CS 477: Operational Program Semantics

Sasa Misailovic

Based on previous slides by Gul Agha, Elsa Gunter, Madhusudan Parthasarathy, Mahesh Viswanathan, and Vikram Adve

University of Illinois at Urbana-Champaign
Transition Semantics Evaluation

• **A sequence of transitions**: trees of justification for each step

\[(C_1, m_1) \rightarrow (C_2, m_2) \rightarrow (C_3, m_3) \rightarrow \ldots \rightarrow (\text{skip, } m) \rightarrow m\]

• **Definition**: let \(\rightarrow^*\) be the transitive closure of \(\rightarrow\) i.e., the smallest transitive relation containing \(\rightarrow\)

• We can define it for final states \((C_1, m_1) \rightarrow^* m\) or intermediate states \((C_1, m_1) \rightarrow^* (C_2, m_2)\).
Small-step vs Big-step

• We can express big-step in terms of small step:
  \[(C, m) \rightarrow^* m' \text{ implies } (C, m) \Downarrow m'.\]

• Can be proved by simple rule induction.

• We can’t go from big-step to express small-step: some information about the execution is lost.
Reasoning: First Intuition

• All end-states reachable from a start state \( m \):
  
  \[
  S(P, \ m) = \{m' \mid (P, m) \rightarrow^* m' \}
  \]

• What if we have a set of start states \( M \)?
  
  \[
  S(P, \ M) = \{m' \mid \exists m_0 \in M. (P, m) \rightarrow^* m' \}
  \]
Reasoning: First Intuition

• How do we give meaning to predicates, e.g.,
  
x = input();
y = x*x + 1;
assert y > 0;

• Let us collect state(s) at the location of the assertion:
  \[ S_{\text{assert}}(m_0) = \{ m' \mid (P, m_0) \rightarrow^* (\text{assert } y > 0, m') \} \]
Reasoning: First Intuition

• Executions that reach the assertion: $S_{\text{assert}}(m_0)$ and those that satisfy the predicate in the assertion:

$$S_{\text{assert, sat}}(m_0) = \{ m' \mid m' \in S_a(m_0) \land m'(y) > 0 \}$$

• If the program is satisfying the assertion, how should the two sets relate?

• If there are violations of the assertion, what is the set we report back to the user?
Reasoning: First Intuition

• How do we claim validity of the program (i.e. it satisfies the assertion for all inputs – e.g. belonging to the set M)?

Extend the definition: \( S_{\text{assert}} = \bigcup_{m_0 \in M} S_{\text{assert}}(m_0) \)

• How do we support other predicates?
Give meaning to predicates in terms of program state (e.g., state \( m \) becomes the valuation)
  • We wander into the First-order theory land (we will discuss Presburger arithmetic later)
Extension: Abort

• Regular execution terminates when program in configuration (skip, m)

• Add another command “abort”.

• If the computation ends in (abort, m), then there is no transition from it => we reached the error state
Extensions: Parallel

- Statement C1 par C2: execute C1 and C2 in parallel
- We can apply multiple rules at the same time!
- (reflects nondeterminism; also hard to express using ↓)

\[
\begin{align*}
(C, m) &\rightarrow (C'', m') \\
(C \text{ par } C', m) &\rightarrow (C'' \text{ par } C', m') \\
(C', m) &\rightarrow (C'', m') \\
(C \text{ par } C', m) &\rightarrow (C \text{ par } C'', m') \\
(C, m) &\rightarrow (C'', m') \\
(C \text{ par skip, } m) &\rightarrow (C'', m') \\
(C', m) &\rightarrow (C'', m') \\
(\text{skip par } C', m) &\rightarrow (C'', m')
\end{align*}
\]
Fun Example

• In what states can this program be after the parallel section?

\[
( Y := 1 ) \text{ par } ( \text{ while } (Y = 0) \text{ do } X := X + 1 )
\]
Extension: Parallel

• Add synchronization: `await B protect C end`
• Command C can only execute if the condition B is true, but it executes as a full block (no interleavings).

\[
\begin{align*}
(B, s) \downarrow (\text{true, m1}) & \quad (C, m1) \rightarrow^* m' \\
(\text{await B protect C end, m}) & \rightarrow^* m'
\end{align*}
\]

• Examples:
  • `x = 1; ((x = 0) par (await x = 0 protect x := 1 ; x := x + 1 end)`
  • `(await true protect l := 1 ; l := k + 1 end) par
    (await true protect k := 2 ; k := l + 1 end)`

From Hillary, 2014
Extension: Nondeterministic

• E.g., nondeterministic assignment $x = E_1 \nondet E_2$
  • Nondeterministically assigns one of the two evaluated values to $x$

• How do we extend the semantics? (e.g., small step)
Symbolic Execution

• So far: we defined the execution of programs for concrete numerical values
• There are many executions so the enumeration is often not tractable

• We can abstract the concrete values of the variables and use symbolic evaluation to execute for a group of states at the same time
Symbolic Execution

