Overview

▶ Today’s lecture: How to formally prove program correctness
▶ Specifically, we’ll look at Hoare logic: a set of logical inference rules for reasoning about program correctness
▶ Named after Tony Hoare: inventor of quicksort, father of formal verification, 1980 Turing award winner
▶ Logic is also known as Floyd-Hoare logic: some ideas introduced by Robert Floyd in 1967 paper “Assigning Meaning to Programs”

Motivation for Verifying Programs

▶ Typical way to ensure software quality: testing
▶ Testing is good at finding presence of bugs
▶ But in general, testing cannot guarantee absence of bugs
▶ Goal of program verification is to mathematically prove program obeys specification
▶ Thus, verification useful for guaranteeing absence of certain classes of errors or security vulnerabilities

Simple Imperative Programming Language

▶ To illustrate Hoare logic, we’ll consider a small imperative programming language IMP
▶ In IMP, we distinguish three program constructs: expressions, conditionals, and statements
▶ Expression $E := Z | V | e_1 + e_2 | e_1 \times e_2$
▶ Conditional $C := true | false | e_1 = e_2 | e_1 \leq e_2$
▶ Statement $S := V := E$ (Assignment)
▶ $S_1 ; S_2$ (Composition)
▶ if $C$ then $S_1$ else $S_2$ (If)
▶ while $C$ do $S$ (While)

Partial Correctness Specification

▶ We will specify partial correctness of programs using Hoare triples of the form: $\{P\}S\{Q\}$
▶ Here, $S$ is a statement in programming language IMP
▶ $P$ and $Q$ are first-order logic formulas over program variables
▶ $P$ is called precondition and $Q$ is called post-condition

Meaning of Hoare Triples

▶ Meaning of Hoare triple $\{P\}S\{Q\}$:
▶ If $S$ is executed in state satisfying $P$
▶ and if execution of $S$ terminates
▶ then the program state after $S$ terminates satisfies $Q$
▶ Is $\{x = 0\}x = x + 1\{x = 1\}$ valid Hoare triple? yes
▶ What about $\{x = 0 \land y = 1\}x = x + 1\{x = 1 \land y = 2\}$? no
▶ What about $\{x = 0\}x = x + 1\{x = 1 \lor y = 2\}$? yes
▶ What about $\{x = 0\}while \ true \ do \ x = 0\{x = 1\}$? yes
Partial vs. Total Correctness

- The specification $\{P\}S\{Q\}$ called partial correctness spec. b/c doesn’t require $S$ to terminate
- There is also a stronger requirement called total correctness
- Total correctness spec. written $[P]S[Q]$
- Meaning of $[P]S[Q]$:  
  - If $S$ is executed in state satisfying $P$
  - then the execution of $S$ terminates     
  - and program state after $S$ terminates satisfies $Q$
- Is $[x = 0] \text{while true do } x = 0 [x = 0]$ valid? no

Proof Rule for Assignment

$\vdash \{Q[E/x]\}x = E \{Q\}$

- To prove $Q$ holds after assignment $x = E$, sufficient to show that $Q$ with $E$ substituted for $x$ holds before the assignment.
- Using this rule, which of these are provable?
  - $\{y = 4\}x = 4 \{y = x\}$ yes
  - $\{x + 1 = n\}x = x + 1 \{x = n\}$ yes
  - $\{y = x\}y = 2 \{y = x\}$ no
  - $\{z = 2\}y = x \{y = x\}$ valid, but not provable just using assignment rule!
- This motivates need for precondition strengthening

Proof Rule for Precondition Strengthening

$\vdash \{P\}S\{Q\} \Rightarrow P' \vdash \{P\}S\{Q\}$

- Says that if we can prove post-condition $Q$ using assumptions $P'$, we can also prove it using stronger (more) assumptions $P$
- Using this rule and rule for assignment, we can now prove $\{z = 2\}y = x \{y = x\}$
- Proof:  
  \[
  \begin{align*}
  \vdash \{y = x[y/y]\}y = x \{y = x\} & \quad z = 2 \Rightarrow true \\
  \vdash \{true\}y = x \{y = x\} & \\
  \vdash \{z = 2\}y = x \{y = x\}
  \end{align*}
  \]

Examples, Safety, Liveness

- What does $\{true\}S\{Q\}$ say? If $S$ terminates, then $Q$ holds
- What about $\{P\}S\{true\}$? holds for any $P$ and any $S$
- What about $\{P\}S\{false\}$? If $P$ holds, then $S$ terminates
- When does $\{true\}S\{false\}$ hold? If $S$ does not terminate
- In the rest of lecture, we’ll focus on only partial correctness

