Programming Language Semantics

David A. Schmidt
Department of Computing and Information Sciences
Kansas State University

January 10, 2012

1 Introduction

A programming language possesses *syntax* and *semantics*. Syntax refers to the spelling of the language's programs, and semantics refers to the meanings of the programs. A language's syntax is formalized by a grammar or syntax chart; such formalizations are found in the back of language manuals. A language's semantics should be formalized, too, and this is the topic of this chapter.

Before we begin, we might ask, "What do we gain by formalizing the semantics of a programming language?" Consider the related question, "What was gained when language syntax was formalized with BNF?"

- A language's syntax definition standardizes the official syntax. This is crucial to users, who require a guide to writing syntactically correct programs, and to implementors, who must write a correct parser for the language's compiler.
- The syntax definition permits a formal analysis of its properties, such as whether the definition is *LL(k)*, *LR(k)*, or ambiguous.
- The syntax definition can be used as input to a compiler front-end generating tool, such as YACC; it becomes, in effect, the implementation.

We derive similar benefits from a formal semantics definition:

- The semantics definition standardizes the official semantics of the language. This is crucial to users, who require a guide to understanding the programs that they write, and to implementors, who must write a correct code generator for the language's compiler.
- The semantics definition permits a formal analysis of its properties, such as whether the definition is strongly typed, block structured, uses single-threaded data structures, is parallelizable, etc.
- The semantics definition can be used as input to a compiler back-end generating tool [26, 31]; it becomes, in effect, the implementation.

Programming-language syntax was studied intensively in the 1960's and 1970's, and programming language semantics is undergoing similar intensive study. Unlike the acceptance of BNF as the standard for syntax definition, it is unlikely that a single definition method will take hold for semantics—semantics is harder to formalize than syntax, and it has a wider variety of applications.

Semantics-definition methods fall roughly into three groups:

- operational: the meaning of a well-formed program is the trace of computation steps that results from processing the program's input. Operational semantics is also called *intensional* semantics, because the sequence of internal computation steps (the "intension") is most important. For example, two differently coded programs that both compute factorial have different operational semantics.

- denotational: the meaning of a well-formed program is a mathematical function from input data to output data. The steps taken to calculate the output are unimportant; it is the relation of input to output that matters. Denotational semantics is also called *extensional* semantics, because only the "extension"—the visible relation between input and output—matters. Thus, two differently coded versions of factorial have the same denotational semantics.

- axiomatic: a meaning of a well-formed program is a logical proposition (a "specification") that states some property about the input and output. For example, the proposition $\forall x.x \geq 0 \supset \exists y.y = x!$ is an axiomatic semantics of a factorial program.

2 Underlying Principles of Semantics Methods

We survey the three semantic methods by applying each in turn to the world's oldest and simplest programming language, arithmetic. The syntax of our arithmetic language is

$$\mathrm{E} \ ::= \ \mathrm{N} \ \mid \ \mathrm{E}_1 + \mathrm{E}_2$$

where N stands for the set of numerals $\{0, 1, 2, ...\}$. Although this language has no notion of input and output, it is computational.

2.1 Operational Semantics

There are several versions of operational semantics for arithmetic. The one that you learned as a child is called a *term rewriting system* [6, 25]. It uses rule schemes generate computation steps. There is just one rule scheme for arithmetic:

$$\mathrm{N}_1 + \mathrm{N}_2 \ \Rightarrow \ \mathrm{N}' \text{ where } \mathrm{N}' \text{ is the sum of the numerals } \mathrm{N}_1 \text{ and } \mathrm{N}_2$$

The rule scheme states that adding two numerals is a computation step, e.g., $1 + 2 \ \Rightarrow \ 3$ is one computation step. An operational semantics of a program is a sequence of such computation steps. For example, an operational semantics of $(1 + 2) + (4 + 5)$ goes as follows:

$$(1 + 2) + (4 + 5) \ \Rightarrow \ 3 + (4 + 5) \ \Rightarrow \ 3 + 9 \ \Rightarrow \ 12$$

Three computation steps led to the answer, 12. An intermediate expression like $3 + (4 + 5)$ is a "state," so this operational semantics traces the states of the computation.

Another semantics for the example is $(1 + 2) + (4 + 5) \ \Rightarrow \ (1 + 2) + 9 \ \Rightarrow \ 3 + 9 \ \Rightarrow \ 12$. The outcome is the same, and a set of rules that has this property is called *confluent* [25].

A *structural operational semantics* is a term-rewriting system plus a set of inference rules that state precisely the context in which a computation step can be undertaken. (A structural operational semantics is sometimes called a "small-step semantics," because each computation step is a small step towards the final answer.) Say that we demand left-to-right computation of arithmetic expressions. This is encoded as follows:

$$\mathrm{N}_1 + \mathrm{N}_2 \ \Rightarrow \ \mathrm{N}' \text{ where } \mathrm{N}' \text{ is the sum of } \mathrm{N}_1 \text{ and } \mathrm{N}_2$$

$$\frac{E_1 \Rightarrow E_1'}{E_1 + E_2 \Rightarrow E_1' + E_2} \qquad \frac{E_2 \Rightarrow E_2'}{N + E_2 \Rightarrow N + E_2'}$$

The first rule goes as before; the second rule states, if the left operand of an addition expression can be rewritten, then do this. The third rule is the crucial one: if the right operand of an addition expression can be rewritten *and* the left operand is already a numeral (completely evaluated), then rewrite the right operand. Working together, the three rules force left-to-right evaluation.

Each computation step must be deduced by the rules. For $(1 + 2) + (4 + 5)$, we deduce this initial computation step:

$$\frac{1 + 2 \Rightarrow 3}{(1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5)}$$

Thus, the first step is $(1+2)+(4+5) \Rightarrow 3+(4+5)$; note that we *cannot* deduce that $(1+2)+(4+5) \Rightarrow (1 + 2) + 9$. The next computation step is justified by this deduction:

$$\frac{4 + 5 \Rightarrow 9}{3 + (4 + 5) \Rightarrow 3 + 9}$$

The last deduction is simply $3 + 9 \Rightarrow 12$, and we are finished. The example shows why the semantics is "structural": each computation step is explicitly embedded into the structure of the overall program.

Operational semantics is often used to expose implementation concepts, like instruction counters, storage vectors, and stacks. For example, say our semantics of arithmetic must show how a stack holds intermediate results. We use a *state* of form $\langle s, c\rangle$, where s is the stack and c is the arithmetic expression to evaluate. A stack containing n items is written $v_1 :: v_2 :: ... :: v_n :: nil$, where v_1 is the topmost item and nil marks the bottom of the stack. The c component will be written as a stack as well. The initial state for an arithmetic expression, p, is written $\langle nil,\ p :: nil\rangle$, and computation proceeds until the state appears as $\langle v :: nil,\ nil\rangle$; we say that the result is v.

The semantics uses three rewriting rules:

$$\langle s,\ \mathrm{N} :: c\rangle \Rightarrow \langle \mathrm{N} :: s,\ c\rangle$$
$$\langle s,\ \mathrm{E}_1 + \mathrm{E}_2 :: c\rangle \Rightarrow \langle s,\ \mathrm{E}_1 :: \mathrm{E}_2 :: add :: c\rangle$$
$$\langle \mathrm{N}_2 :: \mathrm{N}_1 :: s,\ add :: c\rangle \Rightarrow \langle \mathrm{N}' :: s,\ c\rangle \text{ where } \mathrm{N}' \text{ is the sum of } \mathrm{N}_1 \text{ and } \mathrm{N}_2$$

The first rule says that a numeral is evaluated by pushing it on the top of the stack. The second rule decomposes an addition into its operands and operator, which must be computed individually. The third rule removes the top two items from the stack and adds them. Here is the previous example, repeated:

$$\begin{aligned}
&\langle nil,\ (1 + 2) + (4 + 5) :: nil\rangle \\
&\Rightarrow \langle nil,\ 1 + 2 :: 4 + 5 :: add :: nil\rangle \\
&\Rightarrow \langle nil,\ 1 :: 2 :: add :: 4 + 5 :: add :: nil\rangle \\
&\Rightarrow \langle 1 :: nil,\ 2 :: add :: 4 + 5 :: add :: nil\rangle \\
&\Rightarrow \langle 2 :: 1 :: nil,\ add :: 4 + 5 :: add :: nil\rangle \\
&\Rightarrow \langle 3 :: nil,\ 4 + 5 :: add :: nil\rangle \Rightarrow\ ...\ \Rightarrow \langle 12 :: nil,\ nil\rangle
\end{aligned}$$

This form of operational semantics is sometimes called a *state-transition semantics*, because each rewriting rule operates upon the entire state. A state-transition semantics has no need for structural operational-semantics rules.

