CS 477: Operational Program Semantics

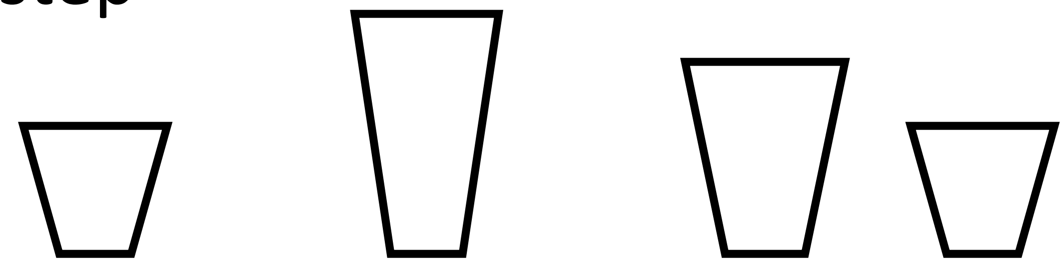
Sasa Misailovic

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) --> (C_2,m_2) --> (C_3,m_3) --> … --> (skip, m) --> m

- **Definition:** let **-->*** be the transitive closure of **-->** i.e., the smallest transitive relation containing **-->**

- We can define it for final states (C_1,m_1) -->* m or intermediate states (C_1,m_1) -->* (C_2,m_2) .

Small-step vs Big-step

- We can express big-step in terms of small step:

  (C, m) -->* m' implies (C, m) ⇓ 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:

$$S(P,\ m) = \{m' \mid (P, m) \to^* m' \}$$

- What if we have a set of start states M?

$$S(P,\ M) = \{m' \mid \ \exists m_0 \in M\ .\ (P, m) \to^* m' \}$$

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:

$$S_{\text{assert}}(\mathrm{m}_0) = \{m' \mid (P, m_0) \rightarrow^* (\texttt{assert y > 0}, m')\}$$

Reasoning: First Intuition

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

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

- 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: $S_{\text{assert}} = \cup_{m_0 \in M} S_{\text{assert}}(m_0)$

- 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 $\Downarrow$)

$$\frac{(C,m) \dashrightarrow (C'', m')}{(C \text{ par } C', m) \dashrightarrow (C'' \text{ par } C', m')} \qquad \frac{(C,m) \dashrightarrow (C'', m')}{(C \text{ par skip}, m) \dashrightarrow (C'', m')}$$

$$\frac{(C',m) \dashrightarrow (C'', m')}{(C \text{ par } C', m) \dashrightarrow (C \text{ par } C'', m')} \qquad \frac{(C',m) \dashrightarrow (C'', m')}{(\text{skip par } C', m) \dashrightarrow (C'', m')}$$

Fun Example

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

```
( Y := 1 )  par  ( while (Y = 0) 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).

$$\frac{(B, s) \Downarrow (\text{true}, m1) \qquad (C, m1) \dashrightarrow^{*} m'}{(\text{await B protect C end}, m) \dashrightarrow^{*} m'}$$

- 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 = E1 [] E2
  - 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 α):

P ::= true | false

| not P | P1 bop P2 | Aexp1 rop Aexpr2

Aexp ::= α | n | Aexp1 + Aexp2 | Aexp1 * Aexp2

| Aexp1 – Aexp2 | Aexp1 / Aexp2

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

Analysis state (P, Σ):

- P is called ***path condition***, and Σ a ***symbolic state***.

Arithmetic And Relational Expressions

$$\frac{(E1, \Sigma) \Downarrow Aexp1' \qquad (E2, \Sigma) \Downarrow Aexp2'}{(E1 \text{ op } E2, \Sigma) \Downarrow Aexp1' \text{ op } Aexp2'}$$

$$\frac{(E1, \Sigma) \Downarrow Aexp1' \qquad (E2, \Sigma) \Downarrow Aexp2' \qquad P = Aexp1' \text{ rop } Aexp2'}{(E \text{ rop } E', \Sigma) \Downarrow P}$$

Statements

Skip: $(P, \text{skip}, \Sigma) \Downarrow (P, \Sigma)$

Assignment:
$$\frac{(E, \Sigma) \Downarrow \text{Aexp}}{(P, k := E, \Sigma) \Downarrow (P, \Sigma\,[k \text{ <-- } \text{Aexp}\,])}$$

Sequencing:
$$\frac{(P, C, \Sigma) \Downarrow (P', \Sigma') \quad (P', C', \Sigma') \Downarrow \Sigma''}{(P, C; C', \Sigma) \Downarrow \Sigma''}$$

If Then Else Statement

$$\frac{(B, \Sigma) \Downarrow Pb \qquad SAT(P \wedge Pb) \quad (P \wedge Pb, C, \Sigma) \Downarrow (P', \Sigma')}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, \Sigma) \Downarrow (P', \Sigma')}$$

$$\frac{(B, \Sigma) \Downarrow Pb \qquad SAT(P \wedge \neg Pb) \qquad (P \wedge \neg Pb, C', \Sigma) \Downarrow (P', \Sigma')}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, \Sigma) \Downarrow (P', \Sigma')}$$

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

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

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

// Question: Is y $\geq$ 0
// after the execution?

(See the lecture video)

Another Example

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

// Question: can the code experience an error?

```
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 Evaluaton for Loops: Rule

Together: Let us derive the rule for the finite loop while_k (condition) -- for a constant k > 0

$$\frac{k > 0 \quad (\Sigma, B) \Downarrow P' \quad SAT(P \wedge P') \quad (P \wedge P', \Sigma, C; \text{while}_{k-1}\ B \text{ do } C) \Downarrow (P'', \Sigma'')}{(P, \Sigma, C; \text{while}_k\ B \text{ do } C) \Downarrow (P'', \Sigma'')}$$

$$\frac{k = 0 \quad (\Sigma, B) \Downarrow P' \quad SAT(P \wedge P')}{(P, \Sigma, C; \text{while}_k\ B \text{ do } C) \Downarrow (P \wedge \neg P', \Sigma'')}$$

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