Program Verification

Akash Lal

Microsoft Research, India

Review

Annotated Program

```
method Eg2 (x, y : int)
  returns (z : int)
{
   assume x < y;
   z := (x+y)/2;
   assert x < z;
}
```

Verification Condition Generation

Verification Condition

Valid iff for all x, y, z:
$((x<y) \wedge (z=x+y/2) \Rightarrow (x<z))$

Foundation: Mathematical Logic

Theorem Prover/ SAT/SMT Solver

Rough Outline

- Programming Language syntax and semantics
  - Operational semantics
- Weakest Preconditions
  - Efficient computation
- (Un)Bounded Verification

Is this program correct?

$x := y + 1;$
$\textbf{assert } x > y$

$$a = b + 1 \wedge \neg(a > b)$$

$x := 2y;$
$\textbf{assert } x > y$

$$a = 2b \wedge \neg(a > b)$$

Syntax

- Program is a “Stmt”

$$
\begin{array}{llcl}
Expr & e & ::= & n \in Int \\
 & & | & x \in Var \\
 & & | & e_1 + e_2 \\
 & & | & e_1 - e_2 \\
 & & & \\
BoolC & b & ::= & true \quad | \quad false \\
 & & | & e_1 = e_2 \\
 & & | & e_1 < e_2 \\
 & & | & \neg b \\
 & & | & b_1 \wedge b_2 \\
 & & | & b_1 \vee b_2
\end{array}
$$

$$
\begin{array}{lcl}
Stmt & ::= & x := e \quad \text{(assignment)} \\
 & | & \textbf{assume } b \\
 & | & \textbf{assert } b \\
 & | & S_1; S_2 \quad \text{(sequence)} \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else} S_2 \\
 & | & \textbf{while } b \textbf{ do } S
\end{array}
$$

$$x := n; y := 1; \textbf{while } x > 1 \textbf{ do } (y := y * x; x := x - 1;)$$

Operational Semantics

- How a program executes

$State \equiv Var \rightarrow Int \cup \{\perp\}$

The $State$ $\perp$ represents *failure*

If σ is a $State$ and $x \in Var$ then $\sigma(x)$ is the *value* of x in σ

Let $f[x \mapsto v]$ be the same as f but with the value of x changed to v

Execution of S changes state from σ to σ': $\langle S, \sigma \rangle \longrightarrow \sigma'$

$x := 5; y := 1; \textbf{while } x > 1 \textbf{ do } (y := y * x; x := x - 1;)$

Operational Semantics

$$
\begin{array}{lcl}
Expr & ::= & n \in Int \\
 & | & x \in Var \\
 & | & e_1 + e_2 \\
 & | & e_1 - e_2
\end{array}
$$

$$
\begin{array}{lcl}
BoolC & ::= & true \quad | \quad false \\
 & | & e_1 = e_2 \\
 & | & e_1 < e_2 \\
 & | & \neg b \\
 & | & b_1 \wedge b_2 \\
 & | & b_1 \vee b_2
\end{array}
$$

$$eval : Expr \times State \rightarrow Int$$

$$
\begin{array}{lcl}
eval(n, \sigma) & = & n \\
eval(x, \sigma) & = & \sigma(x) \\
eval(e_1 + e_2, \sigma) & = & eval(e_1, \sigma) + eval(e_2, \sigma) \\
eval(e_1 - e_2, \sigma) & = & eval(e_1, \sigma) - eval(e_2, \sigma)
\end{array}
$$

$$beval : BoolC \times State \rightarrow Bool$$

$$
\begin{array}{lcl}
beval(e_1 = e_2, \sigma) & = & eval(e_1, \sigma) = eval(e_2, \sigma) \ ? \ true \ : \ false \\
beval(e_1 < e_2, \sigma) & = & eval(e_1, \sigma) < eval(e_2, \sigma) \ ? \ true \ : \ false \\
beval(\neg b, \sigma) & = & \neg beval(b, \sigma) \\
beval(b_1 \wedge b_2, \sigma) & = & beval(b_1, \sigma) \wedge beval(b_2, \sigma) \\
beval(b_1 \vee b_2, \sigma) & = & beval(b_1, \sigma) \vee beval(b_2, \sigma)
\end{array}
$$

