Lecture 23
Verification by Solving Horn Clauses
Instructor: Hossein Hojjat

October 27, 2017
Recap: Automated Verification

Program

Verifier

Specification

Theorem Prover

✓

fail

✗
### Weakest Precondition Rules: Summary

<table>
<thead>
<tr>
<th>Statement</th>
<th>Weakest Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>( \text{wp}(c, Q) )</td>
</tr>
<tr>
<td>( x := e )</td>
<td>( Q[x \mapsto e] )</td>
</tr>
<tr>
<td>( \text{assume}(b) )</td>
<td>( b \rightarrow Q )</td>
</tr>
<tr>
<td>( \text{assert}(b) )</td>
<td>( \text{wp}(b \land Q) )</td>
</tr>
<tr>
<td>( \text{havoc}(x) )</td>
<td>( \forall y. Q[x \mapsto y] )</td>
</tr>
<tr>
<td>( c_1; c_2 )</td>
<td>( \text{wp}(c_1, \text{wp}(c_2, Q)) )</td>
</tr>
<tr>
<td>( \text{if } b \text{ then } c_1 \text{ else } c_2 )</td>
<td>( b \rightarrow \text{wp}(c_1, Q) \land \neg b \rightarrow \text{wp}(c_2, Q) )</td>
</tr>
<tr>
<td>( \text{while } b \text{ do } c )</td>
<td>( I \land \forall \vec{y}. \left( (I \land b \rightarrow \text{wp}(c, I)) \land (I \land \neg b \rightarrow Q) \right)[\vec{x} \mapsto \vec{y}] ) (( \vec{x} ) are variables modified in ( c ) and ( I ) is the loop invariant)</td>
</tr>
</tbody>
</table>
Loop Invariant

\[
\begin{align*}
\vdash \{A \land b\} \ c \ \{A\} \\
\vdash \{A\} \ while \ b \ do \ c \ \{A \land \neg b\}
\end{align*}
\]

\[
\begin{align*}
wp(\text{while } b \ do \ c, \ Q) &= I \land \forall \vec{y}. \left( (I \land b \rightarrow wp(c, I)) \land (I \land \neg b \rightarrow Q) \right)[\vec{x} \mapsto \vec{y}]
\end{align*}
\]

(\vec{x} \text{ are variables modified in } c \text{ and } I \text{ is the loop invariant})

- Unfortunate reality: Inferring invariants automatically is undecideable
- This puts significant limits on the degree to which we can automate verification
• Active research area: how to find loop invariants efficiently and automatically
• The simplest technique: guess-and-check!
• Given template of invariants (e.g., \(? = \?, \ ? \leq \?)\), instantiate the holes with program variables and constants
• Check if it’s an invariant; if not, try a different instantiation
• Abstract interpretation: popular approach to discover invariants
  • We will discuss it later in the course
• Today we discuss an alternative approach: reducing verification to checking the satisfiability of Horn clauses
• First, let’s discuss control-flow graphs
Control Flow Graphs

- **Control Flow Graph (CFG):** graph representation of computation and control flow in the program
- Highlights the possible flow of execution

```plaintext
x = 1
while (x < 50) {
    x = x + 2
}
```

\[
\begin{align*}
x := 1 \\
\neg (x < 50) \quad [x < 50]
\end{align*}
\]
Control-Flow Graph (CFG) of a program $P$ is a rooted directed graph $G = (V, E, \text{entry}, \text{exit})$ where

- $V \subseteq \text{Label}$ is a set of labels
- $E \subseteq \text{Label} \times \text{Action} \times \text{Label}$ is a set of arcs labeled by actions
- $\text{entry} \in V$ is the start state
- $\text{exit} \in V$ is the final state

Each action $f \in \text{Action}$ is a relation from program state to program state:

$\mathbb{G}(f) \subseteq \text{state} \times \text{state}$
Generating Control-Flow Graphs

- Start with a graph that has one **entry** and one **exit** node.
- Draw an edge from entry to exit and label it with the entire program.
- Recursively decompose the program to have more edges with simpler labels.
- When labels cannot be decomposed further, we are done.
Basic Operations

