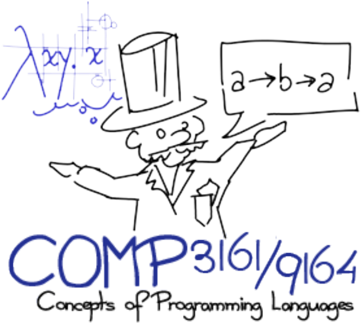
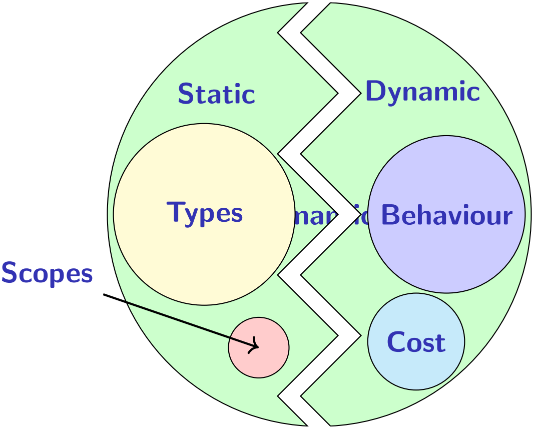


Semantics

Johannes Åman Pohjola
UNSW
Term 3 2022

Semantics

$\sigma\eta\mu\alpha\nu\tau\iota\chi\omega\varsigma$



Static Semantics

Definition

The *static semantics* of a program is those significant aspects of the meaning of P that can be determined by the compiler (or an external lint tool) without running the program.

Recall our arithmetic expression language. What properties might we derive statically about those terms?

The only thing we can check is that the program is well-scoped (assuming FOAS).

Scope-Checking

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Times}\ e_1\ e_2)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2)\ \textbf{ok}}$$

$$\frac{(x\ \textbf{bound}) \in \Gamma}{\Gamma \vdash (\texttt{Var}\ x)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad x\ \textbf{bound}, \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Let}\ x\ e_1\ e_2)\ \textbf{ok}}$$

Key Idea

We keep a *context* Γ, a set of assumptions, on the LHS of our judgement, indicating what is required in order for e to be *well-scoped*.
This could be read as hypothetical derivations for the judgement e **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

Scope-Checking Example

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Times}\ e_1\ e_2)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2)\ \textbf{ok}}$$

$$\frac{(x\ \textbf{bound}) \in \Gamma}{\Gamma \vdash (\texttt{Var}\ x)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad x\ \textbf{bound}, \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash (\texttt{Let}\ x\ e_1\ e_2)\ \textbf{ok}}$$

$$\frac{\dfrac{}{\vdash (\texttt{N 3})} \qquad \dfrac{\dfrac{}{\texttt{"x"} \vdash (\texttt{N 4})} \qquad \dfrac{\dfrac{}{\texttt{"y"}, \texttt{"x"} \vdash (\texttt{V "x"})} \qquad \dfrac{}{\texttt{"y"}, \texttt{"x"} \vdash (\texttt{V "y"})}}{\texttt{"y"}, \texttt{"x"} \vdash (\texttt{Plus (V "x") (V "y")})}}{\texttt{"x"} \vdash (\texttt{Let "y" (N 4) (Plus (V "x") (V "y"))})}}{\vdash (\texttt{Let "x" (N 3) (Let "y" (N 4) (Plus (V "x") (V "y")))})}$$

Dynamic Semantics

Dynamic Semantics can be specified in many ways:

1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. COMP6752 (briefly)
2. *Axiomatic Semantics* is the construction of a *proof calculus* to allow correctness of a program to be verified. COMP2111, COMP6721
3. *Operational Semantics* is the construction of a program-evaluating *state machine* or *transition system*.

In this course

We focus mostly on operational semantics. We will use axiomatic semantics (Hoare Logic) on Thursday in the imperative programming topic. Denotational semantics are mostly an extension topic, except for the very next slide.

Denotational Semantics

$$[\![\cdot]\!] : \textbf{AST} \to (\textbf{Var} \nrightarrow \mathbb{Z}) \to \mathbb{Z}$$

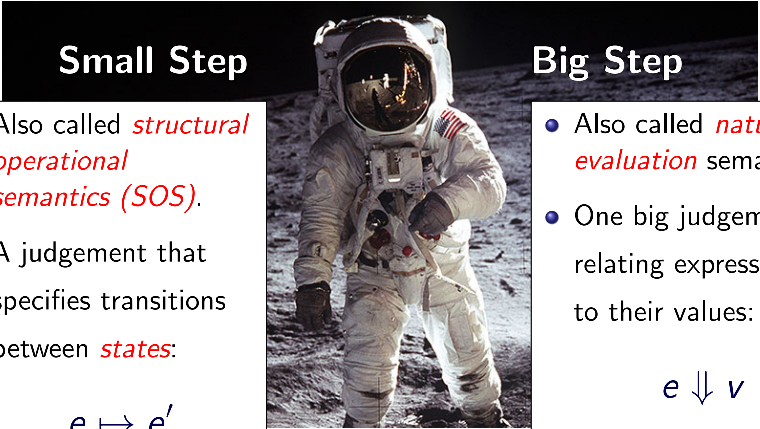
Our denotation for arithmetic expressions is functions from *environments* (mapping from variables to their values) to values.

$$\begin{array}{lcl}
[\![\texttt{Num}\ n]\!] & = & \lambda E.\ n \\
[\![\texttt{Var}\ x]\!] & = & \lambda E.\ E(x) \\
[\![\texttt{Plus}\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_1]\!]E + [\![e_2]\!]E \\
[\![\texttt{Times}\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_1]\!]E \times [\![e_2]\!]E \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_2]\!]\ (E[x := [\![e_1]\!]E])
\end{array}$$

Where $E[x := n]$ is a new environment just like E, except the variable x now maps to n.

