Algorithm for Proving Hoare Triples?

- Have seen in Isabelle that much of proving a Hoare triple is routine
- Will this always work?
- Why not automate the whole process?
  - Can’t (always) calculate needed loop invariants
  - Can’t (always) prove implications (side-conditions) in Rule of Consequence application
- Can we automate all but this?
  1. Annotate all while loops with needed invariants
  2. Use routine to “roll back” post-condition to weakest precondition, gathering side-conditions as we go
  2. called verification condition generation

Relation Between Two Languages

- Hoare Logic for Simple Imperative Programs and Hoare Logic for Annotated Programs almost the same
- Give verification conditions for an annotated version of our simple imperative language
- Add a presumed invariant to each while loop

\[
\begin{align*}
\langle \text{command} \rangle & ::= \langle \text{variable} \rangle ::= \langle \text{term} \rangle \\
& | \langle \text{command} \rangle ; \ldots ; \langle \text{command} \rangle \\
& | \text{if } \langle \text{statement} \rangle \text{ then } \langle \text{command} \rangle \text{ else } \langle \text{command} \rangle \\
& | \text{while } \langle \text{statement} \rangle \text{ inv } \langle \text{statement} \rangle \text{ do } \langle \text{command} \rangle \\
\end{align*}
\]

Example:

```plaintext
while y < n inv x = y * y
do
x := (2 * y) + 1;
y := y + 1
od
```

Relation Between Two Hoare Logics

- Hoare Logic for Simple Imperative Programs and Hoare Logic for Annotated Programs almost the same
- What it precise relationship?
- First need precise relation between the two languages

Definition

- strip(v := e) = v := e
- strip(C1 ; C2) = strip(C1) ; strip(C2)
- strip(if B then C1 else C2 fi) = if B then strip(C1) else strip(C2) fi
- strip(while B inv P do C od) = while B do strip(C) od

- We recursively remove all invariant annotations from all while loops

### Assignment Rule

\[
\langle P \{e/x\} \rangle \times = e \langle P \rangle
\]

### Rule of Consequence

\[
\begin{align*}
P & \Rightarrow P' & (P') & \Rightarrow (P) & \Rightarrow C & \Rightarrow Q' & \Rightarrow Q \\
\end{align*}
\]

### Sequencing Rule

\[
\begin{align*}
\langle P \rangle & \Rightarrow C_1 & \Rightarrow (Q) & \Rightarrow C_2 & \Rightarrow (R) \\
\end{align*}
\]

### If Then Else Rule

\[
\begin{align*}
\langle P \rangle & \Rightarrow C_1 & \Rightarrow (Q) & \Rightarrow C_2 & \Rightarrow (R) \\
\end{align*}
\]

### While Rule

\[
\begin{align*}
\langle P \rangle & \Rightarrow C & \Rightarrow (P) & \Rightarrow P & \text{while } B & \text{ inv } P & \text{ do } C & \Rightarrow (P \land \neg B) \\
\end{align*}
\]
Relation Between Two Hoare Logics

Weakest Justification

Theorem
For all pre- and post-conditions \( P \) and \( Q \), and unannotated programs \( C \), if \( \{ P \} C \{ Q \} \), then there exists an annotated program \( S \) such that \( C = \text{strip}(S) \) and \( \{ P \} S \{ Q \} \).

Proof.
(Sketch) Use rule induction on proof of \( \{ P \} C \{ Q \} \); in case of While Rule, add invariant from precondition as invariant to command.

Weakest Precondition

Theorem
For all pre- and post-conditions \( P \) and \( Q \), and unannotated programs \( C \), if \( \{ P \} C \{ Q \} \), then there exists an annotated program \( S \) such that \( C = \text{strip}(S) \) and \( \{ P \} S \{ Q \} \).

Proof.
(Sketch) Use rule induction on proof of \( \{ P \} C \{ Q \} \); in case of While Rule, add invariant from precondition as invariant to command.

What About Precondition?

