Operational Semantics and the Lambda Calculus

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 8, 2021

WARNING: This lecture is going to be more mathematical than usual.

Table of Contents

Table of Contents

Syntax and Semantics

All programming languages have syntax and semantics.

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Definition (Semantics)

"Semantics reveals the meaning of syntactically valid strings in a language" [Slonneger and Kurtz 1995].

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Definition (Semantics)

"Semantics reveals the meaning of syntactically valid strings in a language" [Slonneger and Kurtz 1995].

This lecture will focus on semantics.

How do we define the semantics of a programming language?

We could point users to the code that implements a reference interpreter or compiler of that language. This is the approach that many real-world programming languages take.

But there are problems with the "show me the code" approach to semantics.

Problems with "Show Me the Code" Semantics

- What if the code has errors?
- What happens when someone decides to write a different implementation of the language?
- What happens when the language gets ported to a different architecture or operating system?

Another approach to defining the semantics of a language is writing official documentation describing in human language the details of the language.

This documentation can take the form of

- Reports
- Books (such as *The C Programming Language* by Brian Kernighan and Dennis Ritchie)
- Standards published by a standards agency such as ANSI

Unfortunately, even with standards, there can still be problems that arise with human-language descriptions of programming language semantics.

An alternative to natural language-defined semantics is *formal semantics*, which makes it possible to reason about the semantics of a programming language in a logical, mathematical fashion.

Why Formal Semantics?

- Provides a degree of precision that natural-language semantic descriptions couldn't provide.
- Facilitates the ability to mathematically prove specific properties of the language.

There are different systems of formal semantics, but in this course we will be focusing on operational semantics.

Operational Semantics

Definition (Operational Semantics)

"[S]pecifies the behavior of a programming language by defining a simple *abstract machine* for it" [Pierce, *Types and Programming Languages*, 2002].

Simple Arithmetic Expression Language: Syntax

```
<operator>   ::= + | -

<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digits>     ::= <digit>
               | <digit><digits>

<integer>    ::= <digits>
               | -<digits>

<expr>       ::= <integer>
               | <expr> <operator> <expr>
```

Small-Step Semantics

- Small-step semantics "shows how individual steps of computation are used to rewrite a term, bit by bit, until it eventually becomes a value" [Pierce 2002, p. 42].
- We go step-by-step starting from non-terminal symbols and eventually working our way down to the point where we have nothing but terminal symbols.

Small-Step Semantics

- In small-step semantics, we define a collection of *evaluation relations*.
- Each evaluation relation has the form $t \rightarrow t'$, which means "t evaluates to t'" [Pierce 2002, p. 34-35].
- If an evaluation relation has the following form

$$\frac{t \rightarrow t'}{u \rightarrow u'}$$

then it means "if t evaluates to t', then u evaluates to u'."

Small-Step Semantics of Simple Arithmetic Expression Language

Let n be an integer, e be an expression, and op be either the $+$ or $-$ operators.

1. $n_1 + n_2 \to n_3$ (where $n_3 = n_1 + n_2$)
2. $n_1 - n_2 \to n_3$ (where $n_3 = n_1 - n_2$)
3. $$\frac{e_1 \to e_1'}{e_1 \ op \ e_2 \to e_1' \ op \ e_2}$$
4. $$\frac{e_2 \to e_2'}{e_1 \ op \ e_2 \to e_1 \ op \ e_2'}$$

Note that we are treating integers as terminal symbols in our semantics despite the fact that the `<integer>` rule is non-terminal in our grammar. This is to simplify matters.

Example of Small-Step Semantics on an Expression

Let's use small-step semantics to evaluate the expression $2 + 3 - 4 \to 1$.

$$\dfrac{\dfrac{\overline{2 + -1 \to 1}}{3 - 4 \to -1}}{2 + 3 - 4 \to 1}$$

When drawing the above derivation, we start from the bottom with the original expression, and then we derive each subexpression, going upward until we have no more subexpressions to derive.

NOTE: You will not be required to perform your own small-step derivations on the midterm or final exams in this class.

Big-Step Semantics

- Recall that in small-step semantics we perform individual derivation steps until we reach terminal symbols.
- In big-step semantics, we go directly from non-terminal rules to terminal values.
- Each evaluation statement has the form $t \Downarrow v$, where t is the original term and v is the resulting value [Pierce 2002, p. 43].

Big-Step Semantics of Simple Expression Language

1. $n \Downarrow n$
2. $$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_3}$$
   where $n_3 = n_1 + n_2$
3. $$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 - e_2 \Downarrow n_3}$$
   where $n_3 = n_1 - n_2$

One nice characteristic of big-step semantics is that it is easy to write interpreters given an abstract syntax tree and a semantic definition of the language.