Symbolic formulas syntax (with symbolic variables $\alpha$):

$$P ::= \text{true} \mid \text{false} \mid \text{not } P \mid P_1 \text{ bop } P_2 \mid Aexp_1 \text{ rop } Aexpr_2$$

$$Aexp ::= \alpha \mid n \mid Aexp_1 + Aexp_2 \mid Aexp_1 \times Aexp_2 \mid Aexp_1 - Aexp_2 \mid Aexp_1 / Aexp_2$$

Memory store: $\Sigma: Var \rightarrow Aexp$

Analysis state $(P, \Sigma)$:

• $P$ is called *path condition*, and $\Sigma$ a *symbolic state*. 
Arithmetic And Relational Expressions

\[(E_1, \Sigma) \Downarrow A_{exp1}' \quad (E_2, \Sigma) \Downarrow A_{exp2}'\]

\[\hline\]

\[(E_1 \text{ op } E_2, \Sigma) \Downarrow A_{exp1}' \text{ op } A_{exp2}'\]

\[(E_1, \Sigma) \Downarrow A_{exp1}' \quad (E_2, \Sigma) \Downarrow A_{exp2}' \quad P = A_{exp1}' \text{ rop } A_{exp2}'\]

\[\hline\]

\[(E \text{ rop } E', \Sigma) \Downarrow P\]
Statements

Skip: \[(P, \text{skip}, \Sigma) \downarrow (P, \Sigma)\]

Assignment: \[
\begin{align*}
\text{(E, } \Sigma\text{) } & \downarrow \text{Aexp} \\
\text{(P, } k := E, \Sigma\text{) } & \downarrow (P, \Sigma [k \leftarrow\text{ Aexp }])
\end{align*}
\]

Sequencing: \[
\begin{align*}
\text{(P, C, } \Sigma\text{) } & \downarrow (P', \Sigma') \\
\text{(P', C', } \Sigma'\text{) } & \downarrow \Sigma'' \\
\text{(P, C; C', } \Sigma\text{) } & \downarrow \Sigma''
\end{align*}
\]
If Then Else Statement

\[(B, \Sigma) \downarrow Pb \quad \text{SAT}(P \land Pb) \quad (P \land Pb, C, \Sigma) \downarrow (P’, \Sigma’)\]

\[\text{(if B then C else C' fi, } \Sigma) \downarrow (P’, \Sigma’)\]

\[(B, \Sigma) \downarrow Pb \quad \text{SAT}(P \land \neg Pb) \quad (P \land \neg Pb, C’, \Sigma) \downarrow (P’, \Sigma’)\]

\[\text{(if B then C else C' fi, } \Sigma) \downarrow (P’, \Sigma’)\]

Both are possibly satisfiable (due to symbolic abstraction)!

Execution is then not a sequence but a tree of instructions!

**Static Symbolic execution**: We “merge” the formulas of both branches and simplify them. This will be clearer after we cover abstract interpretation next!
Example

```python
int x = input()
int y = 0

if x > 0
    y = x + 1
else
    y = -x

// Question: Is y ≥ 0
// after the execution?
```

(See the lecture video)
Another Example

```c
int x = input();
int y = 1/x
```

// Question: can the code experience an error?

```c
int x = input();
if x != 0
    y = 1 / x
else
    abort
```
Symbolic Execution of Loops?

• Most practical tools just “unroll” the loop k times

• Enough for finding various bugs:
  search under “Small scope hypothesis”

• A more general approach will require loop invariants
  (predicates that hold at any point of loop execution)

• Often requires manual intervention by developer!

• We will discuss invariants later when we cover deductive methods for reasoning about programs.
Symbolic Evaluation for Loops: Rule

Together: Let us derive the rule for the finite loop
\( \text{while}_k \text{ (condition)} \) -- for a constant \( k > 0 \)

\[
\begin{align*}
\text{k} > 0 & \quad (\Sigma, B) \Downarrow P' \quad \text{SAT}(P \land P') \quad (P \land P', \Sigma, C; \text{while}_{k-1} B \text{ do C}) \Downarrow (P'', \Sigma'') \\
\hline
(P, \Sigma, C; \text{while}_k B \text{ do C}) \Downarrow (P'', \Sigma'')
\end{align*}
\]

\[
\begin{align*}
\text{k} = 0 & \quad (\Sigma, B) \Downarrow P' \quad \text{SAT}(P \land P') \\
\hline
(P, \Sigma, C; \text{while}_k B \text{ do C}) \Downarrow (P \land \neg P', \Sigma'')
\end{align*}
\]
Symbolic Execution and Testing

• Generalizes testing by using symbolic values and having means to explore all paths: exhaustive exploration

• Scalability is an issue (although the modern tools have made it more practical)

• Concolic execution: combines testing (concrete execution) with symbolic execution
  • Use concrete execution to reach a certain point in the execution (e.g., an important subcomputation)
  • Use then symbolic execution to exhaustively explore the executions within that smaller scope