt{FOR}-command equivalent to:

  \[
  \begin{align*}
  \text{BEGIN } & \text{VAR } V; \quad V := e_1; \quad \ldots \quad C; \quad V := V + 1\quad \ldots \quad V := e_2; \quad \text{END} \quad \text{repeated}
  \end{align*}
  \]

Towards the \texttt{FOR}-Rule

- If \(e_1 \leq e_2\) the \texttt{FOR}-command is equivalent to:

  \[
  \begin{align*}
  \text{BEGIN } & \text{VAR } V; \quad V := e_1; \quad \ldots \quad C; \quad V := V + 1; \quad \ldots \quad V := e_2; \quad \text{END}
  \end{align*}
  \]

- Assume \(C\) doesn't modify \(V\) and \(\vdash \{P\} C \{P(V+1/V)\}\)

  Hence:

  \[
  \begin{align*}
  & \vdash \{P[e_1/V]\} V := e_1 \{P \land V = e_1\} \quad \text{(assign. ax + pre. streng.)} \\
  & \vdash \{P \land V = v\} C; \quad V := V + 1 \{P \land V = v + 1\} \quad \text{(last slide; } V = e_1, e_1+1, \ldots, e_2-1\} \\
  & \vdash \{P \land V = e_2\} C \{P[V+1/V] \land V = e_2\} \quad \text{(assumption + constancy rule)} \\
  & \vdash \{P \land V = e_2\} C \{P[e_2+1/V]\} \quad \text{(post. weak.)}
  \end{align*}
  \]

  Hence by the sequencing and block rules

  \[
  \vdash \{P\} C \{P[V+1/V]\}
  \]

  \[
  \vdash \{P[e_1/V]\} \text{BEGIN } V := e_1; \quad \ldots \quad C; \quad V := V + 1; \quad \ldots \quad V := e_2; \quad \text{END} \{P[e_2+1/V]\}
  \]

The Rule of Constancy (Derived Frame Rule)

- The following derived rule is used on the next slide

  \[
  \begin{align*}
  & \vdash \{P\} C \{Q\} \\
  & \vdash \{P \land R\} C \{Q \land R\} \\
  \end{align*}
  \]

  where no variable assigned to in \(C\) occurs in \(R\)

  Outline of derivation

  - prove \(\{R\} C \{R\}\) by induction on \(C\)
  - then use Specification Conjunction

  Assume \(C\) doesn't modify \(V\) and \(\vdash \{P\} C \{P(V+1/V)\}\) then:

  \[
  \begin{align*}
  & \vdash \{P \land V = v\} C \{P[V+1/V] \land V = v\} \quad \text{(assumption + constancy rule)} \\
  & \vdash \{P[V+1/V] \land V = v\} V := V + 1 \{P \land V = v + 1\} \quad \text{(assign. ax + pre. streng.)} \\
  & \vdash \{P \land V = v\} C; \quad V := V + 1 \{P \land V = v + 1\} \quad \text{(sequencing)}
  \end{align*}
  \]

  So \(C; V := V + 1\) has \(P\) as an invariant and increments \(V\)

Problems with the \texttt{FOR}-rule (i)

- Previous derivation suggests a rule

  \[
  \begin{align*}
  & \vdash \{P\} C \{P[V+1/V]\} \\
  & \vdash \{P[E_i/V]\} \text{FOR } V := E_i \texttt{ UNTIL } E_2 \texttt{ DO } C \{P[E_2+1/V]\}
  \end{align*}
  \]

  - This is a good start, but needs debugging

  Consider:

  \[
  \begin{align*}
  & \vdash \{X = Y\} X := Y + 1 \quad \{X = Y + 1\}
  \end{align*}
  \]

  Taking \(P\) as ‘\(X = Y\)’ this is:

  \[
  \begin{align*}
  & \vdash \{P\} X := Y + 1 \quad \{P[Y + 1/Y]\}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  & \vdash \{X \leq 3\} \text{ FOR } Y := 3 \texttt{ UNTIL } 1 \texttt{ DO } X := Y + 1 \quad \{X \leq 2\} \\
  & \quad \{X \leq 2\} \quad \text{FOR } Y := 3 \texttt{ UNTIL } 1 \texttt{ DO } X := Y + 1 \quad \{X \leq 2\}
  \end{align*}
  \]

  \[
  \begin{align*}
  & \quad \{X \leq 2\} \quad \text{FOR } Y := 3 \texttt{ UNTIL } 1 \texttt{ DO } X := Y + 1 \quad \{X \leq 2\}
  \end{align*}
  \]

  \[
  \begin{align*}
  & \quad \{X \leq 2\} \quad \text{FOR } Y := 3 \texttt{ UNTIL } 1 \texttt{ DO } X := Y + 1 \quad \{X \leq 2\}
  \end{align*}
  \]
Problems with the FOR-rule (ii)

- The conclusion below is clearly undesirable

\[ \vdash \{ X=3 \} \text{ FOR } Y:=3 \text{ UNTIL } X:=Y+1 \{ \begin{array}{l} X=2 \\ Y:=Y+1 \end{array} \} \]

\[ \vdash \{ P \} C \{ P\frac{[V+1]}{V} \} \]

- 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

\[ \vdash \{ P \} C \{ P\frac{[V+1]}{V} \} \]

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

Problems with the FOR-rule (iii)

- FOR-rule so far

\[ \vdash \{ P \} C \{ P\frac{[V+1]}{V} \} \]

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

- On the example just considered this rule results in

\[ \vdash \{ X=3 \wedge \exists Y \text{ s.t. } Y=3 \} \text{ FOR } Y:=3 \text{ UNTIL } X:=Y+1 \{ X=2 \} \]

- 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

\[ \vdash \{ P \} C \{ P\frac{[V+1]}{V} \} \]

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

- Take \( P \) to be ‘\( Y=1 \)’ and note that

\[ \vdash \{ Y=1 \} Y:=Y-1 \{ \begin{array}{l} Y+1 \leq 1 \\ P\frac{[Y+1]}{Y} \end{array} \} \]

- So by this FOR-rule

\[ \vdash \{ X=3 \wedge 1 \leq 1 \} \text{ FOR } Y:=1 \text{ UNTIL } Y:=Y-1 \{ \begin{array}{l} 2 \wedge 1 \\ P\frac{[1]}{Y} \wedge P\frac{[1+1]}{Y} \end{array} \} \]

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
  \[ \vdash \{Z=Y\} Z:=Z+1 \{ \begin{array}{ll} Z=Y+1 \end{array} \} \]
  \[ P \]
  \[ P[Y+1/Y] \]
- Thus the following can be derived
  \[ \vdash \{ Z=Y \land 1 \leq Z \} \text{FOR } Y:=1 \text{ UNTIL } Z \text{ DO } Z:=Z+1 \{ \begin{array}{ll} Z=Z+1 \end{array} \} \]
  \[ P[1/Y] \]
  \[ P[Z+1/Y] \]
- 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

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

**The FOR-Rule**

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

**The FOR-axiom**

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

  **The FOR-axiom**
  \[ \vdash \{ P \land (E_1 \leq E_2) \} \text{FOR } V:=E_1 \text{ UNTIL } E_2 \text{ DO } C \{ P \} \]
- This says that when $E_2$ is less than $E_1$ the FOR-command has no effect
Exercise: understand the example on this slide

- By the assignment axiom and precondition strengthening
  \[ \vdash (X = ((N-1) \times N) \div 2) \]
  \[ X := X + N \quad \{ X = ((N \times (N+1)) \div 2) \} \]

