Program Testing and Analysis: Operational Semantics

Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();
```

Options: 5, 7, or something else
Warm-up Quiz

```
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();

Correct answer: 5
```
Warm-up Quiz

```
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();
```

eval() evaluates JavaScript code given as a string

Correct answer: 5
Warm-up Quiz

Store function into variable (functions are first-class objects)

```javascript
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();
```

Correct answer: 5
Warm-up Quiz

```javascript
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();
```

Define a function and call it immediately

Correct answer: 5
Warm-up Quiz

```javascript
var e = eval;

(function f() {
    var x = 5;
    e("x=7")
    console.log(x);
})();
```

Indirect `eval()`:
Works at global scope rather than local scope

Correct answer: 5
Big Picture

Last lecture:

- Syntax of languages
- Representations of programs

This lecture:

- Assign meaning (= semantics) to programs
- Focus: Operational semantics of imperative languages
- Formal foundation for specifying languages and for describing dynamic analyses
Plan for Today

- Motivation & preliminaries
- Abstract syntax of SIMP
- An abstract machine for SIMP
- Structural operation semantics for SIMP
  - Small-step semantics
  - Big-step semantics
Why Do We Need Operational Semantics?

Example (C code):

```c
int i = 5;
f(i++, --i)  // What are the actual parameters?
```

Option 1: 5, 5  (left-to-right)
Option 2: 4, 4  (right-to-left)

Both options possible in C

→ unspecified semantics!
→ compiler decides
Specifying the Semantics of Programs

- Static semantics, e.g., types

- Dynamic semantics
  - Denotational
  - Axiomatic
  - Operational (our focus)

Useful for:
  - Language implementation
  - Programming
  - Language design
  - Program analysis
7) Preliminaries

a) Transition system

- set Config of configurations or states
- binary relation $\rightarrow \subseteq \text{Config} \times \text{Config}$ ("transition relation")

$L \rightarrow c \rightarrow c'$.. change of state

$\rightarrow^*$.. reflexive-transitive closure of $\rightarrow$

$\forall c. c \rightarrow^* c$  $\forall c, c', c''. c \rightarrow^* c' \land c' \rightarrow^* c''$  $\Rightarrow c \rightarrow^* c''$

deterministic:  $c \rightarrow c_1 \land c \rightarrow c_2 \Rightarrow c_1 = c_2$
5) **Rule induction**

To define a set with:

- a finite set of basic elements (axioms)

\[ \vdash \]

- a finite set of rules that specify how to generate elements

\[ \frac{h_n \vdash h_n}{\vdash h_{n+1}} \]

\(< \) hypotheses

\(< \) conclusion

**Example 1:** set of natural numbers

- axion: \[ 0 \]

- rule: \[ n \vdash n+1 \]
Example 2: Evaluation of expressions

\[ L \rightarrow \text{Set of pairs of AST \& value} \]

Notation: \( E \downarrow n \) \quad \text{expr. E evaluates to number n}

Axioms: \( 1 \downarrow 1, 2 \downarrow 2, \ldots \), etc.

axiom scheme \( n \downarrow n \)

Rule scheme:

\[
\begin{align*}
E_1 & \downarrow m \\
E_2 & \downarrow n
\end{align*}
\]

\[ + (E_1, E_2) \downarrow n \]

if \( n = (m + n_2) \), etc.

\[
\begin{align*}
E_1 & \downarrow m \\
E_2 & \downarrow n_2
\end{align*}
\]

\[ \text{Op} (E_1, E_2) \downarrow n \]

if \( n = (m \text{ Op } n_2) \)
c) Proof tree

Let show that element is in inductive set

Example 1

\[
\begin{align*}
0 & \\
1 & \\
2 & 
\end{align*}
\]

Example 2

Show that \(-(+(3, 4), 1) \downarrow 6\)

\# axioms, \# rules?

3 2

\[
\begin{align*}
3 \downarrow 3 & \\
4 \downarrow 4 & (plus) \\
+ (3, 4) \downarrow 7 & \\
