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?

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 { 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:

intconst $\in$ L [for every **intconst** token]

ident $\in$ L [for every **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 <expr> ::= **intconst** | **ident** | <expr> **+** <expr>

Background: Inference Rules

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

$$\frac{}{0 \in X} \qquad \frac{n \in X}{n+2 \in X}$$

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:

$$\frac{}{\textbf{intconst} \in L} \qquad \frac{}{\textbf{ident} \in L} \qquad \frac{e_1 \in L \quad e_2 \in L}{e_1 \textbf{+} e_2 \in L}$$

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 (related to 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 **σ** from variable names to values

An abstraction of the contents of the physical memory

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

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

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

σ: Vars → Z

Vars is the set of all variable names in the program

Z is the set of integers: *{ 0, -1, 1, -2, 2, ... }*

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**, σ, *v*) but we will write <**ae**, σ> → *v*

ae is a parse subtree derived from <expr>

σ is a state

v is a value from *Z*

Meaning of <**ae**, σ> → *v* : the evaluation of **ae** from state σ completes successfully and produces the value *v*

Example: <**x+y-1**, [x↦5, y↦4]> → *8*

Example: <**x/(y-1)**, [x↦5, y↦1]> → … No triple exists

Evaluation for Arithmetic Expressions

Syntax: **id** | **const** | <expr> **+** <expr> | …

$$\boxed{\langle \textbf{const}, \sigma\rangle \to \mathit{const}}$$ **const** is a parse tree node; *const* $\in Z$

$$\boxed{\langle \textbf{id}, \sigma\rangle \to \sigma(\mathrm{id})}$$ axiom, applicable only if the id has a value in σ

$$\boxed{\dfrac{\langle ae_1, \sigma\rangle \to v_1 \quad \langle ae_2, \sigma\rangle \to v_2}{\langle ae_1 \mathbf{+} ae_2, \sigma\rangle \to v}}$$ $v = v_1 + v_2$

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 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

x + 2*y - z evaluated in state σ = [x↦*9*, y↦*5*, *z*↦*1*]

$$\frac{\dfrac{\langle \mathbf{x}, \sigma\rangle \to 9 \qquad \dfrac{\langle \mathbf{y}, \sigma\rangle \to 5 \qquad \langle \mathbf{2}, \sigma\rangle \to 2}{\langle \mathbf{2*y}, \sigma\rangle \to 10}}{\langle \mathbf{x + 2*y}, \sigma\rangle \to 19} \qquad \langle \mathbf{z}, \sigma\rangle \to 1}{\langle \mathbf{x + 2*y - z}, \sigma\rangle \to 18}$$

Evaluation for Arithmetic Expressions

Syntax: … | <expr> **/** <expr> | …

$$\frac{\langle ae_1, \sigma\rangle \rightarrow v_1 \quad \langle ae_2, \sigma\rangle \rightarrow v_2}{\langle ae_1 / ae_2, \sigma\rangle \rightarrow v}$$

$v_2 \neq 0$ *and* $v = \mathrm{round}(v_1/v_2)$

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

What if we have <**x/(y-1)**, [x↦5, y↦1]>? Of course, we have <**x**, [x↦5, y↦1]> → *5* and <**y-1**, [x↦5, y↦1]> → *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↦5, y↦1]>, 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↦5, y↦2]>: use of uninitialized variable; σ(z) is undefined and rule <**id**, σ> → σ(id) cannot be applied

Interpreter for this Language

The inference rules implicitly define a math function

eval(code,state)

$$eval(\textbf{const}, \sigma) = \textbf{const}.lexval$$

$$eval(\textbf{id}, \sigma) = \sigma(\textbf{id}.lexval)$$

$$eval(ae_1 \mathbf{+} ae_2, \sigma) = eval(ae_1, \sigma) + eval(ae_2, \sigma)$$

An interpreter (e.g., in Project 3) is an implementation of this function

Note: Project 3 has reading of input values from *stdin*; this means that expression could have side effects on *stdin* and evaluation cannot be modeled by such a simple function: e.g., consider **print x + readin x;** *[stdin is part of the state]*