- Strengthening the precondition of this again yields
  \[ \vdash (X = ((N-1) \times N) \div 2) \land (1 \leq N) \land (N \leq M) \]
  \[ X := X + N \quad \{ X = (N \times (N+1)) \div 2 \} \]

- Hence by the FOR-rule
  \[ \vdash (X = 0) \land (1 \leq N) \]
  \[ \text{FOR } N := 1 \text{ UNTIL } M \text{ DO } X := X + N \quad \{ X = (N \times (N+1)) \div 2 \} \]

- Hence
  \[ \vdash (X = 0) \land (1 \leq N) \]
  \[ \text{FOR } N := 1 \text{ UNTIL } M \text{ DO } X := X + N \quad \{ X = (N \times (N+1)) \div 2 \} \]

Deriving the FOR Rule

- The following is a command equivalent to \( \text{FOR } I := E_1 \text{ UNTIL } E_2 \text{ DO } C \)

  \[
  \begin{align*}
  \text{BEGIN} \\
  \text{VAR } I; \\
  \text{VAR } \text{UpperBound}; \\
  I := E_1; \\
  \text{UpperBound} := E_2; \\
  \text{WHILE } I \leq \text{UpperBound} \text{ DO } (C; I := I + 1) \\
  \text{END}
  \end{align*}
  \]

  - 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

Note on using the FOR-Rule

- Note that if any of the following hold
  1. \( \vdash \{ P \} C \{ P[V + 1/V] \} \)
  2. \( \vdash \{ P \land (E_1 \leq V) \} C \{ P[V + 1/V] \} \)
  3. \( \vdash \{ P \land (V \leq E_2) \} C \{ P[V + 1/V] \} \)

  - Then by precondition strengthening:
    \[ \vdash \{ P \land (E_1 \leq V) \land (V \leq E_2) \} C \{ P[V + 1/V] \} \]

  - So any of (i), (ii) or (iii) above is a sufficient hypothesis for FOR Rule
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:
  1. define the language by the axioms and rules of the logic
  2. 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. \( Q \Rightarrow Q' \land (A/h) \)
2. \( P[\text{const } K; \text{ var } V; S_i] Q(A/h) \land Q \)
3. \( P^2(A/h) \land Q[S_i] Q^2(A/h) \land Q \)
4. \( g \left( P^3(A/h) \land Q[S_i] Q^3(A/h) \land g = g(A, c, d) \right) \)
5. \( g \left( P^3(A/h) \land Q \Rightarrow Q^3(A/h) \right) \)
6. \( P^6(A/h) \land Q[\text{var } V; S_i] Q \)
7. \( \forall \theta P(h, \theta) \)\)
8. \( \left( Q0(a/c, x/h, x'[h]) = \left( P^2(x/h, x'[h], a_2/x_2, c_2/c_2, a/c) \right) \land \left( Q^2(x_2/h', x'[h'], a_2/x_2, c_2/c_2, a/c, y_2/y_2, a_2/x_2', x_2'[y_2')] \Rightarrow \left( R_1(x_2'[y_2'], a_2/a_2, y_2/y_2) \right) \right) \right) \land Q_0(a/c, x/h, x'[h]) \)
8. \( \left( Q0(a/c, x/h) \Rightarrow P^3(x/h, x'[h], a/c) \land Q^3(a/c, x/h, x'[h]) \right) \)\)
8. \( P^3(a/c) \land \left( Q^3(x_1/h, x_1'[h], a/c, y_3/y_3) \Rightarrow \left( R^3(x_1/h, x_1'/h', y_3/y_3) \right) \right) \)\)
8. \( Q_0(a/c, x/h) \Rightarrow P^3(a/c, x/h, x'[h], a/c) \land \left( Q_1(a/c, y_3/y_3) \Rightarrow \left( R_1(y_3/y_3) \right) \right) \)\)
8. \( P(x/h) \land x. \text{ Initially; } S; x. \text{ Finally; } R(x/h) \)

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 errors are spotted earlier in the design process.
  - the reasons for design decisions are available.
- Programming becomes less of a black art and more like an engineering discipline.
- Rigorous development methods such as the B-Method, SPARK and the Vienna Development Method (VDM) are based on this idea.
- The approach here is based on “Programming From Specifications”:
  - a book by Carroll Morgan.
  - simplified and with a more concrete semantics.
Refinement Laws

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

Refinement Specifications

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

Example

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

A Little Wide Spectrum Programming Language

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

\[
E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots
\]
\[
B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots
\]
\[
C ::= \text{SKIP} \quad \text{(does nothing, SKIP-Axiom is } \vdash [P \text{ SKIP } P])
\]
  | \(V := E\)
  | \(C_1 ; C_2\)
  | \(\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2\)
  | \(\text{BEGIN VAR } V_1 ; \ldots \text{ VAR } V_n ; C \text{ END}\)
  | \([P, Q]\)
Specifications as Sets of Commands

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

- Example
  \[ R:=X; 
  Q:=0; 
  \{ [ R=X \land Y>0 \land Q=0, \ X=R+Y \times Q]} \]

- Think of a specification as defining the set of implementations
  \[ [P, Q] = \{ C \mid \vdash [P] C [Q] \} \]

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

- Don’t confuse use of \{\ldots\} as set brackets and in Hoare triples.

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 \( p_1 \sqsubseteq p_2 \) 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.
  \[ spec \sqsupseteq mixed_1 \sqsupseteq \ldots \sqsupseteq \text{mixed_n} \sqsupseteq \text{code} \]

Notation for combining sets of commands

- Wide spectrum language commands are sets of ordinary commands.
- Let \( c_1, c_2, \ldots \) denote sets of commands, then define:
  \[ c_1; \ldots ; c_n = \{ C \mid \exists c_1 \ldots c_n : C = C_1; \ldots ; C_n \land C_1 \in c_1 \land \ldots \land C_n \in c_n \} \]

Skip Law

The Skip Law
\[ [P, P] \sqsupseteq \{ \text{SKIP} \} \]

- Derivation:
  \[ C \in \{ \text{SKIP} \} \]
  \[ \iff C = \text{SKIP} \]
  \[ \iff \vdash [P] C [P] \text{ (Skip Axiom)} \]
  \[ \iff C \in [P, P] \text{ (Definition of } [P, P]) \]

- Examples
  \[ [X=1, X=1] \sqsupseteq \{ \text{SKIP} \} \]
  \[ [T, T] \sqsupseteq \{ \text{SKIP} \} \]
  \[ [X=R+Y \times Q, X=R+Y \times Q] \sqsupseteq \{ \text{SKIP} \} \]
Notational Convention

- Omit { and } around individual commands
- Skip law becomes: 
  \[ P, P \uparrow \subseteq \text{SKIP} \]
- Examples become:
  \[ X=1, X=1 \uparrow \subseteq \text{SKIP} \]
  \[ T, T \uparrow \subseteq \text{SKIP} \]
  \[ X=R+Y \times Q, X=R+Y \times Q \uparrow \subseteq \text{SKIP} \]

Assignment Law

The Assignment Law

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

- Derivation
  \[ C \in \{ V := E \} \iff C = V := E \]
  \[ \vdash P \Rightarrow R \]
  \[ \vdash P[E/V], P \]
  \[ \vdash C \in [P[E/V], P] \text{ (Assignment Axiom)} \]
  \[ \vdash C \in [P[E/V], P] \text{ (Definition of } [P[E/V], P] \text{)} \]

