Operational Semantics

Slonneger and Kurtz Ch 8.4, 8.6 (only big-step semantics)

Nielsen and Nielson, Ch 2.1
Uses of Operational Semantics

**Correctness:** does this program have a run-time error?

**Equivalence:** given two programs, are they *always* semantically equivalent? Essential question for the correctness of compiler optimizations

**Conditions for equivalence:** given two programs, under what restrictions/conditions are they semantically equivalent? Needed to define compiler analyses that prove these conditions before optimizations can be applied

**Correctness of code generation:** given any program and a translation algorithm to create low-level code (e.g., assembly code or Java bytecode), is the low-level program semantically equivalent to the original program? That is, can we prove the correctness of the translation algorithm?
Inductive definition:

**Example:** a set $X$ defined as follows:

- $0 \in X$
- If $n \in X$, then $n+2 \in X$

$X$ is the smallest set with these properties

All **even natural numbers** $\{ 0, 2, 4, \ldots \}$. Note that $\{ 0, 1, 2, 3, \ldots \}$ also satisfies the first two rules, but is not the smallest such set

**Example:** a set $L$ defined as follows:

- $\texttt{intconst} \in L$ [for every $\texttt{intconst}$ token]
- $\texttt{ident} \in L$ [for every $\texttt{ident}$ token]
- If $e_1 \in L$ and $e_2 \in L$, then $e_1 + e_2 \in L$ [$e_1$, $e_2$ are token sequences]

$L$ is the smallest set with these properties

Language for $\langle \text{expr} \rangle ::= \texttt{intconst} \mid \texttt{ident} \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle$
Background: Inference Rules

The same thing, written as **inference rules** [from formal logic]

\[
\begin{align*}
0 \in X \\
n \in X \\
n+2 \in X
\end{align*}
\]

The boxes are for readability only; not part of the inference rule

Over the bar: zero or more **premises**
Below the bar: **conclusion**

If the premises are true, we can derive the conclusion
[For example: If we know that \( n \in X \), we can conclude that \( n+2 \in X \)]

If there are no premises: the rule is an **axiom**
[For example: we know that \( 0 \in X \) “by itself”]

The second example:

\[
\begin{align*}
\text{intconst} \in L \\
\text{ident} \in L \\
e_1 \in L \\
e_2 \in L
\end{align*}
\]

\[
e_1 + e_2 \in L
\]
Simple Language  (related to the programming projects)

<program> ::= <stmtList>

<stmtList> ::= <stmt> ; <stmtList> | <stmt>

<stmt> ::= int id = <expr>  [for brevity, only consider integer vars/consts]
            | id = <expr>
            | if ( <cond> ) <stmt>
            | if ( <cond> ) <stmt> else <stmt>
            | while ( <cond> ) <stmt>
            | { <stmtList> }
            | skip
Simple Language (from the programming projects)

<expr> ::= const | id  [for brevity, only consider integer vars/consts]
   |  <expr> + <expr>  |  <expr> - <expr>
   |  <expr> * <expr>  |  <expr> / <expr>
   |  (  <expr>  )

<cond> ::= true  |  false  |  <expr> < <expr>  [also <=, >, >=, ==, !=]
   |  <cond> && <cond>  |  <cond> || <cond>
   |  ! <cond>  |  (  <cond>  )
Memory State (we will just say “State”)

**State**: a map $\sigma$ from variable names to values

An abstraction of the contents of the physical memory

Example: program with two variables $x$ and $y$

$$\sigma(x) = 9 \text{ and } \sigma(y) = 5$$

Sometimes will denote with $[x \mapsto 9, \ y \mapsto 5]$  $\mapsto$ means “maps to”

$\sigma$: Vars $\rightarrow$ Z

- **Vars** is the set of all variable names in the program
- **Z** is the set of integers: $\{0, -1, 1, -2, 2, \ldots\}$

Note: we will ignore issues of **finite-precision arithmetic**. In all standard hardware and languages, the built-in types are limited: e.g. Java **int** is $-2,147,483,648 (-2^{31})$ to $2,147,483,647 (2^{31}-1)$

[Interesting paper on the web page under Resources: “Understanding Integer Overflow in C/C++”]
Evaluation for Arithmetic Expressions

