SWEN430 - Compiler Engineering

Lecture 7 - Operational Semantics

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Operational Semantics?

“The meaning of a program in the strict language is explained in terms of a hypothetical computer which performs the set of actions which constitute the elaboration of that program”

– Report on the Algorithmic Language ALGOL 68.

- Semantics specifies the meaning of programs in a language.
- Operational semantics does so by describing how a program executed.
- Provides a basis for implementing the language, reasoning about programs, showing soundness of the type system, ...

Towards a While Language Calculus ($\lambda_{\mathtt{W}}$)

- **Objective:** to define **operational semantics** of WHILE
- **Problem:** WHILE is **quite complex**
- **Therefore:** we develop a simplified **calculus**, starting with a tiny subset and then expanding it, and hope that this characterises the interesting bits
- **Unfortunately:**
  - Need lots of **simplifications** to make calculus manageable!
  - As a result, it barely resembles **original language**
  - But, there’s not much else we can do in the space/time available!

Semantics for Expressions

- We'll start by considering expressions with variables, numeric and Boolean constants and operators:

e	::=			*expressions*
		|	b	*logical constants*
		|	c	*numeric constants*
		|	n	*variables*
		|	e_1 op e_2	*binary*

- We define the semantics in terms of a *relation*, $\longrightarrow$, which specifies the steps by which an expression is evaluated.
- Since an expression can contain variables, we need a way to refer to their values.
- We let Σ be a mapping from variables to values, called the *store*.
- Thus, $\longrightarrow$ maps $($*Store*, *Expression*$)$ pairs to $($*Store*, *Expression*$)$ pairs, written $\langle \Sigma, \texttt{e} \rangle$.

Semantics for Expressions

$$\frac{\Sigma(\mathtt{n}) = \mathtt{v}}{\langle \Sigma, \mathtt{n} \rangle \longrightarrow \langle \Sigma, \mathtt{v} \rangle} \quad \text{(R-VAR)}$$

$$\frac{\langle \Sigma, \mathtt{e}_1 \rangle \longrightarrow \langle \Sigma, \mathtt{e}_1' \rangle}{\langle \Sigma, \mathtt{e}_1 \ \mathtt{op} \ \mathtt{e}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{e}_1' \ \mathtt{op} \ \mathtt{e}_2 \rangle} \quad \text{(R-BINARY1)}$$

$$\frac{\langle \Sigma, \mathtt{e}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{e}_2' \rangle}{\langle \Sigma, \mathtt{v}_1 \ \mathtt{op} \ \mathtt{e}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{v}_1 \ \mathtt{op} \ \mathtt{e}_2' \rangle} \quad \text{(R-BINARY2)}$$

$$\frac{\vdash \mathtt{v}_1 \ \mathtt{op} \ \mathtt{v}_2 = \mathtt{v}_3}{\langle \Sigma, \mathtt{v}_1 \ \mathtt{op} \ \mathtt{v}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{v}_3 \rangle} \quad \text{(R-BINARY3)}$$

- These rule reduce each expression to a constant, or a variable whose value is retrieved, and then the operator is applied to the resulting values.
- This style of definition is called *small step semantics* because each rule application performs one step in the computation.
- By contrast, *big step semantics* describes computation in terms of larger steps.

Examples

- $$\langle \{\mathtt{x} \mapsto \mathtt{1}\}, \mathtt{x} + \mathtt{1} \rangle$$
  $$\longrightarrow$$
  $$\langle \{\mathtt{x} \mapsto \mathtt{1}\}, \mathtt{1} + \mathtt{1} \rangle$$
  $$\longrightarrow$$
  $$\langle \{\mathtt{x} \mapsto \mathtt{1}\}, \mathtt{2} \rangle$$

- $$\langle \{\mathtt{x} \mapsto \mathtt{1}, \mathtt{y} \mapsto \mathtt{2}, \mathtt{z} \mapsto \mathtt{3}\}, \mathtt{x} + \mathtt{1} > \mathtt{2} * \mathtt{y} + \mathtt{z} \rangle$$
  $$\longrightarrow$$

Semantics for Assignment Statements

For a simple assignment, `n = e;`, we need to:

- evaluate the expression on the right hand side, and
- update the store with the new value.

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{e}' \rangle}{\langle \Sigma, \mathtt{n} = \mathtt{e}\ ; \rangle \longrightarrow \langle \Sigma, \mathtt{n} = \mathtt{e}'\ ; \rangle} \quad \text{(R-Assign1)}$$

$$\frac{\Sigma' = \Sigma[\mathtt{n} \mapsto \mathtt{v}]}{\langle \Sigma, \mathtt{n} = \mathtt{v}\ ; \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle} \quad \text{(R-Assign2)}$$

The function update $\Sigma[n \mapsto v]$ replaces a mapping already in the store, or if there is none adds a new mapping for *n* to the store.
Note that we use `skip` to denote an empty program; i.e. nothing else to do.