Proof Rule for Post-Condition Strengthening

$\vdash \{P\}S\{Q'\} \Rightarrow Q' \Rightarrow Q \vdash \{P\}S\{Q\}$

- We also need a dual rule for post-conditions called post-condition weakening:
- Suppose we can prove $\{true\}S(x = y \land z = 2)$.
- Using post-condition weakening, which of these can we prove?
  - $\{true\}S\{z = y\}$ yes
  - $\{true\}S\{z = 2\}$ yes
  - $\{true\}S\{z > 0\}$ yes
Proof Rule for Composition

\[ \vdash (P) S_1 (Q) \quad \vdash (Q) S_2 (R) \]
\[ \vdash (P) S_1 ; S_2 (R) \]

- Using this rule, let’s prove validity of Hoare triple:
  \[ \{\text{true}\} \ x = 2; y = x \ (y = 2 \land x = 2) \]

- What is appropriate Q? \( x = 2 \)

Proof Rule for While and Loop Invariants

- Last proof rule of Hoare logic is that for while loops.

- But to understand proof rule for while, we first need concept of a loop invariant

- A loop invariant \( I \) has following properties:
  1. \( I \) holds initially before the loop
  2. \( I \) holds after each iteration of the loop

- Question: Suppose \( I \) is a loop invariant. Does \( I \) also hold after loop terminates?

- Yes because, by definition, \( I \) holds after every loop iteration, including after the last one

Proof Rule for If Statements

\[ \vdash (P \land C) \ S_1 \ {Q} \]
\[ \vdash (P \land \neg C) \ S_2 \ {Q} \]
\[ \vdash (P) \text{ if } C \text{ then } S_1 \text{ else } S_2 \ {Q} \]

- Suppose we know \( P \) holds before if statement and want to show \( Q \) holds afterwards.

- At beginning of then branch, what facts do we know? \( P \land C \)

- Thus, in the then branch, we want to show \( (P \land C) S_1 {Q} \)

- At beginning of else branch, what facts do we know? \( P \land \neg C \)

- What do we need to show in else branch? \( (P \land \neg C) S_2 {Q} \)

Proof Rule for While

- Consider the statement \( \text{while } C \text{ do } S \)

- Suppose \( I \) is a loop invariant for this loop. What is guaranteed to hold after loop terminates? \( I \land \neg C \)

- Putting all this together, proof rule for while is:

  \[ \vdash (P \land C) S {P} \land \neg C \]

- This rule simply says “If \( P \) is a loop invariant, then \( P \land \neg C \) must hold after loop terminates”

- Based on this rule, why is \( P \) a loop invariant?

- Because \( P \) holds initially and is preserved after each iteration

Example

- Consider the statement \( S = \text{while } x < n \text{ do } x = x + 1 \)

- Let’s prove validity of \( \{x \leq n\} S \{x = n\} \)

- What is appropriate loop invariant? \( x \leq n \)

- First, let’s prove \( x \leq n \) is loop invariant. What do we need to show? \( \{x \leq n \land x < n\} x = x + 1 \{x \leq n\} \)

- What proof rules do we need to use to show this? assignment, precondition strengthening

  \[ \vdash \{x \leq n \mid x + 1/x\} x = x + 1 \{x \leq n\} \quad \vdash \{x + 1 \leq n\} x = x + 1 \{x \leq n\} \]

Example, cont.

The full proof:

\[ \vdash (x + 1 \leq n) x = x + 1 \{x \leq n\} \]
\[ \vdash (x \leq n \land x < n) x = x + 1 \{x \leq n\} \]
\[ \vdash \{x \leq n\} S \{x \leq n \land \neg (x < n) \} \]
\[ \vdash \{x \leq n\} S \{x = n\} \]
Summary of Proof Rules

1. $\vdash \{Q'[x/E]\} x = E \{Q\}$ (Assignment)
2. $\vdash \{P'\}S\{Q\} \quad P \Rightarrow P'$ (Strengthen P)
3. $\vdash \{P\}S\{Q\} \quad Q' \Rightarrow Q$ (Weaken Q)
4. $\vdash \{P\}C_1\{Q\} \quad \vdash \{Q\}C_2\{R\}$ (Composition)
5. $\vdash \{P\} \text{if } C \text{ then } S_1 \text{ else } S_2 \{Q\}$ (If)
6. $\vdash \{P\} \text{while } C \text{ do } S\{P \land \neg C\}$ (While)

