Programming Language Semantics

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

October 24, 1995

A programming language possesses two fundamental features: syntax and semantics. Syntax refers to the appearance of the well-formed programs of the language, and semantics refers to the meanings of these programs. A language's syntax can be formalized by a grammar or syntax chart; such a formalization is found in the back of almost every language manual. A language's semantics should be formalized as well, so that it can appear in the language manual, too. This is the topic of this chapter.

It is traditional for computer scientists to calculate the semantics of a program by using a test-case input and tracing the program's execution with a state table and flow chart. This is one form of semantics, called *operational semantics*, but there are other forms of semantics that are not tied to test cases and traces; we will study several such approaches.

Before we begin, we might ask, "What do we gain by formalizing the semantics of a programming language?" Before we answer, we might consider the related question, "What was gained when language syntax was formalized?" The formalization of syntax, via BNF rules, produced these benefits:

- The syntax definition standardizes the official syntax of the language. 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. In this way, the syntax definition is also the implementation of the front end of the language's compiler.

There are similar benefits to providing a formal semantics definition of a programming language:

- 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, or single threaded.
- The semantics definition can be used as input to a compiler back-end generating tool, such as SIS or MESS [19, 22]. In this way, the semantics definition is also the implementation of the back end of the language's compiler.

Programming language syntax was studied intensively in the 1960's and 1970's, and presently programming language semantics is undergoing similar intensive study. Unlike the acceptance of BNF as a standard definition method for syntax, it appears 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 nonetheless 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.

1 A Survey of Semantics Methods

We survey the three semantic methods by applying each of them 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 data and output data, it does contain the notion of computation, so it will be a useful example for our initial case studies.

1.1 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*. A term rewriting system uses rewriting rule schemes to generate computation steps. There is just one rewriting rule scheme for arithmetic:

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

This rule scheme states that the addition of two numerals is a computation step. One use of the scheme would be to rewrite $1+2$ to 3, that is, $1+2 \Rightarrow 3$. An operational semantics of a program is the sequence of computation steps generated by the rewriting rule schemes. For example, an operational semantics of the program $(1+2)+(4+5)$ goes as follows:

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

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

Perhaps you noticed that another legal semantics for the example is $(1{+}2){+}(4{+}5) \Rightarrow (1{+}2){+}9 \Rightarrow 3+9 \Rightarrow 12$. The outcome is the same in both cases, but sometimes an operational semantics must be forced to be *deterministic*, that is, a program has exactly one operational semantics.

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[1]. Say that we desire left-to-right computation of arithmetic expressions. This is encoded as follows:

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

$$\frac{\mathrm{E}_1 \;\Rightarrow\; \mathrm{E}_1'}{\mathrm{E}_1 + \mathrm{E}_2 \;\Rightarrow\; \mathrm{E}_1' + \mathrm{E}_2} \qquad \frac{\mathrm{E}_2 \;\Rightarrow\; \mathrm{E}_2'}{\mathrm{N} + \mathrm{E}_2 \;\Rightarrow\; \mathrm{N} + \mathrm{E}_2'}$$

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

Now, each computation step must be deduced by these rules. For our example, $(1+2)+(4+5)$, we must 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 $(1{+}2){+}(4{+}5) \Rightarrow (1 + 2) + 9$. (Try.) 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": a computation step, like an addition, which affects a small part of the overall program, is explicitly embedded into the structure of the overall program.

Operational semantics can also be used to represent internal data structures, like instruction counters, storage vectors, and stacks. For example, say that our semantics of arithmetic must show that a stack is used to hold intermediate results. So, we use a state of the form $\langle s, c\rangle$, where s is the stack and c is the arithmetic expression to be executed. A stack containing n items is written $v_1 :: v_2 :: \ldots :: v_n :: nil$, 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:

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

[1] A structural operational semantics is sometimes called a "small-step semantics," because each computation step is a small step towards the final answer.