Evaluation relation (3-way relation) for expressions:
set of triples \((ae, \sigma, v)\) but we will write \(\langle ae, \sigma \rangle \rightarrow v\)

- \(ae\) is a parse subtree derived from \(<\text{expr}>\)
- \(\sigma\) is a state
- \(v\) is a value from \(\mathbb{Z}\)

Meaning of \(\langle ae, \sigma \rangle \rightarrow v\): the evaluation of \(ae\) from state \(\sigma\)
completes successfully and produces the value \(v\)

Example: \(\langle x+y-1, [x\mapsto 5, y\mapsto 4] \rangle \rightarrow 8\)

Example: \(\langle x/(y-1), [x\mapsto 5, y\mapsto 1] \rangle \rightarrow ... \) No triple exists
Evaluation for Arithmetic Expressions

Syntax: $\text{id} | \text{const} | <\text{expr}> + <\text{expr}> | ...$

$<\text{const}, \sigma> \rightarrow \text{const}$  \hspace{1cm} \text{const} is a parse tree node; $\text{const} \in \mathbb{Z}$

$<\text{id}, \sigma> \rightarrow \sigma(\text{id})$  \hspace{1cm} \text{axiom, applicable only if the id has a value in}\ \sigma$

\[
\begin{align*}
<\text{ae}_1, \sigma> & \rightarrow v_1 \\
<\text{ae}_2, \sigma> & \rightarrow v_2 \\
\hline \\
<\text{ae}_1+\text{ae}_2, \sigma> & \rightarrow v
\end{align*}
\]

$v = v_1 + v_2$

Last one is an example of an inference rule with a condition ($v = v_1 + v_2$); the rule is applicable only when the condition is satisfied

Nothing in the rule for $\text{ae}_1+\text{ae}_2$ tells us in which order the operands of $+$ will be evaluated. In fact, their evaluation could be interleaved – do a bit of work for $\text{ae}_1$ then do a bit of work for $\text{ae}_2$ then go again to $\text{ae}_1$ etc. (or even evaluate them in parallel)
Example: Derivation Tree

\[ x + 2*y - z \text{ evaluated in state } \sigma = [x\mapsto9, y\mapsto5, z\mapsto1] \]

\[
\begin{align*}
\langle 2, \sigma \rangle &\rightarrow 2 & \langle y, \sigma \rangle &\rightarrow 5 \\
\hline
\langle x, \sigma \rangle &\rightarrow 9 & \langle 2*y, \sigma \rangle &\rightarrow 10 \\
\hline
\langle x+2*y, \sigma \rangle &\rightarrow 19 & \langle z, \sigma \rangle &\rightarrow 1 \\
\hline
\langle x+2*y-z, \sigma \rangle &\rightarrow 18
\end{align*}
\]
Evaluation for Arithmetic Expressions

Syntax: ... | <expr> / <expr> | ...

\[
\begin{array}{c}
\text{<ae}_1, \sigma \rightarrow v_1 & \text{<ae}_2, \sigma \rightarrow v_2 \\
\hline
\text{<ae}_1 / \text{ae}_2, \sigma \rightarrow v
\end{array}
\]

\(v_2 \neq 0\) and \(v = \text{round}(v_1/v_2)\)

\(v_1/v_2\) is division for real numbers; then round toward 0

What if we have \(<x/(y-1), [x\mapsto5, y\mapsto1]>\)? Of course, we have \(<x, [x\mapsto5, y\mapsto1]> \rightarrow 5\) and \(<y-1, [x\mapsto5, y\mapsto1]> \rightarrow 0\)

But the rule is not applicable because of the condition. This is the only rule we could use to derive something for \(<x/(y-1), [x\mapsto5, y\mapsto1]>\), so we are basically “stuck” – no way to derive anything. This happens because the run-time execution has an error

Similar example: \(<z/(y-1), [x\mapsto5, y\mapsto2]>\): use of uninitialized variable; \(\sigma(z)\) is undefined and rule \(<\text{id}, \sigma> \rightarrow \sigma(\text{id})\) cannot be applied
Interpreter for the Language

The inference rules implicitly define a math function \( \text{eval}(\text{code},\text{state}) \) and also the code of an interpreter

