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.

Proof Rule for While

- Consider the statement while $C$ 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:
  $$\vdash \{P \land C\}S\{P\} \Rightarrow \vdash \{P\}\text{while}C\text{ do } S\{P \land \neg C\}$$
- 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?

Example

Consider the statement $S = \text{while } x < n \text{ do } x = x + 1$
- Let’s prove validity of $\{x \leq n\}S\{x \geq n\}$
- 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
  $$\vdash \{x \leq n\}[x + 1/x] \Rightarrow \vdash x = x + 1\{x \leq n\} \Rightarrow \vdash x + 1\{x \leq n\} \Rightarrow \vdash x = x + 1\{x \leq n\}$$

Example, cont

- Ok, we’ve shown $x \leq n$ is loop invariant, now let’s instantiate proof rule for while with this loop invariant:
  $$\vdash \{x \leq n \land x < n\}S'[x \leq n] \Rightarrow \vdash \{x \leq n\}\text{while}x < n\text{ do } S'[x \leq n \land \neg(x < n)]$$
- Recall: We wanted to prove the Hoare triple $\{x \leq n\}S\{x \geq n\}$
- In addition to proof rule for while, what other rule do we need? postcondition weakening
Example, cont.

The full proof:

\[ \vdash \{ x + 1 \leq n \} x = x + 1 \{ x \leq n \} \]
\[ x \leq n \land x < n \Rightarrow x + 1 < n \]
\[ \vdash \{ x \leq n \land x < n \} x = x + 1 \{ x \leq n \} \]
\[ \vdash \{ x \leq n \} S \{ x \leq n \land \neg(x < n) \} \text{ holds} \]
\[ x \leq n \land \neg(x < n) \Rightarrow x \geq n \]
\[ \vdash \{ x \leq n \} S \{ x \geq n \} \]

Exercise

Find inductive loop invariant to prove the following Hoare triple:

\[ \{ i = 0 \land j = 0 \land n = 5 \} \]
\[ \text{while } i < n \text{ do } i := i + 1; \ j := j + 1 \]
\[ \{ j = 15 \} \]

- Inductive loop invariant I:
  \[ 2j = i(i + 1) \land i \leq n \land n = 5 \]
- Weakest precondition P w.r.t loop body:
  \[ 2j = i(i + 1) \land i + 1 \leq n \land n = 5 \]
- Since \( I \land C \Rightarrow P, \) I is inductive.

Meta-theory: Soundness of Proof Rules

- It can be show that the proof rules for Hoare logic are sound:
  \[ \text{If } \vdash \{ P \} S \{ Q \}, \text{ then } \models \{ P \} S \{ Q \} \]
- That is, if a Hoare triple \( \{ P \} S \{ Q \} \) is provable using the proof rules, then \( \{ P \} S \{ Q \} \) is indeed valid
- Completeness of proof rules means that if \( \{ P \} S \{ Q \} \) is a valid
  Hoare triple, then it can be proven using our proof rules, i.e.,
  \[ \text{If } \models \{ P \} S \{ Q \}, \text{ then } \vdash \{ P \} S \{ Q \} \]
- Unfortunately, completeness does not hold!

Invariant vs. Inductive Invariant

- Suppose I is a loop invariant for while C do S.
- Does it always satisfy \( \{ I \land C \} S \{ I \} \)?
- Counterexample: Consider I = \( j \geq 1 \) and the code:
  \[ i := 1; \ j := 1; \text{ while } i < n \text{ do } \{ j := j + 1; \ i := i + 1 \} \]
- But strengthened invariant \( j \geq 1 \land i \geq 1 \) does satisfy it
- Such invariants are called inductive invariants, and they are the only invariants that we can prove
- Key challenge in verification is finding inductive loop invariants

