Operational Semantics

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

Nielson 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 Definitions

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 \(\{ 0, 2, 4, ... \}\). Note that \(\{ 0, 1, 2, 3, ... \}\) also satisfies the first two rules, but is not the smallest such set

Example: a set L defined as follows:

- \(\text{intconst} \in L\) [for every \text{intconst} token]
- \(\text{ident} \in L\) [for every \text{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 \(<\text{expr}> ::= \text{intconst} \mid \text{ident} \mid <\text{expr}> + <\text{expr}>\)
Background: Inference Rules

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

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

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:

\[
\begin{align*}
\text{intconst} \in L & \quad \text{ident} \in L & \quad e_1 \in L \quad e_2 \in L & \quad e_1 + e_2 \in L
\end{align*}
\]
Simple Language (related to the programming projects)

\[
\text{<program>} ::= \text{<stmtList>}
\]

\[
\text{<stmtList>} ::= \text{<stmt> ; <stmtList>} \mid \text{<stmt>}
\]

\[
\text{<stmt>} ::= \text{int id = <expr>} \quad \text{[for brevity, only consider integer vars/consts]}
\]

\[
\quad \mid \text{id = <expr>}
\]

\[
\quad \mid \text{if ( <cond> ) <stmt>}
\]

\[
\quad \mid \text{if ( <cond> ) <stmt> } \text{else} \text{ <stmt>}
\]

\[
\quad \mid \text{while ( <cond> ) <stmt>}
\]

\[
\quad \mid \{ \text{<stmtList>} \}
\]

\[
\quad \mid \text{skip}
\]
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 $y$

$\sigma(x) = 9$ and $\sigma(y) = 5$

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

$\sigma: \text{Vars} \rightarrow \mathbb{Z}$

$\text{Vars}$ is the set of all variable names in the program
$\mathbb{Z}$ 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 \(<ae, \sigma> \rightarrow v\)

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

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

Example: \(<x+y-1, \{x\mapsto 5, y\mapsto 4\}> \rightarrow 8\)

Example: \(<x/(y-1), \{x\mapsto 5, y\mapsto 1\}> \rightarrow \ldots \quad \text{No triple exists}\)
Evaluation for Arithmetic Expressions

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

\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{<const, \sigma \rightarrow const} \\
\text{const is a parse tree node; const} \in \mathbb{Z} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{<id, \sigma \rightarrow \sigma(id)} \\
\text{axiom, applicable only if the id has a value in } \sigma \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{<ae_1, \sigma \rightarrow v_1} \quad \texttt{<ae_2, \sigma \rightarrow v_2} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{equation}
<ae_1 + ae_2, \sigma \rightarrow v \\
v = v_1 + v_2
\end{equation}
\end{center}

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 \(ae_1 + ae_2\) tells us in which order the operands of + will be evaluated. In fact, their evaluation could be interleaved – do a bit of work for \(ae_1\) then do a bit of work for \(ae_2\) then go again to \(ae_1\) etc. (or even evaluate them in parallel)
Example: Derivation Tree

\( x + 2*y - z \) evaluated in state \( \sigma = [x \mapsto 9, y \mapsto 5, z \mapsto 1] \)

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

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

\[ \text{<ae}_1, \sigma \rightarrow v_1 \quad \text{<ae}_2, \sigma \rightarrow v_2 \]
\[ \text{<ae}_1 / \text{ae}_2, \sigma \rightarrow v \]

\( v_2 \neq 0 \) 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 \( \langle x/(y-1), [x\mapsto 5, y\mapsto 1]\rangle \)? Of course, we have
\( \langle x, [x\mapsto 5, y\mapsto 1]\rangle \rightarrow 5 \) and \( \langle y-1, [x\mapsto 5, y\mapsto 1]\rangle \rightarrow 0 \)

But the rule is not applicable because of the condition. This is the only rule we could use to derive something for \( \langle x/(y-1), [x\mapsto 5, y\mapsto 1]\rangle \), so we are basically “stuck” – no way to derive anything. This happens because the run-time execution has an error

