Last Time

- When program S executes it switches to a different state
- We need to express assertions on the states of the program S before and after its execution
- We can do it using a Hoare triple written as \{P\}S{Q}, where P is a precondition, S is a program, and Q is a postcondition
- We used flowchart diagrams to prove partial correctness and termination of two programs

Inference Rules

- An inference rule maps one or more wffs, called premises, to a single wff, called the conclusion

\[
\begin{align*}
\frac{A, A \rightarrow B}{\therefore B} & \quad \text{modus ponens (MP)} \\
\frac{A \lor B, \neg A}{\therefore B} & \quad \text{disjunctive syllogism (DS)} \\
\frac{\neg B, A \rightarrow B}{\therefore \neg A} & \quad \text{modus tollens (MT)} \\
\frac{A \rightarrow B, B \rightarrow C}{\therefore A \rightarrow C} & \quad \text{hypothetical syllogism (HS)} \\
\frac{A, B}{\therefore A \land B} & \quad \text{conjunction intro (CI)} \\
\frac{A \lor B, A \rightarrow C, B \rightarrow D}{\therefore C \lor D} & \quad \text{constructive dilemma (CD)} \\
\frac{A}{\therefore A \lor B} & \quad \text{disjunction intro (DI)} \\
\frac{\neg C \lor \neg D, A \rightarrow C, B \rightarrow D}{\therefore \neg A \lor \neg B} & \quad \text{destructive dilemma (DD)}
\end{align*}
\]
Proofs

- A proof is a finite sequence of wffs s.t. each wff in the sequence is either an axiom or a premise or can be inferred from previous wffs in the sequence.
- A formal reasoning system is also called a formal theory.
- If a formal theory enables the proof of both wffs P and ¬P, then this theory is inconsistent (not sound).
- How to build consistent theories?
  - Choose axioms to be tautologies.
  - Choose inference rules to map tautologies onto tautologies.

Examples

- Prove \((A \lor B) \land (A \lor C) \land \neg A \rightarrow B \land C\)
  1. \(A \lor B\) \(P\)
  2. \(A \lor C\) \(P\)
  3. \(\neg A\) \(P\)
  4. \(B\) \(1,3,DS\)
  5. \(C\) \(2,3,DS\)
  6. \(B \land C\) \(4,5,CI\)
  7. QED \(1,2,3,6\)

Our Strategy

- Recall proof calculi for propositional and predicate logic.
  - Formula to prove, inference rules, axioms.
  - For example, to prove \(\phi \rightarrow \psi\) we assume \(\phi\) and manage to show \(\psi\) using given inference rules.
- What if we replace a logic formula with a piece of code?
- Can we prove fragments of code and these small proofs compose a final proof?
Partial Correctness, Termination, and Total Correctness

- **Partial correctness**: if for all states that satisfy the precondition, the state resulting from program’s execution satisfies the postcondition, provided that the program terminates
- **Termination**: if the precondition holds, then the program terminates
- **Total correctness**: if for all states in which P is executed which satisfy the precondition, P is guaranteed to terminate and the resulting state satisfies the postcondition

Proof Calculus For Partial Correctness

- Goes back to R.Floyd and C.A.R. Hoare
- Given a language grammar
- Given proof rules for each of the grammar clauses for commands
- We construct our proofs in a form of proof tableaux
A Core Programming Language

- S ::=  
  x=E |  
  S;S |  
  if B {S} else {S} |  
  while B {S}  
- B ::= true | false | (!B) | (B&B) | (B||B) | (E<E)  
- E ::= n | x | (-E) | (E-E) | (E+E) | (E*E)  
- n is any numeral  
- x is any variable

A Program For Computing a Factorial

Factorial( x ) {  
  y = 1;  
  z = 0;  
  while( z != x ) {  
    z = z + 1;  
    y = y * z;  
  }  
}  

0! ≡ 1  
(n+1)! ≡ (n+1) · n!
**Composition Rule**

\[
\begin{array}{c}
\{P\} S_1 \{Q\} \\
\{Q\} S_2 \{R\} \\
\{P\} S_1 ; S_2 \{R\}
\end{array}
\]

- S₁ and S₂ are program fragments
- In order to prove \{P\} S₁ ; S₂ {R} we need to find an appropriate Q
- Then we prove \{P\} S₁ {Q} and \{Q\} S₂ {R} separately

**Assignment**

\[
\{P[E/x]\} x = E(P)
\]

- No premises => it is an axiom!
- We wish to know that P holds in the state after the assignment x = E
- P[E/x] means the formula obtained by taking P and replacing all occurrences of x with E
  - P with E in place of x
Assignment: Flawed Understanding

\[
\left\{ P\left[\frac{E}{x}\right]\right\} x = E\{P\} 
\]

- If P holds in a state in which we perform the assignment \(x = E\), then \(P[E/x]\) holds in the resulting state
  - We replace \(x\) by \(E\)
  - Do we perform this replacement of occurrences of \(x\) in a condition on the starting state by \(E\)?