The formulations presented here are typical of operational semantics. When one wishes to prove properties of an operational-semantics definition, the standard proof technique is *induction on the*

```
def eval(t, stack):
    """eval computes the meaning of  t  using  stack.
       parameters:  t - an arithmetic term, parsed into a nested list of form:
                      term ::=  N  |  ["+", term, term],  where N is a string of digits
          stack - a list of integers
       returns:  stack  with the integer meaning of  t  pushed on its top.
    """
    print "Evaluate", t, "with stack =", stack
    if isinstance(t, str) and t.isdigit():  # is  t  a string of digits?
        newstack = [int(t)] + stack  # push the int onto the stack
    else:  # t is a list, ["+", t1, t2]
        stack1 = eval(t[1], stack)
        stack2 = eval(t[2], stack1)
        answer = stack2[1] + stack2[0]  # add top two stack values
        newstack = [answer] + stack2[2:]   # push answer onto popped stack
    print "Evaluated", t, " Updated stack =", newstack
    return newstack
```

Figure 1: Definitional interpreter in Python for stack-semantics of arithmetic

length of the computation. That is, to prove that a property, P, holds for an operational semantics, one must show that P holds for all possible computation sequences that can be generated from the rewriting rules. For an arbitrary computation sequence, it suffices to show that P holds no matter how long the computation runs. Therefore, one shows *(i)* P holds after zero computation steps, that is, at the outset, and *(ii)* if P holds after n computation steps, it holds after $n + 1$ steps. See Nielson and Nielson [34] for examples.

Operational semantics lies close to implementation, and early forms of operational semantics were *definitional interpreters* — implementations that generated the state-transition steps [13, 44]. Figure 1 displays a Python-coded definitional interpreter that generates the computation steps of the stack-based arithmetic semantics, modelling the stack with a list argument and modelling subexpression evaluation with recursive-function calls. (As an exercise, you should prove the interpreter prints a sequence of stacks that matches the one computed by the state-transition rules.) A definitional interpreter is a powerful, practical tool for language definition and prototyping.

2.2 Denotational Semantics

Operational semantics emphasizes internal state transitions. For the arithmetic language, we were distracted by questions about order of evaluation of subphrases, even though this issue is not at all important to arithmetic. Further, the key property that the meaning of an expression is built from the meanings of its subexpressions was obscured.

We use denotational semantics to establish that a program has an underlying *mathematical meaning* that is independent of the computation strategy used to compute it. In the case of arithmetic, an expression like $(1 + 2) + (4 + 5)$ has the meaning, 12. The implementation that computes the 12 is a separate issue, perhaps addressed by an operational semantics.

The assignment of meaning to programs is performed *compositionally*: the meaning of a phrase is built from the meanings of its subphrases. We now see this in the denotational semantics of the arithmetic language. First, we assert that meanings of arithmetic expressions must be taken from the *domain* ("set") of natural numbers, $Nat = \{0, 1, 2, ...\}$, and there is a binary, mathematical function, $plus : Nat \times Nat \rightarrow Nat$, which maps a pair of natural numbers to their sum.

The denotational semantics definition of arithmetic is simple and elegant:

$$\begin{aligned}&\mathcal{E} : Expression \ \rightarrow \ Nat\\&\quad \mathcal{E}[\![\mathrm{N}]\!] = \mathrm{N}\\&\quad \mathcal{E}[\![E_1 + E_2]\!] = plus(\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!])\end{aligned}$$

The first line states that $\mathcal{E}$ is the name of the function that maps arithmetic expressions to their meanings. Since there are two BNF constructions for expressions, $\mathcal{E}$ is completely defined by the two equational clauses. (This is a Tarksi-style interpretation, as used in symbolic logic to give meaning to logical propositions [49].) The interesting clause is the one for $\mathrm{E}_1 + \mathrm{E}_2$; it says that the meanings of E_1 and E_2 are combined compositionally by *plus*. Here is the denotational semantics of our example program:

$$\begin{aligned}&\mathcal{E}[\![(1+2)+(4+5)]\!] = plus(\mathcal{E}[\![1+2]\!], \mathcal{E}[\![4+5]\!])\\&= plus(plus(\mathcal{E}[\![1]\!], \mathcal{E}[\![2]\!]), plus(\mathcal{E}[\![4]\!], \mathcal{E}[\![5]\!]))\\&= plus(plus(1,2), plus(4,5)) = plus(3,9) = 12\end{aligned}$$

Read the above as follows: the meaning of $(1+2)+(4+5)$ equals the meanings of $1+2$ and $4+5$ added together. Since the meaning of $1+2$ is 3, and the meaning of $4+5$ is 9, the meaning of the overall expression is 12. This reading says nothing about order of evaluation or run-time data structures—it states only mathematical meaning.

Here is an alternative way of understanding the semantics; write a set of simultaneous equations based on the denotational definition:

$$\begin{array}{rcl}\mathcal{E}[\![(1+2)+(4+5)]\!] &=& plus(\mathcal{E}[\![1+2]\!], \mathcal{E}[\![4+5]\!])\\ \mathcal{E}[\![1+2]\!] &=& plus(\mathcal{E}[\![1]\!], \mathcal{E}[\![2]\!])\\ \mathcal{E}[\![4+5]\!] &=& plus(\mathcal{E}[\![4]\!], \mathcal{E}[\![5]\!])\\ \mathcal{E}[\![1]\!] = 1 && \mathcal{E}[\![2]\!] = 2\\ \mathcal{E}[\![4]\!] = 4 && \mathcal{E}[\![5]\!] = 5\end{array}$$

Now, solve the equation set to discover that $\mathcal{E}[\![(1+2)+(4+5)]\!]$ is 12.

Since denotational semantics states the meaning of a phrase in terms of the meanings of its subphrases, its associated proof technique is structural induction. That is, to prove that a property, P, holds for all programs in the language, one must show that the meaning of each construction in the language has property P. Therefore, one must show that each equational clause in the semantic definition produces a meaning with property P. In the case that a clause refers to subphrases (e.g., $\mathcal{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!]$), one may assume that the meanings of the subphrases have property P. Again, see Nielson and Nielson [34] for examples.

2.3 Natural Semantics

A semantics method has been proposed that is halfway between operational semantics and denotational semantics; it is called *natural semantics*. Like structural operational semantics, natural semantics shows the context in which a computation step occurs, and like denotational semantics, natural semantics emphasizes that the computation of a phrase is built from the computations of its subphrases.

A natural semantics is a set of inference rules, and a complete computation in natural semantics is a single, large derivation. The natural semantics rules for the arithmetic language are:

$$\mathrm{N} \ \Rightarrow \ \mathrm{N}$$

$$\frac{\mathrm{E}_1 \Rightarrow n_1 \quad \mathrm{E}_2 \Rightarrow n_2}{\mathrm{E}_1 + \mathrm{E}_2 \Rightarrow m} \text{ where } m \text{ is the sum of } n_1 \text{ and } n_2$$

Read a configuration of the form $\mathrm{E} \Rightarrow n$ as "E evaluates to n." The rules resemble a denotational semantics written in inference rule form; this is no accident—natural semantics can be viewed as a denotational-semantics variant where the internal calculations of meaning are made explicit and the domains are left implicit. The internal calculations are seen in the natural semantics of our example expression:

$$\frac{\dfrac{1 \Rightarrow 1 \quad 2 \Rightarrow 2}{(1+2) \Rightarrow 3} \qquad \dfrac{4 \Rightarrow 4 \quad 5 \Rightarrow 5}{(4+5) \Rightarrow 9}}{(1+2)+(4+5) \Rightarrow 12}$$

Unlike denotational semantics, natural semantics does not claim that the meaning of a program is necessarily "mathematical." And unlike structural operational semantics, where a configuration $e \Rightarrow e'$ says that e transits to an intermediate state, e', in natural semantics $e \Rightarrow v$ asserts that the final answer for e is v. For this reason, a natural semantics is sometimes called a "big-step semantics." An interesting drawback of natural semantics is that semantics derivations can be drawn only for terminating programs.

The usual proof technique for proving properties of a natural semantics definition is induction on the height of the derivation trees that are generated from the semantics. Once again, see Nielson and Nielson [34].

2.4 Axiomatic Semantics

An axiomatic semantics produces *properties* of programs rather than meanings. The derivation of these properties is done by an inference rule set that looks somewhat like a natural semantics.