The first rule says that a numeral is evaluated by pushing it on the top of the stack. The second rule states that the addition of two expressions is decomposed into first evaluating the two expressions and then adding them. The third rule removes the top two items from the stack and adds them. Here is the previous example, repeated:

$$
\begin{array}{l}
\langle nil,\ (1+2)+(4+5) :: nil\rangle \\
\ \Rightarrow\ \langle nil,\ 1+2 :: 4+5 :: add :: nil\rangle \\
\ \Rightarrow\ \langle nil,\ 1 :: 2 :: add :: 4+5 :: add :: nil\rangle \\
\ \Rightarrow\ \langle 1 :: nil,\ 2 :: add :: 4+5 :: add :: nil\rangle \\
\ \Rightarrow\ \langle 2 :: 1 :: nil,\ add :: 4+5 :: add :: nil\rangle \\
\ \Rightarrow\ \langle 3 :: nil,\ 4+5 :: add :: nil\rangle\ \Rightarrow\ \ldots\ \Rightarrow\ \langle 12 :: nil,\ nil\rangle
\end{array}
$$

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

The three example semantics shown above are typical of operational semantics. When one wishes to prove properties of an operational semantics definition, the standard proof technique is *induction on the 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 [27] for examples.

1.2 Denotational Semantics

A drawback of operational semantics is the emphasis it places upon state sequences. For the arithmetic language, we were distracted by questions regarding order of evaluation of subphrases, even though this issue is not central to arithmetic. Further, a key aspect of arithmetic, the property that the meaning of an expression is built from the meanings of its subexpressions, was obscured by the operational semantics.

Denotational semantics handles these issues by emphasizing that a program has an underlying mathematical meaning that is independent of whatever computation strategy is taken to uncover it. In the case of arithmetic, an expression like $(1+2)+(4+5)$ has the meaning, 12, and we need not worry about the internal computation steps that were taken to discover this.

The assignment of meaning to programs is performed in a *compositional* manner: the meaning of a phrase is built from the meanings of its subphrases. We can see this in the denotational semantics of the arithmetic language: first, we note that meanings of arithmetic expressions are natural numbers, $Nat = \{0, 1, 2, ...\}$, and we note that there is a binary 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:

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

The first line merely states that $\mathcal{E}$ is the name of the function that maps arithmetic expressions to their meanings. Since there are just two BNF constructions for expressions, $\mathcal{E}$ is completely defined by the two equational clauses. 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:

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

One might 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 emphasizes underlying 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., $\mathcal{E}[\![E_1 + E_2]\!]$), one may assume that the meanings of the subphrases have property P. Again, see Nielson and Nielson [27] for examples.

1.3 Natural Semantics

Recently, 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:

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

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 + E_2 \Rightarrow m} \text{ where } m \text{ is the sum of } n_1 \text{ and } n_2$$

Read a configuration of the form $E \Rightarrow n$ 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. These 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 [27].

1.4 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 calculate even-odd properties of programs in arithmetic and our set of properties is simply $\{is\_even, is\_odd\}$. 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:

$$\frac{\dfrac{1 : is\_odd \quad 2 : is\_even}{1+2 : is\_odd} \quad \dfrac{4 : is\_even \quad 5 : is\_odd}{4+5 : is\_odd}}{(1+2)+(4+5) : is\_even}$$

In the usual case, the properties to be proved of programs are expressed in the language of predicate logic; see Section 2.8. Also, axiomatic semantics has strong ties to the *abstract interpretation* of denotational and natural semantics definitions [1, 6].

2 Semantics of Programming Languages

The semantics methods shine when they are applied to a realistic programming language—the primary features of the programming language are proclaimed loudly, and subtle features receive proper mention. Ambiguities and anomalies stand out like the proverbial sore thumb. In this section, we give the semantics of a block-structured imperative language. Emphasis will be placed upon the denotational semantics method, but excerpts from the other semantics formalisms will be provided for comparison.

