Introduction to Deductive Program Verification

Iｓｉｌ Dillig

Hoare Logic I

Example specs: safety (no crashes), absence of arithmetic overflow, complex behavioral property (e.g., "sorts an array")

Verification condition: An SMT formula $\phi$ s.t. program is correct iff $\phi$ is valid

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 $E := Z \mid V \mid e_1 + e_2 \mid e_1 \times e_2$

Conditional $C := true \mid false \mid e_1 = e_2 \mid e_1 \leq e_2$

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

In Hoare logic, we specify partial correctness of programs using Hoare triples:

$\{P\} S \{Q\}$

Here, $S$ is a statement in programming language IMP

$P$ and $Q$ are SMT formulas

$P$ 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?

What about $\{x = 0 \land y = 1\} x := x + 1 \{x = 1 \land y = 2\}$?

What about $\{x = 0\} x := x + 1 \{x = 1 \lor y = 2\}$?

What about $\{x = 0\}$ while true do $x := 0 \{x = 1\}$?
Inference Rules

Proof rules in Hoare logic are written as inference rules:

\[ \vdash \{P_1\}S_1\{Q_1\} \ldots \vdash \{P_n\}S_n\{Q_n\} \vdash \{P\}S\{Q\} \]

- Says if Hoare triples \( \{P_1\}S_1\{Q_1\}, \ldots, \{P_n\}S_n\{Q_n\} \) are provable in our proof system, then \( \{P\}S\{Q\} \) is also provable.
- Not all rules have hypotheses: these correspond to bases cases in the proof
- Rules with hypotheses correspond to inductive cases in proof
- One inference rule for every statement in the IMP language

Proving Partial Correctness

Key problem: How to prove valid Hoare triples?

- If a Hoare triple is valid, written \( \models \{P\}S\{Q\} \), we want a proof system to prove its validity
- Use notation \( \vdash \{P\}S\{Q\} \) to indicate that we can prove validity of Hoare triple
- Hoare also gave a sound and (relatively-) complete proof system that allows semi-mechanizing correctness proofs
- Soundness: If \( \vdash \{P\}S\{Q\} \), then \( \models \{P\}S\{Q\} \)
- Completeness: If \( \models \{P\}S\{Q\} \), then \( \vdash \{P\}S\{Q\} \)

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 specification 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 \( \{x = 0\} \) while true do \( x := 0 \)[\( x = 1 \)] valid?

Example Specifications

- What does \( \{true\}S\{Q\} \) say?
- What about \( \{P\}S[true] \)?
- What about \( [P]S[true] \)?
- When does \( \{true\}S[false] \) hold?
- When does \( [false]S[Q] \) hold?
- We’ll only focus on only partial correctness (safety)
- Total correctness = Partial correctness + termination

More Examples

Valid or invalid?

- \( \{0\} \) while \( i < n \) do \( i++ \); \( \{i = n\} \)
- \( \{0\} \) while \( i < n \) do \( i++ \); \( \{i >= n\} \)
- \( \{0 \land j = 0\} \) while \( i < n \) do \( i++ \); \( j++ \) \( \{2j = n(n + 1)\} \)
- How can we strengthen the precondition so it’s valid?

Understanding Proof Rule for Assignment

- Consider the assignment \( x := y \) and post-condition \( x > 2 \)
- What do we need before the assignment so that \( x > 2 \) holds afterwards?
- Consider \( i := i + 1 \) and post-condition \( i > 10 \)
- What do we need to know before the assignment so that \( i > 10 \) holds afterwards?
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?
  
  - \( \{ y = 4 \} \ x := 4 \ \{ y = x \} \)
  
  - \( \{ x + 1 = n \} \ x := x + 1 \ \{ x = n \} \)
  
  - \( \{ y = x \} \ y := 2 \ \{ y = x \} \)
  
  - \( \{ z = 3 \} \ y := x \ \{ z = 3 \} \)

Exercise

- Your friend suggests the following proof rule for assignment:
  \[ \vdash \{(x = E) \rightarrow Q\} \ x := E \ \{ Q \} \]

- Is the proposed proof rule correct?