As an example, say that we wish to prove even-odd properties of programs in arithmetic and our set of properties is simply $\{is_even, is_odd\}$. For example, $2 : is_even$ and $(2+3) : is_odd$ both hold true. We can define an axiomatic semantics to do this:

$$\mathrm{N} : is_even \text{ if } \mathrm{N}\ mod\ 2 = 0 \qquad\qquad \mathrm{N} : is_odd \text{ if } \mathrm{N}\ mod\ 2 = 1$$

$$\frac{\mathrm{E}_1 : p_1 \quad \mathrm{E}_2 : p_2}{\mathrm{E}_1 + \mathrm{E}_2 : p_3} \text{ where } p_3 = \begin{cases} is_even \text{ if } p_1 = p_2 \\ is_odd \text{ otherwise} \end{cases}$$

The derivation of the even-odd property of our example program is:

$$\frac{\dfrac{1 : is_odd \quad 2 : is_even}{1+2 : is_odd} \qquad \dfrac{4 : is_even \quad 5 : is_odd}{4+5 : is_odd}}{(1+2)+(4+5) : is_even}$$

In the usual case, the properties to be proved of programs are expressed in the language of predicate logic; see Section 3.7. Also, axiomatic semantics has strong ties to the *abstract interpretation* of denotational and natural semantics definitions; see Section 4.

3 Semantical Principles of Programming Languages

The semantics methods shine when they are applied to programming languages—primary features of a language are made prominent, and subtle features receive proper mention. Ambiguities and anomalies stand out like the proverbial sore thumb. In this section, we use a classic block-structured imperative language to demonstrate. Denotational semantics will be emphasized, but excerpts from the other semantics formalisms will be provided for comparison.

P ∈ Program I ∈ Identifier = alphabetic strings
D ∈ Declaration V ∈ Variable = { X, Y, Z } ⊆ Identifier
C ∈ Command N ∈ Numeral = { 0, 1, 2, ... }
E ∈ Expression

P ::= C.
D ::= proc I = C
C ::= V:=E | C_1 ; C_2 | begin D in C end | call I | while E do C od
E ::= N | $E_1 + E_2$ | E_1 not= E_2 | V

Figure 2: Language Syntax Rules

3.1 Language Syntax and Informal Semantics

The syntax of the programming language is presented in Figure 2. As stated in the figure, there are four "levels" of syntax constructions in the language, and the topmost level, Program, is the primary one. The language is a while-loop language with local, nonrecursive procedure definitions. For simplicity, variables are predeclared and there are just three of them—X, Y, and Z. A program, C., operates as follows: an input number is read and assigned to X's location. Then the body, C, of the program is evaluated, and upon completion, the storage vector holds the results. For example, this program computes n^2 for a positive input n; the result is found in Z's location:

```
begin proc INCR = Z:= Z+X; Y:= Y+1
  in  Y:= 0; Z:= 0; while Y not= X do call INCR od end.
```

It is possible to write nonsense programs in the language; an example is: `A:=0; call B.` Such programs have no meaning, and we will not attempt to give semantics to them.

3.2 Domains for Denotational Semantics

To give a denotational semantics to the sample language, we must state the sets of meanings, called *domains*, that we use. Our imperative, block-structured language has two primary domains: *(i)* the domain of storage vectors, called *Store*, and *(ii)* the domain of symbol tables, called *Environment*. There are also secondary domains of booleans and natural numbers. The primary domains and their operations are displayed in Figure 3.

The domains and operations deserve study. First, the *Store* domain states that a storage vector is a triple. (Recall that programs have exactly three variables.) The operation *lookup* extracts a value from the store, e.g., $lookup(2, \langle 1,3,5\rangle) = 3$, and *update* updates the store, e.g., $update(2, 6, \langle 1,3,5\rangle) = \langle 1,6,5\rangle$. Operation *init_store* creates a starting store. We examine *check* momentarily.

The environment domain states that a symbol table is a list of identifier-value pairs. For example, if variable X is the name of location 1, and P is the name of a procedure that is a "no-op," then the environment that holds this information would appear $(\mathtt{X},1) :: (\mathtt{P}, id) :: nil$, where $id(s) = s$. (Procedures will be discussed momentarily.) Operation *find* locates the binding for an identifier in the environment, e.g., $find(\mathtt{X},\ (\mathtt{X},1) :: (\mathtt{P}, id) :: nil) = 1$, and *bind* adds a new binding, e.g., $bind(\mathtt{Y}, 2,\ (\mathtt{X},1) :: (\mathtt{P}, id) :: nil) = (\mathtt{Y},2) :: (\mathtt{X},1) :: (\mathtt{P}, id) :: nil$. Operation *init_env* creates an environment to start the program.

In the next section, we will see that the job of a command, e.g., an assignment, is to update the store—the meaning of a command is a function that maps the current store to the updated

$$
\begin{aligned}
&\mathit{Store} = \{\langle n_1, n_2, n_3\rangle \mid n_i \in \mathit{Nat},\ i \in 1..3\}\\
&\quad \mathit{lookup} : \{1,2,3\} \times \mathit{Store} \rightarrow \mathit{Nat}\\
&\quad\quad \mathit{lookup}(i, \langle n_1, n_2, n_3\rangle) = n_i\\
&\quad \mathit{update} : \{1,2,3\} \times \mathit{Nat} \times \mathit{Store} \rightarrow \mathit{Store}\\
&\quad\quad \mathit{update}(1, n, \langle n_1, n_2, n_3\rangle) = \langle n, n_2, n_3\rangle\\
&\quad\quad \mathit{update}(2, n, \langle n_1, n_2, n_3\rangle) = \langle n_1, n, n_3\rangle\\
&\quad\quad \mathit{update}(3, n, \langle n_1, n_2, n_3\rangle) = \langle n_1, n_2, n\rangle\\
&\quad \mathit{init_store} : \mathit{Nat} \rightarrow \mathit{Store}\\
&\quad\quad \mathit{init_store}(n) = \langle n, 0, 0\rangle\\
&\quad \mathit{check} : (\mathit{Store} \rightarrow \mathit{Store}_\bot) \times \mathit{Store}_\bot \rightarrow \mathit{Store}_\bot \text{ where } \mathit{Store}_\bot = \mathit{Store} \cup \{\bot\}\\
&\quad\quad \mathit{check}(c, a) = \text{ if } (a = \bot) \text{ then } \bot \text{ else } c(a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Environment} = (\text{Identifier} \times \mathit{Denotable})^*\\
&\quad \text{where } A^* \text{ is a list of } A\text{-elements, } a_1 :: a_2 :: \ldots :: a_n :: \mathit{nil}, n \geq 0\\
&\quad \text{and } \mathit{Denotable} = \{1,2,3\} \cup (\mathit{Store} \rightarrow \mathit{Store}_\bot)\\
&\quad\quad \mathit{find} : \text{Identifier} \times \mathit{Environment} \rightarrow \mathit{Denotable}\\
&\quad\quad\quad \mathit{find}(i, \mathit{nil}) = 0\\
&\quad\quad\quad \mathit{find}(i, (i', d) :: \mathit{rest}) = \text{if } (i = i') \text{ then } d \text{ else } \mathit{find}(i, \mathit{rest})\\
&\quad\quad \mathit{bind} : \text{Identifier} \times \mathit{Denotable} \times \mathit{Environment} \rightarrow \mathit{Environment}\\
&\quad\quad\quad \mathit{bind}(i, d, e) = (i, d) :: e\\
&\quad\quad \mathit{init_env} : \mathit{Environment}\\
&\quad\quad\quad \mathit{init_env} = (\mathtt{X}, 1) :: (\mathtt{Y}, 2) :: (\mathtt{Z}, 3) :: \mathit{nil}
\end{aligned}
$$

Figure 3: Semantic Domains

one. (That's why a "no-op" command is the identity function, $id(s) = s$, where $s \in Store$.) But sometimes commands "loop," and no updated store appears. We use the symbol, $\bot$, read "bottom," to stand for a looping store, and we use $Store_\bot = Store \cup \{\bot\}$ as the possible outputs from commands. Therefore, the meaning of a command is a function of form $Store \rightarrow Store_\bot$.

It is impossible to recover from looping, so if there is a command sequence, $\mathrm{C}_1;\mathrm{C}_2$, and C_1 is looping, then C_2 cannot proceed. The *check* operation is used to watch for this situation.

Finally, here are two commonly used notations. First, functions like $id(s) = s$ are often reformatted to read $id = \lambda s.s$; in general, for $f(a) = e$, we write $f = \lambda a.e$, that is, we write the argument to the function to the right of the equals sign. This is called *lambda notation*, and stems from the *lambda calculus*, an elegant formal system for functions [4]. The notation $f = \lambda a.e$ emphasizes that *(i)* the function $\lambda a.e$ is a value in its own right, and *(ii)* the function's name is f.

Second, it is common to revise a function that takes multiple arguments, e.g., $f(a, b) = e$, so that it takes the arguments one at a time: $f = \lambda a.\lambda b.e$. So, if the arity of f was $A \times B \rightarrow C$, its new arity is $A \rightarrow (B \rightarrow C)$. This reformatting trick is named *currying*, after Haskell Curry, one of the developers of the lambda calculus.