Evaluation for Boolean Expressions

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

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

| **!** <cond> | **(** <cond> **)**

<**be**, σ> → *v*

be is a parse subtree derived from <cond>

σ is a state

v is a value from *{ true, false }*

Evaluation for Boolean Expressions

Syntax: **true** | **false**| <expr>**==**<expr> | **!**<cond> | <cond> **&&** <cond> | …

$$\langle \textbf{true}, \sigma\rangle \rightarrow \mathit{true} \qquad \langle \textbf{false}, \sigma\rangle \rightarrow \mathit{false}$$

$$\frac{\langle ae_1, \sigma\rangle \rightarrow v_1 \quad \langle ae_2, \sigma\rangle \rightarrow v_2}{\langle ae_1 \mathbf{==} ae_2, \sigma\rangle \rightarrow \mathit{true}} \quad v_1 = v_2$$

[similar rule for $v_1 \neq v_2$, evaluates to *false*]

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

$$\frac{\langle be, \sigma\rangle \rightarrow \mathit{true}}{\langle \mathbf{!}be, \sigma\rangle \rightarrow \mathit{false}} \qquad \frac{\langle be, \sigma\rangle \rightarrow \mathit{false}}{\langle \mathbf{!}be, \sigma\rangle \rightarrow \mathit{true}}$$

$$\frac{\langle be_1, \sigma\rangle \rightarrow \mathit{true} \quad \langle be_2, \sigma\rangle \rightarrow \mathit{true}}{\langle be_1 \,\mathbf{\&\&}\, be_2, \sigma\rangle \rightarrow \mathit{true}}$$

and three more similar rules, for true/false, false/true, false/false

Also, similar rules for <be> **||** <be>

Short-Circuit Evaluation

$$\frac{\langle be_1, \sigma\rangle \rightarrow \mathit{true} \quad \langle be_2, \sigma\rangle \rightarrow \mathit{true}}{\langle be_1 \;\&\&\; be_2, \sigma\rangle \rightarrow \mathit{true}}$$

$$\frac{\langle be_1, \sigma\rangle \rightarrow \mathit{true} \quad \langle be_2, \sigma\rangle \rightarrow \mathit{false}}{\langle be_1 \;\&\&\; be_2, \sigma\rangle \rightarrow \mathit{false}}$$

$$\frac{\langle be_1, \sigma\rangle \rightarrow \mathit{false}}{\langle be_1 \;\&\&\; be_2, \sigma\rangle \rightarrow \mathit{false}}$$

How about the rules for <be> || <be>?

Execution of Statements

Expression: produces a value; does not change the memory σ (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 some existing var)

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

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

Semantics: **<s, σ> → σ′**

Starting from initial state σ, the execution of s completes successfully, and the final state is σ′

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

$$<\textbf{skip}, \sigma> \rightarrow \sigma$$

$$\frac{<ae, \sigma> \rightarrow v}{<\textbf{id=}ae, \sigma> \rightarrow \sigma[id \mapsto v]}$$

id is mapped (or remapped) to *v*; the rest of the vars are not changed

$$\frac{<be, \sigma> \rightarrow \mathit{true} \quad <s_1, \sigma> \rightarrow \sigma'}{<\textbf{if (}be\textbf{)}\ s_1\ \textbf{else}\ s_2, \sigma> \rightarrow \sigma'}$$

$$\frac{<be, \sigma> \rightarrow \mathit{false} \quad <s_2, \sigma> \rightarrow \sigma'}{<\textbf{if (}be\textbf{)}\ s_1\ \textbf{else}\ s_2, \sigma> \rightarrow \sigma'}$$

This is for if-then-else; how about for if-then?