Question: Do we have \( \{ wp \ C \} Q \{ Q \} \)?
Answer: Not always - need to check while-loop side-conditions – verification conditions

Question: How to calculate verification conditions?
What About Precondition?

**Question:** Do we have \( \{wp \ C \ Q\} \ C \ \{Q\} \)?

**Answer:** Not always - need to check while-loop side-conditions - verification conditions

**Question:** How to calculate verification conditions?

**Definition**

\[
\text{vcg (} x := e \text{) } Q \text{ = true}
\]

Verification Condition Guarantees \( wp \) Precondition

**Theorem**

\[
\text{vcg } C \ Q \Rightarrow \{wp \ C \ Q\} \ C \ \{Q\}
\]

**Proof.**

(Sketch)

- Induct on structure of \( C \)
- For each case, wind back as we did in specific examples:
  - Assignment: \( wp \ C \ Q \) exactly what is needed for Assignment Axiom
  - Sequence: Follows from inductive hypotheses, all elim, and modus ponens
  - If Then Else: Need to use Precondition Strengthening with each branch of conditional; \( wp \) and inductive hypotheses give the needed side conditions
  - While: Need to use Postcondition Weakening, While Rule and Precondition Strengthening

What About Precondition?

**Question:** Do we have \( \{wp \ C \ Q\} \ C \ \{Q\} \)?

**Answer:** Not always - need to check while-loop side-conditions - verification conditions

**Question:** How to calculate verification conditions?

**Definition**

\[
\begin{align*}
\text{vcg (} x := e \text{) } Q &= \text{true} \\
\text{vcg (} C_1 ; C_2 \text{) } Q &= (\text{vcg } C_1 \ (wp \ C_2 \ Q)) \land (\text{vcg } C_2 \ Q) \\
\text{vcg (if } B \text{ then } C_1 \text{ else } C_2 \text{ fi) } Q &= (\text{vcg } C_1 \ Q) \land (\text{vcg } C_2 \ Q)
\end{align*}
\]

Verification Condition Guarantees \( wp \) Precondition

**Corollary**

\[
\{(P \Rightarrow wp \ C \ Q) \land (vcg \ C \ Q)\} \Rightarrow \{P\} \ C \ \{Q\}
\]

This amounts to a method for proving Hoare triple \( \{P\} \ C \ \{Q\} \):

1. Annotate program with loop invariants
2. Calculate \( wp \ C \ Q \) and \( vcg \ C \ Q \) (automated)
3. Prove \( P \Rightarrow wp \ C \ Q \) and \( vcg \ C \ Q \)

Basic outline of interaction with Boogie: Human does 1, Boogie does 2, Z3 / Simplify / Isabelle + human / ... does 3

For more information

Model For Hoare Logic

- Seen proof system for Hoare Logic
- What about models?
- Informally, triple modeled by pairs of assignments of program variables to values where executing program starting with initial assignment results in a memory that gives the final assignment
- Calls for alternate definition of execution

Elsa L Gunter
CS477 Formal Software Dev Methods
March 16, 2018

Natural Semantics

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

  - (C, m) \downarrow m'
  - (E, m) \downarrow v

Simple Imperative Programming Language

- I ∈ Identifiers
- N ∈ Numerals
- B ::= true | false | B & B | B or B | not B | E < E | E = E
- E ::= N | I | E + E | E * E | E - E | - E
- C ::= skip | C;C | I ::= E | if B then C else C fi | while B do C od

Booleans:

- (B, m) \downarrow false
- (B & B', m) \downarrow false
- (B & B', m) \downarrow b
- (B, m) \downarrow true
- (B or B', m) \downarrow true
- (B or B', m) \downarrow b
- (not B, m) \downarrow false
- (not B, m) \downarrow true

Relations

- (E, m) \downarrow U
- (E', m) \downarrow V
- U \sim V = b

- By U \sim V = b, we mean does (the meaning of) the relation \sim hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V
Arithmetic Expressions

\[(E, m) \downarrow U \ (E', m) \downarrow V \ U \ op \ V = N\]
\[(E \ op \ E', m) \downarrow N\]
where \(N\) is the specified value for \(U \ op \ V\)