2.1 Language Syntax and Informal Semantics

The syntax of the programming language is presented in Figure 1. As stated in the figure, there are four "levels" of syntax constructions in the language, and the topmost level, Program, is the primary one[2]. Basically, 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

[2]The Identifier and Numeral sets are collections of words—terminal symbols—and not phrase-level "syntax constructions" in the sense of this chapter.

$$
\begin{array}{l}
\mathrm{P} \in \text{Program} \\
\mathrm{D} \in \text{Declaration} \\
\mathrm{C} \in \text{Command} \\
\mathrm{E} \in \text{Expression} \\
\mathrm{I} \in \text{Identifier} = \text{upper-case alphabetic strings} \\
\mathrm{N} \in \text{Numeral} = \{\ 0,\ 1,\ 2,\ \ldots\ \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{P} & ::= & \mathrm{C}. \\
\mathrm{D} & ::= & \texttt{proc}\,\mathrm{I} = \mathrm{C} \\
\mathrm{C} & ::= & \mathrm{I} := \mathrm{E} \ \mid\ \mathrm{C}_1\,;\mathrm{C}_2 \ \mid\ \texttt{begin}\,\mathrm{D}\,\texttt{in}\,\mathrm{C}\,\texttt{end} \ \mid\ \texttt{call}\,\mathrm{I} \ \mid\ \texttt{while}\,\mathrm{E}\,\texttt{do}\,\mathrm{C}\,\texttt{od} \\
\mathrm{E} & ::= & \mathrm{N} \ \mid\ \mathrm{E}_1 + \mathrm{E}_2 \ \mid\ \mathrm{E}_1\,\texttt{not=}\,\mathrm{E}_2 \ \mid\ \mathrm{I}
\end{array}
$$

Figure 1: Language Syntax Rules

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. Nonsense programs are trapped by a type checker, and an elegant way of defining a type checker is by a set of typing rules for the programming language; See Chapter 140 for details [5].

2.2 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 2.

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. That is, the meaning of a command is a function that maps the current store to the updated 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, $\perp$, read

$Store = \{\langle n_1, n_2, n_3 \rangle \mid n_i \in Nat,\ i \in 1..3\}$

$$lookup : \{1,2,3\} \times Store \to Nat$$
$$lookup(i, \langle n_1, n_2, n_3 \rangle) = n_i$$
$$update : \{1,2,3\} \times Nat \times Store \to Store$$
$$update(1, n, \langle n_1, n_2, n_3 \rangle) = \langle n, n_2, n_3 \rangle$$
$$update(2, n, \langle n_1, n_2, n_3 \rangle) = \langle n_1, n, n_3 \rangle$$
$$update(3, n, \langle n_1, n_2, n_3 \rangle) = \langle n_1, n_2, n \rangle$$
$$init\_store : Nat \to Store$$
$$init\_store(n) = \langle n, 0, 0 \rangle$$
$$check : (Store \to Store_\perp) \times Store_\perp \to Store_\perp \text{ where } Store_\perp = Store \cup \{\perp\}$$
$$check(c, a) = \text{ if } (a = \perp) \text{ then } \perp \text{ else } c(a)$$

$Environment = (\text{Identifier} \times Denotable)^*$
where A^* is a list of A-elements, $a_1 :: a_2 :: ... :: a_n :: nil, n \geq 0$
and $Denotable = \{1,2,3\} \cup (Store \to Store_\perp)$

$$find : \text{Identifier} \times Environment \to Denotable$$
$$find(i, nil) = 0$$
$$find(i, (i', d) :: rest) = \text{if } (i = i') \text{ then } d \text{ else } find(i, rest)$$
$$bind : \text{Identifier} \times Denotable \times Environment \to Environment$$
$$bind(i, d, e) = (i, d) :: e$$
$$init\_env : Environment$$
$$init\_env = (\mathtt{X}, 1) :: (\mathtt{Y}, 2) :: (\mathtt{Z}, 3) :: nil$$

Figure 2: Semantic Domains

"bottom," to stand for a looping store, and we use $Store_\perp$ to stand for the set of possible outputs of commands. Therefore, the meaning of a command is a function of the form $Store \to Store_\perp$.