Example 1

w = x + 2*y - z executed in state σ = [x↦*9*, y↦*5*, z↦*1*]

$$\dfrac{\langle \mathbf{x}, \sigma\rangle \rightarrow 9 \qquad \dfrac{\langle \mathbf{y}, \sigma\rangle \rightarrow 5 \quad \langle \mathbf{2}, \sigma\rangle \rightarrow 2}{\langle \mathbf{2*y}, \sigma\rangle \rightarrow 10}}{\dfrac{\langle \mathbf{x + 2*y}, \sigma\rangle \rightarrow 19 \qquad \langle \mathbf{z}, \sigma\rangle \rightarrow 1}{\dfrac{\langle \mathbf{x + 2*y - z}, \sigma\rangle \rightarrow 18}{\langle \mathbf{w = x + 2*y - z}, \sigma\rangle \rightarrow \sigma'}}}$$

where σ' = [x↦*9*, y↦*5*, z↦*1*, *w*↦*18*]

Note: we could have written σ[*w*↦*18*] instead of σ'

Example 2

if (x>0) then w = x + 2*y - z else skip in $\sigma = [x \mapsto 9,\ y \mapsto 5,\ z \mapsto 1]$

$$
\dfrac{
  \dfrac{\langle x, \sigma\rangle \to 9 \quad \langle 0, \sigma\rangle \to 0}{\langle x>0, \sigma\rangle \to \mathit{true}}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \langle x, \sigma\rangle \to 9 \quad
        \dfrac{\langle y, \sigma\rangle \to 5 \quad \langle 2, \sigma\rangle \to 2}{\langle 2*y, \sigma\rangle \to 10}
      }{\langle x + 2*y, \sigma\rangle \to 19}
      \quad \langle z, \sigma\rangle \to 1
    }{\langle x + 2*y - z, \sigma\rangle \to 18}
  }{\langle w = x + 2*y - z, \sigma\rangle \to \sigma'}
}{\langle \textbf{if } (x>0) \textbf{ then } w = x + 2*y - z \textbf{ else skip}, \sigma\rangle \to \sigma'}
$$

where $\sigma' = [x \mapsto 9,\ y \mapsto 5,\ z \mapsto 1,\ w \mapsto 18]$

Statements

$$\frac{\langle be, \sigma\rangle \rightarrow \mathit{false}}{\langle \textbf{while } (be)\ s, \sigma\rangle \rightarrow \sigma}$$

$$\frac{\langle be, \sigma\rangle \rightarrow \mathit{true} \quad \langle s, \sigma\rangle \rightarrow \sigma' \quad \langle \textbf{while } (be)\ s, \sigma'\rangle \rightarrow \sigma''}{\langle \textbf{while } (be)\ s, \sigma\rangle \rightarrow \sigma''}$$

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>

$$\frac{\langle sl, \sigma \rangle \rightarrow \sigma'}{\langle p, \sigma \rangle \rightarrow \sigma'}$$

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

$$\frac{\langle s, \sigma \rangle \rightarrow \sigma' \quad \langle sl, \sigma' \rangle \rightarrow \sigma''}{\langle s \; ; \; sl, \sigma \rangle \rightarrow \sigma''}$$

$$\frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle sl, \sigma \rangle \rightarrow \sigma'}$$

<stmt> ::= { <stmtList> }

$$\frac{\langle sl, \sigma \rangle \rightarrow \sigma'}{\langle s, \sigma \rangle \rightarrow \sigma'}$$

Properties of This Operational Semantics

Determinism: suppose a given program c terminates normally (without a run-time error or infinite loop) when executed from initial state σ. Then there exists a unique state σ′ 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 σ, $\langle p_1, \sigma\rangle \rightarrow \sigma'$ if and only if $\langle p_2, \sigma\rangle \rightarrow \sigma'$

Note: If for some σ program p_1 terminates normally but p_2 does not (or vice versa), they are not equivalent. Either both succeed (with the same final state), or both fail.

Semantic Equivalence

Simple example of 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?

Simple examples of movement of loop-invariant code

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

$\boxed{x=1+1;}$ **while** be **do** { y=y+x }

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

$\boxed{x=1+1;}$ **do** { y=y+x } **while** be

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

$\boxed{x=1+1;}$ **do** { y=y+x } **while** be