- Examples
  \[ Y=1, X=1 \uparrow \supseteq X:=Y \]
  \[ X+1=n+1, X=n+1 \uparrow \supseteq X:=X+1 \]

Laws of Consequence

Precondition Weakening

\[ [P, Q] \uparrow \supseteq [R, Q] \]
provided \( \vdash P \Rightarrow R \)

Postcondition Strengthening

\[ [P, Q] \uparrow \supseteq [P, R] \]
provided \( \vdash R \Rightarrow Q \)

- We are now “weakening the precondition” and “strengthening the post condition”
  - this is the opposite terminology to the Hoare rules
  - refinement consequence rules are ‘backwards’

Derivation of Consequence Laws

- Derivation of Precondition Weakening
  \[ C \in [R, Q] \]
  \[ \iff \vdash [R] C [Q] \text{ (Definition of } [R, Q] \text{)} \]
  \[ \Rightarrow \vdash [P] C [Q] \text{ (Precondition Strengthening } \vdash P \Rightarrow R \text{)} \]
  \[ \iff C \in [P, Q] \text{ (Definition of } [P, Q] \text{)} \]

- Derivation of Postcondition Strengthening
  \[ C \in [P, R] \]
  \[ \iff \vdash [P] C [R] \text{ (Definition of } [P, R] \text{)} \]
  \[ \Rightarrow \vdash [P] C [Q] \text{ (Postcondition Weakening } \vdash R \Rightarrow Q \text{)} \]
  \[ \iff C \in [P, Q] \text{ (Definition of } [P, Q] \text{)} \]
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. \( \supseteq \)
  - if \( c \supseteq c' \), \( c_1 \supseteq c'_1 \), \ldots, \( c_n \supseteq c'_n \)
  - then:
    \[
    \begin{align*}
    &c_1; \ldots; c_n \\
    \text{BEGIN VAR } V_1; \ldots \text{ VAR } V_n; \ c \text{ END} \\
    \text{IF } S \text{ THEN } c_1; \ldots; \text{ ELSE } c' \ldots; \text{ ELSE } c'_n \\
    \text{WHILE } S \text{ DO } c \\
    \end{align*}
    \]
    \( \supseteq \)\[
    \begin{align*}
    &c'_1; \ldots; c'_n \\
    \text{BEGIN VAR } V_1; \ldots \text{ VAR } V_n; \ c' \text{ END} \\
    \text{IF } S \text{ THEN } c'_1; \ldots; \text{ ELSE } c'_2 \\
    \text{WHILE } S \text{ DO } c' \\
    \end{align*}
    \]

- Laws of refinement for non-atomic commands now follow

Examples (illustrates refinement notation)

- A previous example:
  \[\begin{align*}
  &\begin{align*}
  &X=1, \ X=1 \\
  \supseteq &\text{(Skip)} \\
  \end{align*}
  \end{align*}\]

- An alternative refinement:
  \[\begin{align*}
  &\begin{align*}
  &Y=1, \ X=1 \\
  \supseteq &\text{(Precondition Weakening} \ \vdash \ \ Y=1 \ \Rightarrow \ 1=1) \\
  \supseteq &\text{(Assignment)} \\
  \end{align*}
  \end{align*}\]

- Another example
  \[\begin{align*}
  &\begin{align*}
  &T, \ R=X \\
  \supseteq &\text{(Precondition Weakening} \ \vdash \ \ T \ \Rightarrow \ X=X) \\
  \supseteq &\text{(Assignment)} \\
  \end{align*}
  \end{align*}\]

Derived Assignment Law

\[\begin{align*}
&P, Q \supseteq [V := E] \\
&\text{provided} \ \vdash P \Rightarrow Q[E/V] \\
\end{align*}\]

- Derivation
  \[\begin{align*}
  &\begin{align*}
  &P, Q \\
  \supseteq &\text{(Precondition Weakening} \ \vdash \ P \Rightarrow Q[E/V]) \\
  \supseteq &\text{(Assignment)} \\
  \end{align*}
  \end{align*}\]

- Example
  \[\begin{align*}
  &\begin{align*}
  &T, R=X \\
  \supseteq &\text{(Derived Assignment} \ \vdash \ T \Rightarrow X=X) \\
  \supseteq &\text{(Assignment)} \\
  \end{align*}
  \end{align*}\]

Sequencing

The Sequencing Law

\[\begin{align*}
&P, Q \supseteq [P, R]; [R, Q] \\
\end{align*}\]

- Derivation of Sequencing Law
  \[\begin{align*}
  &C \in [P, R]; [R, Q] \\
  \Rightarrow &C \in \{ C_1 ; C_2 | C_1 \in [P, R] \land C_2 \in [R, Q] \} \quad \text{(Definition of } C_1 \land C_2) \\
  \Rightarrow &C \in \{ C_1 ; C_2 | \vdash [P] C_1 | R \land \vdash [R] C_2 | Q \} \quad \text{(Definition of } [P, R] \text{ and } [R, Q]) \\
  \Rightarrow &\vdash [P] C \subseteq [Q] \quad \text{(Sequencing Rule)} \\
  \Rightarrow &C \in [P, Q] \quad \text{(Definition of } [P, Q]) \\
\end{align*}\]

- Example
  \[\begin{align*}
  &\begin{align*}
  &T, R=X \land Q=0 \\
  \supseteq &\text{(Sequencing)} \\
  \supseteq &\text{(Derived Assignment} \ \vdash \ T \Rightarrow X=X) \\
  \supseteq &\text{(Assignment)} \\
  \supseteq &\text{(Derived Assignment} \ \vdash \ R=X \Rightarrow R=X \land 0=0) \\
  \supseteq &\text{(Assignment)} \\
  \end{align*}
  \end{align*}\]
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

\[ [T, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{Sequencing}) \]
\[ [T, \ Q=0] ; [Q=0, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{Derived Assignment} \vdash T \Rightarrow 0=0) \]
\[ Q:=0; [Q=0, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{Derived Assignment} \vdash Q=0 \Rightarrow X=X \land Q=0) \]
\[ Q:=0; R:=X \]

Inefficient Programs

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

Example

\[ [T, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{Sequencing}) \]
\[ [T, \ R=X \land Q=0] ; [R=X \land Q=0, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{as previous example}) \]
\[ Q:=0; R:=X; [R=X \land Q=0, \ R=X \land Q=0] \]
\[ \supseteq \ (\text{Skip}) \]
\[ Q:=0; R:=X; \text{SKIP} \]

Blind Alleys

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

Example

\[ [X=x \land Y=y, \ X=y \land Y=x] \]
\[ \supseteq \ (\text{Sequencing}) \]
\[ [X=x \land Y=y, \ X=x \land Y=x]; [X=x \land Y=x, \ X=y \land Y=x] \]
\[ \supseteq \ (\text{Derived Assignment} \vdash X=x \land Y=y \Rightarrow X=x \land X=x) \]
\[ Y:=X; [X=x \land Y=x, \ X=y \land Y=x] \]
\[ \supseteq \ (\text{Sequencing}) \]
\[ Y:=X; \]
\[ [X=x \land Y=x, \ Y=y \land Y=x]; [Y=y \land Y=x, \ X=y \land Y=x] \]
\[ \supseteq \ (\text{Derived Assignment}) \]
\[ Y:=X; \]
\[ [X=x \land Y=x, \ Y=y \land Y=x]; X:=Y \]