3.3 Denotational Semantics of Programs

Figure 4 gives the denotational semantics of the programming language. Since the syntax of the language has four levels, the semantics is organized into four levels of meaning. For each level,

$$
\begin{aligned}
&\mathcal{P} : \mathit{Program} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}_\perp \\
&\quad \mathcal{P}[\![\mathrm{C}.]\!] = \lambda n.\mathcal{C}[\![\mathrm{C}]\!]\mathit{init_env}\ (\mathit{init_store}\ n) \\
&\mathcal{D} : \mathit{Declaration} \rightarrow \mathit{Environment} \rightarrow \mathit{Environment} \\
&\quad \mathcal{D}[\![\texttt{proc}\ \mathrm{I} \texttt{=} \mathrm{C}]\!] = \lambda e.\mathit{bind}(\mathrm{I}, \mathcal{C}[\![\mathrm{C}]\!]e, e) \\
&\mathcal{C} : \mathit{Command} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Store}_\perp \\
&\quad \mathcal{C}[\![\mathrm{V} \texttt{:=} \mathrm{E}]\!] = \lambda e.\lambda s.\mathit{update}(\mathit{find}(\mathrm{V}, e), \mathcal{E}[\![\mathrm{E}]\!]e\ s, s) \\
&\quad \mathcal{C}[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!] = \lambda e.\lambda s.\mathit{check}(\mathcal{C}[\![\mathrm{C}_2]\!]e,\ \mathcal{C}[\![\mathrm{C}_1]\!]e\ s) \\
&\quad \mathcal{C}[\![\texttt{begin}\ \mathrm{D}\ \texttt{in}\ \mathrm{C}\ \texttt{end}]\!] = \lambda e.\lambda s.\mathcal{C}[\![\mathrm{C}]\!](\mathcal{D}[\![\mathrm{D}]\!]e)s \\
&\quad \mathcal{C}[\![\texttt{call}\ \mathrm{I}]\!] = \lambda e.\mathit{find}(\mathrm{I}, e) \\
&\quad \mathcal{C}[\![\texttt{while}\ \mathrm{E}\ \texttt{do}\ \mathrm{C}\ \texttt{od}]\!] = \lambda e.\textstyle\bigcup_{i \geq 0} w_i \\
&\qquad\quad \text{where} \quad \begin{array}{l} w_0 = \lambda s.\perp \\ w_{i+1} = \lambda s.\text{if } \mathcal{E}[\![\mathrm{E}]\!]e\ s \text{ then } \mathit{check}(w_i, \mathcal{C}[\![\mathrm{C}]\!]e\ s) \text{ else } s \end{array} \\
&\mathcal{E} : \mathit{Expression} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{Nat} \cup \mathit{Bool}) \\
&\quad \mathcal{E}[\![\mathrm{N}]\!] = \lambda e.\lambda s.\mathrm{N} \\
&\quad \mathcal{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = \lambda e.\lambda s.\mathit{plus}(\mathcal{E}[\![\mathrm{E}_1]\!]e\ s, \mathcal{E}[\![\mathrm{E}_2]\!]e\ s) \\
&\quad \mathcal{E}[\![\mathrm{E}_1\ \texttt{not=}\ \mathrm{E}_2]\!] = \lambda e.\lambda s.\mathit{notequals}(\mathcal{E}[\![\mathrm{E}_1]\!]e\ s, \mathcal{E}[\![\mathrm{E}_2]\!]e\ s) \\
&\quad \mathcal{E}[\![\mathrm{V}]\!] = \lambda e.\lambda s.\mathit{lookup}(\mathit{find}(\mathrm{V}, e), s)
\end{aligned}
$$

Figure 4: Denotational Semantics

we define a *valuation function*, which produces the meanings of constructions at that level. For example, at the Expression level, the constructions are mapped to their meanings by $\mathcal{E}$.

What is the meaning of the expression, say, `X+5`? This would be $\mathcal{E}[\![\texttt{X+5}]\!]$, and the meaning depends on which location is named by `X` and what number is stored in that location. Therefore, the meaning is dependent on the current value of the environment and the current value of the store. So, if the current environment is $e_0 = (\texttt{P}, \lambda s.s) :: (\texttt{X}, 1) :: (\texttt{Y}, 2) :: (\texttt{Z}, 3) :: \mathit{nil}$ and the current store is $s_0 = \langle 2, 0, 0 \rangle$, then the meaning of `X+5` is 7:

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{X+5}]\!]e_0\ s_0 = \mathit{plus}(\mathcal{E}[\![\texttt{X}]\!]e_0\ s_0, \mathcal{E}[\![\texttt{5}]\!]e_0\ s_0) \\
&= \mathit{plus}(\mathit{lookup}(\mathit{find}(\texttt{X}, e_0), s_0), 5) \\
&= \mathit{plus}(\mathit{lookup}(1, s_0), 5) = \mathit{plus}(2, 5) = 7
\end{aligned}
$$

As this simple derivation shows, data structures like the symbol table and storage vector are modelled by the environment and store arguments. This pattern is used throughout the semantics definition.

As noted in the previous section, a command updates the store. Precisely stated, the valuation function for commands is: $\mathcal{C} : \mathit{Command} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Store}_\perp$. For example, for e_0 and s_0 given above, we see that

$$\mathcal{C}[\![\texttt{Z:=X+5}]\!]e_0\ s_0 = \mathit{update}(\mathit{find}(\texttt{Z}, e_0), \mathcal{E}[\![\texttt{X+5}]\!]e_0\ s_0, s_0) = \mathit{update}(3, 7, s_0) = \langle 2, 0, 7 \rangle$$

But a crucial point about the meaning of the assignment is that it is a function upon stores. That is, if we are uncertain of the current value of store, but we know that the environment for the assignment is e_0, then we can conclude

$$\mathcal{C}[\![\texttt{Z:=X+5}]\!]e_0 = \lambda s.\mathit{update}(3, \mathit{plus}(\mathit{lookup}(1, s), 5), s)$$

That is, the assignment with environment e_0 is a function that updates a store at location 3.

Next, consider this example of a command sequence:

$$\begin{aligned}&\mathcal{C}[\![\texttt{Z:=X+5; call P}]\!]e_0\, s_0 = \mathit{check}(\mathcal{C}[\![\texttt{call P}]\!]e_0, \mathcal{C}[\![\texttt{Z:=X+5}]\!]e_0\, s_0)\\ &= \mathit{check}(\mathit{find}(\texttt{P}, e_0), \langle 2,0,7\rangle) = \mathit{check}(\lambda s.s, \langle 2,0,7\rangle)\\ &= (\lambda s.s)\langle 2,0,7\rangle = \langle 2,0,7\rangle\end{aligned}$$

As noted in the earlier section, the *check* operation verifies that the first command in the sequence produces a proper output store; if so, the store is handed to the second command in the sequence. Also, we see that the meaning of `call P` is the store updating function bound to `P` in the environment.

Procedures are placed in the environment by declarations, as we see in this example: let e_1 denote $(\texttt{X}, 1) :: (\texttt{Y}, 2) :: (\texttt{Z}, 3) :: \mathit{nil}$:

$$\begin{aligned}&\mathcal{C}[\![\texttt{begin proc P = Y:=Y in Z:=X+5; call P end}]\!]e_1\, s_0\\ &= \mathcal{C}[\![\texttt{Z:=X+5; call P}]\!](\mathcal{D}[\![\texttt{proc P = Y:=Y }]\!]e_1)s_0\\ &= \mathcal{C}[\![\texttt{Z:=X+5; call P}]\!](\mathit{bind}(\texttt{P}, \mathcal{C}[\![\texttt{Y:=Y}]\!]e_1, e_1))s_0\\ &= \mathcal{C}[\![\texttt{Z:=X+5; call P}]\!](\mathit{bind}(\texttt{P}, \lambda s.\mathit{update}(2, \mathit{lookup}(2, s), s), e_1))s_0\\ &= \mathcal{C}[\![\texttt{Z:=X+5; call P}]\!]((\texttt{P}, \mathit{id}) :: e_1)s_0\\ &\quad \text{where } \mathit{id} = \lambda s.\mathit{update}(2, \mathit{lookup}(2, s), s) = \lambda s.s \qquad\qquad (*)\\ &= \mathcal{C}[\![\texttt{Z:=X+5; call P}]\!]e_0\, s_0 = \langle 2,0,7\rangle\end{aligned}$$

The equality marked by $(*)$ is significant; we can assert that the function $\lambda s.\mathit{update}(2, \mathit{lookup}(2, s), s)$ is identical to $\lambda s.s$ by appealing to the *extensionality* law of mathematics: if two functions map identical arguments to identical answers, then the functions are themselves identical. The extensionality law can be used here because in denotational semantics the meanings of program phrases are mathematical—functions. In contrast, the extensionality law cannot be used in operational semantics calculations.