- Base cases

Assignment

\[ x := e \]

assume\((e)\)

\[ [e] \]

assert\((e)\)

\[ [e], [\neg e] \]

error

- Sequence of statements

\[ c_1 ; c_2 \]

\[ c_2 \]

\[ c_1 \]
Control Structures

- Conditional statement

```plaintext
if (e) c_1 else c_2 \implies [e] c_1 [\neg e] c_2 [e]
```

- While loop

```plaintext
while (e) \{c\} \implies c [e] [\neg e]
```
while (c2) {
    x = y - 1;
    y = z * 2;
    if (c1) x = y - z;
    z = 10;
}

z = x;
while (c2) {
    x = y - 1;
    y = z * 2;
    if (c1) x = y - z;
    z = 10;
}
z = x;
How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert(n >= -10);
    n = n - 1;
}
```
Example

• How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert (n \geq -10);
    n = n - 1;
}
```

Control Flow Graph

- \( P_1 \): \( n := n + 1 \)  \[ n = 0 \]
- \( P_2 \): \( n := n - 1 \)  \[ n < -10 \]
- Error \( \times \)
• How to prove that the assertion does not fail in this program?

```java
int n = 0;
while (true) {
    n = n + 1;
    assert (n ≥ −10);
    n = n − 1;
}
```

• Let \( P_i(n) \) denotes the reachable values of \( n \) in state \( P_i \)

• Initially we do not know the set of reachable values of \( n \) in each state

• \( P_i \)'s can be any Boolean formula on \( n \), for example:
  • \( P_1(n) = (n ≥ 0) \) and \( P_2(n) = (n = −5) \lor (n > 0) \)
  • \( P_1(n) = (n = 0) \) and \( P_2(n) = \text{true} \) (any value can reach it)
  • ...

• We can write constraints between \( P_i \)'s according to CFG
Example

- How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert (n >= -10);
    n = n - 1;
}
```

Control Flow Graph

```
P1
n := n + 1

P2
n := n - 1
[n < -10]
```

\[(n = 0) \rightarrow P_1(n)\]
Example

- How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert(n ≥ −10);
    n = n − 1;
}
```

![Control Flow Graph]

\[\begin{align*}
(n = 0) \quad &\rightarrow \quad P_1(n) \\
P_1(n) \land (n' = n + 1) \quad &\rightarrow \quad P_2(n')
\end{align*}\]
• How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert (n ≥ -10);
    n = n - 1;
}
```

Control Flow Graph:

- $P_1$: $n := n + 1$
- $P_2$: $n := n - 1$

Initial state: $[n = 0]$

Transitions:
- $(n = 0) \rightarrow P_1(n)$
- $P_1(n) \land (n' = n + 1) \rightarrow P_2(n')$
- $P_2(n) \land (n' = n - 1) \rightarrow P_1(n')$
- Error: $[n < -10]$


• How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert (n \geq -10);
    n = n - 1;
}
```

```plaintext

\[
\begin{align*}
(n = 0) & \rightarrow P_1(n) \\
P_1(n) \land (n' = n + 1) & \rightarrow P_2(n') \\
P_2(n) \land (n' = n - 1) & \rightarrow P_1(n') \\
P_2(n) \land (n < -10) & \rightarrow false
\end{align*}
\]

```
• How to prove that the assertion does not fail in this program?

```java
int n = 0;
while (true) {
    n = n + 1;
    assert (n \geq -10);
    n = n - 1;
}
```

\[
\forall n. \ (n = 0) \quad \rightarrow \quad P_1(n)
\]
\[
\forall n, n'. \ P_1(n) \land (n' = n + 1) \quad \rightarrow \quad P_2(n')
\]
\[
\forall n, n'. \ P_2(n) \land (n' = n - 1) \quad \rightarrow \quad P_1(n')
\]
\[
\forall n. \ P_2(n) \land (n < -10) \quad \rightarrow \quad false
\]
How to prove that the assertion does not fail in this program?

```c
int n = 0;
while (true) {
    n = n + 1;
    assert (n ≥ -10);
    n = n - 1;
}
```