Blocks

The Block Law

\[ [P, \ Q] \supseteq \text{BEGIN VAR } V_1; \ldots; \text{VAR } V_n; [P, \ Q] \text{ END} \]
where \( V_1, \ldots, V_n \) do not occur in \( P \) or \( Q \)

- Derivation: exercise

Example

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

\[ [P, Q] \supseteq IF S THEN [P \land S, Q] \quad ELSE \quad [P \land \neg S, Q] \]

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

Example

\[ [T, M=\text{max}(X,Y)] \supseteq \text{(Derived Assignment)} \quad \text{IF} \quad X \geq Y \quad \text{THEN} \quad [T \land X \geq Y, M=\text{max}(X,Y)] \quad \text{ELSE} \quad [T \land \neg (X \geq Y), M=\text{max}(X,Y)] \]

- \text{Derived Assignment} \quad \vdash \quad T \land X \geq Y \quad \Rightarrow \quad X=\text{max}(X,Y)
- \text{IF} \quad X \geq Y \quad \text{THEN} \quad M:=X
- \text{ELSE} \quad [T \land \neg X \geq Y, M=\text{max}(X,Y)]
- ( Derived Assignment ) \quad \vdash \quad T \land \neg X \geq Y \quad \Rightarrow \quad Y=\text{max}(X,Y)
- \text{IF} \quad X \geq Y \quad \text{THEN} \quad M:=X \quad \text{ELSE} \quad M:=Y

Derivation of the Conditional Law

\[ C \in IF S THEN [P \land S, Q] \quad ELSE \quad [P \land \neg S, Q] \]

\( \Leftrightarrow \quad C \in \{ IF S THEN C_1 ELSE C_2 \quad | \quad C_1 \in [P \land S, Q] \quad \text{AND} \quad C_2 \in [P \land \neg S, Q] \} \quad \) (Two-armed Conditional Rule)

\( \quad \vdash \quad [P \land S] \quad C_1 \quad \text{AND} \quad \vdash \quad [P \land \neg S] \quad C_2 \quad \) (Definition of \( [P \land S, Q] \quad \text{AND} \quad [P \land \neg S, Q] \))

\( \quad \Rightarrow \quad \vdash \quad [P] \quad C \quad [Q] \quad \) (Definition of \( [P, Q] \))

While

The While Law

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

provided \( \vdash \quad R \land S \quad \Rightarrow \quad E \geq 0 \)

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

- Example

\[ \quad [X=R+Y \times Q \quad \land \quad Y>0, \quad X=R+Y \times Q \quad \land \quad Y>0 \quad \land \quad \neg \quad Y \leq R] \]

\( \quad \supseteq \quad \text{(While)} \quad \vdash \quad X=R+Y \times Q \quad \land \quad Y>0 \quad \land \quad Y \leq R \quad \Rightarrow \quad R \geq 0 \)

\[ \quad \text{WHILE} \quad Y \leq R \quad \text{DO} \quad [X=R+Y \times Q \quad \land \quad Y>0 \quad \land \quad Y \leq R \quad \land \quad R=n, \quad X=R+Y \times Q \quad \land \quad Y>0 \quad \land \quad R<n] \]
Derivation of the While Law

\[ C \in \text{WHILE } S \text{ DO } [P \land S \land (E = n), \ P \land (E < n)] \]
\[ \Leftrightarrow C \in \{ \text{WHILE } S \text{ DO } C' \mid C' \in [P \land S \land (E = n), \ P \land (E < n)] \} \quad \text{(Definition of WHILE } S \text{ DO \{ \ldots \})} \]
\[ C \in \{ \text{WHILE } S \text{ DO } C' \mid \vdash [P \land S \land (E = n)] C' [P \land (E < n)] \} \quad \text{(Definition of } C') \]
\[ \vdash [P] C [P \land \neg S] \quad \text{(While Rule & } \vdash P \land S \Rightarrow E \geq 0) \]
\[ \vdash [P] C [P \land \neg S] \quad \text{(Definition of } [P \land \neg S]) \]

Example (i)

\[ [Y > 0, \ X = R + Y \times Q \land R \leq Y] \]
\[ \supseteq \text{(Block)} \]
BEGIN \[ Y > 0, \ X = R + Y \times Q \land R \leq Y \] END
\[ \supseteq \text{(Sequencing)} \]
BEGIN \[ Y > 0, \ R = X \land Y > 0 \] \[ \supseteq \text{(Derived Assignment)} \vdash Y > 0 \Rightarrow X = X \land Y > 0 \]
BEGIN \[ R := X \; \]
\[ [R = X \land Y > 0, \ X = R + Y \times Q \land R \leq Y] \]
END

Example (ii)

BEGIN \[ R := X \; \]
\[ [R = X \land Y > 0, \ X = R + Y \times Q \land R \leq Y] \]
\[ \supseteq \text{(Derived Assignment)} \vdash Y > 0 \Rightarrow X = X \land Y > 0 \; \]
BEGIN \[ R := X \; \]
\[ Q := 0 \; \]
\[ [R = X \land Y > 0 \land Q = 0, \ X = R + Y \times Q \land R \leq Y] \]
END

Example (iii)

\[ \supseteq \left( \begin{array}{l}
\text{Precondition Weakening} \\
\vdash R = X \land Y > 0 \land Q = 0 \Rightarrow X = R + Y \times Q \land Y > 0
\end{array} \right) \]
BEGIN \[ R := X; \ Q := 0; \]
\[ [X = R + Y \times Q \land Y > 0, \ X = R + Y \times Q \land R \leq Y] \]
END
\[ \supseteq \left( \begin{array}{l}
\text{Postcondition Strengthening} \\
\vdash R = X \land Y > 0 \land \neg (Y \leq R) \Rightarrow X = R + Y \times Q \land R \leq Y
\end{array} \right) \]
BEGIN \[ R := X; \ Q := 0; \]
\[ [X = R + Y \times Q \land Y > 0, \ X = R + Y \times Q \land Y > 0 \land \neg (Y \leq R)] \]
END
\[ \supseteq \left( \begin{array}{l}
\text{While} \\
\vdash X = R + Y \times Q \land Y > 0 \land Y \leq R \Rightarrow R \geq 0
\end{array} \right) \]
BEGIN \[ R := X; \ Q := 0; \]
WHILE \[ Y \leq R \] DO
\[ [X = R + Y \times Q \land Y > 0 \land Y \leq R \land R \geq 0, \]
\[ = X = R + Y \times Q \land Y > 0 \land R \geq 0] \]
END

Exercise: complete the refinement (see next few slides)
The notation 
\[P_1, P_2, P_3, \ldots, P_{n-1}, P_n\]
is used to abbreviate:
\[P_1, P_2\] ; \[P_2, P_3\] ; \ldots ; \[P_{n-1}, P_n\]

- Brackets around specifications \{C\} omitted
- If \(C\) is a set of commands, then
  \(R := X ; \{C\}\)
  abbreviates
  \[\{R := X\} ; C\]

Example (iv)
\[\begin{array}{l}
  \text{BEGIN} \\
  R := X; Q := 0; \\
  \text{WHILE } Y \leq R \text{ DO} \\
  \quad \begin{cases} \\
    X = R + Y \times Q \wedge Y > 0 \wedge Y \leq R \wedge R = n, \\
    X = (R - Y) + Y \times (Q + 1) \wedge Y > 0 \wedge (R - Y) < n \\
  \end{cases} \\
  \quad Q := Q + 1; \\
  R := R - Y \\
  \end{array}
\]

