Operational Semantics and the Lambda Calculus

Michael McThrow

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

August 31, 2020

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.

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

Problems with "Show Me the Code" Semantics

Problems with "Show Me the Code" Semantics

- What if the code has errors?

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?

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 writting 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?

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

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].

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].

Our example will demonstrate *big-step semantics*, also known as *natural semantics*.

Arith Programming Language [Pierce 2002]

```
<t> ::= true
      | false
      | if <t> then <t> else <t>
      | 0
      | succ <t>
      | pred <t>
      | iszero <t>
```

Examples of Valid Expressions in Arith

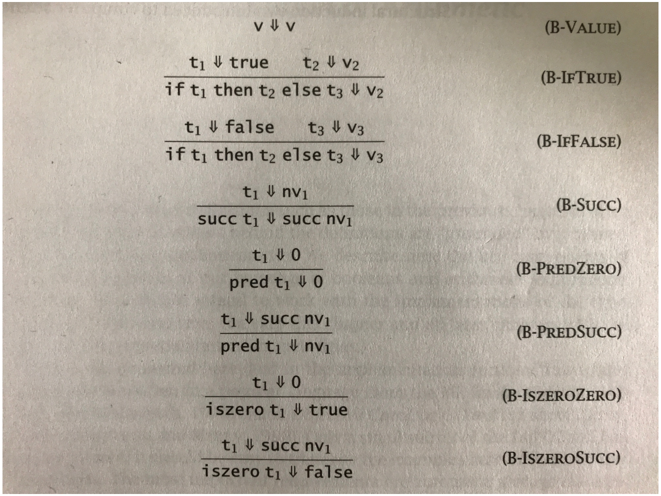
```
true
succ 0
pred 0
succ succ pred succ pred 0
if iszero succ pred 0 then true else false
```

Examples of Semantically Incorrect Expressions in Arith

Note: these expressions are syntactically valid, but semantically incorrect.

```
iszero false
if 0 then true else false
succ true
```

Big-Step Definition of Arith [Pierce 2002, p. 43]



$$\mathtt{v} \Downarrow \mathtt{v} \qquad \text{(B-VALUE)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{true} \qquad \mathtt{t}_2 \Downarrow \mathtt{v}_2}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{false} \qquad \mathtt{t}_3 \Downarrow \mathtt{v}_3}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{nv}_1}{\mathtt{succ}\ \mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1} \qquad \text{(B-SUCC)}$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow \mathtt{nv}_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{false}} \qquad \text{(B-ISZEROSUCC)}$$

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.

Is this lambda expression syntactically correct?

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

Is this lambda expression syntactically correct?

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

No.

Is this lambda expression syntactically correct?

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

No. The reason why it is not syntactically correct is because an application can only have one argument; there are two in the application that is part of the body of the above abstraction.

Is this lambda expression syntactically correct?

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

Is this lambda expression syntactically correct?

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

Yes.

Is this lambda expression syntactically correct?

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

Is this lambda expression syntactically correct?

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

Yes.

Is this lambda expression syntactically correct?

$$\lambda v \; ((v \; u) \; (\lambda v.y \; (v \; y)))$$

Is this lambda expression syntactically correct?

$$\lambda v\ ((v\ u)\ (\lambda v.y\ (v\ y)))$$

No. A λ and a variable must be followed by a dot.

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.

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.

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.

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.

Substitution Rules for λ-Expressions

Here are some simple rules:

- $v[v \rightarrow E_1] = E_1$
- When $v \neq x$, $x[v \rightarrow E_1] = x$
- $(E_A\ E_B)[v \rightarrow E_1] = ((E_A[v \rightarrow E_1])\ (E_B[v \rightarrow E_1]))$
- $(\lambda v.E)[v \rightarrow E_1] = (\lambda v.E)$

Substitution Rules for Abstractions when $x \neq v$

- If x is not free in E_1, then $(\lambda x.E)[v \to E_1] = \lambda x.(E[v \to E_1])$
- Else, we need to perform an α-conversion, which is $(\lambda x.E)[v \to E_1] = \lambda z.(E[x \to z][v \to E_1])$ where $z \neq v$.

β-redex

Definition (β-redex)

A β-redex is an application where the first term is an abstraction (e.g., $(\lambda x.x\ y)$ is a β-redex, but not $(x\ y)$).

β-reduction

Definition (β-reduction)

Given a β-redex, if v is a variable and E and E_1 are λ-expressions, then

$$(\lambda v.E\ E_1) \Rightarrow_\beta E[v \rightarrow E_1]$$

provided that the substitution $E[v \rightarrow E_1]$ is carried out according to the rules for a safe substitution.

Evaluating λ-expressions

We can evaluate lambda expressions by repeatedly applying β-reductions until all β-redexes have been removed.

Evaluating λ-expressions

We can evaluate lambda expressions by repeatedly applying β-reductions until all β-redexes have been removed.

- The resulting lambda expression would be in *normal form*.
- However, not all lambda expressions are reducible to normal form.

Lambda Expression Evaluation Strategies

- In full β-reduction, any redex may be reduced at any time.
- In normal order reduction, the leftmost, outermost redex is always reduced first.
- In applicative normal order reduction, the leftmost, innermost redex is always reduced first.

Call by Name Semantics

- Call by name semantics is equivalent to normal order reduction, except there are no reductions inside abstractions.
- Used in ALGOL-60.

Call by Value Semantics

- Call by value semantics is equivalent to applicable order reduction, except no redex in a λ expression that is inside an abstraction is reduced.
- Used in the vast majority of programming languages today.

Table of Contents

On Wednesday, 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.