Examples

$$\langle \{\mathtt{x} \mapsto 1\}, \mathtt{x = x + 1;} \rangle$$

$$\longrightarrow \langle \{\mathtt{x} \mapsto 1\}, \mathtt{x = 1 + 1;} \rangle$$

$$\longrightarrow \langle \{\mathtt{x} \mapsto 1\}, \mathtt{x = 2;} \rangle$$

$$\longrightarrow \langle \{\mathtt{x} \mapsto 2\}, \mathtt{skip;} \rangle$$

Semantics for If Statements

For an if statement, `if (e) s1 else s2`, we need to:

- evaluate the condition, and
- execute either the true branch or the false branch according to the value of the condition.

$$\frac{\langle \Sigma, \texttt{e} \rangle \longrightarrow \langle \Sigma, \texttt{e}' \rangle}{\langle \Sigma, \texttt{if (e) s}_1\ \texttt{else s}_2 \rangle \longrightarrow \langle \Sigma, \texttt{if (e}'\texttt{) s}_1\ \texttt{else s}_2 \rangle} \quad \text{(R-IF1)}$$

$$\frac{}{\langle \Sigma, \texttt{if (true) s}_1\ \texttt{else s}_2 \rangle \longrightarrow \langle \Sigma, \texttt{s}_1 \rangle} \quad \text{(R-IF2)}$$

$$\frac{}{\langle \Sigma, \texttt{if (false) s}_1\ \texttt{else s}_2 \rangle \longrightarrow \langle \Sigma, \texttt{s}_2 \rangle} \quad \text{(R-IF3)}$$

Semantics for While Statements

For a while statement, `while (e) do s`, we need to:

- evaluate the condition, and
- either exit the loop, if the condition is true; or execute the loop body and then the loop again.

This is based on the law: while (*e*) *s* = if (*e*) { *s* ; while (*e*) *s* }

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{e}' \rangle}{\langle \Sigma, \mathtt{while\ (e)\ s} \rangle \longrightarrow \langle \Sigma, \mathtt{while\ (e')\ s} \rangle} \quad \text{(R-WHILE1)}$$

$$\frac{}{\langle \Sigma, \mathtt{while\ (false)\ s} \rangle \longrightarrow \langle \Sigma, \mathtt{skip} \rangle} \quad \text{(R-WHILE2)}$$

$$\frac{}{\langle \Sigma, \mathtt{while\ (true)\ s} \rangle \longrightarrow \langle \Sigma, \mathtt{s\ ;\ while\ (e)\ s} \rangle} \quad \text{(R-WHILE3)}$$

Note that we again need a representation for an empty program (skip), and now also need a way of handling sequences of statements.

Semantics for Sequences of Statements

For a sequence of statements, we need to execute the first statement in the initial store, then the second statement in the store resulting from that, etc.
There are two different ways we can do this:

- Use a separate rule for sequence, with a special rule to discard skip.

$$\frac{\langle \Sigma, \mathtt{s}_0 \rangle \longrightarrow \langle \Sigma', \mathtt{s}_0' \rangle}{\langle \Sigma, \mathtt{s}_0;\ \overline{\mathtt{s}} \rangle \longrightarrow \langle \Sigma', \mathtt{s}_0';\ \overline{\mathtt{s}} \rangle} \quad \text{(R-SEQ1)}$$

$$\frac{}{\langle \Sigma, \mathtt{skip};\ \overline{\mathtt{s}} \rangle \longrightarrow \langle \Sigma, \overline{\mathtt{s}} \rangle} \quad \text{(R-SEQ2)}$$

Semantics for Sequences of Statements

- Add a "rest of program" part to each of the other rules.

$$\frac{\Sigma' = \Sigma[\mathtt{n} \mapsto \mathtt{v}]}{\langle \Sigma, \mathtt{n} = \mathtt{v}\ ;\ \overline{\mathtt{s}} \rangle \longrightarrow \langle \Sigma', \overline{\mathtt{s}} \rangle} \quad \text{(R-ASSIGN2)}$$

$$\frac{}{\langle \Sigma, \mathtt{while}\ (\mathtt{false})\ \mathtt{s};\ \overline{\mathtt{s}} \rangle \longrightarrow \langle \Sigma, \overline{\mathtt{s}} \rangle} \quad \text{(R-WHILE2)}$$

$$\frac{}{\langle \Sigma, \mathtt{while}\ (\mathtt{true})\ \mathtt{s};\ \overline{\mathtt{s}} \rangle \longrightarrow \langle \Sigma, \mathtt{s}\ ;\ \mathtt{while}\ (\mathtt{e})\ \mathtt{s};\ \overline{\mathtt{s}} \rangle} \quad \text{(R-WHILE3)}$$

Ex: Adapt the other rules.

Ex: Which approach is easier to extend to include continue and break?

Ex: How can we add methods/functions?