Example (v)
\[\begin{array}{l}
  \text{BEGIN} \\
  R := X; Q := 0; \\
  \text{WHILE } Y \leq R \text{ DO} \\
  \quad \begin{cases} \\
    X = R + Y \times Q \wedge Y > 0 \wedge Y \leq R \wedge R = n, \\
    X = (R - Y) + Y \times (Q + 1) \wedge Y > 0 \wedge (R - Y) < n \\
  \end{cases} \\
  \quad Q := Q + 1; \\
  R := R - Y \\
  \end{array}
\]

Exercise: check the refinement on this slide
- Let \(I\) stand for \(X = R + (Y \times Q)\), then:
  \[\begin{array}{l}
   [Y \geq 0, I \wedge R \leq Y]\end{array}\]
  2 (Sequencing)
  \[\begin{array}{l}
   [Y = 0, R = X \wedge Y > 0 \wedge I \wedge R \leq Y] \\
  \end{array}\]
  2 (Assignment)
  \[\begin{array}{l}
   R = X; [R = X \wedge Y > 0, I \wedge R \leq Y] \\
  \end{array}\]
  2 (Sequencing)
  \[\begin{array}{l}
   R := X; Q := 0; [R = X \wedge Y > 0 \wedge Q = 0, I \wedge R \leq Y] \\
  \end{array}\]
  2 (Assignment)
  \[\begin{array}{l}
   R := X; Q := 0; [I \wedge Y > 0, I \wedge R \leq Y] \\
  \end{array}\]
  2 (Postcondition Strengthening)
  \[\begin{array}{l}
   R := X; Q := 0; [I \wedge Y > 0, I \wedge R \leq Y] \\
  \end{array}\]
  2 (While)
  \[\begin{array}{l}
   R := X; Q := 0; \\
   \text{WHILE } Y \leq R \text{ DO} [Z \wedge Y > 0 \wedge Y \leq R \wedge R = n, \end{array}\]
  2 (Sequencing)
  \[\begin{array}{l}
   Z \wedge Y > 0 \wedge R < n] \\
  \end{array}\]
  2 (Derived Assignment)
  \[\begin{array}{l}
   R := X; Q := 0; \\
   \text{WHILE } Y \leq R \text{ DO} [Z \wedge Y > 0 \wedge Y \leq R \wedge R = n, \end{array}\]
  2 (Derived Assignment)
  \[\begin{array}{l}
   X = (R - Y) + (Y \times Q) \wedge Y > 0 \wedge (R - Y) < n; \end{array}\]
  2 (Derived Assignment)
  \[\begin{array}{l}
   R := R - Y \\
  \end{array}\]
  2 (Derived Assignment)
  \[\begin{array}{l}
   R := X; Q := 0; \\
   \text{WHILE } Y \leq R \text{ DO} [Q = Q + 1; R := R - Y \\
   \end{array}\]
  2 (Derived Assignment)
  \[\begin{array}{l}
   R := X; Q := 0; \\
   \text{WHILE } Y \leq R \text{ DO} [Q = Q + 1; R := R - Y \\
   \end{array}\]
  2 (Derived Assignment)
**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-command

**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 → 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

<table>
<thead>
<tr>
<th>X</th>
<th>l₁</th>
<th>l₂</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td>heap</td>
<td>heap</td>
<td></td>
</tr>
</tbody>
</table>

Heap semantics

\[\text{Store} = \text{Var} → \text{Val} \quad \text{(assume \text{Num} ⊆ \text{Val}; \text{nil} ∈ \text{Val} and \text{nil} ∉ \text{Num})}\]
\[\text{Heap} = \text{Num} → \text{Val}\]
\[\text{State} = \text{Store} × \text{Heap}\]

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

Adding pointer operations to our language

Expressions:

\[E::= N | V | E₁ + E₂ | E₁ − E₂ | E₁ × E₂ \ | \ldots\]

Boolean expressions:

\[B::= T | F | E₁ = E₂ | E₁ ≤ E₂ \ | \ldots\]

Commands:

\[C::= V := E \quad \text{value assignments}\]
\[V := [E] \quad \text{fetch assignments}\]
\[[E₁] := E₂ \quad \text{heap assignments (heap mutation)}\]
\[V := \text{cons}(E₁, \ldots, Eₙ) \quad \text{allocation assignments}\]
\[\text{dispose}(E) \quad \text{pointer disposal}\]
\[C₁ ; C₂ \quad \text{sequences}\]
\[\text{IF } B \text{ THEN } C₁ \text{ ELSE } C₂ \quad \text{conditionals}\]
\[\text{WHILE } B \text{ DO } C \quad \text{while commands}\]

Pointers 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: \([E₁] := E₂\)
  - evaluate \(E₁\) to get a location \(l\)
  - fault if the \(l\) is not in the heap
  - otherwise store the value of \(E₁\) as the new contents of \(l\) in the heap
- Pointer disposal: \(\text{dispose}(E)\)
  - evaluate \(E\) to get a pointer \(l\) (a number)
  - fault if \(l\) is not in the heap
  - otherwise remove \(l\) from the heap

Allocation assignments

- Allocation assignments: \(V := \text{cons}(E₁, \ldots, Eₙ)\)
  - 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 \(E₁, E₂, \ldots\) 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)

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

- 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 \( l \), i.e. contents of \( l \), i.e. 1

Want to prove:

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

- Need additional axioms for fetch, store and allocation assignments
- Need assertions in specification language to describe contents of heap

Intuitively:

\[ \{ T \} \ X := \text{cons}(0); \ X := l \land H(l) = 0 \]

where \( l \) is a new location and \( H \) is the heap

- \( x := \text{cons}(0) \)
- \( X := l \land H(l) = 0 \)
- \( Z := 1 \)
- \( Y := X \)
- \( [Y] := Z \)
- \( Y := [X] \)

How can this be formalised? The tricky bit is the heap mutation:

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

The rule of constancy (Reynolds’ name)

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

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 \( \vdash \{ P \} C_1 \{ Q_1 \} \)
    - Suppose no variable modified by \( C_1 \) occurs free in \( P \)
    - Then by rule of constancy: \( \vdash \{ P \} \times \nu \{ C \} \{ Q_1 \} \)
    - Suppose no variable modified by \( C \) occurs free in \( Q_1 \)
    - Then by rule of constancy: \( \vdash \{ P \} \times \nu \{ C \} \{ Q_1 \} \)
    - Hence by commutativity of \( \land \) and sequencing rule: \( \vdash \{ P \} \land \nu \{ C \} \{ Q \land Q_1 \} \)

Rule of constancy not valid for heap assignments:

\[ \vdash \{ T \} \{ X := 0 \} \{ \{ H \} = 0 \} \]

But not

\[ \vdash \{ T \land H(Y) = 1 \} \{ X := 0 \} \{ \{ H \} = 0 \land H(Y) = 1 \} \]

Because \( X = Y \) is possible

Heap assignment (mutation)

- A plausible Floyd-style forward heap assignment axiom:
  \[ \{ E_1 = l \land E_2 = v \} \ [E_1] := E_2 \{ H(l) = v \} \]

- How can we get from this to:
  \[ \{ X = l \land H(l) = 0 \land Z = 1 \land Y = l \} \ [Y] := Z \{ X = l \land H(l) = 1 \land Z = 1 \land Y = l \} \]