Finally, we can combine our series of little examples into the semantics of a complete program:

$$\begin{aligned}&\mathcal{P}[\![\texttt{begin proc P = Y:=Y in Z:=X+5; call P end.}]\!]2\\ &= \mathcal{C}[\![\texttt{begin proc P = Y:=Y in Z:=X+5; call P end }]\!]\mathit{init_env}\ (\mathit{init_store}\ 2)\\ &= \mathcal{C}[\![\texttt{begin proc P = Y:=Y in Z:=X+5; call P end}]\!]e_1\, s_0\\ &= \langle 2,0,7\rangle\end{aligned}$$

3.4 Semantics of the While Loop

The most difficult clause in the semantics definition is the one for the while-loop. Here is some intuition: to produce an output store, the loop `while` E `do` C `od` must terminate after some finite number of iterations. To measure this behavior, let $\texttt{while}_i$ E `do` C `od` be a loop that can iterate at most i times—if the loop runs more than i iterations, it becomes exhausted, and its output is $\bot$. For example, for input store $\langle 4,0,0\rangle$, the loop $\texttt{while}_k$ `Y not= X do Y:=Y+1 od` can produce the output store $\langle 4,4,0\rangle$ only when k is greater than 4. (Otherwise, the output is $\bot$.)

It is easy to conclude that the family, $\texttt{while}_i$ E `do` C `od`, for $i \geq 0$, can be written equivalently as:

$$\begin{aligned}\texttt{while}_0\ \text{E}\ \texttt{do}\ \text{C}\ \texttt{od} &= \textit{“exhausted”}\ (\text{that is, its meaning is } \lambda s.\bot)\\ \texttt{while}_{i+1}\ \text{E}\ \texttt{do}\ \text{C}\ \texttt{od} &= \texttt{if}\ \text{E}\ \texttt{then}\ \text{C}\ ;\ \texttt{while}_i\ \text{E}\ \texttt{do}\ \text{C}\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}\end{aligned}$$

When we refer back to Figure 3, we draw these conclusions:

$$\begin{aligned}\mathcal{C}[\![\texttt{while}_0\ \text{E}\ \texttt{do}\ \text{C}\ \texttt{od}]\!]e &= w_0\\ \mathcal{C}[\![\texttt{while}_{i+1}\ \text{E}\ \texttt{do}\ \text{C}\ \texttt{od}]\!]e &= w_{i+1}\end{aligned}$$

$$e \vdash \texttt{proc}\,\mathrm{I}\,\texttt{=}\,\mathrm{C} \Rightarrow \mathit{bind}(\mathrm{I}, (e, \mathrm{C}), e) \qquad \frac{e \vdash \mathrm{D} \Rightarrow e' \quad e', s \vdash \mathrm{C} \Rightarrow s'}{e, s \vdash \texttt{begin}\,\mathrm{D}\,\texttt{in}\,\mathrm{C}\,\texttt{end} \Rightarrow s'}$$

$$\frac{l = \mathit{find}(V, e) \quad e, s \vdash \mathrm{E} \Rightarrow n}{e, s \vdash \mathrm{V}\texttt{:=}\mathrm{E} \Rightarrow \mathit{update}(l, n, s)} \qquad \frac{e, s \vdash \mathrm{C}_1 \Rightarrow s' \quad e, s' \vdash \mathrm{C}_2 \Rightarrow s''}{e, s \vdash \mathrm{C}_1\,;\,\mathrm{C}_2 \Rightarrow s''}$$

$$\frac{(e', C') = \mathit{find}(\mathrm{I}, e) \quad e', s \vdash C' \Rightarrow s'}{e, s \vdash \texttt{call}\,\mathrm{I} \Rightarrow s'} \qquad \frac{e, s \vdash \mathrm{E} \Rightarrow \mathit{false}}{e, s \vdash \texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od} \Rightarrow s}$$

$$\frac{e, s \vdash \mathrm{E} \Rightarrow \mathit{true} \quad e, s \vdash \mathrm{C} \Rightarrow s' \quad e, s' \vdash \texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od} \Rightarrow s''}{e, s \vdash \texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od} \Rightarrow s''}$$

Figure 5: Natural Semantics

Since the behavior of a while loop must be the "union" of the behaviors of the $\texttt{while}_i$-loops, we conclude that $\mathcal{C}[\![\texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od}]\!]e = \bigcup_{i \geq 0} w_i$. The semantic union operation is well defined because each w_i is a function from the set $\mathit{Store} \rightarrow \mathit{Store}_\perp$, and a function can be represented as a set of argument-answer pairs. (This is called the *graph of the function*.) So, $\bigcup_{i \geq 0} w_i$ is the union of the graphs of the w_i functions[1].

The definition of $\mathcal{C}[\![\texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od}]\!]$ is succinct, but it is awkward to use in practice. An intuitive way of defining the semantics is:

$$\begin{array}{l} \mathcal{C}[\![\texttt{while}\ \mathrm{E}\ \texttt{do}\ \mathrm{C}\ \texttt{od}]\!]e = w \\ \quad \text{where } w = \lambda s.\text{if } \mathcal{E}[\![\mathrm{E}]\!]e\ s \text{ then } \mathit{check}(w, \mathcal{C}[\![\mathrm{C}]\!]e\ s) \text{ else } s \end{array}$$

The problem here is that the definition of w is circular, and circular definitions can be malformed. Fortunately, this definition of w can be claimed to denote the function $\bigcup_{i \geq 0} w_i$ because the following equality holds:

$$\bigcup_{i \geq 0} w_i = \lambda s.\text{if } \mathcal{E}[\![\mathrm{E}]\!]e\ s \text{ then } \mathit{check}(\bigcup_{i \geq 0} w_i, \mathcal{C}[\![\mathrm{C}]\!]e\ s) \text{ else } s$$

So, $\bigcup_{i \geq 0} w_i$ is a solution—a *fixed point*—of the circular definition, and in fact it is the smallest function that makes the equality hold. Therefore, it is the *least fixed point*.

Typically, the denotational semantics of the while-loop is presented by the circular definition, and the claim is then made that the circular definition stands for the least fixed point. This is called *fixed-point semantics*. We have omitted many technical details regarding fixed-point semantics; these are available in several texts [15, 45, 47, 51].

3.5 The Natural Semantics of the Language

We can compare the denotational semantics of the imperative language with a natural semantics formulation. The semantics of several constructions appear in Figure 5.

A command configuration has the form $e, s \vdash \mathrm{C} \Rightarrow s'$, where e and s are the "inputs" to command C and s' is the "output." To understand the inference rules, read them "bottom up." For example, the rule for I:=E says, given the inputs e and s, one must first find the location, l,

[1]Several important technical details have been glossed over. First, pairs of the form $(s, \perp)$ are ignored when the union of the graphs is performed. Second, for all $i \geq 0$, the graph of w_i is a subset of the graph of w_{i+1}; this ensures the union of the graphs is a function.

$$
\begin{array}{l}
\text{let } e_0 = (\texttt{X}, 1) :: (\texttt{Y}, 2) :: (\texttt{Z}, 3) :: nil \\
\quad s_0 = \langle 2,0,0 \rangle, \quad s_1 = \langle 2,1,0 \rangle \\
\quad \mathrm{E}_0 = \texttt{Y not=1}, \quad \mathrm{C}_0 = \texttt{Y:=Y+1} \\
\quad \mathrm{C}_{00} = \texttt{while}\ \mathrm{E}_0\ \texttt{do}\ \mathrm{C}_0\ \texttt{od}
\end{array}
$$

$$
\dfrac{e_0, s_0 \vdash \mathrm{E}_0 \ \Rightarrow\ true \qquad \dfrac{2 = \mathit{find}(\texttt{Y}, e_0) \quad e_0, s_0 \vdash \texttt{Y+1} \ \Rightarrow\ 1}{e_0, s_0 \vdash \mathrm{C}_0 \ \Rightarrow\ s_1} \qquad \dfrac{e_0, s_1 \vdash \mathrm{E}_0 \ \Rightarrow\ \mathit{false}}{e_0, s_1 \vdash \mathrm{C}_{00} \ \Rightarrow\ s_1}}{e_0, s_0 \vdash \mathrm{C}_{00} \ \Rightarrow\ s_1}
$$

Figure 6: Natural Semantics Derivation

bound to I and then calculate the output, n, for E. Finally, l and n are used to update s, producing the output.

The rules are denotational-like, but differences arise in several key constructions. First, the semantics of a procedure declaration binds I not to a function but to an environment-command pair called a *closure*. When procedure I is called, the closure is disassembled, and its text and environment are executed. Since a natural semantics does not use function arguments, it is called a *first-order semantics*. (Denotational semantics is sometimes called a *higher-order semantics*.)