Similar example: \( \langle z/(y-1), [x\mapsto 5, y\mapsto 2]\rangle \): use of uninitialized variable; \( \sigma(z) \) is undefined and rule \( \langle \text{id}, \sigma \rangle \rightarrow \sigma(\text{id}) \) 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

$\text{eval}(\text{const}, \sigma) = \text{const}$

$\text{eval}(\text{id}, \sigma) = \sigma(\text{id})$

$\text{eval}(\text{ae}_1 + \text{ae}_2, \sigma) = \text{eval}(\text{ae}_1, \sigma) + \text{eval}(\text{ae}_2, \sigma)$

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

```java
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)
}
```
Evaluation for Boolean Expressions

<cond> ::= true | false | <expr> < <expr> [also <=, >, >=, ==, !=]

| <cond> && <cond> | <cond> || <cond>

| ! <cond> | ( <cond> )

<be, σ> → ν

be is a parse subtree derived from <cond>
σ is a state
ν is a value from {true, false}
Evaluation for Boolean Expressions

Syntax: \texttt{true} | \texttt{false} | \texttt{<expr>=<expr>} | \texttt{!<cond>} | \texttt{<cond> \&\& <cond>} | ...  

\[
\begin{array}{ll}
\texttt{true}, \sigma & \rightarrow \texttt{true} \\
\texttt{false}, \sigma & \rightarrow \texttt{false}
\end{array}
\]

\[
\begin{array}{ll}
\texttt{ae}_1, \sigma & \rightarrow v \_1 \\
\texttt{ae}_2, \sigma & \rightarrow v \_2
\end{array}
\]

\[
\frac{v \_1 = v \_2}{\texttt{ae}_1=\texttt{ae}_2, \sigma \rightarrow \texttt{true}}
\]

Also, similar rules for \texttt{<, <=, >, >=, !=}

\[
\begin{array}{ll}
\texttt{be}, \sigma & \rightarrow \texttt{true} \\
\texttt{!be}, \sigma & \rightarrow \texttt{false}
\end{array}
\]

\[
\begin{array}{ll}
\texttt{be}, \sigma & \rightarrow \texttt{false} \\
\texttt{!be}, \sigma & \rightarrow \texttt{true}
\end{array}
\]

\[
\begin{array}{ll}
\texttt{be}_1, \sigma & \rightarrow \texttt{true} \\
\texttt{be}_2, \sigma & \rightarrow \texttt{true}
\end{array}
\]

\[
\frac{\texttt{be}_1 \&\& \texttt{be}_2, \sigma \rightarrow \texttt{true}}{	exttt{be}_1 \&\& \texttt{be}_2, \sigma \rightarrow \texttt{true}}
\]

Also, similar rules for \texttt{be} \texttt{||} \texttt{be}

\[
\begin{array}{ll}
\texttt{be}_1, \sigma & \rightarrow \texttt{true} \\
\texttt{be}_2, \sigma & \rightarrow \texttt{true}
\end{array}
\]

\[
\frac{\texttt{be}_1 \&\& \texttt{be}_2, \sigma \rightarrow \texttt{true}}{	exttt{be}_1 \&\& \texttt{be}_2, \sigma \rightarrow \texttt{true}}
\]

Also, similar rules for \texttt{be} \texttt{||} \texttt{be}

Also, similar rules for \texttt{true/false, false/true, false/false}
Short-Circuit Evaluation

\[
\begin{align*}
<\text{be}_1, \sigma> &\rightarrow true \quad <\text{be}_2, \sigma> \rightarrow true \\
\hline
<\text{be}_1 \&\& \text{be}_2, \sigma> &\rightarrow true \\
\end{align*}
\]

\[
\begin{align*}
<\text{be}_1, \sigma> &\rightarrow true \quad <\text{be}_2, \sigma> \rightarrow false \\
\hline
<\text{be}_1 \&\& \text{be}_2, \sigma> &\rightarrow false \\
\end{align*}
\]

\[
\begin{align*}
<\text{be}_1, \sigma> &\rightarrow false \\
\hline
<\text{be}_1 \&\& \text{be}_2, \sigma> &\rightarrow false \\
\end{align*}
\]

