G54FOP: Lecture 3

Programming Language Semantics: Introduction

Henrik Nilsson

University of Nottingham, UK

Why Does PL Semantics Matter? (1)

- Documentation
  - Programmers ("What does X mean? Did the compiler get it right?")
  - Implementers ("How to implement X?")
- Formal Reasoning
  - Proofs about programs
  - Proofs about programming languages (E.g. "Well-typed programs do not go wrong")
  - Proofs about tools (E.g. compiler correctness)

Why Does PL Semantics Matter? (2)

- Language Design
  - Semantic simplicity is a good guiding principle
  - Ensure desirable meta-theoretical properties hold (like "well-typed programs do not go wrong")
- Education
  - Learning new languages
  - Comparing languages
- Research

Static vs. Dynamic Semantics (1)

- ***Static Semantics***: "compile-time" meaning
  - Scope rules
  - Type rules

  Example: the meaning of `1+2` is an integer value (its ***type*** is Integer)
- ***Dynamic Semantics***: "run-time" meaning
  - Exactly what value does a term evaluate to?
  - What are the effects of a computation?

  Example: the meaning of `1+2` is the integer 3.

Static vs. Dynamic Semantics (2)

Distinction between static and dynamic semantics not always clear cut. E.g.

- Multi-staged languages ("more than one run-time")
- Dependently typed languages (computation at the type level)

Styles of Semantics (1)

Main examples:

- ***Operational Semantics***: Meaning given by ***Abstract Machine***, often a ***Transition Function*** mapping a state to a "more evaluated" state.

  Kinds:
  - ***small-step*** semantics: each step is atomic; more machine like
  - ***structural operational semantics (SOS)***: compound, but still simple, steps
  - ***big-step*** or ***natural*** semantics: Single, compound step evaluates term to final value.

Styles of Semantics (2)

- ***small-step*** semantics:

$$t \rightarrow t^{(1)} \rightarrow t^{(2)} \rightarrow \ldots \rightarrow v$$

- ***structural operational semantics***:

$$t \xrightarrow{\nabla} t^{(1)} \xrightarrow{\nabla} t^{(2)} \xrightarrow{\nabla} \ldots \xrightarrow{\nabla} v$$

- ***big-step*** or ***natural*** semantics:

$$t \xrightarrow{\nabla} v$$

where ∇ suggests a proof (tree) justifying the step.

Example: Small Step Semantics (1)

Consider a simple machine for evaluating arithmetic expressions. Its state is denoted

$$(\bar{t}, \bar{v})$$

where

- $\bar{t}$ is a sequence of expressions (including values) and operators
- $\bar{v}$ is a stack (sequence) of values

Example: Small-Step Semantics (2)

Idea of the machine. Given state $(\bar{t}, \bar{v})$:

- If $\bar{t}$ is empty, we're done; whatever is on the value stack $\bar{v}$ is the result.
- If head of $\bar{t}$ is a value, push it onto $\bar{v}$.
- If head of $\bar{t}$ is an expression, prepend the individual subexpressions and the operator to the tail of $\bar{t}$.
- If head of $\bar{t}$ is an operator, apply it to appropriate number of arguments on top of the value stack and replace them with the result.

Example: Small-Step Semantics (3)

An evaluation might proceed as follows:

$$
\begin{aligned}
(\langle \texttt{1+(2-3)} \rangle, \langle\rangle) &\rightarrow (\langle \texttt{2-3}, \texttt{1}, \texttt{+} \rangle, \langle\rangle) \\
&\rightarrow (\langle \texttt{3}, \texttt{2}, \texttt{-}, \texttt{1}, \texttt{+} \rangle, \langle\rangle) \\
&\rightarrow (\langle \texttt{2}, \texttt{-}, \texttt{1}, \texttt{+} \rangle, \langle \texttt{3} \rangle) \\
&\rightarrow (\langle \texttt{-}, \texttt{1}, \texttt{+} \rangle, \langle \texttt{2}, \texttt{3} \rangle) \\
&\rightarrow (\langle \texttt{1}, \texttt{+} \rangle, \langle \texttt{-1} \rangle) \\
&\rightarrow (\langle \texttt{+} \rangle, \langle \texttt{1}, \texttt{-1} \rangle) \\
&\rightarrow (\langle\rangle, \langle \texttt{0} \rangle)
\end{aligned}
$$

Exercise: Evaluate $(\langle \texttt{(5-3)*(1+2)} \rangle, \langle\rangle)$

Example: Small-Step Semantics (4)

Note:

- Each step describes a small, essentially mechanical, ***syntactic*** transformation of the machine state; i.e., a very operational view of computation.
- Our description was informal. We will discuss at length how to formalise operational semantics in a mathematically precise way, typically using ***inference rules***.