**Control Flow Graph**

$$
\forall n. \quad (n = 0) \quad \rightarrow \quad P_1(n)
$$

$$
\forall n, n'. \quad P_1(n) \wedge (n' = n + 1) \quad \rightarrow \quad P_2(n')
$$

$$
\forall n, n'. \quad P_2(n) \wedge (n' = n - 1) \quad \rightarrow \quad P_1(n')
$$

$$
\forall n. \quad P_2(n) \wedge (n < -10) \quad \rightarrow \quad false
$$

Solvable: $P_1(n) \equiv (n \geq 0)$ and $P_2(n) \equiv (n \geq 1)$
Demo

- Try https://rise4fun.com/

\[
\forall n. \quad (n = 0) \quad \rightarrow \quad P_1(n)
\]
\[
\forall n, n'. \quad P_1(n) \land (n' = n + 1) \quad \rightarrow \quad P_2(n')
\]
\[
\forall n, n'. \quad P_2(n) \land (n' = n - 1) \quad \rightarrow \quad P_1(n')
\]
\[
\forall n. \quad P_2(n) \land (n < -10) \quad \rightarrow \quad false
\]

(set-logic HORN)

(declare-fun P1 (Int) Bool)
(declare-fun P2 (Int) Bool)
(assert (forall ((n Int)) (=> (= n 0) (P1 n))))
(assert (forall ((n Int)(np Int)) (=> (and (P1 n) (= np (+ n 1)))
                                  (P2 np))))
(assert (forall ((n Int)(np Int)) (=> (and (P2 n) (= np (- n 1)))
                                  (P1 np))))
(assert (forall ((n Int)) (=> (and (P2 n) (< n (- 10))) false)))
(check-sat)
(get-model)
• Horn clause is an implication of the form:

\[
\forall \vec{v}. \quad \Phi(\vec{v}) \land R_1(\vec{v}) \land \cdots \land R_n(\vec{v}) \rightarrow R_0(\vec{v})
\]

- \(\Phi(\vec{v})\) is an arithmetic formula (e.g. \(x + 2y \leq z\))
- \(R_i(\vec{v})\) is a relation symbol
- Head of the clause is either a relation symbol or \textit{false}
- A solution for the Horn clause is an assignment of formulae to relation symbols for which the implication is valid
Verification by Solving Horn Clauses

\[ \forall \bar{v}. \Phi^0(\bar{v}) \land R^0_1(\bar{v}) \land \cdots \land R^0_n(\bar{v}) \rightarrow R^0_0(\bar{v}) \]

\[ \forall \bar{v}. \Phi^1(\bar{v}) \land R^1_1(\bar{v}) \land \cdots \land R^1_n(\bar{v}) \rightarrow R^1_0(\bar{v}) \]

\[ \vdots \]

\[ \forall \bar{v}. \Phi^m(\bar{v}) \land R^m_1(\bar{v}) \land \cdots \land R^m_n(\bar{v}) \rightarrow R^m_0(\bar{v}) \]

\[ \forall \bar{v}. \Phi^i(\bar{v}) \land R^i_1(\bar{v}) \land \cdots \land R^i_n(\bar{v}) \rightarrow false \]
Exercise

- Convert to Horn clauses

```c
int x, y;
assume(x ≥ 0 ∧ y ≥ 0);
while(x ≠ y) {
    if (x > y) then x := x − y;
    else y := y − x;
}
assert(x ≠ −1);
```
• Convert to Horn clauses

```
int x,y;
assume(x ≥ 0 ∧ y ≥ 0);
while(x ≠ y) {
    if (x > y) then x := x − y;
    else y := y − x;
}
assert(x ≠ −1);
```
Exercise

- Convert to Horn clauses

\[
\begin{align*}
1) & \quad P_0(x, y) \land (x \geq 0) \land (y \geq 0) \quad \to \quad P_0(x, y) \\
2) & \quad P_1(x, y) \land (x \neq y) \quad \to \quad P_1(x, y) \\
3) & \quad P_2(x, y) \land (x > y) \quad \to \quad P_2(x, y) \\
4) & \quad P_2(x, y) \land (x \leq y) \quad \to \quad P_1(x', y) \\
5) & \quad P_3(x, y) \land (x' = x - y) \quad \to \quad P_1(x', y) \\
6) & \quad P_4(x, y) \land (y' = y - x) \quad \to \quad P_1(x, y') \\
7) & \quad P_1(x, y) \land (x = y) \quad \to \quad P_5(x, y) \\
8) & \quad P_5(x, y) \land (x = -1) \quad \to \quad false \\
\end{align*}
\]