Operational Semantics

$$
\begin{array}{lll}
Stmt & ::= & x := e \quad \text{(assignment)} \\
& | & \textbf{assume } b \\
& | & \textbf{assert } b \\
& | & S_1; S_2 \quad \text{(sequence)} \\
& | & \textbf{if } b \textbf{ then } S_1 \textbf{ else} S_2 \\
& | & \textbf{while } b \textbf{ do } S
\end{array}
$$

$$\frac{}{\langle x := e, \sigma \rangle \longrightarrow \sigma[x \mapsto \mathit{eval}(e)]}\text{(Assign)}$$

$$\frac{\langle S_1, \sigma \rangle \longrightarrow \sigma' \qquad \langle S_2, \sigma' \rangle \longrightarrow \sigma''}{\langle S_1; S_2, \sigma \rangle \longrightarrow \sigma''}\text{(Sequence)}$$

$$\frac{}{\langle x := y; z := x + 4, [x \mapsto 1, y \mapsto 2, z \mapsto 3] \rangle \longrightarrow ???}$$

Operational Semantics

$$
\begin{array}{lll}
Stmt & ::= & x := e \quad \text{(assignment)} \\
 & | & \textbf{assume } b \\
 & | & \textbf{assert } b \\
 & | & S_1; S_2 \quad \text{(sequence)} \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else} S_2 \\
 & | & \textbf{while } b \textbf{ do } S
\end{array}
$$

$$\frac{}{\langle S, \bot \rangle \longrightarrow \bot}(\textsc{Failed})$$

$$\frac{beval(b, \sigma) = true}{\langle \textbf{assert } b, \sigma \rangle \longrightarrow \sigma}(\textsc{Assert})$$

$$\frac{beval(b, \sigma) = false}{\langle \textbf{assert } b, \sigma \rangle \longrightarrow \bot}(\textsc{Assert})$$

$$\frac{beval(b, \sigma) = true}{\langle \textbf{assume } b, \sigma \rangle \longrightarrow \sigma}(\textsc{Assume})$$

If there is no σ' such that $\langle S, \sigma \rangle \longrightarrow \sigma'$ then the execution of S on σ is said to *block*

Operational Semantics

$$\frac{beval(b) = true \qquad \langle S_1, \sigma\rangle \longrightarrow \sigma'}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, \sigma\rangle \longrightarrow \sigma'}(\textsc{Ite})$$

$$\frac{beval(b) = false \qquad \langle S_2, \sigma\rangle \longrightarrow \sigma'}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, \sigma\rangle \longrightarrow \sigma'}(\textsc{Ite})$$

$$\frac{beval(b) = false}{\langle \textbf{while } b \textbf{ do } S, \sigma\rangle \longrightarrow \sigma}(\textsc{While})$$

$$\frac{beval(b) = true \qquad \langle S, \sigma\rangle \longrightarrow \sigma' \qquad \langle \textbf{while } b \textbf{ do } S, \sigma'\rangle \longrightarrow \sigma''}{\langle \textbf{while } b \textbf{ do } S, \sigma\rangle \longrightarrow \sigma''}(\textsc{While})$$

Example

$$\frac{}{\langle z := 0; \mathbf{while}\ y < x\ \mathbf{do}\ z := z + 1; x := x - y, [x \mapsto 17, y \mapsto 5, z \mapsto 2]\rangle \longrightarrow ???}\,(\textsc{While})$$

Example

$$\frac{}{\langle \textbf{while } true \textbf{ do } x := x, \sigma\rangle \longrightarrow ???}\ (\textsc{While})$$

Semantic Equivalence

Two programs S_1 and S_2 are **semantically equivalent** if for all states σ, $\langle S_1, \sigma \rangle \longrightarrow \sigma'$ *if and only if* $\langle S_2, \sigma \rangle \longrightarrow \sigma'$.

The following programs are equivalent:
(1) $(S_1; S_2); S_3$
(2) $S_1; (S_2; S_3)$

Semantic Equivalence

Two programs S_1 and S_2 are **semantically equivalent** if for all states σ, $\langle S_1, \sigma \rangle \longrightarrow \sigma'$ *if and only if* $\langle S_2, \sigma \rangle \longrightarrow \sigma'$.

The following programs are equivalent:
(1) $\textbf{while}\ b\ \textbf{do}\ S$
(2) $\textbf{if}\ b\ \textbf{then}\ (S; \textbf{while}\ b\ \textbf{do}\ S)\ \textbf{else}\ skip$

Semantic Equivalence

Two programs S_1 and S_2 are **semantically equivalent** if for all states σ, $\langle S_1, \sigma\rangle \longrightarrow \sigma'$ *if and only if* $\langle S_2, \sigma\rangle \longrightarrow \sigma'$.