Commands

Skip: \((\text{skip}, m) \downarrow m\)

Assignment: \((E,m) \downarrow V\)
\[(I::=E,m) \downarrow m[I <-- V]\]

Sequencing: \((C,m) \downarrow m\)
\[(C,m) \downarrow m\]
\[(C;C, m) \downarrow m\]

If Then Else Command

\[(B,m) \downarrow \text{true} \ (C,m) \downarrow m'\]
\[(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \downarrow m'\]

\[(B,m) \downarrow \text{false} \ (C',m) \downarrow m'\]
\[(\text{if } B \text{ then } C' \text{ else } C' \text{ fi}, m) \downarrow m'\]

While Command

\[(B,m) \downarrow \text{false}\]
\[(\text{while } B \text{ do } C \text{ od}, m) \downarrow m\]

\[(B,m) \downarrow \text{true} \ (C,m) \downarrow m'\]
\[(\text{while } B \text{ do } C \text{ od}, m) \downarrow m''\]

Example: If Then Else Rule

\((x > 5, \{x -> 7\})\)?
\[(x > 5 \text{ then } y:= 2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x -> 7\}) \downarrow ?\]
Example: Arith Relation

? > ? = ?
(x, {x -> 7}) ⊥ ?  (5, {x -> 7}) ⊥ ?
(x > 5, {x -> 7}) ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?

Example: Identifier(s)

7 > 5 = true
(x, {x -> 7}) ⊥  (5, {x -> 7}) ⊥ 5
(x > 5, {x -> 7}) ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?

Example: Arith Relation

7 > 5 = true
(x, {x -> 7}) ⊥  (5, {x -> 7}) ⊥ 5
(x > 5, {x -> 7}) ⊥ true ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?

Example: Assignment

7 > 5 = true
(x, {x -> 7}) ⊥  (5, {x -> 7}) ⊥ 5
(x > 5, {x -> 7}) ⊥ true ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?

Example: If Then Else Rule

7 > 5 = true
(x, {x -> 7}) ⊥  (5, {x -> 7}) ⊥ 5
(x > 5, {x -> 7}) ⊥ true ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?

Example: Arith Op

? + ? = ?
(x, {x -> 7}) ⊥ ?  (3, {x -> 7}) ⊥ ?
(x > 5, {x -> 7}) ⊥ 2 + 3 ⊥ ?
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊥ ?
Natural Semantics Models Hoare Logic (Soundness)

**Definition**

Say a pair of states (aka assignments) \((m_1, m_2)\) satisfies, or models the Hoare triple \([P] C [Q]\) if \(m_1 \models P\) and \(m_2 \models Q\). Write \((m_1, m_2) \models [P] C [Q]\)

**Theorem**

Let \([P] C [Q]\) be a valid Hoare triple (i.e., it's provable). Let \(m_1\) be a state (aka assignment) such that \(m_1 \models P\). Let \(m_2\) be a state such that \((C, m_1) \downarrow m_2\). Then \((m_1, m_2) \models [P] C [Q]\)

Natural Semantics Models Hoare Logic (Completeness)

**Theorem**

Let \([P] C [Q]\) be a such that for all \(m_1\) and \(m_2\), if (we can prove) \(m_1 \models P\) and \((C, m_1) \downarrow m_2\) then (we can prove) \(m_2 \models Q\). Then \([P] C [Q]\) is provable in Hoare logic.
Simple Imperative Programming Language #2

I ∈ Identifiers
N ∈ Numerals

E ::= N | I | E + E | E ⇤ E | E E | I

B ::= true | false | B & B | B or B | not B | E < E | E = E

C ::= skip | C ; C | { C } | E | if B then C else C fi | while B do C

Changes for Expressions

- Need new type of result for expressions
  
  \[(E, m) \downarrow (v, m')\]

- Modify old rules for expressions:
  