Summary of Proof Rules

1. \[ \vdash \{ Q[E/x] \} x = E \{ Q \} \] (Assignment)
2. \[ \vdash \{ P' \} S \{ Q \} \]
   \[ P \Rightarrow P' \]
   \[ \vdash \{ P \} S \{ Q \} \] (Strengthen P)
3. \[ \vdash \{ P \} S \{ Q' \} \]
   \[ Q' \Rightarrow Q \]
   \[ \vdash \{ P \} S \{ Q \} \] (Weaken Q)
4. \[ \vdash \{ P \} C_1 \{ Q \} \]
   \[ \vdash \{ Q \} C_2 \{ R \} \]
   \[ \vdash \{ P \} C_1 \{ Q \} \]
   \[ \vdash \{ P \} C_1 \{ Q \} \]
   \[ \vdash \{ P \} \] (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)

Meta-theory: Relative Completeness

- Recall: Rules for precondition strengthening and postcondition weakening require checking \( A \Rightarrow B \)
- In general, these formulas belong to Peano arithmetic
- Since PA is incomplete, there are implications that are valid but cannot be proven
- However, Hoare’s proof rules still have important goodness guarantee: relative completeness
- If we have an oracle for deciding whether an implication \( A \Rightarrow B \) holds, then any valid Hoare triple can be proven using our proof rules
Automating Reasoning in Hoare Logic

- Manually proving correctness is tedious, so we'd like to automate the tedious parts of program verification
- Idea: Assume an oracle gives loop invariants, but automate the rest of the reasoning
- This oracle can either be a human or a static analysis tool (e.g., abstract interpretation)

Generating VCs: Forwards vs. Backwards

- Two ways to generate verification conditions: forwards or backwards
- A forwards analysis starts from precondition and generates formulas to prove postcondition
- Forwards technique computes strongest postconditions (sp)
- In contrast, backwards analysis starts from postcondition and tries to prove precondition
- Backwards technique computes weakest preconditions (wp)
- We'll use the backwards method

Weakest Preconditions

- Idea: Suppose we want to verify Hoare triple \{P\}S\{Q\}
- We'll start with Q and going backwards, compute formula wp(S, Q) called weakest precondition of Q w.r.t. to S
- 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:
  \[ P \Rightarrow \text{wp}(S, Q) \]
- Why? Because if triple \{P'\}S\{Q\} is valid and \( P \Rightarrow P' \), then \{P\}S\{Q\} is also valid

Example

- Consider the following code S:
  \[
x := y + 1; \text{ if } x > 0 \text{ then } z := 1 \text{ else } z := -1
  \]
- What is wp(S, z > 0)? \( y \geq 0 \)
- What is wp(S, z \leq 0)? \( y < 0 \)
- Can we prove post-condition \( z = 1 \) if precondition is \( y \geq -1 \)?
- What if precondition is \( y > -1 \)?

Defining Weakest Preconditions

- Weakest preconditions are defined inductively and follow Hoare's proof rules
  - wp(x := E, Q) = Q[E/x]
  - wp(a_1; a_2, Q) = wp(a_1, wp(a_2, Q))
  - wp(if C then s_1 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, wp of then branch must hold; otherwise, wp of else branch must hold"
Verification with Approximate Weakest Preconditions

- If $P \Rightarrow awp(S, Q)$, does this mean $\{P\} S \{Q\}$ is valid?
  
  No, two problems with $awp(\text{while } C \text{ do } \{I\} S, Q)$
  
  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 $Q$!

- For each statement $S$, generate verification condition $VC(S, Q)$ that encodes additional conditions to prove

Generating Verification Conditions

- Most interesting VC generation rule is for loops:
  
  $VC(\text{while } C \text{ do } \{I\} S, Q) = \text{?}$

- 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 Condition for Loops

- To summarize, to show $I$ is preserved in loop, need:
  
  $I \land C \Rightarrow awp(S, I) \land VC(S, I)$

- To show $I$ is strong enough to establish $Q$, need:

  $I \land \neg C \Rightarrow Q$

- Putting this together, verification condition for a while loop $S' = \text{while } C \text{ do } \{I\} S$ is:

  $VC(S', Q) = (I \land C \Rightarrow awp(S, I) \land VC(S, I)) \land (I \land \neg C \Rightarrow Q)$

Verification Condition for Other Statements

- We also need rules to generate VC’s for other statements because there might be loops nested in them

  $VC(x := E, Q) = \text{true}$

  $VC(s_1; s_2, Q) = VC(s_2, Q) \land VC(s_1, awp(s_2, Q))$

  $VC(\text{if } C \text{ then } s_1 \text{ else } s_2, Q) = VC(s_1, Q) \land VC(s_2, Q)$
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:
  \[ VC(S, Q) \land P \rightarrow awp(S, Q) \] (*)

Thus, if we can prove validity of (*), we have shown that program obeys specification.