Relative Completeness

- However, Hoare’s logic still has nice property: relative completeness
- Recall: Rules for precondition strengthening and postcondition weakening require checking $A \Rightarrow B$
- In general, these formulas belong to Peano arithmetic (undecidable in general)
- Relative completeness: If we have an oracle for deciding validity, then any valid Hoare triple can be proven using our proof rules

Automating Reasoning in Hoare Logic

- Hoare’s proof rules provide a sound system to show properties about programs, but not immediately amenable to automation
- Number of places in proof system that require insight:
  1. When do we apply precondition strengthening?
  2. When do we apply postcondition weakening?
  3. Most difficult: How do we come up with loop invariants to use in rule for while?

Loop Invariants

- In rest of lecture, we’ll assume some oracle gives us appropriate invariants for each program
- This oracle can either be a human or a static analysis tool
- Static analysis techniques, such as abstract interpretation, can automatically infer loop invariants
- However, we won’t discuss how to automate loop invariant discovery in this lecture

Basic Idea Behind Program Verification

- Automating Hoare logic is based on generating verification conditions (VC)
- A verification condition is a formula $\phi$ generated automatically from source code and annotated loop invariants
- Furthermore, the program obeys specification if $\phi$ is valid
- In this model, program verification has two components:
  1. Generate VC’s from source code
  2. Use theorem prover to check validity of formulas from step 1

Soundness and Completeness?

- It can be shown that the proof rules for Hoare logic are sound:
  If $\vdash \{P\}S\{Q\}$, then $\models \{P\}S\{Q\}$
- That is, if a Hoare triple $\{P\}S\{Q\}$ is provable using the proof rules, then $\{P\}S\{Q\}$ is indeed valid
- Completeness: Any valid Hoare triple can be proven using our proof system:
  If $\models \{P\}S\{Q\}$, then $\vdash \{P\}S\{Q\}$
- Unfortunately, completeness does not hold!
Generating VCs: Forwards vs. Backwards

- Two ways to generate verification conditions: forwards or backwards
- Forwards technique starts from precondition and generates strongest postconditions (sp)
- Backwards technique starts from postcondition and computes weakest precondition
- We'll discuss backwards, wp-based method for generating verification conditions

Weakest Preconditions

- Idea: Suppose we want to verify Hoare triple \( \{P\}S\{Q\} \)
- We'll start with \( Q \) and going backwards, compute formula \( \text{wp}(S, Q) \) called weakest precondition of \( Q \) w.r.t. to \( S \)
- \( \text{wp}(S, Q) \) has the property that it is the weakest condition that guarantees \( Q \) will hold after \( S \) in any execution
- Thus, Hoare triple \( \{P\}S\{Q\} \) is valid iff:
  \[ P \Rightarrow \text{wp}(S, Q) \]

Defining Weakest Preconditions

- Weakest preconditions are defined inductively and follow Hoare’s proof rules
- \( \text{wp}(x = E, Q) = Q[E/x] \)
- \( \text{wp}(s_1; s_2, Q) = \text{wp}(s_1, \text{wp}(s_2, Q)) \)
- \( \text{wp}(\text{if } C \text{ then } s_1 \text{ else } s_2, Q) = C \Rightarrow \text{wp}(s_1, Q) \land \neg C \Rightarrow \text{wp}(s_2, Q) \)
- This says “If \( C \) holds, \( \text{wp} \) of then branch must hold; otherwise, \( \text{wp} \) of else branch must hold”

Defining Weakest Preconditions, cont.

- We have one case left for while loops: \( \text{wp}(\text{while } C \text{ do } s, Q) \)
- In general, not possible to compute weakest preconditions for loops exactly
- Instead of computing exact \( \text{wp}(S, Q) \), we will approximate it using \( \text{awp}(S, Q) \)
- \( \text{awp}(S, Q) \) may be stronger than \( \text{wp}(S, Q) \) but not weaker
- Thus, to verify \( \{P\}S\{Q\} \), necessary to show \( P \Rightarrow \text{awp}(S, Q) \)
- Hope is that \( \text{awp}(S, Q) \) is weak enough to be implied by \( P \) although it may not be the weakest

