Operational Semantics

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

Nielson and Nielson, Ch 2.1
Background: Inductive Definitions

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

Example: a set $L$ defined as follows:

– $\text{IntConst} \subseteq L$ [IntConst is the set of all $\text{intconst}$ tokens]
– $\text{Ident} \subseteq L$ [Ident is the set of all $\text{ident}$ tokens]
– 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 $\langle \text{expr} \rangle ::= \text{intconst} \mid \text{ident} \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle$
Background: Inference Rules
The same thing, written as inference rules [from formal logic]

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:

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

<program> ::= <stmtList>

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

<stmt> ::= \textbf{int} id = <expr> \quad \text{[for brevity, only consider integer vars/consts]}
         | id = <expr>
         | if ( <cond> ) <stmt>
         | if ( <cond> ) <stmt> \textbf{else} <stmt>
         | \textbf{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 $y$

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

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

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

- **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 \(<ae, \sigma> \rightarrow v\)

- \(ae\) 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\mapsto5, y\mapsto4]> \rightarrow 8\>

Example: \(<x/(y-1), [x\mapsto5, y\mapsto1]> \rightarrow ... \quad \text{No triple exists}\>
Evaluation for Arithmetic Expressions

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

\[<\texttt{const}, \sigma> \rightarrow \text{const}\]

\texttt{const} is a parse tree node; \(\texttt{const} \in \mathbb{Z}\)

\[<\texttt{id}, \sigma> \rightarrow \sigma(\text{id})\]

axiom, applicable only if the id has a value in \(\sigma\)

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

\[v = v_1 + v_2\]

\[<\text{ae}_1 + \text{ae}_2, \sigma> \rightarrow v\]

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

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

\[ <2, \sigma> \rightarrow 2 \quad <y, \sigma> \rightarrow 5 \]

\[ <x, \sigma> \rightarrow 9 \quad <2*y, \sigma> \rightarrow 10 \]

\[ <x+2*y, \sigma> \rightarrow 19 \quad <z, \sigma> \rightarrow 1 \]

\[ <x+2*y-z, \sigma> \rightarrow 18 \]
Evaluation for Arithmetic Expressions

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

\[
\begin{align*}
\langle ae_1, \sigma \rangle & \rightarrow v_1 \\
\langle ae_2, \sigma \rangle & \rightarrow v_2 \\
\langle ae_1 / ae_2, \sigma \rangle & \rightarrow v
\end{align*}
\]

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

\[v_1 / v_2 \text{ is division for real numbers; then round toward 0}\]

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

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]>: \text{use of uninitialized variable; } \sigma(z) \text{ is undefined and rule } <\text{id}, \sigma> \rightarrow \sigma(\text{id}) \text{ cannot be applied}\]
Interpreter for the Language

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

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

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

```c
int eval(TreeNode n, State st) {
    if (n is a \texttt{const} token) return n.lexval
    if (n is an \texttt{id} token) return st.getVal(n.lexval) \text{ [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:**

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

<table>
<thead>
<tr>
<th><code>&lt;true, σ&gt;</code></th>
<th><code>true</code></th>
<th><code>&lt;false, σ&gt;</code></th>
<th><code>false</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;ae_1, σ&gt;</code></td>
<td><code>v_1</code></td>
<td><code>&lt;ae_2, σ&gt;</code></td>
<td><code>v_2</code></td>
</tr>
<tr>
<td><code>v_1 = v_2</code></td>
<td><code>&lt;ae_1 = ae_2, σ&gt;</code></td>
<td><code>true</code></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><code>&lt;be, σ&gt;</code></th>
<th><code>true</code></th>
<th><code>&lt;be, σ&gt;</code></th>
<th><code>false</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=!be, σ</code></td>
<td><code>false</code></td>
<td><code>=!be, σ</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;be_1, σ&gt;</code></th>
<th><code>true</code></th>
<th><code>&lt;be_2, σ&gt;</code></th>
<th><code>true</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;be_1 &amp;&amp; be_2, σ&gt;</code></td>
<td><code>true</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Also, similar rules for `||` `be`:

- `true/false`, `false/true`, `false/false`
- `true/true`, `false/false`, `false/true`
- `true/false`, `false/true`, `false/false`
Short-Circuit Evaluation

\[
<\text{be}_1, \sigma> \rightarrow true \quad <\text{be}_2, \sigma> \rightarrow true
\]

\[
<\text{be}_1 \&\& \text{be}_2, \sigma> \rightarrow true
\]

\[
<\text{be}_1, \sigma> \rightarrow true \quad <\text{be}_2, \sigma> \rightarrow false
\]

\[
<\text{be}_1 \&\& \text{be}_2, \sigma> \rightarrow false
\]

\[
<\text{be}_1, \sigma> \rightarrow false
\]

\[
<\text{be}_1 \&\& \text{be}_2, \sigma> \rightarrow false
\]

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: $<\text{stmt}> ::= \text{skip} \mid \text{id} = <\text{expr}> \mid \ldots$

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

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

- \( <\text{skip}, \sigma> \rightarrow \sigma \)

- \( <\text{ae}, \sigma> \rightarrow v \)
  
  \( <\text{id}=\text{ae}, \sigma> \rightarrow \sigma[\text{id} \mapsto v] \)
  
  \( \text{id is mapped (or remapped) to } v; \) 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?
Statements

<be, σ> → false

<while (be) s, σ> → σ

<be, σ> → true  <s, σ> → σ’  <while (be) s, σ'> → σ''

< while (be) s, σ> → σ''

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>

<table>
<thead>
<tr>
<th>&lt;sl, σ&gt; → σ'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;p, σ&gt; → σ'</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>&lt;s, σ&gt; → σ'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sl, σ'&gt; → σ''</td>
</tr>
<tr>
<td>&lt;s sl, σ&gt; → σ''</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;s, σ&gt; → σ'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sl, σ&gt; → σ'</td>
</tr>
</tbody>
</table>

<stmt> ::= { <stmtList> }

<table>
<thead>
<tr>
<th>&lt;sl, σ&gt; → σ'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;s, σ&gt; → σ'</td>
</tr>
</tbody>
</table>
Properties of Big-Step Operational Semantics

**Determinism:** suppose a given program c 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 **while (be) s**

Modified: **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 σ, <c,σ> → σ’ if and only if <c’,σ> → σ’
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

- `<be, σ>→false` (σ and σ’ are the same state), or
- `<be, σ>→true    <s, σ>→σ_{interm}  <while (be) s, σ_{interm}>→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: while (be) do { [x:=1+1; y:=y+x] } equivalent to
[x:=1+1; while (be) do { y:=y+x } ]

Example: do { [x:=1+1; y:=y+x] } while (be) equivalent to
[x:=1+1; do { y:=y+x } while (be) ]

Example: do { y:=y+x; [x:=1+1] } while (be) equivalent to
[x:=1+1; do { y:=y+x } while (be) ]