1 \downarrow 1 & (minus) \\
-(+(3, 4), 1) \downarrow 6 &
\end{align*}
\]
Plan for Today

- Motivation & preliminaries
- Abstract syntax of SIMP
- An abstract machine for SIMP
- Structural operation semantics for SIMP
  - Small-step semantics
  - Big-step semantics
Abstract syntax of SIMP = simple imperative PL

- features: assignments, sequencing, conditionals, loops, integer variables

abstract syntax:

\[ P ::= C | E | B \]
a) Commands:

- `C₁ ; C₂` (sequence)
- `l := E` (assignment to label `l`)
- `if B ; C₁ ; C₂` (two-way selector)
- `while B ; C` (while loop)
- `ship` (do nothing)

Textual notation:

\[ C ::= l := E \mid C₁ ; C \mid \text{if } B \text{ then } C₁ \text{ else } C₂ \mid \text{while } B \text{ do } C \mid \text{ship} \]
b) Integer expressions:

\[ E ::= !l \mid n \mid E \, \text{op} \, E \]

\[ \text{op} ::= + \mid - \mid \ast \mid / \]

where \( n \) -- integers

\( l \in L = \{ l_0, l_2, \ldots \} \) -- memory locations

\( !l \) -- value stored at address \( l \)

c) Boolean expressions:

\[ B ::= \text{True} \mid \text{False} \mid E \, \text{bop} \, E \mid \neg B \mid B \land B \]

\[ \text{bop} ::= < \mid > \mid = \]
Example 1: \[ z := !x ; ( x := !y ; y := !z ) \]

-- swaps values of x and y

AST:
Example 2: while (!l > 0) do
    ( factorial := !factorial * !l ;
      l := !l - 1 )

AST:
Plan for Today

- Motivation & preliminaries
- Abstract syntax of SIMP
- An abstract machine for SIMP
- Structural operation semantics for SIMP
  - Small-step semantics
  - Big-step semantics
An Abstract Machine for SIMP

4 main elements:
- control stack s: stores instructions to execute
- auxiliary/result stack r: stores intermediate values
- processor: performs arithmetic operations, etc.
- memory/store m: partial function mapping locations to integers

L> Notation: \( m[e \mapsto n] \) - update \( m \) with the new mapping

L> i.e. \( m[e \mapsto n] (e) = n \)
\( m[e \mapsto n] (e') = m(e') \quad \forall e', e' \neq e \)
Abstract machine = transition system

Configuration: \( \langle c, i, r \rangle \)

- \( c ::= \text{nil} \mid i \oplus c \) (instruction \( i \) pushed on top of \( c \))

- \( \text{empty stack} \)

- \( i ::= P \mid \text{op} \mid - \mid \land \mid \text{bop} \)

- \( r ::= \text{nil} \mid P \lor r \mid \text{for} \lor r \)
Model execution of SIMP programs as sequences of transitions from initial to final configuration.

\[ \langle C, \text{nil}, \text{nil}, m \rangle \]
Execute C in given memory state m.

Transition rules:
\[ \langle c, r, m \rangle \rightarrow \langle c', r', m' \rangle \]

\[ \langle \text{nil}, \text{nil}, m \rangle \]
Stops when stacks are empty.
a) Evaluating expressions

\[
\langle \text{op} \circ c, r, m \rangle \rightarrow \langle c, \text{op} \circ r, m \rangle
\]

- pop \( \text{op} \) from control stack,
- push onto result stack

\[
\langle !l \circ c, r, m \rangle \rightarrow \langle c, \text{op} \circ r, m \rangle \quad \text{if } m(l) = n
\]

- read from memory & push onto \( r \)

\[
\langle (E_1 \circ E_2) \circ c, r, m \rangle \rightarrow \langle E_1 \circ E_2 \circ \text{op} \circ c, r, m \rangle
\]

\[
\langle \text{op} \circ c, \text{op} \circ r, m \rangle \rightarrow \langle c, \text{op} \circ r, m \rangle \quad \text{if } m_1 \circ \text{op} \circ m_2 = n
\]

( similar rules for Boolean expressions)
Semantics of SIMP expressions

Value of $E$ in state $m$ is $v$

if there is a sequence of transitions

$$ < E \circ c, r, m > \xrightarrow{*} < c, v \circ r, m' > $$

Example: What is the value of $C = a + b$

in state where $m = \{ a \rightarrow 3, b \rightarrow 1 \}$?

# transitions?