Second, the while-loop rules are circular. The second rule states, in order to derive a while-loop computation that terminates in s'', one must derive *(i)* the test, E is true, *(ii)* the body, C, outputs s', and *(iii)* using e and s', one can derive a terminating while-loop computation that outputs s''. The rule makes one feel that the while-loop is "running backwards" from its termination to its starting point, but a complete derivation, like the one shown in Figure 6, shows that the iterations of the loop can be read from the root to the leaves of the derivation tree.

One important aspect of the natural semantics definition is that derivations can be drawn only for terminating computations. A nonterminating computation is equated with no computation at all.

3.6 The Operational Semantics of the Language

A fragment of the structural operational semantics of the imperative language is presented in Figure 7. For expressions, a computation step takes the form $e \vdash \langle \mathrm{E}, s \rangle \ \Rightarrow\ \mathrm{E}'$, where e is the environment, E is the expression that is evaluated, s is the current store, and E' is E rewritten. In the case of a command, C, a step appears $e \vdash \langle \mathrm{C}, s \rangle \ \Rightarrow\ \langle \mathrm{C}', s' \rangle$, because computation on C might also update the store. If the computation step on C "uses up" the command, the step appears $e \vdash \langle \mathrm{C}, s \rangle \ \Rightarrow\ s'$.

The rules in the figure are more tedious than those for a natural semantics, because the individual computation steps must be defined, and the order in which the steps are undertaken must also be defined. This complicates the rules for command composition, for example. On the other hand, the rewriting rule for the while-loop merely decodes the loop as a conditional command.

The rules for procedure call are awkward; as with the natural semantics, a procedure, I, is represented as a closure of the form (e', C'). Since C' must execute with environment, e', which is different from the environment that exists where procedure I is called, the rewriting step for `call` I must retain *two* environments; a new construct, `use` e' `in` C', remembers that C' must use e' (and not e). A similar trick is used in `begin` D `in` C `end`.

Unlike a natural semantics definition, a computation can be written for a nonterminating program; the computation is a state sequence of countably infinite length.

$$e \vdash \langle n_1 + n_2, s\rangle \;\Rightarrow\; n_3 \text{ where } n_3 \text{ is the sum of } n_1 \text{ and } n_2$$

$$\frac{e \vdash \langle \mathrm{E}, s\rangle \;\Rightarrow\; \mathrm{E}'}{e \vdash \langle \mathrm{I}\texttt{:=}\mathrm{E}, s\rangle \;\Rightarrow\; \langle \mathrm{I}\texttt{:=}\mathrm{E}', s\rangle}$$

$$e \vdash \langle \mathrm{I}\texttt{:=}n, s\rangle \;\Rightarrow\; \mathit{update}(l, n, s) \text{ where } \mathit{find}(\mathrm{I}, e) = l$$

$$\frac{e \vdash \langle \mathrm{C}_1, s\rangle \;\Rightarrow\; \langle \mathrm{C}_1', s'\rangle}{e \vdash \langle \mathrm{C}_1\,;\,\mathrm{C}_2, s\rangle \;\Rightarrow\; \langle \mathrm{C}_1'\,;\,\mathrm{C}_2, s'\rangle} \qquad \frac{e \vdash \langle \mathrm{C}_1, s\rangle \;\Rightarrow\; s'}{e \vdash \langle \mathrm{C}_1\,;\,\mathrm{C}_2, s\rangle \;\Rightarrow\; \langle \mathrm{C}_2, s'\rangle}$$

$$e \vdash \langle \texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od}, s\rangle \;\Rightarrow\; \langle \texttt{if}\,\mathrm{E}\,\texttt{then}\,\mathrm{C}\,;\,\texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od}\,\texttt{else}\,\texttt{skip}\,\texttt{fi}, s\rangle$$

$$e \vdash \langle \texttt{call}\,\mathrm{I}, s\rangle \;\Rightarrow\; \langle \texttt{use}\,e'\,\texttt{in}\,\mathrm{C}', s\rangle \text{ where } \mathit{find}(\mathrm{I}, e) = (e', \mathrm{C}')$$

$$\frac{e' \vdash \langle \mathrm{C}, s\rangle \;\Rightarrow\; \langle \mathrm{C}', s'\rangle}{e \vdash \langle \texttt{use}\,e'\,\texttt{in}\,\mathrm{C}, s\rangle \;\Rightarrow\; \langle \texttt{use}\,e'\,\texttt{in}\,\mathrm{C}', s'\rangle} \qquad \frac{e' \vdash \langle \mathrm{C}, s\rangle \;\Rightarrow\; s'}{e \vdash \langle \texttt{use}\,e'\,\texttt{in}\,\mathrm{C}, s\rangle \;\Rightarrow\; s'}$$

$$e \vdash \texttt{proc}\,\mathrm{I}\,\texttt{=}\,\mathrm{C} \;\Rightarrow\; \mathit{bind}(\mathrm{I}, (e, \mathrm{C}), e)$$

$$\frac{e \vdash \mathrm{D} \;\Rightarrow\; e'}{e \vdash \langle \texttt{begin}\,\mathrm{D}\,\texttt{in}\,\mathrm{C}\,\texttt{end}, s\rangle \;\Rightarrow\; \langle \texttt{use}\,e'\,\texttt{in}\,\mathrm{C}, s\rangle}$$

Figure 7: Structural Operational Semantics

$$\{[\mathrm{E}/\mathrm{I}]P\}\,\mathrm{I}\texttt{:=}\mathrm{E}\,\{P\}$$

$$\frac{P \supset P' \quad \{P'\}\,\mathrm{C}\,\{Q'\} \quad Q' \supset Q}{\{P\}\,\mathrm{C}\,\{Q\}} \qquad \frac{\{P\}\,\mathrm{C}_1\,\{Q\} \quad \{Q\}\,\mathrm{C}_2\,\{R\}}{\{P\}\,\mathrm{C}_1\,;\,\mathrm{C}_2\,\{R\}}$$

$$\frac{\{P \wedge \mathrm{E}\}\,\mathrm{C}_1\,\{Q\} \quad \{P \wedge \neg\mathrm{E}\}\,\mathrm{C}_2\,\{Q\}}{\{P\}\,\texttt{if}\,\mathrm{E}\,\texttt{then}\,\mathrm{C}_1\,\texttt{else}\,\mathrm{C}_2\,\texttt{fi}\,\{Q\}} \qquad \frac{\{P \wedge \mathrm{E}\}\,\mathrm{C}\,\{P\}}{\{P\}\,\texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od}\,\{P \wedge \neg\mathrm{E}\}}$$

Figure 8: Axiomatic Semantics Inference Rules

3.7 An Axiomatic Semantics of the Language

An axiomatic semantics computes upon *properties* of stores, because programs are understood as "knowledge transformers." For example, the property $\texttt{X} = 3 \wedge \texttt{Y} > \texttt{X}$ asserts knowledge about the values of `X` and `Y` (namely, the store holds 3 at `X`'s location and a number in `Y`'s location that is even larger). The meaning of a command, C, is stated in terms of configurations of form, $\{P\}\,\mathrm{C}\,\{Q\}$, stating that, if predicate (knowledge) P holds true then C generates knowledge Q upon termination (if C does indeed terminate). For example, all of these configurations are true assertions of the command, `X:=X+1`: *(i)* $\{\texttt{X} > 2\}\,\texttt{X:=X+1}\,\{\texttt{X} > 3\}$; *(ii)* $\{\texttt{Y} > \texttt{X}\}\,\texttt{X:=X+1}\,\{\texttt{Y} > \texttt{X} - 1\}$; *(iii)* $\{\texttt{X} = 3 \wedge \texttt{Y} > \texttt{X}\}\,\texttt{X:=X+1}\,\{\texttt{X} = 4 \wedge \texttt{Y} > \texttt{X} - 1\}$.