- Appropriate instance of plausible heap assignment axiom:
  \[ \{ Y = l \land Z = 1 \} \ [Y] := Z \{ H(l) = 1 \} \]

The rule of constancy (derived rule of Hoare logic)

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

where no variable modified by \( C \) occurs free in \( R \)

- Hence:
  \[ \{ Y = l \land Z = 1 \land \{ X = l \land H(l) = 0 \} \} \ [Y] := Z \{ H(l) = 1 \land \{ X = l \land H(l) = 0 \} \} \]

Fail!
### 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:
  
  \[
  V := [E] \quad \Rightarrow \quad V := H(E) \quad \text{(assign value of} \ E \ \text{in} \ H \ \text{to} \ V) \\
  [E_1] := E_2 \quad \Rightarrow \quad H := H[E_2/E_1] \quad \text{(change} \ H \ \text{at} \ E_1 \ \text{to} \ E_2) \\
  \]

  - (not sure about this case)

  dispose(\( E \)) \quad \Rightarrow \quad H := H - E \quad \text{(delete} \ E \ \text{from domain of} \ H) \\

- If \( [E_1] := E_2 \) is \( H := H[E_2/E_1] \), then \( [E_1] := E_2 \) 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

### Rule of constancy again

- If \( [E_1] := E_2 \) is translated to \( H := H[E_2/E_1] \) 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 \( \vdash (P_1 \land P_2)C_1(Q_1 \land P_2) \)
  - suppose no variable modified by \( C_2 \) occurs in \( Q_1 \)
  - then by rule of constancy \( \vdash (P_1 \land Q_1)C_2(Q_1 \land Q_1) \)
  - hence by commutativity of \( \land \) and sequencing rule \( \vdash (P_1 \land P_2)C_1C_2(Q_1 \land Q_2) \)

- 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 \( C_1 \)
  - suppose no variable modified by \( C_1 \) occurs in \( P_2 \)
  - then by some rule \( \vdash (P_1 \land P_2)C_1C_2(Q_1 \land Q_2) \)
  - suppose no variable modified by \( C_2 \) occurs in \( Q_1 \)
  - then by some rule \( \vdash (P_1 \land Q_1)C_1C_2(Q_1 \land Q_1) \)
  - hence by commutativity of \( \land \) and sequencing rule \( \vdash (P_1 \land P_2)C_1C_2(Q_1 \land Q_2) \)

### Heap assignment axiom again

- Translating \( [E_1] := E_2 \) to \( H := H[E_2/E_1] \) yields by assignment axiom:
  \[
  \{Q[H[E_2/E_1]/H]\}[E_1] := E_2(Q) \\
  \]

- An instance is:
  \[
  \{(x = l \land H(l) = 1 \land z = 1 \land y = l)[H[Z/Y]/H]\} \quad [y] := z \quad \{(x = l \land H(l) = 1 \land z = 1 \land y = l) q \} \\
  \]

performing the substitution \( Q[H[Z/Y]/H] \)

\[
\{(x = l \land H(l) = 1 \land z = 1 \land y = l) q \} \quad [y] := z \quad \{(x = l \land H(l) = 1 \land z = 1 \land y = l) q \} \\
\]

then by precondition strengthening:

\[
\{(x = l \land H(l) = 0 \land z = 1 \land y = l) q \} \quad [y] := z \quad \{(x = l \land H(l) = 1 \land z = 1 \land y = l) q \} \\
\]

as wanted!

### 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 $\neq E_i$ unchanged
  - clumsy ... but used successfully by some (e.g. ACL2 + Rockwell Collins)
  - need to always reason about the whole heap
- Analogy from AI

**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

**Sneak preview of the Frame Rule**

The frame rule

$$\Gamma \vdash \{P\}C\{Q\}$$

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

**Notation for separation assertions**

- Expressions $E$ evaluated in the store $s$, just like before
  - $E(s)$ means $E$ true in $s$ – i.e. $\text{Sem}_s E s$
- In general an assertion depends of store $s$ and heap $h$
  - $P(s,h)$ means $P$ true in state $(s,h)$ (see $P(s,h)$ in background reading)
  - semantics of first-order logic statement $S$ (doesn’t depend on heap) is $\text{Sem}_s S s$
- Notation and terminology for finite functions
  - $\text{dom}(f)$ is domain of finite function $f$, so if $f : A \rightarrow B$ then $\text{dom}(f) = A$
  - $f[b/a]$ is same as $f$ except $a$ maps to $b$, $\text{dom}(f[b/a]) = \text{dom}(f) \cup \{a\}$
  - $f\neg a$ is the result of deleting $a$ from $\text{dom}(f)$, so $\text{dom}(f\neg a) = \text{dom}(f) \setminus \{a\}$
  - $\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ 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 $l \in \text{dom}(h)$
  - $\text{dom}(h_1 \cup h_2) = \text{dom}(h_1) \cup \text{dom}(h_2)$ and $(h_1 \cup h_2)[l] = \text{if } l \in \text{dom}(h_1) \text{ then } h_1(l) \text{ else } h_2(l)$
  - $h_1 \star h_2$ only defined if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$, 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:
  \[
  (E_1 \mapsto E_2)(s, h) \iff \text{dom}(h) = \{E_1(s)\} \land h(E_1(s)) = E_2(s)
  \]
- Example: $(X \mapsto Y+1)(s, \{20 \mapsto 43\})$ is true iff $s(X) = 20$ and $s(Y) = 42$
- Abbreviation: $E \mapsto _\bot =_{df} \exists X. E \mapsto X$ (where $X$ does not occur in $E$)
- By semantics: $(E \mapsto _\bot)(s, h) \iff \text{dom}(h) = \{E(s)\}$

 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-heap $h_1$, $h_2$ so that: $h = h_1 \ast h_2$
  (Note: "$\ast$" in $h_1 \ast h_2$ 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 conjunction $P \ast Q$ is defined by:
  \[
  (P \ast Q)(s, h) \iff \exists h_1, h_2. h = h_1 \ast h_2 \land P(s, h_1) \land Q(s, h_2)
  \]
- Example: $(X \mapsto 0 \ast X+1 \mapsto 0)(s, \{20 \mapsto 0, 21 \mapsto 0\})$ is true iff $s(X) = 20$
- Abbreviation: $E \mapsto E_0, \ldots, E_n =_{df} (E \mapsto E_0) \star \cdots \star (E \mapsto E_n)$
  - specifies contents of $n+1$ contiguous locations starting at $E$
  - for $0 \leq i \leq n$ the contents of location $E+i$ is value of $E_i$
- Example: $(X \mapsto Y, Z)(s, \{x \mapsto y, x+1 \mapsto z\})$ is true iff $s(X) = y \land s(Y) = y \land s(Z) = z$

 Separation logic assertions: emp

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

Summary of separation logic assertions (there are more)

- Points-to $E_1 \mapsto E_2$
  \[
  E_1 \mapsto E_2 \iff \text{dom}(h) = \{E_1(s)\} \land h(E_1(s)) = E_2(s)
  \]
- Separating conjunction $P \ast Q$
  \[
  (P \ast Q)(s, h) \iff \exists h_1, h_2. h = h_1 \ast h_2 \land P(s, h_1) \land Q(s, h_2)
  \]
- Empty heap emp
  \[
  \text{emp}(s, h) \iff h = \{\} \quad \text{(where \{\} is the empty heap)}
  \]
- Abbreviation: $E \mapsto _\bot =_{df} \exists X. E \mapsto X$ (where $X$ does not occur in $E$)
- Abbreviation: $E \mapsto F_0, \ldots, F_n =_{df} (E \mapsto F_0) \star \cdots \star (E \mapsto F_n)$
- Abbreviation: $E_1 \parallel E_2 =_{df} (E_1 = E_2) \land \text{emp}$
Example: reversing a linked list