Math definition

\[
\begin{align*}
\text{eval}(\text{const}, \sigma) &= \text{const} \\
\text{eval}(\text{id}, \sigma) &= \sigma(\text{id}) \\
\text{eval}(\text{ae}_1 + \text{ae}_2, \sigma) &= \text{eval}(\text{ae}_1, \sigma) + \text{eval}(\text{ae}_2, \sigma)
\end{align*}
\]

Code implementation in an interpreter (e.g., for the programming projects)

```cpp
int eval(TreeNode n, State st) {
    if (n is a const token) return n.lexval
    if (n is an id token) return st.getVal(n.lexval) [also check for uninitialized vars]
    if (n is a plus expression) return
        eval(left subexpression, st) + eval(right subexpression, st)
} 12
```
Evaluation for Boolean Expressions

\(<\text{cond}> ::= \text{true} \mid \text{false} \mid <\text{expr}> < <\text{expr}> \quad [\text{also} \leq, \geq, \approx, \not=] \)

\(\mid <\text{cond}> \&\& <\text{cond}> \mid <\text{cond}> || <\text{cond}>\)

\(\mid ! <\text{cond}> \mid ( <\text{cond}> )\)

\(<\text{be}, \sigma> \rightarrow \nu\)

\(<\text{be}> is a parse subtree derived from <\text{cond}>\)

\(<\sigma> is a state\)

\(<\nu> is a value from \{\text{true, false}\}\)
Evaluation for Boolean Expressions

**Syntax:**

```
true | false | <expr>=<expr> | !<cond> | <cond> & & <cond> | ...
```

| `<true, σ>` | `true` |
| `<false, σ>` | `false` |

\[
\frac{\text{<ae}_1, σ \rightarrow v_1 \quad \text{<ae}_2, σ \rightarrow v_2}{\text{<ae}_1=\text{ae}_2, σ \rightarrow true}
\]

\[v_1 = v_2\]

[Similar rule for \(v_1 \neq v_2\), evaluates to \(false\)]

Also, similar rules for `<`, `<=`, `>`, `>=`, `!=`

| `<be, σ>` | `true` |
| `<be, σ>` | `false` |
| `<!be, σ>` | `false` |
| `<!be, σ>` | `true` |

\[
\frac{\text{<be}_1, σ \rightarrow true \quad \text{<be}_2, σ \rightarrow true}{\text{<be}_1 & & \text{be}_2, σ \rightarrow true}
\]

And three more similar rules, for \(true/false, false/true, false/false\)

Also, similar rules for `<be> || <be>`
Short-Circuit Evaluation

<table>
<thead>
<tr>
<th>&lt;be₁, σ&gt; → true</th>
<th>&lt;be₂, σ&gt; → true</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;be₁ &amp;&amp; be₂, σ&gt; → true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;be₁, σ&gt; → true</th>
<th>&lt;be₂, σ&gt; → false</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;be₁ &amp;&amp; be₂, σ&gt; → false</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;be₁, σ&gt; → false</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;be₁ &amp;&amp; be₂, σ&gt; → false</td>
</tr>
</tbody>
</table>

How about the rules for <be> || <be>?
Execution of Statements

Expression: produces a value; does not change the memory $\sigma$ (the evaluation does not have side effects on the memory)

Note: in imperative languages, some expressions can have side effects (e.g. in C: `x++` or `f()` if function `f` changes existing var)

Statement: does not produce a value; changes the memory $\sigma$; so, we evaluate an expression but we execute a statement

Syntax: `<stmt> ::= skip | id = <expr> | ...`

Semantics: `<s, \sigma> \rightarrow \sigma'`

Starting from initial state $\sigma$, the execution of $s$ completes successfully, and the final state is $\sigma'$
Statements: \(<s, \sigma> \rightarrow \sigma'\)

- **<skip, \sigma> \rightarrow \sigma**

- **<ae, \sigma> \rightarrow \nu**

  "\<id=ae, \sigma> \rightarrow \sigma[id↦\nu]\"  

  "id is mapped (or remapped) to \nu; the rest of the vars are not changed"