Figure 7 displays the rules for the primary command constructions. The rule for I:=E states that an assignment transforms knowledge about E into knowledge about E's "new name," I. (To understand this, recall that $[\mathrm{E}/\mathrm{I}]P$ stands for the substitution of E for all free occurrences of I in P. For example, `[Y+1/X](X>3)` is `Y+1>3`.) We can use the rule to prove that `X:=X+1` transforms `X+1>3` into `X>3`, that is, $\{\texttt{X+1>3}\}\,\texttt{X:=X+1}\,\{\texttt{X>3}\}$. Again, I:=E transforms knowledge about E into

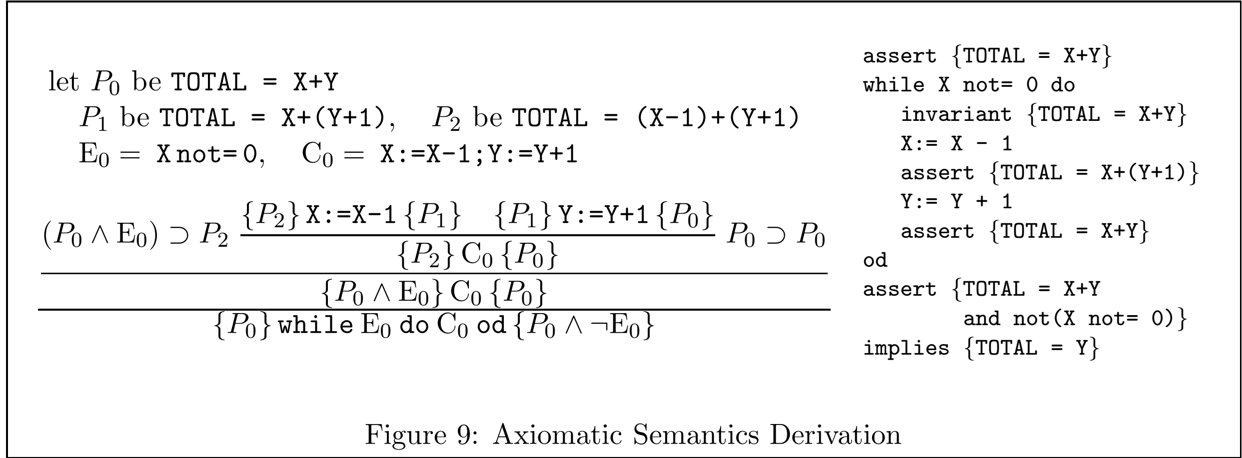

Figure 9: Axiomatic Semantics Derivation

knowledge about E's "new name," I.

The second rule in the Figure uses logical implication to deduce new knowledge; we deduce that {X>2} X:=X+1 {X>3}, because X+1>3 implies X>2. The properties of command composition are defined by the third rule: knowlege generated by command C_1 passes to command C_2.

The fourth rule, for the if-command, shows how to make a property hold upon termination regardless of which arm of the conditional is evaluated. For example, the rule proves $\{X \neq 0\}$ if $X < 0$ then $Y := -X$ else $Y := X$ fi $\{Y > 0\}$, because knowledge from the test, $X < 0$, lets us prove both $\{X < 0 \wedge X \neq 0\}\, Y := -X \,\{Y > 0\}$ and also $\{\neg(X < 0) \wedge X \neq 0\}\, Y := X \,\{Y > 0\}$.

The most fascinating rule is the last one, for while E do C od. The loop's body, C, is "repeatable code," which means whatever knowledge, I, is required to *use* C must be regenerated to *reuse* C — $\{E \wedge I\}\, C \,\{I\}$! I is called an *invariant*, because it remains true for all the repetitions of the loop. We conclude that I holds true if and when the loop terminates.

Another viewpoint is that a loop is kind of "game" played in rounds. If C lists the moves taken in one round, then the invariant, I, is the strategy one uses to "keep the lead" and eventually win. The game ends when the loop's test, E, goes false.

Here is an example: We play a pebble game, where pebbles can be moved only one at a time. There are two boxes of pebbles, named X and Y. To win the game, all the pebbles must be moved into Y's box without losing any. Here is how we might play one round of the game: `X:= X-1; Y:= Y+1`. Our strategy is encoded by the invariant, `TOTAL = X+Y`, which says that we do not lose any pebbles when we move them:

$$\{\texttt{TOTAL = X+Y}\}\ \texttt{X:= X-1; Y:= Y+1}\ \{\texttt{TOTAL = X+Y}\}$$

The completed game is this loop program, `while X not= 0 do X:= X-1; Y:= Y+1 od`, and the proof that the loop generates a victory is listed in Figure 9, which shows at the end that `TOTAL = Y`, that is, all the pebbles rest in Y's box. The derivation is difficult to read, but when reformatted vertically, as seen in the right half of the Figure, it shows clearly how the input knowledge, `TOTAL = X+Y`, is transformed into the output knowledge, `TOTAL = Y`.

The key to building the proof is determining a loop invariant. Since the goal was to prove `TOTAL = Y`, this made `TOTAL = X+Y` useful, because `X:= X-1` eventually lowers X to zero.[2] With the invariant in hand, it is easy to work backwards, from the end of the loop body, to the top, to calculate that the loop body is indeed "repeatable code": The rule for assignment proves both

[2]The loop terminates only if X's value begins as a nonnegative. The laws in Figure 8 *do not guarantee termination*. This requires additional logical machinery.

$\{\texttt{TOTAL} = \texttt{X}+(\texttt{Y}+1)\}$ `Y:=Y+1` $\{\texttt{TOTAL} = \texttt{X}+\texttt{Y}\}$, and also $\{\texttt{TOTAL} = (\texttt{X}-1)+(\texttt{Y}+1)\}$ `X:=X-1` $\{\texttt{TOTAL} = \texttt{X}+(\texttt{Y}+1)\}$. Finally, $\texttt{TOTAL} = (\texttt{X}-1)+(\texttt{Y}+1)$ implies `TOTAL = X+Y`.

There are three ways of stating the semantics of a command in an axiomatic semantics:

- relational semantics: The meaning of C is the set of P, Q pairs for which $\{P\}\,\mathrm{C}\,\{Q\}$ holds. Termination is not demanded of C. (This is called "partial correctness.")
- postcondition semantics: The meaning of C is a function from an input predicate to an output predicate. We write $slp(P, \mathrm{C}) = Q$; this means that $\{P\}\,\mathrm{C}\,\{Q\}$ holds, and for all Q' such that $\{P\}\,\mathrm{C}\,\{Q'\}$ holds, it is the case that Q implies Q'. This is also called *strongest liberal postcondition semantics*. When termination ("total correctness") is demanded also of C, the name becomes *strongest postcondition semantics*.
- precondition semantics: The meaning of C is a function from an output predicate to an input predicate. We write $wlp(\mathrm{C}, Q) = P$; this means that $\{P\}\,\mathrm{C}\,\{Q\}$ holds, and for all P' such that $\{P'\}\,\mathrm{C}\,\{Q\}$ holds, it is the case that P' implies P. This is also called *weakest liberal precondition semantics*. When termination is demanded also of C, the name becomes *weakest precondition semantics*.

4 Practical Impact

Research on programming-language semantics showed how a language can be defined precisely, its correctness properties proved, and its implementation realized. Indeed, the area of *formal methods* (cf. Chapter 116) is an adaptation of the semantics methods from computer languages to computer programs — how a program can be specified precisely, its correctness properties proved, and its specification implemented.

Operational semantics, having the longest life of the semantic approaches, is most entrenched within software development. From operational semantics came virtual machines to which languages can be translated. Definitional interpreters are now the norm for prototyping new languages, especially "little" or "domain-specific" languages, which are designed for problem solving in a limited application domain, e.g., file-linking (Make), web-page layout (HTML), spreadsheet layout (Excel), or linear algebra (Matlab). Scripting languages (see Chapter 110) are also implemented in definitional-interpreter style.

Axiomatic semantics has undergone a significant revival, not only as the starting point for specification writing and verification in formal-methods work, but as the language for description and analysis of secure and safety-critical software systems, where correctness properties, above all, must be stated so that there can validation or justification, whether by testing, monitoring, or theorem proving. Indeed, the Object Control Language (OCL) used in the Unified Modelling Language for software blueprinting is a derivative of the classic axiomatic semantics notation. (See Volume 2 of this Handbook.) A wide variety of *proof assistants* and *logical frameworks* are available for helping a designer state and validate crucial properties of software, e.g., ACL2 [24], PVS [38], Isabelle/HOL [35], and Coq [41].

The primary impact of denotational semantics has been to the design of modern programmming languages. The modelling of programming constructions as functions that operate on storage motivated many designers to add such functions as primitive constructions to the programming language itself — the results are *(i) higher-order* constructions, such as the anonymous functions in ML and Java; *(ii)* generic and *polymorphic* constructions, such as class templates in C++ and Scala; and *(iii) reflective constructions*, like that found in JavaBeans and 3-Lisp. The importance of poly-

morphic constructions to modern-day software assembly has stimulated extensions of denotational semantics into "higher-order" formats that are expressed within *type theory* [39, 36].

Within the subarea of programming-languages research, the semantics methods play a central role. Language designers use semantics definitions to formalize their creations, as was done during the development of Ada [10] and after development for Scheme [43] and Standard ML [27]. Software tools are readily available to prototype a syntax-plus-semantics definition into an implementation [31, 26, 5, 8].