- Motivation for Precondition Strengthening

- Is the Hoare triple \( \{ z = 2 \} \ y := x \ \{ y = x \} \) valid?

- Is this Hoare triple provable using our assignment rule?

- Instantiating the assignment rule, we get:
  \[ \{ y = x \ y := x \ \{ y = x \} \} \]

- But intuitively, if we can prove \( y = x \) w/o any assumptions, we should also be able to prove it if we do make assumptions!

Proof Rule for Precondition Strengthening
\[ \vdash \{ P \} \ S \ \{ Q \} \quad P \Rightarrow P' \]
\[ \vdash \{ P \} \ S \ \{ Q \} \]

- Recall: \( P \Rightarrow P' \) means the formula \( P \rightarrow P' \) is valid

- Hence, need to use SMT solver every time we use precondition strengthening!

Example

- Using this rule and rule for assignment, we can now prove \( \{ z = 2 \} \ y := x \ \{ y = x \} \)

- Proof:
  \[ \vdash \{ y = x \ y := x \ \{ y = x \} \} \]
  \[ \vdash \{ \text{true} \} \ y := x \ \{ y = x \} \]
  \[ \vdash \{ z = 2 \} \ y := x \ \{ y = x \} \]

Proof Rule for Post-Condition Weakening
\[ \vdash \{ P \} \ S \ \{ Q \} \quad Q' \Rightarrow Q \]
\[ \vdash \{ P \} \ S \ \{ Q \} \]

- We also need a dual rule for post-conditions called **post-condition weakening**:

- If we can prove post-condition \( Q' \), we can always relax it to something weaker

- Again, need to use SMT solver when applying post-condition weakening
Proof Rule for If Statements

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

- 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?
- 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?
- What do we need to show in else branch?

Exercise, cont

- Yes, this rule can be derived from existing rules.
- From premises, we know (1) \(\{P \land C\} S_1; S_3\{Q\}\) and (2) \(\{P \land \neg C\} S_2; S_3\{Q\}\)
- Let \(Q'\) be the weakest precondition we need for \(Q\) to hold after executing \(S_3\), i.e., (3) \(\{Q'\} S_3\{Q\}\)
- Using premises, this means \(\{P \land C\} S_1\{Q'\}\) and \(\{P \land \neg C\} S_2\{Q'\}\)
- From these and existing if rule, we can derive:
  \(\{P\} \text{if } C \text{ then } S_1 \text{ else } S_2 \{Q'\}\)
- Conclusion follows from these and (3) using Seq

Post-condition Weakening Examples

- Suppose we can prove \(\{\text{true}\} S(x = y \land z = 2)\).
- Using post-condition weakening, which of these can we prove?
  - \(\{\text{true}\} S(x = y)\)
  - \(\{\text{true}\} S(z = 2)\)
  - \(\{\text{true}\} S(z > 0)\)
  - \(\{\text{true}\} S(x = y)\)
  - \(\{\text{true}\} S(\exists y. x = y)\)

Proof Rule for Composition

\[
\frac{\vdash \{P\} S_1 \{Q\} \quad \vdash \{Q\} S_2 \{R\}}{
\vdash \{P\} S_1; S_2 \{R\}}
\]

- Using this rule, let’s prove validity of Hoare triple:
  \(\{\text{true}\} x = 2; y = x \{y = 2 \land x = 2\}\)
- What is appropriate \(Q\)?
  \(\{x = 2; y = 2\} \quad x = 2 \quad y = x \{x = 2 \land y = 2\}\)

Example

Prove the correctness of this Hoare triple:
\(\{\text{true}\} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \{y \geq 0\}\)

Exercise

- Your friend suggests the following proof rule:
  \(\{P \land C\} S_1; S_3\{Q\}\)
  \(\{P \land \neg C\} S_2; S_3\{Q\}\)
  \(\{P\} \text{ if } C \text{ then } S_1 \text{ else } S_2\{Q\}\)
- Is this proof rule correct? If so, prove your answer. Otherwise, give a counterexample.
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 $\text{while } C \text{ do } S$.
- Suppose $I$ is a loop invariant for this loop. What is $I$?