Are there languages defined using operational semantics?

Scheme R6RS is defined via operational semantics; check out Appendix A of *The Revised^6 Report on the Algorithmic Language Scheme*.

Summary

- Semantics defines the meaning of sentences in a programming language.
- Formal semantics allow us to define programming languages in a logical, mathematical fashion.
- Operational semantics specifies the behavior of a programming language by specifying an artificial machine for it.
- Big-step operational semantics is ideal for programming language implementers.

Table of Contents

This is the very beginning of our transition from procedural programming to functional programming, which will be our focus for the next six weeks.

I like to think about the development of programming languages as two schools of thought: one rooted in a hardware-oriented point of view, and one rooted in a mathematical point of view.

Procedural programming was developed largely under pragmatic concerns: how do we save ourselves from the tedium of performing low-level programming tasks?

Functional programming, however, approaches programming from a different point of view: how do we express our programs as mathematical functions, and how do we run them efficiently on computer hardware?

Von Neumann computer architectures can be thought of as the reification of the Turing machine model of computation.

Functional programming languages can be thought of as the reification of the *lambda calculus*.

Some Background

- In the beginning of the 20th century there were a lot of research efforts by mathematicians and logicians in the area of *metamathematics*.
- Hilbert's program (by mathematician David Hilbert) was an initiative to see if all of the theorems of mathematics can be built upon a set of axioms that were proven to be consistent.
- However, logician Kurt Gödel proved that it is impossible to prove the consistency of axioms within the same logical system; this result is known as Gödel's Second Incompleteness Theorem.

Some Background

- Logician Alonzo Church formulated the lambda calculus as part of his research on metamathematics.
- The purpose of the lambda calculus is to develop a mathematical model for expressing computation.
- Theoretically, any computable function can be expressed as a lambda calculus expression.
- In addition, the Church-Turing Thesis is a hypothesis stating that any function expressed by the lambda calculus is computable by a Turing machine.

Lambda Calculus Syntax

$$\begin{aligned} <\lambda expr> ::= \; & <var> \\ & |\lambda <var> . <\lambda expr> \\ & |(<\lambda expr> <\lambda expr>) \end{aligned} \tag{1}$$

The first rule represents a variable. The second represents an abstraction, which is a function definition. The third represents an application, which is a function call.

Abstraction

$$\lambda <var> . <\lambda expr> \tag{2}$$

$<var>$ is the function parameter and $<\lambda expr>$ is the function body. All functions in the lambda calculus only have one parameter, and all functions are anonymous.

Application

$$(f\ x) \tag{3}$$

Call the function f with argument x; equivalent to $f(x)$ in standard mathematical notation. This type of notation is known as prefix notation.

Examples of Lambda Calculus Expressions

- x
- $\lambda x.x$
- $\lambda x.\lambda y.(\lambda u.v\ \lambda u.z)$

Variable Scoping in the Lambda Calculus

- If a variable x occurs within the body t of an abstraction $\lambda x.t$, then x is bound, and λx is a binder whose scope is t.
- If x is not bound by an enclosing abstraction on x, then it is free.
- If a term has no free variables, it is closed. A combinator is a closed term.

Example: In the λ-expression $(\lambda y.x\ y)$, x is free and y is bound.

Evaluating Lambda Calculus Expressions

Here are some simple examples:

- $x \Rightarrow x$
- $(\lambda x.x\ y) \Rightarrow y$
- $\lambda x.x \Rightarrow \lambda x.x$

However, not all evaluations are straightforward applications.

Function applications often involve *substitutions* of terms. However, we must make sure that no free variables become mistakenly bound as a result of substitution, or else we cause the problem of variable capture.

To avoid variable capture, we perform α-conversion, which is renaming in such a way where the semantic meaning of a function abstraction does not change. We accomplish this by using a new variable name, one that does not occur in the body of the function being α-converted.

Concluding Thoughts

There is a lot more that can be said about the lambda calculus; in fact, it is possible to teach an entire semester-long class on the lambda calculus and its applications to mathematics and computer science. The lambda calculus is used sometimes by programming language researchers as a means of defining semantics.

Table of Contents

On Monday, we will begin our lessons on Scheme, a functional programming language that is part of the Lisp family of programming languages. Lisp can be thought of as a reification of the lambda calculus, except it's much easier to code in than the lambda calculus. We will also cover the core tenets of functional programming.