Assignment: Correct Understanding

\[
\left\{ P\left[\frac{E}{x}\right]\right\} x = E\{P\} 
\]

- Do we perform this replacement of occurrences of \(x\) in a condition on the starting state by \(E\)?
- No, we need to prove something about the initial state in order to prove that \(P\) holds in the resulting state
- Whatever \(P\) says about \(x\) but applied to \(E\) must be true in the initial state
Assignment: Examples

\[ \{2 = 2\} \ x = 2 \ \{x = 2\} \]

- If we want to prove \(x=2\) after the assignment \(x=2\), then we must be able to prove that \(2=2\) before it

\[ \{2 = y\} \ x = 2 \ \{x = y\} \]

- If we want to prove \(x=y\) after the assignment \(x=2\), then we must be able to prove that \(2=y\) before it

Assignment: Exercises

\[ \{x + 1 = 2\} \ x = x + 1 \ \{x = 2\} \]

\[ \{x + 1 = y\} \ x = x + 1 \ \{x = y\} \]

\[ \{x + 1 > 0 \land y > 0\} \ x = x + 1 \ \{x > 0 \land y > 0\} \]
Assignment

$$\left\{ P\left[ \frac{E}{x} \right] \right\} x = E\{ P \}$$

- This assignment axiom is best applied backward than forward in the verification process
- We know Q and wish to find P s.t. \{P\}x=E \{Q\} – easy
  - Set P to be Q[E/x]
- If we know P and want to find Q s.t. \{P\} x=E \{Q\} – very difficult!!!
WHILE-Statement Rule

\[
\begin{align*}
\{P \land B\} & \xrightarrow{S} \{P\} \\
\{P\} \quad \text{while} \quad B & \quad \{S\} \quad \{P \land \neg B\}
\end{align*}
\]

- S is a program fragment that is executed multiple times in the while loop.
- We don't know how many times S is gonna be executed or whether it terminates at all.
- P is a loop invariant.

Implied Rule

\[
\begin{align*}
\vdash P' & \rightarrow P \\
\{P\} & \xrightarrow{S} \{Q\} \\
\vdash Q & \rightarrow Q'
\end{align*}
\]

- Implied rule allows the precondition to be strengthened.
  - We assume more than we need to.
- The postcondition may be weakened.
  - We conclude less than we are entitled to.
A Program For Computing a Factorial

Factorial( x ) {
    y = 1;
    z = 0;
    while( z != x) {
        z = z + 1;
        y = y * z;
    }
}

\[
\begin{align*}
0! &\triangleq 1 \\
(n+1)! &\triangleq (n+1) \cdot n!
\end{align*}
\]

Let’s Prove It!!!

Proof Tableaux

- What is good about them?
  - Tree structure
  - We think of a program as a sequence of code fragments
  - We interleave the program code with intermediate formulae called midconditions

- Is it easy to read proof tableaux?
- Is there an alternative?
Division With Remainder Example

\{x \geq 0 \land y \geq 0\}

\[
\begin{align*}
a &= 0; \\
b &= x; \\
\text{while } (b \geq y) \{ \\
\quad b &= b - y; \\
\quad a &= a + 1; \\
\} \\
\{x = a \cdot y + b \land b \geq 0\}
\end{align*}
\]

Invariant:

\[
\begin{align*}
x &= a \cdot y + b \land b \geq 0\}
\end{align*}
\]

DivProg

Invariant

- How to start the proof?
- Heuristics: Find invariant for each loop.

- For this example: \(x = a \cdot y + b \land x \geq 0\)
- Note: total correctness does not hold for \(y = 0\)
- Total correctness (with \(y > 0\)) should be proved separately.
Proof

\( \{x = a \cdot y + x \land x \geq 0\} b = x \{x = a \cdot y + b \land b \geq 0\} \) \hspace{1cm} (1)

\( \{x = 0 \cdot y + x \land x \geq 0\} a = 0 \{x = a \cdot y + x \land x \geq 0\} \) \hspace{1cm} (2)

\( \{x = 0 \cdot y + x \land x \geq 0\} a = 0; b = x \{x = a \cdot y + b \land x \geq 0\} \) \hspace{1cm} (3)

---

Proof

\( \{x = (a+1) \cdot y + b \land b \geq 0\} a = a+1 \{x = a \cdot y + b \land b \geq 0\} \) \hspace{1cm} (4)

\( \{x = (a+1) \cdot y + b \land b \geq 0\} b = b - y \)
\( \{x = (a+1) \cdot y + b \land b \geq 0\} \)

\( \{x = (a+1) \cdot y + b \land b \geq 0\} b = b - y; a = a+1 \)
\( \{x = a \cdot y + b \land b \geq 0\} \) \hspace{1cm} (5)
Consequence rules