Discussion

Theorem: \( \{P\}S\{Q\} \) is valid if following formula is valid:

\[ VC(S, Q) \land P \rightarrow awp(S, Q) \] (*)

- Question: If \( \{P\}S\{Q\} \) is valid, is (*) 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.

Example

- Consider the following code:
  
  ```
  i := 1; sum := 0;
  while i ≤ n do [sum ≥ 0] {
    j := 1;
    while j ≤ i do [sum ≥ 0 ∧ j ≥ 0] {
      sum := sum + j; j := j + 1
    }
    i := i + 1
  }
  ```

- Show the VC’s generated for this program for post-condition \( \sum \geq 0 \) – can it be verified?
- What is the post-condition we need to show for inner loop? \( \sum \geq 0 \)

Example, cont.

- Generate VC’s for inner loop:
  
  \[ \begin{align*}
  (1) \quad (\sum \geq 0 \land j \geq 0 \land j > i) \Rightarrow \sum \geq 0 \\
  (2) \quad (j \leq i \land \sum \geq 0 \land j \geq 0) \Rightarrow (\sum + j \geq 0 \land j + 1 \geq 0)
  \end{align*} \]

- Finally, compute \( awp \) for outer loop: \( (5) \ 0 \geq 0 \)

- Feed the formula \((1) \land (2) \land (3) \land (4) \land (5)\) to SMT solver

- It’s valid; hence program is verified!

Example: Variant

- Suppose annotated invariant for inner loop was \( \sum \geq 0 \) instead of \( \sum \geq 0 \land j \geq 0 \)
- Could the program be verified then? No, because loop invariant not strong enough
- While VC generation handles many tedious aspects of the proof, user must still come up with loop invariants...

Guess-and-Check

- Fortunately, there are many automated techniques for loop invariant generation
  
  - The simplest technique is guess-and-check
  
  - Given template of invariants (e.g., \( ? = ?\), \( ? \leq \)), instantiate the holes with program variables and constants

  - Then, check if it’s an invariant; if not, try a different instantiation
Abstract Interpretation

- Symbolically execute the program over an abstraction until we reach a fixed point

- **Example:** In sign abstract domain, only track if a variable \( x \) is positive, non-negative, negative, or zero

- This defines a lattice:

```
non-neg  neg
pos    zero
1
```

- Initialize everything to \( \bot \) and then take the join of the new value with old value; repeat until you reach fixed point

Fixed-Point Computation

```
x = 0;
y = 0;
while (y <= n)
{
   if (z == 0)
      x = x + 1;
   else
      x = x + y;
   y = y + 1;
}
```

Abstract Interpretation, cont.

- The sign abstract domain allows inferring simple invariants of the form \( x \geq 0, x < 0 \) etc.

- More interesting abstract domains:
  - Intervals: Tracks ranges (e.g., \( x \in [0,100] \))
  - Polyhedra: Tracks linear inequalities (e.g., \( x \leq y + z \))
  - Karr’s domain: Tracks linear equalities (e.g., \( x = y + z \))

- In these domains, we may not reach a fixed point; apply so-called widening operation to force fixed-point

Conclusion

- Program verification automates reasoning about program correctness

- In this lecture, we assumed oracle provides loop invariants

- Many different techniques for automating loop invariant generation; active research area

- Some other challenges: how to reason about the heap, concurrency, recursive functions ...

- Since program verification is undecidable, we can’t always verify every correct program, but can verify many