Loop Invariants

- Based on this rule, why is $P$ a loop invariant?

Example

- Consider the following code:
  
  ```plaintext
  i := 0; j := 0; n := 10; while i<n do i:= i + 1; j := i + j
  ```

Example, cont.

- Which of the following are loop invariants?
  - $i \leq n$
  - $i < n$
  - $j \geq 0$

Example, cont.

- Suppose $I$ is a loop invariant. Does $I$ also hold after loop terminates?

The full proof:

\[
\begin{align*}
\vdash\{x + 1 \leq n\} x = x + 1 \{x \leq n\} \\
\vdash\{x \leq n\} \Rightarrow x + 1 < n \\
\vdash\{x \leq n\} S \{x \leq n \wedge (x < n)\} \\
\vdash\{x \leq n\} \Rightarrow x \geq n \\
\vdash\{x \leq n\} S \{x \geq n\}
\end{align*}
\]
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

Another Exercise

- Suppose we add a for loop construct to IMP:
  \[
  \text{for } v := e_1 \text{ until } e_2 \text{ do } S
  \]
- Initializes \( v \) to \( e_1 \), increments \( v \) by 1 in each iteration and terminates when \( v > e_2 \)
- Write a proof rule for this for loop construct
  \[
  \{ I \land v \leq e_2 \} S; \; v := v + 1 \{ I \}
  \]
- We can de-sugar into while loop:
  \[
  v := e_1; \; \text{while } v \leq e_2 \text{ do } \{ S; \; v := v + 1 \}
  \]

Exercise, cont.

- Putting all this together, we get the following proof rule:
  \[
  \begin{align*}
  \{ P \} \; v := e_1 \{ I \}
  \\
  \{ I \land v \leq e_2 \} \; S; \; v := v + 1 \{ I \}
  \\
  \{ P \} \; \text{for } v := e_1 \text{ until } e_2 \text{ do } S \{ I \land v > e_2 \}
  \end{align*}
  \]

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 \( I \):
- 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.

Arrays

- Let’s add arrays to our IMP language:
  \[
  v[e_1] := e_2
  \]
- What is the proof rule for this statement?
- Idea 1: Treat array write just like assignment:
  \[
  \{ Q[e_2/v[e_1]] \} \; v[e_1] := e_2 \{ Q \}
  \]
- Is this rule correct?
Counterexample

- No, counterexample:
  \( \{ i = 1 \} \ v[1] := 3; v[1] := 2 \ \{ v[i] = 3 \} \)
- What is the value of \( v[1] \) after this code?
- But using previous “proof rule”, we can “prove” this Hoare triple
- Clearly, this rule is unsound

Correct Proof Rule for Arrays

- The correct proof rule:
  \( \{ Q|v(e_1 \circ e_2)/v \} \ v[e_1] := e_2 \ \{ Q \} \)
- Effectively assigns \( v \) to a new array that is the same as \( v \) except at index \( e_1 \)
- We now require theory of arrays

Array Example

- Consider again this example:
  \( \{ i = 1 \} \ v[1] := 3; v[1] := 2 \ \{ v[i] = 3 \} \)
- Applying the array write rule, we obtain:
  \( \{ v(1 \triangleleft 2)[i] = 3 \} \ v[1] := 2 \ \{ v[i] = 3 \} \)
- Use composition and apply array rule to first statement:
  \( \{ (v(i \triangleleft 3))(1 \triangleleft 2)[i] = 3 \} \ v[1] := 3 \ \{ v(1 \triangleleft 2)[i] = 3 \} \)
- But the following implication is not valid:
  \( i = 1 \Rightarrow (v(i \triangleleft 3))(1 \triangleleft 2)[i] = 3 \)