A major success of formal semantics is the analysis and synthesis of data-flow analysis and type-inference algorithms from semantics definitions. This subject area, called *abstract interpretation* [3, 7, 33], uses algebra techniques to map semantic definitions from their "concrete" (execution) definition to their "abstract" (homomorphic property) definition. The technique can be used to extract properties from the definitions, generate data flow and type inference, and prove program correctness or code-improvement transformations. Most automated methods for program validation use abstract interpretation.

5 Research Issues

The techniques in this chapter have proved successful for defining, improving, and implementing sequential programming languages. But new language and software paradigms present new challenges to the semantics methods.

Challenging issues arise in the object-oriented programming paradigm. Not only can objects be arguments ("messages") to other objects' procedures ("methods"), but coercion laws based on *inheritance* allow controlled mismatches between arguments and parameters. For example, an object argument that contains methods for addition, subtraction, and multiplication is bound to a method's parameter that expects an object with just addition and subtraction methods. Carelessly defined coercions lead to unsound programs, so semantics definitions must be extended with inheritance hierarchies [16, 40]. See Chapter 107 for details.

Yet another challenging topic is defining communication as it arises in distributed programming, where multiple processes (threads of execution) synchronize through communication. Structural operational semantics has been adapted to formalize such systems. More importantly, the protocol used by a system lies at the center of the system's semantics; semantics approaches based on *process algebra* [12, 20, 28] have been developed that express a protocol as a family of simultaneous algebra equations that must be "solved" to yield a convergent solution. See Chapter 112.

A current challenge is applying semantics methods to define and explain programs that exploit the architecture of multi-core processors, where subcomputations and their storage requirements must be allocated to processor cores and their associated cache hierarchy [18]. See Chapter 34 for background.

Finally, a longstanding crucial topic is the relationship between different forms of semantic definitions: If one has, say, both a denotational semantics and an axiomatic semantics for the same programming language, in what sense do the semantics agree? Agreement is crucial, since a programmer might use the axiomatic semantics to reason about the properties of programs, whereas a compiler writer might use the denotational semantics to implement the language. This question generalizes to the problem of *refinement* of software specifications — see Chapter 116. In mathematical logic, one uses the concepts of *soundness* and *completeness* to relate a logic's proof system to its interpretation, and in semantics there are similar notions of *soundness* and *adequacy* to relate one semantics to another [15, 37]. Important as these properties are, they are quite difficult to realize in practice [1].

6 Defining Terms

Axiomatic semantics: The meaning of a program as a property or specification in logic.

Denotational semantics: The meaning of a program as a compositional definition of a mathematical function from the program's input data to its output data.

Fixed-point semantics: A denotational semantics where the meaning of a repetitive structure, like a loop or recursive procedure, is expressed as the smallest mathematical function that satisfies a recursively defined equation.

Loop invariant: In axiomatic semantics, a logical property of a while-loop that holds true no matter how many iterations the loop executes.

Natural semantics: A hybrid of operational and denotational semantics that shows computation steps performed in a compositional manner. Also known as a "big-step semantics."

Operational semantics: The meaning of a program as calculation of a trace of its computation steps on input data.

Strongest postcondition semantics: A variant of axiomatic semantics where a program and an input property are mapped to the strongest proposition that holds true of the program's output.

Structural operational semantics: A variant of operational semantics where computation steps are performed only within prespecified contexts. Also known as a "small-step semantics."

Weakest precondition semantics: A variant of axiomatic semantics where a program and an output property are mapped to the weakest proposition that is necessary of the program's input to make the output property hold true.

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. In *Proc. Theoretical Aspects of Computer Software, TACS94*, Lecture Notes in Computer Science 789, pages 1–15. Springer, 1994.

[2] K. Apt. Ten years of Hoare's logic: A survey–part I. *ACM Trans. Programming Languages and Systems*, 3:431–484, 1981.

[3] B. Blanchet, et al. A static analyzer for large safety-critical software. In *Proc. Programming Language Design and Implementation*, pages 196–207. ACM Press, 2003.

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam, 1984.

[5] D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proceedings of the 4th International Conference on Compiler Construction, Paderborn*, Lecture Notes in Computer Science 641, pages 95–109. Springer-Verlag, 1992.

[6] M. Clavel, et al. *All About Maude - A High-Performance Logical Framework*. Springer-Verlag, 2007.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[8] Th. Despeyroux. Executable specification of static semantics. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 215–234. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[9] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] V. Donzeau-Gouge. On the formal description of Ada. In N.D. Jones, editor, *Semantics-Directed Compiler Generation.* Lecture Notes in Computer Science 94, Springer-Verlag, 1980.

[11] G. Dromey. *Program Derivation.* Addison-Wesley, Sydney, 1989.

[12] J. Fokkink. *Introduction to Process Algebra.* Springer-Verlag, 2010.

[13] D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages, 2d ed.* MIT Press, 2001.

[14] D. Gries. *The Science of Programming.* Springer, 1981.

[15] C. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1992.

[16] C. Gunter and J. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, MA, 1994.

[17] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structured Operational Semantics.* Wiley, New York, 1991.

[18] M. Herlihy and N. Shavit. *Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[19] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[20] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[21] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[22] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1980.

[23] G. Kahn. Natural semantics. In *Proc. STACS '87*, pages 22–39. Lecture Notes in Computer Science 247, Springer, Berlin, 1987.

[24] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer, 2000.

[25] J.W. Klop. Term rewriting systems. In T. Maibaum S. Abramsky, D. Gabbay, editor, *Handbook of Logic in Computer Science*, volume 2, pages 2–117. Oxford University Press, 1992.

[26] P. Lee. *Realistic Compiler Generation.* The MIT Press, Cambridge, Massachusetts, 1989.

[27] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[28] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[29] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[30] C. Morgan. *Programming from Specifications, 2d. Ed.* Prentice Hall, 1994.

[31] P.D. Mosses. Compiler generation using denotational semantics. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 45, pages 436–441. Springer, Berlin, 1976.

[32] P.D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–632. Elsevier, 1990.

[33] S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[34] H. Riis Nielson and F. Nielson. *Semantics with Applications, a formal introduction.* Wiley Professional Computing. John Wiley and Sons, 1992.

[35] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, 2002.

[36] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford, 1990.

[37] C.H.-L. Ong. Correspondence between operational and denotational semantics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Computer Science, Vol. 4.* Oxford Univ. Press, 1995.

[38] S. Owre, et al. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV '87*, pages 411–414. Springer-Verlag, 1996.

[39] B. Pierce. Formal models/calculi of programming languages. In Allen Tucker, editor, *CRC Handbook of Computer Science and Engineering.* CRC Press, Boca Raton, FL, 1996.

[40] B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[41] B. Pierce, et al. Software Foundations. Technical Report `http://www.cis.upenn.edu/~bcpierce/sf`, University of Pennsylvania, 2011.

[42] G.D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus, Denmark, September 1981.

[43] J. Rees and W. Clinger. Revised3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21:37–79, 1986.

[44] J. Reynolds. Definitional interpreters for higher-order programming languages. *J. of Higher-Order and Symbolic Computation*, 11:363–397, 1998.

[45] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[46] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach.* Addison-Wesley, Reading, MA, 1995.

[47] J.E. Stoy. *Denotational Semantics.* MIT Press, Cambridge, MA, 1977.

[48] R.D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

[49] D. van Dalen. *Logic and Structure, 3d edition.* Springer, Berlin, 1994.

[50] D.A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

[51] G. Winskel. *Formal Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1993.

7 Further Information

The best starting point for further reading is the comparative semantics text of Nielson and Nielson [34], which thoroughly develops the topics in this chapter. See also the texts by Slonneger and Kurtz[46] and Watt[50].

Operational semantics has a long history; modern introductions are Hennessey's text [17] and Plotkin's report on structural operational semantics [42]. The principles of natural semantics are documented by Kahn [23]. A good example of the interpreter-based approach to semantics is the text by Friedman, Wand, and Haynes [13].

Mosses's paper [32] is a useful introduction to denotational semantics; textbook-length treatments include those by Schmidt [45], Stoy [47], Tennent [48], and Winskel [51]. Gunter's text [15] uses denotational-semantics-based mathematics to compare several of the semantics approaches. Pierce [39] and Mitchell [29] provide foundational supporting material.

Of the many textbooks on axiomatic semantics, one might start with books by Dromey [11] or Gries [14]; both emphasize precondition semantics, which is most effective at deriving correct code. Apt's paper [2] is an excellent description of the formal properties of relational semantics, and Dikstra's text [9] is the standard reference on precondition semantics. Hoare's landmark papers on relational semantics [19, 21] are worth reading as well. Many texts have been written on the application of axiomatic semantics to systems development; two samples are by Jones [22] and Morgan [30].

You are also urged to read the other chapters in Section 9 of this Volume.