SWEN430 - Compiler Engineering

Lecture 8 - Operational Semantics II

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Big Step Semantics

Big step semantics defines semantics in terms of bigger steps.

$$\frac{\langle \Sigma, \mathtt{e}_1 \rangle \longrightarrow \langle \Sigma, \mathtt{v}_1 \rangle, \quad \langle \Sigma, \mathtt{e}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{v}_2 \rangle, \quad \vdash \mathtt{v}_1 \; \mathtt{op} \; \mathtt{v}_2 = \mathtt{v}_3}{\langle \Sigma, \mathtt{e}_1 \; \mathtt{op} \; \mathtt{e}_2 \rangle \longrightarrow \langle \Sigma, \mathtt{v}_3 \rangle} \quad (\text{B-B}\textsc{inary})$$

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{v} \rangle, \quad \Sigma' = \Sigma[\mathtt{n} \mapsto \mathtt{v}]}{\langle \Sigma, \mathtt{n} = \mathtt{v}\,; \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle} \quad (\text{B-A}\textsc{ssign})$$

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{true} \rangle, \quad \langle \Sigma, \mathtt{s}_1 \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle}{\langle \Sigma, \mathtt{if}\; (\mathtt{e}) \; \mathtt{s}_1 \; \mathtt{else} \; \mathtt{s}_2 \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle} \quad (\text{B-I}\textsc{f}1)$$

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{false} \rangle, \quad \langle \Sigma, \mathtt{s}_2 \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle}{\langle \Sigma, \mathtt{if}\; (\mathtt{e}) \; \mathtt{s}_1 \; \mathtt{else} \; \mathtt{s}_2 \rangle \longrightarrow \langle \Sigma', \mathtt{skip} \rangle} \quad (\text{B-I}\textsc{f}2)$$

- Closer to the structure of a recursive interpreter.
- Each rule describes complete execution of a statement.
- Sometimes write $\langle \Sigma \rangle$ instead of $\langle \Sigma', \mathtt{skip} \rangle$, and $\Downarrow$ instead ot $\longrightarrow$.
- Ex: Write big step rules for other constructs.

Semantics for Method Calls

$$\frac{\Sigma(\mathtt{n}_1) = \mathtt{T}_1\ \mathtt{n}_1(\mathtt{T}_2\ \mathtt{n}_2)\{\overline{\mathtt{s}}\},\ \langle\Sigma, \mathtt{e}\rangle \longrightarrow \langle\Sigma, \mathtt{v}\rangle, \quad \langle\Sigma \cup \{\mathtt{n}_2 \mapsto \mathtt{v}\}, \overline{\mathtt{s}}\rangle \longrightarrow \langle\Sigma', \mathtt{skip}\rangle}{\langle\Sigma, \mathtt{n}_1(\mathtt{e})\rangle \longrightarrow \langle\Sigma'|_{\mathtt{dom}(\Sigma)}, \mathtt{skip}\rangle} \quad \text{(R-CALL)}$$

- Look up $\mathtt{n}$ in Σ — may use separate environment for declarations, as in the type rules.
- Evaluate *tte* in current store.
- Execute method body in store augmented with $\mathtt{n}_2$ bound to $\mathtt{e}$'s value.
- Execution continues in new store, without $\mathtt{n}_2$.
- Need a rule to collect method declarations in Σ, as in type rule.

Semantics for Write Statements

$$\frac{\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma, \mathtt{v} \rangle, \quad \mathtt{v}' = \mathtt{toString}(\mathtt{v})}{\langle \Sigma, \mathtt{write(e)} \rangle \longrightarrow \langle \Sigma[\mathtt{output} \mapsto \Sigma(\mathtt{output}) + + \mathtt{v}'], \mathtt{skip} \rangle} \quad \text{(R-WRITE)}$$

- Use a special variable for output, initialised to `""`.
- Evaluate `e`, convert to string representation and appended to the value of `output`.
- Need to retain value of `output` when program terminates.
- Write is the only statement in While which has an effect beyond the current function/method.
- Use similar approach for input, with special variable initialised to program input.

Semantics for While Statements — Take 2

How can we handle `beak` and `continue` statements?

- Let (s_1, s_2) the program part of a configuration mean that the program needs to execute s_1 and then s_2, where s_2 is a loop.

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

- Now a `continue` or `break` or discards the rest of the first component of the pair, and for `break` the second as well.

$$\frac{}{\langle \Sigma, (\texttt{continue};\overline{\texttt{s}},\ \texttt{while (e) s}) \rangle \longrightarrow \langle \Sigma, \texttt{while (e}'\texttt{) s} \rangle} \quad \text{(R-CONTINUE)}$$

$$\frac{}{\langle \Sigma, (\texttt{break};\overline{\texttt{s}},\ \texttt{while (e) s}) \rangle \longrightarrow \langle \Sigma, \texttt{skip} \rangle} \quad \text{(R-BREAK)}$$

- Other statements update the first component as usual. (Ooops! ;)

$$\frac{\langle \Sigma, \texttt{s}_1;\overline{\texttt{s}} \rangle \longrightarrow \langle \Sigma', \overline{\texttt{s}'} \rangle, \quad \texttt{s} \notin \{\texttt{continue}, \texttt{break}\}}{\langle \Sigma, (\texttt{s}_1;\overline{\texttt{s}},\ \texttt{while (e) s}) \rangle \longrightarrow \langle \Sigma', (\overline{\texttt{s}'},\ \texttt{while (e) s}) \rangle} \quad \text{(R-LOOPBODY)}$$

- Ex: Can we handle `return` in a similar way?

Handling Error Conditions

- Many of the rules have implicit conditions that certain operations are well defined. Eg:
  - $\Sigma(\mathrm{n})$ in R-VAR and R-CALL
  - $\mathrm{v}_1 \mathrm{~op~} \mathrm{v}_2$ in expression in B-BINARY
  - Similarly for field and array access when included
- A configuration is *terminated* if there is nothing left to do (a term has been reduced to a value, a program has been reduced to `skip`).
  If there is no rule that can be applied to a non-terminated configuration (either no match or antecedents don't hold), we say that execution is *stuck*.
- Type checking is intended to ensure that certain kinds of errors never occur during program execution (execution doesn't get stuck).
- Ex: What is the difference between the rules for assignment and variable declaration with inialisation?

Soundness of Type Checking

Progress

A well-typed term $\mathtt{t}$ is not stuck (either $\mathtt{t}$ is a value or there exists some transition $\mathtt{t} \longrightarrow \mathtt{t}'$)

Preservation

If a well-typed term is evaluated one step, then the resulting term is also well typed (in fact, it has the same type)

- This provides a link between (small-step) semantics and typing
- Stated in terms of terms (from lambda calculus); easily adapt to programs
- This characterises *strong* typing; can't always achieve this