- Strengthen a precondition
  \[ R \rightarrow P \quad \{ P \} S \{ Q \} \quad \{ R \} S \{ Q \} \]

- Weaken a postcondition
  \[ \{ P \} S \{ Q \} \quad Q \rightarrow R \quad \{ P \} S \{ R \} \]

Proof

\( x = a \cdot y + b \wedge b \geq 0 \wedge b \geq y \rightarrow (x = (a + 1) \cdot y + b - y \wedge b - y \geq 0) \)

\( \{ x = a \cdot y + b \wedge b \geq 0 \wedge b \geq y \} b = b - y; a = a + 1 \)

\( \{ x = a \cdot y + b \wedge b \geq 0 \} \)

consequence, 6, 7

\( \{ x = a \cdot y + b \wedge b \geq 0 \} \) while \( (b \geq y) \) \{
  b = b - y; a = a + 1
\}

\( \{ x = a \cdot y + b \wedge b \geq 0 \wedge b < y \} \)

while, 8
Proof

\{ x = 0 \cdot y + x \land x \geq 0 \} \text{DivProg}
\{ x = a \cdot y + b \land b \geq 0 \land b < y \}

composition, 3,9

\begin{align*}
(x \geq 0 \land y \geq 0) & \rightarrow (x = 0 \cdot y + x \land x \geq 0) \\
\{ x = 0 \cdot y + x \land x \geq 0 \} \text{DivProg} \\
\{ x = a \cdot y + b \land b \geq 0 \land b < y \}
\end{align*}

consequence

Soundness

- Hoare logic is sound in the sense that everything that can be proved is correct!

- This follows from the fact that each axiom and proof rule preserves soundness
Completeness

- A proof system is called complete if every correct assertion can be proved
- Propositional logic is complete
- No deductive system for the standard arithmetic can be complete (Godel)

And for Hoare’s logic?

- Let S be a program and P its precondition

  Then \( \{P\} S \{\bot\} \) means that S never terminates when started from P
  - This is undecidable
  - Thus, Hoare’s logic cannot be complete
General Observations

- If we can prove programs then we represent them as mathematical objects
- Does it mean that computer programmers are like mathematicians?
- Mathematicians try to improve their confidence in the correctness of theorems
- They use chain of formal logic statements to achieve this goal

Is Proof = Program?

- By verifying a program we increase our confidence in it
- So, it is like verifying the correctness of a theorem, right?
- The critical piece here is a social process that governs the acceptance of a theorem
- It is completely different between mathematical theorems and verified program
Mathematical Process

- Mathematicians publish about 200,000 theorems each year
- Are all of them correct and/or accepted?
- Multiple examples of famous mathematicians who announced and published proofs of theorems that were discredited later
  - Sometimes after many, many years!
- Mathematicians make a lot of mistakes!

Who Corrects Those Mistakes?

- Examples of contradictory results from published complicated proofs are well-known
- Only mathematicians can correct their errors, but who verifies the correctness of corrections?

- A proof does not in itself significantly raise our confidence in the probable truth of the theorem it purports to prove
What About Algebraic Proof?

- Many examples confirm that proofs that consist solely of calculations are not necessarily correct.
- It is not the question of “how do theorems get believed?”
- It is a question of “what is it we believe when we believe a theorem?”

Long Proofs

- Given a proof that occupies 2,000 pages, how long would it take to verify its correctness?
- What is a value of a long and complicated proof?
- How social process works for mathematicians?
- What is a fundamental difference between mathematicians and computer scientists doing proofs?
Why Do We Need Program Verification?

- Testing can never show the absence of errors, only their presence
- Software errors can cause major disasters especially in critical systems
- Math is used to state program properties and to prove program correct for all inputs
- However, program verification is expensive and has other drawbacks

Man and Machines

- What parts of program verifications cannot be replaced by machines?
- How to choose what properties to prove?
- How to find errors in specifications?
- Is the proof process correct?
Tool-Assisted Verification

- We can use tools that mechanize the deduction process
- If we have executable specifications then we can use tools that assist us in debugging these specifications
- When doing proof of program correctness we can use theorem provers to ensure proof correctness

Limitations of Program Verification

- We have only limited ways to convince ourselves that we are given a correct spec
- Even with the right specification we can prove only the correctness of mathematical abstraction, never of the system running in the real world
- There is a significant cost associated with program correctness proofs
- Not all systems are equally critical
Cost and Assurance of Formal Methods

Believing Software

- People cannot create perfect mechanisms
- Use social processes to create reliable structures
  - This is what most engineers do
- Computing structures are not
  - Perfect
  - The energy that can be wasted to make them perfect, is limited
Homework

- **Mandatory**
    - Downloadable from http://www.swt.edu/~mg43/reading.html

- **Optional**