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

\[
\begin{array}{c}
\text{l} \quad \text{m} \quad \text{n} \\
\text{a} \quad \text{b} \quad \text{c}
\end{array}
\]

- Assume \(\text{nil} \in \text{Val}\) and \([a_1, \ldots, a_n]\) \(\in \text{Val}\) for \(a_i \in \text{Val}\)

- Define list \(x \in E\) to mean \(x\) is stored as a linked list at location \(E\):

\[
\text{list} \[\text{nil} \Rightarrow (E \equiv \text{nil}) \\
\text{list} \[(a_0, a_1, \ldots, a_n)] \Rightarrow \exists E'. (E \rightarrow a_0, E') \times \text{list} \[a_1, \ldots, a_n]\ E'
\]

- Can then specify:

\[
\{\text{list} x \ X\}
\]

\[
Y := \text{nil}; \quad \text{WHILE } \neg (X = \text{nil}) \text{ DO } (Z := [X + 1]; \ [X + 1] := Y; \ Y := X; \ X := Z)
\]

\[
\{\text{list \ (rev}\(x\)) \ Y\}
\]

Separation logic: small axioms and faulting

- One might expect a heap assignment axiom to entail:

\[
\vdash \{T\}[0] := 0 \rightarrow 0
\]

i.e., after executing \([0] := 0\) the contents of location 0 in the heap is 0

- Recalling the sneak preview of the frame rule:

\[
\begin{array}{c}
\text{The frame rule} \\
\vdash \{P\} C \{Q\} \\
\vdash \{P \ast R\} C \{Q \ast R\}
\end{array}
\]

where no variable modified by \(C\) occurs free in \(R\).

- Taking \(R\) to be the points-to statement \(0 \rightarrow 1\) yields:

\[
\vdash \{T \ast 0 \rightarrow 1\}[0] := 0 \rightarrow 0 \ast 0 \rightarrow 1
\]

something is wrong with the conclusion!

- Solution: define Hoare triple so \(\vdash \{T\}[0] := 0 \rightarrow 0\) 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: $\text{Sem } C(s,h)(s',h')$
- $C(s,h)\text{fault}$ means executing $C$ in state $(s,h)$ can fault
  - in the background reading: $\text{Sem } C(s,h)$ fault
- Sometimes $C(s,h)r$ where $r$ (for “result”) is a state or fault

Semantics of store assignments (only store changed):

\[(V:=E)(s,h)r = (r = (s[[E(s)]/V], h))\]

Semantics of fetch assignments (only store changed):

\[(V:=E)(s,h)r = (r = \begin{cases} E(s) \in \text{dom}(h) \text{ then } (s[h(E(s)]/V), h) \text{ else fault} \end{cases})\]

Semantics of heap assignments (only heap changed):

\[(E_1):=E_2)(s,h)r = (r = \begin{cases} E_1(s) \in \text{dom}(h) \text{ then } (s, h[E_2(s)/E_1(s)]) \text{ else fault} \end{cases})\]

Semantics of pointer disposal (only heap changed):

\[(\text{dispose}(E))(s,h)r = (r = \begin{cases} E(s) \in \text{dom}(h) \text{ then } (s, h-(E(s))) \text{ else fault} \end{cases})\]

Semantics of commands (ii)

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

Semantics of commands (iii)

- Semantics of sequences:
  \[(C_1;C_2)(s,h)r = \begin{cases} \exists s', h'. r = (s', h') \text{ then } (\exists s', h'. C_1(s,h)(s',h') \land C_2(s',h')r) \text{ else } ((C_1(s,h)r \land (r = \text{fault})) \lor (\exists s', h'. C_1(s,h)(s',h') \land C_2(s',h')r \land (r = \text{fault}))) \end{cases} \]
- As in simple language, but propagate faults
  - if $C_1(s,h)\text{fault}$ then $C_1;C_2(s,h)\text{fault}$

Semantics of conditionals:

\[(\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2)(s,h)r = \begin{cases} S \text{ s then } C_1(s,h)r \text{ else } C_2(s,h)r \end{cases} \]
- $S$ is a first-order logic statement (doesn’t depend on heap), hence $\text{Sem } S \text{ s}$

Semantics of commands (iv)

- Semantics of while-commands:
  \[(\text{WHILE } S \text{ DO } C)(s,h)r = \exists n. \text{Iter } n (\text{Sem } S) (\text{Sem } C) (s,h) r \text{ where the recursive function } \text{Iter is redefined to handle faulting:} \]
  \[\text{Iter } n (\text{Sem } S) (\text{Sem } C) (s,h) r = \begin{cases} p \in \text{dom}(h) \land (r = (s,h)) \text{ then } n \text{ Iter } n p c (s,h)r \text{ else } ((c(s,h)r \land (r = \text{fault})) \lor (\exists s', h'. c(s,h)(s',h') \land \text{Iter } n p c (s',h') r)) \end{cases} \]
- Looks horrible ... but is just the obvious fault-propagating semantics
  - $\text{Iter } : \text{Num} \rightarrow (\text{Store} \rightarrow \text{Bool}) ightarrow (\text{State} \rightarrow \text{Result} \rightarrow \text{Bool}) \rightarrow \text{State} \rightarrow \text{Result} \rightarrow \text{Bool}$
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
  \[ \models \{P\}C\{Q\} = \forall s\ h.\ P(s,h) \Rightarrow \neg(C(s,h)\text{fault}) \land \forall s' h',\ C(s,h)(s',h') \Rightarrow Q(s',h') \]

- Now \( \models \{T\}[0] := 0\{0 \rightarrow 0\} \) is not true as \( \{0\} := 0\{s,\{\}\}\text{fault} \)

- Recall the sneak preview of the frame rule:

\[ \text{The frame rule} \]

\[ \frac{\vdash \{P\}C\{Q\}}{\vdash \{P \ast R\}C\{Q \ast R\}} \]

where no variable modified by \( C \) occurs free in \( R \).

- So can’t use frame rule to get \( \vdash \{T \ast 0 \rightarrow 1\}[0] := 0\{0 \rightarrow 0 \ast 0 \rightarrow 1\} \)

Purely logical rules

- Following rules apply to both Hoare logic and separation logic

  \[ \text{Rules of consequence} \]

  \[ \frac{\vdash P \Rightarrow P'}{\vdash \{P\}C\{Q\} \Rightarrow \{P'\}C\{Q\}} \]

  \[ \frac{\vdash \{P\}C\{Q\} \quad \vdash Q' \Rightarrow Q}{\vdash \{P\}C\{Q'\}} \]

  \[ \frac{\vdash \{P\}C\{Q\}}{\vdash \{\exists x.\ P\}C\{\exists x.\ Q\}} \]

where \( x \) does not occur in \( C \).

- For separation logic, need to think about faulting

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

Store assignment axiom

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

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

where \( v \) is an auxiliary variable not occurring in \( E \).

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

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

where \( v \) is an auxiliary variable not occurring in \( E \).