The following programs are equivalent:
(1) **while** $true$ **do** $skip$
(2) **assume** $false$

More Constructs

$\mathbf{repeat}\ S\ \mathbf{until}\ b$

More Constructs

$\textbf{for } x := a_1 \textbf{ to } a_2 \textbf{ do } S$

More Constructs

$S_1 \parallel S_2$

Structural Operational Semantics

- Focus on execution steps

$$\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle$$
$$\langle S, \sigma \rangle \Rightarrow \sigma'$$

$$\langle x := e, \sigma \rangle \Rightarrow \sigma[x \mapsto \mathit{eval}(e, \sigma)]$$

$$\langle \mathbf{assume}\ b, \sigma \rangle \Rightarrow \sigma \text{ when } \mathit{beval}(b, \sigma) = \mathit{true}$$

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \langle S_1', \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S_1'; S_2, \sigma' \rangle}(\textsc{Sequence})$$

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma' \rangle}(\textsc{Sequence})$$

$$\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, \sigma \rangle \Rightarrow \langle S_1, \sigma \rangle \text{ when } \mathit{beval}(b, \sigma) = \mathit{true}$$

$$\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma \rangle \text{ when } \mathit{beval}(b, \sigma) = \mathit{false}$$

$$\langle \mathbf{while}\ b\ \mathbf{do}\ S, \sigma \rangle \Rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ (S; \mathbf{while}\ b\ \mathbf{do}\ S)\ \mathbf{else}\ \mathit{skip}, \sigma \rangle$$

Structural Operational Semantics

Let us use γ to denote either $\langle S, \sigma \rangle$ or σ

Define $\Rightarrow^*$ as the smallest relations such that $\gamma_0 \Rightarrow^* \gamma_n$ only if there exists $\gamma_1, \gamma_2, \cdots \gamma_{n-1}$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for all $i = 0$ to $n - 1$.

$$\langle z := x; x := y; y := z, [x \mapsto 5, y \mapsto 7] \rangle \Rightarrow^* [x \mapsto 7, y \mapsto 5, z \mapsto 5]$$

Equivalence

Two programs S_1 and S_2 are **semantically equivalent** if for all states σ,
(1) $\langle S_1, \sigma\rangle \Rightarrow^* \gamma$ *if and only if* $\langle S_1, \sigma\rangle \Rightarrow^* \gamma$.
(2) there is an infinite derivation sequence from $\langle S_1, \sigma\rangle$ if and only if there is one from $\langle S_2, \sigma\rangle$

The following programs are equivalent:
(1) **while** b **do** S
(2) **if** b **then** $(S;$ **while** b **do** $S)$ **else** $skip$

The following programs are not equivalent:
(1) **while** $true$ **do** $skip$
(2) **assume** $false$

The following programs are not equivalent:
(1) $S \parallel$ **while** $true$ **do** $skip$
(2) S

Exercises

- Introduce construct random(x) in the language and give its natural semantics
  - Is it possible to compile away “random” if we have the choose construct?

- Introduce parallelism and give its structural operational semantics
  - Natural semantics?

$$
\begin{array}{lcll}
Stmt & ::= & x := e & \text{(assignment)} \\
 & | & \textbf{assume } b & \\
 & | & \textbf{assert } b & \\
 & | & S_1; S_2 & \text{(sequence)} \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else} S_2 & \\
 & | & \textbf{while } b \textbf{ do } S & \\
 & | & S_1 \textbf{ par } S_2 & \text{(parallelism)}
\end{array}
$$

$$x := 1 \textbf{ par } x := 2; x := x + 2;$$

Procedures

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & x := e \quad \text{(assignment)} \\
 & | & \textbf{assume } b \\
 & | & \textbf{assert } b \\
 & | & S_1; S_2 \quad \text{(sequence)} \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else} S_2 \\
 & | & \textbf{while } b \textbf{ do } S \\
 & | & \textbf{call } p \\
 & & \\
\mathit{ProcDecl} & ::= & \textbf{procedure } p \ \{ \ \textbf{let } V \textbf{ in } S \ \} \\
 & & \\
\mathit{Program} & ::= & \mathit{ProcDecl}^*; S
\end{array}
$$