Styles of Semantics (3)

- ***Denotational Semantics***: More abstract view:
  - meaning of a term is a mathematical object (like a ***number*** (e.g. $\mathbb{N}$ or $\mathbb{Z}$) or ***function*** (e.g. $\mathbb{Z} \rightarrow \mathbb{Z}$);
  - an ***interpretation function*** maps terms ((abstract) syntax) to their meaning (semantics).

Example: Denotational Semantics (1)

Given the abstract syntax for expressions:

$$
\begin{array}{lll}
e & \rightarrow & \textit{expressions:} \\
 & \mid \ \mathbb{Z} & \textit{integer literals} \\
 & \mid \ e \ \texttt{+} \ e & \textit{addition} \\
 & \mid \ e \ \texttt{-} \ e & \textit{subtraction} \\
 & \mid \ e \ \texttt{*} \ e & \textit{multiplication}
\end{array}
$$

Example: Denotational Semantics (2)

the denotational semantics (the interpretation function mapping the (abstract) syntax of an expression to its meaning) might be specified as:

$$
\begin{aligned}
[\![\cdot]\!] &: e \rightarrow \mathbb{Z} \\
[\![n]\!] &= n \\
[\![e_1 \ \texttt{+} \ e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![e_1 \ \texttt{-} \ e_2]\!] &= [\![e_1]\!] - [\![e_2]\!] \\
[\![e_1 \ \texttt{*} \ e_2]\!] &= [\![e_1]\!] \times [\![e_2]\!]
\end{aligned}
$$

Not vacuous: e.g., note the difference between **+**, ***syntax***, and $+$, the ordinary function plus.

Example: Denotational Semantics (3)

Exercise. Given:

$$
\begin{aligned}
[\![\cdot]\!] &: e \rightarrow \mathbb{Z} \\
[\![n]\!] &= n \\
[\![e_1 \ \texttt{+} \ e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![e_1 \ \texttt{-} \ e_2]\!] &= [\![e_1]\!] - [\![e_2]\!] \\
[\![e_1 \ \texttt{*} \ e_2]\!] &= [\![e_1]\!] \times [\![e_2]\!]
\end{aligned}
$$

Calculate the denotation (meaning) of:

`1 + (2 - 3)`

Styles of Semantics (4)

- ***Axiomatic Semantics***: More abstract still:
  - An operational or denotational semantics ***implies*** certain properties or ***laws***.
  - An axiomatic semantics takes such laws as the starting point: the laws defines the semantics and the meaning is just what can be proved.
  - Closely related to Hoare logic.

Example: Axiomatic Semantics (1)

The meaning of a command c is given by specifying a ***precondition*** and a ***postcondition***:

$$\{\mathit{PRE}\} \ c \ \{\mathit{POST}\}$$

This says: If PRE holds for a program state, then executing c in that state will terminate and POST will hold in the resulting state.

Example: Axiomatic Semantics (2)

For example, the meaning of assignment is given by:

$$\{P[v \mapsto e]\} \ v \ \texttt{:=} \ e \ \{P\}$$

This says: For ***any*** predicate P whatsoever that holds in a program state when e is substituted for free occurrences of v, it is the case that P holds in the state resulting after the assignment.

Note: Nothing is said about ***how*** execution is carried out, but the focus is on ***what*** its effect is.

Example: Axiomatic Semantics (3)

We can now prove e.g.

$$\{x = 7\}\ \texttt{x := x+1}\ \{x = 8\}$$

(often easier to work backwards from postcondition):

$\{x = 8\}$	postcondition
$\{x + 1 = 8\}$	substituting $x + 1$ for x
$\{x = 8 - 1\}$	arithmetic
$\{x = 7\}$	simplification yields precondition

Example: Axiomatic Semantics (4)

Works for ***any*** predicate. Consider proving:

$$\{x \geq 0\}\ \texttt{x := x+1}\ \{x > 0\}$$

$\{x > 0\}$	postcondition
$\{x + 1 > 0\}$	substituting $x + 1$ for x
$\{x + 1 \geq 1\}$	$n > 0 \equiv n \geq 1$ for any integer n
$\{x \geq 1 - 1\}$	arithmetic
$\{x \geq 0\}$	simplification yields precondition

Example: Axiomatic Semantics (5)

Exercise. Prove:

$$\{i = 6 \wedge j = 7\}\ \texttt{i := i * j}\ \{i = 42 \wedge j = 7\}$$