  Atomic Expressions:
  
  \[(I, m) + (true, m) + (false, m)\]

  \[(E, m) + (V , m0) (E0, m0) + (V , m00) U ⇠ V = N\]

  \[(E \downarrow E', m) \downarrow (N, m''')\]

  \[\langle E, m \rangle \downarrow \{ V, m' \}\]

  \[\{ I := E, m \} \downarrow \{ V, m'[I \leftarrow V]\}\]

New Rule for Expressions

Changes for Commands

- Replace rule for Assignment by one for Expressions as Commands:
  
  \[(E, m) \downarrow (v, m')\]

- Unfortunately, can’t stop there
  
  - Relations use Expressions; must be changed
  - Relations produce Booleans; all Booleans must be changed
  - if, then, else and while use Booleans; must be changed

Changes for Boolean Expressions

- Arithmetic Expressions occur in Boolean Expression; must change type of result for Booleans:
  
  \[(B, m) \downarrow (b, m')\]

- Modify old rules for Booleans to reflect new type:
  
  Atomic Booleans:
  
  \[(true, m) \downarrow (true, m)\]

  \[(false, m) \downarrow (false, m)\]
### Changes for Boolean Expressions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (B, m) \downarrow (false, m') ) &amp; ( (B, m) \downarrow (true, m') )</td>
<td>( (B', m') \downarrow (b, m'') )</td>
</tr>
<tr>
<td>( (B &amp; B', m) \downarrow (false, m') )</td>
<td>( (B &amp; B', m) \downarrow (true, m') )</td>
</tr>
</tbody>
</table>
| \( (B 

### Revised if_then_else Rule

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (B, m) \downarrow (true, m') )</td>
<td>( (C, m') \downarrow m'' )</td>
</tr>
<tr>
<td>( (if B \ then \ C \ else \ C0 \ fi, m) \downarrow m'' )</td>
<td>( (if B \ then \ C \ else \ C0 \ fi, m) \downarrow m'' )</td>
</tr>
</tbody>
</table>

### Revised while Rule

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (B, m) \downarrow (false, m') )</td>
<td>( (B, m) \downarrow (true, m') )</td>
</tr>
<tr>
<td>( (B \or B', m) \downarrow (false, m') )</td>
<td>( (B \or B', m) \downarrow (true, m') )</td>
</tr>
<tr>
<td>( (not \ B', m) \downarrow (true, m') )</td>
<td>( (not \ B, m) \downarrow (true, m') )</td>
</tr>
</tbody>
</table>

### Transition Semantics

- Aka “small step structured operational semantics”
- Defines a relation of “one step” of computation, instead of complete evaluation
  - Determines granularity of atomic computations
- Typically have two kinds of “result”: configurations and final values
- Written \( (C, m) \rightarrow (C', m') \) or \( (C, m) \rightarrow m' \)

### Simple Imperative Programming Language #1 (SIMPL1)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I \in )</td>
<td>Identifiers</td>
</tr>
<tr>
<td>( N \in )</td>
<td>Numerals</td>
</tr>
<tr>
<td>( E \ ::= \ N \</td>
<td>\ I \</td>
</tr>
<tr>
<td>( B \ ::= \ true \</td>
<td>\ false \</td>
</tr>
<tr>
<td>( C \ ::= \ skip \</td>
<td>C \</td>
</tr>
<tr>
<td>( if \ B \ then \ C \ else \ C \ fi )</td>
<td></td>
</tr>
<tr>
<td>( while \ B \ do \ C )</td>
<td></td>
</tr>
</tbody>
</table>
Transitions for Atomic Expressions

Identifiers: \( (I, m) \rightarrow m(I) \)
Numerals are values: \( (N, m) \rightarrow N \)
Booleans: \( (true, m) \rightarrow true \)
\( (false, m) \rightarrow false \)

Commands - in English

- **skip** means done evaluating
- When evaluating an assignment, evaluate expression first
- If the expression being assigned is a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

Booleans:

- Values = \{true, false\}
- Operators: (short-circuit)
  - \( (false \& B, m) \rightarrow false \)
  - \( (true \& B, m) \rightarrow (B, m) \)
  - \( (false \lor B, m) \rightarrow (false, m) \)
  - \( (true \lor B, m) \rightarrow (B, m) \)
  - \( (not B, m) \rightarrow (not B, m) \)

Relations

- Let \( U, V \) be arithmetic values
  \[ (E, m) \rightarrow (E', m) \]
  \[ (E \sim E', m) \rightarrow (E' \sim E', m) \]
  \[ (E, m) \rightarrow (E', m) \]
  \[ (V \sim E, m) \rightarrow (V \sim E', m) \]
  \[ (U \sim V, m) \rightarrow b \]
  where \( U \sim V = b \)

Arithmetic Expressions

- \( (E, m) \rightarrow (E', m) \)
- \( (E \oplus E', m) \rightarrow (E' \oplus E', m) \)
- \( (E, m) \rightarrow (E', m) \)
- \( (V \oplus E, m) \rightarrow (V \oplus E', m) \)
- \( (U \oplus V, m) \rightarrow N \)
  where \( N \) is the specified value for \( U \oplus V \)

Commands

- **skip**: \( (skip, m) \rightarrow m \)
- Assignment: \( (I ::= E, m) \rightarrow (I ::= E', m) \)
  \( (I ::= E, m) \rightarrow m[I \leftarrow V] \)
- Sequencing: \( (C, m) \rightarrow (C', m') \)
  \( (C, C', m) \rightarrow (C', m') \)
  \( (C, C', m') \rightarrow (C', m') \)
  \( (C, C', m) \rightarrow (C', m') \)
Block Command

- Choice of level of granularity:
  - Choice 1: Open a block is a unit of work
    \[ (\{C\}, m) \rightarrow (C, m) \]
  - Choice 2: Blocks are syntactic sugar
    \[ (C, m) \rightarrow (C', m') \]

If Then Else Command - in English

- If the boolean guard in an if.then.else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

- (if true then C else C0 fi, m) \rightarrow (C, m)
- (if false then C else C0 fi, m) \rightarrow (C0, m)

While Command

- (while B do C, m)
- (if B then C; while B do C else skip fi, m)
  - In English: Expand a while into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example

- (y := i; while i > 0 do {i := i - 1; y := y * i}; (i := 3))
  \[ \rightarrow ? \]

Alternate Semantics for SIMPL1

- Can mix Natural Semantics with Transition Semantics to get larger atomic computations
- Use \((E, m) \downarrow v\) and \((B, m) \downarrow b\) for arithmetics and boolean expressions
- Revise rules for commands
Revised Rules for SIMPL1

Skip: \((\text{skip}, m) \rightarrow m\)

Assignment: \[((E, m) \downarrow v) \quad (I ::= E, m) \rightarrow m[I \leftarrow V]\]

Sequencing: \(((C, m) \rightarrow (C', m')) \quad (C, m) \rightarrow m'\)

Blocks: \((\{C, m\} \rightarrow (C', m')) \quad ([C, m] \rightarrow m')\)

If Then Else Command

\[(B, m) \downarrow \text{true} \quad (\text{if } B \text{ then } C \text{ else } C_0 \text{ fi}, m) \rightarrow (C, m)\]

\[(B, m) \downarrow \text{false} \quad (\text{if } B \text{ then } C \text{ else } C_0 \text{ fi}, m) \rightarrow (C_0, m)\]

Transition Semantics for SIMPL2?

- What are the choices and consequences for giving a transition semantics for the Simple Concurrent Imperative Programming Language #2, SIMP2?

Simple Concurrent Imperative Programming Language

\[I \in \text{Identifiers}\]
\[N \in \text{Numerals}\]
\[E ::= N \mid I \mid E + E \mid E - E\]
\[B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B\]
\[C ::= \text{skip} \mid C; C \mid \{C\} \mid I ::= E \mid C || C'\]
\[\text{if } B \text{ then } C \text{ else } C \text{ fi}\]
\[\text{while } B \text{ do } C\]