- Separation logic: exists introduction valid, rule of constancy invalid
### Fetch assignment axiom

\[ \vdash \{(V = v_1) \land E \rightarrow v_2\} [E] \{(V = v_2) \land E[v_1/V] \rightarrow v_2\} \]

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

\[ \vdash \{E \rightarrow v\} V := [E] \{(V = v) \land E \rightarrow v\} \quad \text{(proof: instantiate } v_1 \text{ to } V \text{ and } v_2 \text{ to } v \text{ and then simplify)} \]

### Heap assignment axiom (heap mutation)

\[ \vdash \{E \rightarrow \_\} [E] := F \{E \rightarrow F\} \]

- 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 deallocation

\[ \vdash \{E \rightarrow \_\} dispose(E) \{emp\} \]

- Attempting to deallocate a pointer not in the heap faults
- Small axiom: singleton precondition heap, empty postcondition heap
- Sanity checking example proof:
  \[ \vdash \{E_1 \rightarrow \_\} dispose(E_1) \{emp\} \quad \text{dispose axiom} \]
  \[ \vdash \{emp\} V := cons(E_2) \{V \rightarrow E_2\} \quad \text{derived allocation assignment axiom} \]
  \[ \vdash \{E_1 \rightarrow \_\} dispose(E_1); V := cons(E_2) \{V \rightarrow E_2\} \quad \text{sequencing rule} \]

### Pointer allocation

\[ \vdash \{V .= v\} V := cons(E_1, \ldots, E_n) \{V \rightarrow E_1[v/V], \ldots, E_n[v/V]\} \]

where \(v\) doesn’t occur in \(E_1, \ldots, E_n\).

- Never faults
- If \(V\) doesn’t occur in \(E_1, \ldots, E_n\) then:
  \[ \vdash \{V .= v\} V := cons(E_1, \ldots, E_n) \{V \rightarrow E_1[v/V], \ldots, E_n[v/V]\} \quad \text{alloc. assign. ax} \]
  \[ \vdash \{V .= v\} V := cons(E_1, \ldots, E_n) \{V \rightarrow E_1, \ldots, E_n\} \quad \text{V not in } E_i \text{, assump.} \]
  \[ \vdash \{\exists v. \; V .= v\} V := cons(E_1, \ldots, E_n) \{\exists v. \; V \rightarrow E_1, \ldots, E_n\} \quad \text{exists intro} \]
  \[ \vdash \{\exists v. \; V .= v \land \text{emp}\} V := cons(E_1, \ldots, E_n) \{\exists v. \; V \rightarrow E_1, \ldots, E_n\} \quad \text{definition of } \exists \]
  \[ \vdash \{\text{emp}\} V := cons(E_1, \ldots, E_n) \{V \rightarrow E_1, \ldots, E_n\} \quad \text{predicate logic} \]

- Which is a derivation of:

\[ \vdash \{\text{emp}\} V := cons(E_1, \ldots, E_n) \{V \rightarrow E_1, \ldots, E_n\} \]

where \(V\) doesn’t occur in \(E_1, \ldots, E_n\).
### Summary of pointer manipulating axioms

#### Store assignment axiom
\[ \vdash \{ V \leftarrow v \} \ V \leftarrow E \ \{ V \leftarrow E[v/V] \} \]
where \( v \) is an auxiliary variable not occurring in \( E \).

#### Fetch assignment axiom
\[ \vdash \{(V = v_1 \land E \mapsto v_2) \} \ V \leftarrow [E] \ \{(V = v_2) \land E[v_1/V] \mapsto v_2 \} \]
where \( v_1, v_2 \) are auxiliary variables not occurring in \( E \).

#### Heap assignment axiom
\[ \vdash \{ E \leftarrow \_ \} \ E \leftarrow F \ {E \leftarrow F} \]

#### Allocation assignment axiom
\[ \vdash \{ V \leftarrow v \} \ V \leftarrow \text{cons}(E_1, \ldots, E_n) \ \{ V \leftarrow E_1[v/v], \ldots, E_n[v/v] \} \]
where \( v \) is an auxiliary variable not equal to \( V \) or occurring in \( E_1, \ldots, E_n \).

#### Dispose axiom
\[ \vdash \{ E \leftarrow \_ \} \ \text{dispose}(E) \ {\text{emp}} \]

### Compound rules

#### The sequencing rule
\[ \vdash \{ P \} \ C_1 \ \{ Q \} \quad \vdash \{ Q \} \ C_2 \ \{ R \} \]
\[ \vdash \{ P \} \ C_1 ; C_2 \ \{ R \} \]

#### The conditional rule
\[ \vdash \{ P \land S \} \ C_1 \ \{ Q \} \quad \vdash \{ P \land \neg S \} \ C_2 \ \{ Q \} \]
\[ \vdash \{ P \} \ \text{IF} \ S \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2 \ \{ Q \} \]

#### The while-rule
\[ \vdash \{ P \land S \} \ C \ \{ P \} \]
\[ \vdash \{ P \} \ \text{WHILE} \ S \ \text{DO} \ C \ \{ P \land \neg S \} \]

For separation logic, need to think about faulting

### The frame rule

**The rule of constancy**
\[ \vdash \{ P \} \ C \ \{ Q \} \]
\[ \vdash \{ P \land R \} \ C \ \{ Q \land R \} \]
where no variable modified by \( C \) occurs free in \( R \).

- Not valid for heap assignments
  \[ \vdash \{ X \leftarrow \_ \} \ X \leftarrow 0 \ \{ X \leftarrow 0 \} \]
  but not
  \[ \vdash \{ X \leftarrow \_ \land Y \leftarrow 1 \} \ X \leftarrow 0 \land \ Y \leftarrow 1 \]
  as \( X \) and \( Y \) could initially both be bound to the same location

- **Using \( \ast \) instead of \( \land \) forces \( X \) and \( Y \) to point to different locations**

**The frame rule**
\[ \vdash \{ P \} \ C \ \{ Q \} \]
\[ \vdash \{ P \land R \} \ C \ \{ Q \land R \} \]
where no variable modified by \( C \) occurs free in \( R \).

- Soundness a little tricky due to faulting
Logic of separating assertions, soundness, completeness

- To use separation logic various properties of $\star$, $\Rightarrow$ etc. are needed
- For rule of consequence in proof on preceding slide need:
  \[
  (\exists v. X \Rightarrow v \star Y \Rightarrow v) \Rightarrow \exists v. x y. ((X = x) \land x \Rightarrow v \star ((Y = y) \land Y \Rightarrow v)) \Rightarrow (X = Y)
  \]
- 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:
  \[
  \{\text{list } x \ X\} \\
  Y:=\text{nil}; \\
  \text{WHILE } \neg (X = \text{nil}) \text{ DO (Z:=}[X+1]; \ [X+1]:=Y; \ Y:=X; \ X:=Z) \\
  \{\text{list } (\text{rev } x) \ Y\}
  \]
- 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.:
    \[
    \{\exists z. \text{list } x \ X\} \\
    Y:=\text{nil}; \\
    \text{WHILE } \neg (X = \text{nil}) \text{ DO (Z:=}[X+1]; \ [X+1]:=Y; \ Y:=X; \ X:=Z) \\
    \{\exists z. \text{list } y \ Y\}
    \]
    automatically finds memory usage errors
- Finally, something to think about:
  should we be verifying code in old fashioned languages (pragmatism) or creating new methods to create correct software (idealism)?

“\text{The tension between idealism and pragmatism is as profound (almost) as that between good and evil (and just as pervasive).}”
[Tony Hoare]