Operational Semantics

There are two main kinds of operational semantics.



Small Step

- Also called *structural operational semantics (SOS)*.
- A judgement that specifies transitions between *states*:

$$e \mapsto e'$$

Big Step

- Also called *natural* or *evaluation* semantics.
- One big judgement relating expressions to their values:

$$e \Downarrow v$$

Big-Step Semantics

We need:

- A set of evaluable expressions E
- A set of values V
- A relation $\Downarrow \subseteq E \times V$

Example (Arithmetic Expressions)

E is the set of all closed expressions $\{e \mid e\ \textbf{ok}\}$. V is the set of integers $\mathbb{Z}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow n} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

To Code Let's do it in Haskell!

Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$

Any other ways to evaluate `Let`?

The above is called *call-by-value* or strict evaluation. Below we have *call-by-name*:

$$\frac{e_2[x := e_1] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$

This can be computationally very expensive, for example:

$$\texttt{let}\ x = \langle \textit{very expensive computation} \rangle\ \texttt{in}\ x + x + x + x$$

In confluent languages like this or λ-calculus, this only matters for performance. In other languages, this is not so. Why?

Haskell uses *call-by-need* or lazy evaluation, which optimises cases like this.

Small Step Semantics

For small step semantics, we need:

- A set of states Σ
- A set of initial states $I \subseteq \Sigma$
- A set of final states $F \subseteq \Sigma$
- A relation $\mapsto \subseteq \Sigma \times \Sigma$, which specifies only "one step" of the execution.

An *execution* or *trace* $\sigma_1 \mapsto \sigma_2 \mapsto \sigma_3 \mapsto \cdots \mapsto \sigma_n$ is called *maximal* if there exists no σ_{n+1} such that $\sigma_n \mapsto \sigma_{n+1}$; and is called *complete* if it is maximal and $\sigma_n \in F$.

Example

Example (Arithmetic Expressions)

Σ and I are the set of all closed expressions $\{e \mid e\ \textbf{ok}\}$, F is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{Plus}\ e_1'\ e_2)} \qquad \frac{e_2 \mapsto e_2'}{(\texttt{Plus}\ (\texttt{Num}\ n)\ e_2) \mapsto (\texttt{Plus}\ (\texttt{Num}\ n)\ e_2')}$$

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ (n+m))}$$

(Similarly for $\texttt{Times}$)

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.\ e_2))}$$

$$\frac{}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.\ e_2)) \mapsto e_2[x := \texttt{Num}\ n]}$$

To Code Let's do it in Haskell!

Equivalence

Comparing small step and big step

Small step semantics are lower-level, they clearly specify the order of evaluation. Big step semantics give us a result without telling us explicitly how it was computed.

Having specified the dynamic semantics in these two ways, it becomes desirable to show they are equivalent, that is:

If there exists a trace $e \mapsto \cdots \mapsto (\texttt{Num}\ n)$, then $e \Downarrow n$, and vice versa.

We will need to define some notation to remove those blasted magic dots.

Notation

Let $\overset{\star}{\mapsto}$ be the *reflexive, transitive closure* of $\mapsto$.

$$\frac{}{e \overset{\star}{\mapsto} e} \qquad \frac{e_1 \mapsto e_2 \qquad e_2 \overset{\star}{\mapsto} e_n}{e_1 \overset{\star}{\mapsto} e_n}$$

We can now state our property formally as:

$$e \overset{\star}{\mapsto} (\texttt{Num}\ n) \iff e \Downarrow n$$

Doing the Proof

The proof will be done on the "board", with typeset versions uploaded later.
The big-step to small-step direction can be proven by reasonably straightforward rule induction:

$$\frac{e \Downarrow n}{e \overset{\star}{\mapsto} (\texttt{Num}\ n)}$$

The other direction requires the lemma:

$$\frac{e \mapsto e' \qquad e' \Downarrow n}{e \Downarrow n}$$

The abridged proof is presented in this lecture, with all cases left for the course website.

Big and small (eliding some small-step rules)

$$\frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{Plus}\ e_1'\ e_2)} \qquad \frac{e_2 \mapsto e_2'}{(\texttt{Plus}\ (\texttt{Num}\ n)\ e_2) \mapsto (\texttt{Plus}\ (\texttt{Num}\ n)\ e_2')}$$

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ (n+m))}$$

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.\ e_2))}$$

$$\frac{}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.\ e_2)) \mapsto e_2[x := \texttt{Num}\ n]}$$

$$\frac{}{(\texttt{Num}\ n) \Downarrow n} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$