- **<be, \sigma> \rightarrow true \<s_1, \sigma> \rightarrow \sigma'**

  "\<if (be) s_1 else s_2, \sigma> \rightarrow \sigma'\"

- **<be, \sigma> \rightarrow false \<s_2, \sigma> \rightarrow \sigma'**

  "\<if (be) s_1 else s_2, \sigma> \rightarrow \sigma'\"

This is for if-then-else; how about for if-then?
What happens with infinite loops? We will not be able to create a derivation tree: e.g., no tree for \textbf{while (true) skip}
Statements

<program> ::= <stmtList>

\[<sl, \sigma> \rightarrow \sigma'\]
\[<p, \sigma> \rightarrow \sigma'\]

<stmtList> ::= <stmt> ; <stmtList> | <stmt>

\[<s, \sigma> \rightarrow \sigma'\]
\[<sl, \sigma'> \rightarrow \sigma''\]
\[<s ; sl, \sigma> \rightarrow \sigma''\]

<stmt> ::= \{<stmtList>\}

\[<sl, \sigma> \rightarrow \sigma'\]
\[<s, \sigma> \rightarrow \sigma'\]
Properties of Big-Step Operational Semantics

**Determinism**: suppose a given program terminates normally (without a run-time error or infinite loop) when executed from initial state $\sigma$. Then there exists a unique state $\sigma'$ such that $<p, \sigma> \rightarrow \sigma'$

Note: If there is a run-time error or infinite loop, it is impossible to derive $<p, \sigma> \rightarrow \sigma'$

**Semantic equivalence**: programs $p_1$ and $p_2$ are equivalent if, for any initial state $\sigma$, $<p_1, \sigma> \rightarrow \sigma'$ if and only if $<p_2, \sigma> \rightarrow \sigma'$

Note: If for some $\sigma$ program $p_1$ terminates normally but $p_2$ does not (or vice versa), they are no equivalent. Either both succeed, or both fail.
Example of Semantic Equivalence

Loop peeling: transform \texttt{while (be) s}

Modified: \texttt{if (be) \{ s ; while (be) do s \}}

Take the first iteration out of the loop
Common compiler optimization; enables a variety of other optimizations

Can we prove that this transformation is semantics-preserving?

For every initial state $\sigma$, $<p_1,\sigma> \rightarrow \sigma'$ if and only if $<p_2,\sigma> \rightarrow \sigma'$
First Half of the Proof (the other half is similar)

If <while..., σ>→σ’ is derivable, so is <if..., σ>→σ’

[slide 18] <while..., σ>→σ’ can be derived in only 2 ways

[Case 1] <be, σ>→false (σ and σ’ are the same state), or

[Case 2] <be, σ>→true  <s, σ>→σ_{intermediate}  <while (be) s, σ_{intermediate}>→σ’

Case 1: Suppose we derived <while..., σ>→σ the first way; then <be, σ>→false is derivable with its own tree; using that, we can build a tree for <if..., σ>→σ [similarly for Case 2]

... some tree here ...

<be, σ>→false

<if (be) s, σ>→σ
Other Examples (in their general form, advanced compiler optimizations)

Partial redundancy elimination

If (be) then { x:=e₁ } else { y:=e₂ }; x:=e₁ transformed to
If (be) then { x:=e₁ } else { y:=e₂; x:=e₁ }

Under what conditions are these two programs equivalent?

Movement of loop-invariant code

Example: \(\textbf{while} \ (\text{be}) \ \textbf{do} \ \{ x:=1+1; \ y:=y+x \} \) equivalent to
\(x:=1+1; \ \textbf{while} \ (\text{be}) \ \textbf{do} \ \{ y:=y+x \}\)?

Example: \(\textbf{do} \ \{ x:=1+1; \ y:=y+x \} \ \textbf{while} \ (\text{be}) \) equivalent to
\(x:=1+1; \ \textbf{do} \ \{ y:=y+x \} \ \textbf{while} \ (\text{be})\)?

Example: \(\textbf{do} \ \{ y:=y+x; x:=1+1 \} \ \textbf{while} \ (\text{be}) \) equivalent to
\(x:=1+1; \ \textbf{do} \ \{ y:=y+x \} \ \textbf{while} \ (\text{be})\)?