How about the rules for \(<\text{be}> \mid\mid <\text{be}>\)?
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: $<s,\sigma> \rightarrow \sigma'$

- $<\text{skip}, \sigma> \rightarrow \sigma$

- $<\text{ae}, \sigma> \rightarrow \nu$
  
  - $<\text{id}=\text{ae}, \sigma> \rightarrow \sigma[\text{id} \mapsto \nu]$ (id is mapped (or remapped) to $\nu$; the rest of the vars are not changed)

- $<\text{be}, \sigma> \rightarrow \text{true}$  $<s_1, \sigma> \rightarrow \sigma'$
  
  - $<\text{if (be)} \ s_1 \ \text{else} \ s_2, \sigma> \rightarrow \sigma'$

- $<\text{be}, \sigma> \rightarrow \text{false}$  $<s_2, \sigma> \rightarrow \sigma'$
  
  - $<\text{if (be)} \ s_1 \ \text{else} \ s_2, \sigma> \rightarrow \sigma'$

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 while (true) skip
Statements

<program> ::= <stmtList>

<sl, σ> → σ’

<pr, σ> → σ’

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

<s, σ> → σ’  <sl, σ’> → σ’’

<s ; sl, σ> → σ’’

<stmt> ::= { <stmtList> }

<sl, σ> → σ’

<s, σ> → σ’

<s, σ> → σ’
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 $\langle p, \sigma \rangle \rightarrow \sigma'$

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

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

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 \textbf{while (be) s}

Modified: \textbf{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 \( <\text{while...}, \sigma> \rightarrow \sigma' \) is derivable, so is \( <\text{if...}, \sigma> \rightarrow \sigma' \)

[slide 18] \( <\text{while...}, \sigma> \rightarrow \sigma' \) can be derived in only 2 ways

[Case 1] \( <\text{be, } \sigma> \rightarrow \textit{false} \) (\( \sigma \) and \( \sigma' \) are the same state), or

[Case 2] \( <\text{be, } \sigma> \rightarrow \textit{true} <s, \sigma> \rightarrow \sigma_{\text{intermediate}} <\text{while (be) } s, \sigma_{\text{intermediate}}> \rightarrow \sigma' \)

Case 1: Suppose we derived \( <\text{while...}, \sigma> \rightarrow \sigma \) the first way; then \( <\text{be, } \sigma> \rightarrow \textit{false} \) is derivable with its own tree; using that, we can build a tree for \( <\text{if...}, \sigma> \rightarrow \sigma \) [similarly for Case 2]

... some tree here ...

\[ \begin{align*}
\text{Case 1:} & \quad <\text{be, } \sigma> \rightarrow \textit{false} \\
\text{Case 2:} & \quad <\text{if (be) } s, \sigma> \rightarrow \sigma
\end{align*} \]
Other Examples (in their general form, advanced compiler optimizations)

Partial redundancy elimination

if be then \{ x=e_1 \} else \{ y=e_2 \}; x=e_1 transformed to
if be then \{ x=e_1 \} else \{ y=e_2; x=e_1 \}

Under what conditions are these two programs equivalent?

Movement of loop-invariant code

Example: while be do \{ x=1+1; y=y+x \} is it equivalent to
\boxed{x=1+1; \text{while be do \{ y=y+x \}}

Example: do \{ x=1+1; y=y+x \} while be is it equivalent to
\boxed{x=1+1; \text{do \{ y=y+x \} while be}

Example: do \{ y=y+x; \boxed{x=1+1} \} while be is it equivalent to
\boxed{x=1+1; \text{do \{ y=y+x \} while be}