Approximate Weakest Preconditions

- For all statements except for while loops, computation of \( \text{awp}(S, Q) \) same as \( \text{wp}(S, Q) \)
- To compute, \( \text{awp}(S, Q) \) for loops, we will rely on loop invariants provided by oracle
- Thus, we’ll assume all loops are of the form \( \text{while } C \text{ do } \{I\} S \) where \( I \) is a loop invariant
- Now, we’ll just define \( \text{awp}(\text{while } C \text{ do } \{I\} S, Q) \equiv I \)
- This is ok because for \( I \) to be loop invariant, it must hold before the loop

Verification with Approximate Weakest Preconditions

- Now suppose we want to verify \( \{P\}S\{Q\} \)
- If \( P \Rightarrow \text{awp}(S, Q) \), does this mean \( \{P\}S\{Q\} \) is valid?
- No, two problems with \( \text{awp}(\text{while } C \text{ do } \{I\} S, Q) \)
  1. We haven’t checked \( I \) is an actual loop invariant
  2. We also haven’t made sure \( I \land \neg C \) is sufficient to establish \( Q \)
- Thus, we’ll generate additional verification conditions to ensure these two conditions hold!
- For each statement \( S \), we’ll define \( VC(S, Q) \) that encodes additional conditions that must be checked
Generating Verification Conditions

- Most interesting VC generation rule is for loops: 
  \[ VC(while\ C\ do\ \{I\}\ S\ Q) =? \]
- To ensure \(Q\) is satisfied after loop, what condition must hold? \(I \land \neg C \Rightarrow Q\)
- Assuming \(I\) holds initially, need to check \(I\) is loop invariant
  - i.e., need to prove \(\{I \land C\} S(I)\)
- How can we prove this? check validity of \(I \land C \Rightarrow awp(S,I) \land VC(S,I)\)

Verification Of Hoare Triple

- Thus, to show validity of \(\{P\} S \{Q\}\), need to do following:
  1. Compute \(awp(S,Q)\)
  2. Compute \(VC(S,Q)\)
- Theorem: \(\{P\} S \{Q\}\) is valid if following formula is valid:
  \[ VC(S,Q) \land (P \to awp(S,Q)) \quad (\ast) \]
- Thus, if we can prove validity of \((\ast)\), we have shown that program obeys specification

Discussion

- Theorem: \(\{P\} S \{Q\}\) is valid if following formula is valid:
  \[ VC(S,Q) \land P \to awp(S,Q) \quad (\ast) \]
- Question: If \(\{P\} S \{Q\}\) is valid, is \((\ast)\) valid?
- No, for two reasons:
  1. Loop invariant might not be strong enough
  2. Loop invariant might be bogus
- Thus, even if program obeys specification, might not be able to prove it b/c loop invariants we use are not strong enough

Verification Condition for Loops

- Putting this together, verification condition for a while loop \(S' = while\ C\ do\ \{I\}\ S\ Q\) is:
  \[ VC(S', Q) = (I \land C \Rightarrow awp(S, I) \land VC(S, I)) \land (I \land \neg C \Rightarrow Q) \]
- We also need rules to generate VC’s for other statements because there might be loops nested in them
- Thus, VC generation rules for other statement simply propagate VC’s for nested loops up
  - \(VC(x = E, Q) = true\)
  - \(VC(s_1; s_2, Q) = VC(s_2, Q) \land VC(s_1, awp(s_2, Q))\)
  - \(VC(if\ C\ then\ s_1\ else\ s_2, Q) = VC(s_1, Q) \land VC(s_2, Q)\)

Example

- Show how to prove correctness of the following code:
  \[ \{x = 0 \land n = 10\} while(x < n) do \ x++ \{x = 10\}\]
- What loop invariant do we need? \(\{x \leq n \land n = 10\}\)
- What formula do we give to the theorem prover?
  1. \((x = 0 \land n = 10) \Rightarrow (x \leq n \land n = 10)\) \(\text{(awp)}\)
  2. \((x < n \land x \leq n \land n = 10) \Rightarrow (x + 1 \leq n \land n = 10)\) \(\text{(inv)}\)
  3. \((\neg(x < n) \land x \leq n \land n = 10) \Rightarrow x = 10\) \(\text{(post)}\)

Discussion, cont.

- Generating verification conditions using weakest preconditions gives a way to automate reasoning in Hoare logic
- But assuming either a human or some other tool gives us appropriate loop invariants
- In general, finding inductive loop invariants not easy even for humans – requires intuition, understanding of program
- Static analysis techniques can automate task of finding some loop invariants, but not all
- If you want fully automated verification, you have to tolerate false alarms!