Example with Arrays and Loops

- Consider the following code snippet:
  \( \text{while } i < n \text{ do } \{ a[i] := 0; i := i + 1; \} \)
- Suppose the precondition is \( i = 0 \land n > 0 \) and the postcondition is:
  \( \forall j. 0 \leq j < n \Rightarrow a[j] = 0 \)
- Find an inductive loop invariant and show the correctness proof
- Inductive invariant:

Summary of Proof Rules

1. \( \vdash \{ Q|E/x \} \ z = E \ \{ Q \} \) (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. \( \vdash \{ P \} C_1(Q) \ \vdash \{ Q \} C_2(R) \ \vdash \{ P \} C_1; C_2(R) \) (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)

Meta-theory: Soundness of Proof Rules

- It can be show that the proof rules for Hoare logic are sound:
  \[ \vdash \{ P \} S(Q) \]
  If \( \vdash \{ P \} S(Q) \)
- 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.,
  \[ \vdash \{ P \} S(Q) \]
  If \( \vdash \{ P \} S(Q) \)
- Unfortunately, completeness does not hold!
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).

Basic Idea Behind Program Verification

- Automating Hoare logic is based on generating verification conditions (VC).
- A verification condition is a formula $\phi$ such that program is correct iff $\phi$ is valid.
- Deductive verification has two components:
  1. Generate VC’s from source code.
  2. Use theorem prover to check validity of formulas from step 1.

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:
  $$ P \Rightarrow wp(S, Q) $$

Defining Weakest Preconditions

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

- Consider the following code $S$:
  
  $$x := y + 1; \text{if } x > 0 \text{ then } z := 1 \text{ else } z := -1$$

- What is $wp(S, z > 0)$?
- What is $wp(S, z < 0)$?
- Can we prove post-condition $z = 1$ if precondition is $y \geq -1$?
- What if precondition is $y > -1$?

Weakest Preconditions for Loops

- Unfortunately, we can’t compute weakest preconditions for loops exactly...

  - Idea: approximate it using $awp(S, Q)$

  - $awp(S, Q)$ may be stronger than $wp(S, Q)$ but not weaker

  - To verify $\{P\}S\{Q\}$, show $P \Rightarrow awp(S, Q)$

  - Hope is that $awp(S, Q)$ is weak enough to be implied by $P$
    although it may not be the weakest

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:
  
  $$VC(\text{while } C \text{ do } [I] S, Q) = ?$$

- 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:
  $$VC(S, Q) \land P \rightarrow awp(S, Q) \quad (*)$$
- Thus, if we can prove of validity of $(*)$, we have shown that program obeys specification

Discussion

- Theorem: $\{P\}S\{Q\}$ is valid if following formula is valid:
  $$VC(S, Q) \land P \rightarrow awp(S, Q) \quad (*)$$
- 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
    }
  }
  ```
- What is the post-condition we need to show for inner loop? $sum ≥ 0$

Example, cont.

- Generate VC's for inner loop:
  1. $(sum ≥ 0 ∧ j ≥ 0 ∧ j > i) \Rightarrow sum ≥ 0$
  2. $(i ≤ n ∧ sum ≥ 0 ∧ j ≥ 0) \Rightarrow (sum + j ≥ 0 ∧ j + 1 ≥ 0)$
- Now, generate VC's for outer loop:
  3. $(i ≤ n ∧ sum ≥ 0) \Rightarrow (sum ≥ 0 ∧ 1 ≥ 0)$
  4. $(i > n ∧ sum ≥ 0) \Rightarrow sum ≥ 0$
- Finally, compute awp for outer loop: $(5) \ 0 ≥ 0$
- 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 $sum ≥ 0$ instead of $sum ≥ 0 ∧ j ≥ 0$
- 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 (more on this in next few lectures)