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 in the next subsection 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 [29]. 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 $A \times B \to C$, its new arity is $A \to (B \to C)$. This reformatting trick is called *currying*, after Haskell Curry, one of the developers of the lambda calculus.

2.3 Denotational Semantics of Programs

Figure 3 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, we define a *valuation function*, which produces the meanings of constructions at that level. For

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

Figure 3: Denotational Semantics

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 $\mathcal{E}[\![\mathtt{X+5}]\!]$, 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 $e_0 = (\mathtt{P}, \lambda s.s) :: (\mathtt{X}, 1) :: (\mathtt{Y}, 2) :: (\mathtt{Z}, 3) :: nil$ and the current store is $s_0 = \langle 2, 0, 0 \rangle$, then the meaning of `X+5` is 7:

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

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: $\mathcal{C} : Command \rightarrow Environment \rightarrow Store \rightarrow Store_\perp$. For example, for e_0 and s_0 given above, we see that

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

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

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

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 $(\texttt{X}, 1) :: (\texttt{Y}, 2) :: (\texttt{Z}, 3) :: \mathit{nil}$:

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

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:

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

2.4 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 $\perp$. 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 $\perp$.)

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

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

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

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

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 $Store \to Store_{\perp}$, 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[3].

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:

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

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:

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

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 [12, 32, 36, 39].

2.5 Action Semantics

One disadvantage of denotational semantics is its dependence on functions to describe all forms of computation. As a result, the denotational semantics of a large language is often too dense to read and too low level to modify. *Action semantics* is an easy-to-read denotational-semantics variant that rectifies these problems by using a family of standard operators to describe standard forms of computation in standard languages [24].

In action semantics, the standard domains are called *facets* and are predefined for expressions (the *functional facet*), for declarations (the *declarative facet*), and for commands (the *imperative facet*). Each facet includes a set of standard operators for consuming values of the facet and producing new ones. The operators are connected together by combinators ("pipes"), and the resulting action semantics definition resembles a data flow program. For example, the semantics of assignment reads as follows:

$$\mathsf{execute}[\![\mathrm{I} := \mathrm{E}]\!] = (\mathsf{find}\ \mathrm{I}\ \mathsf{and}\ \mathsf{evaluate}[\![\mathrm{E}]\!])\ \mathsf{then}\ \mathsf{update}$$

One can naively read the semantics as an English sentence, but each word is an operator or a combinator: execute is $\mathcal{C}$; evaluate is $\mathcal{E}$; find is a declarative facet operator; update is an imperative facet operator; and and and then are combinators. The equation accepts as its inputs a declarative facet argument (that is, an environment) and an imperative facet argument (that is, a store) and pipes them to the operators. So, find consumes its declarative argument and produces a functional-facet answer, and independently, evaluate$[\![\mathrm{E}]\!]$ consumes declarative and imperative arguments and produces a functional answer. The and combinator pairs these, and the then combinator transmits

[3]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.

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

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

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

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

Figure 4: Natural Semantics

the pair to the **update** operator, which uses the pair and the imperative-facet argument to generate a new imperative result.

The important aspects of an action semantics definition are *(i)* standard arguments, like environments and stores, are implicit; *(ii)* standard operators are used for standard computation steps (e.g., **find** and **update**); and *(iii)* combinators connect operators together seamlessly and pass values implicitly. Lack of space prevents a closer examination of action semantics, but see Watt [38] for an introduction.

2.6 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 4.

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, 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 5, 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.

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

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

Figure 5: Natural Semantics Derivation

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

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

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

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

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

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

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

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

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

Figure 6: Structural Operational Semantics

2.7 The Operational Semantics of the Language

A fragment of the structural operational semantics of the imperative language is presented in Figure 6.

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

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

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

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

Figure 7: Axiomatic Semantics

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.

2.8 An Axiomatic Semantics of the Language

An axiomatic semantics uses properties of stores, rather than stores themselves. For example, we might write the predicate $\texttt{X} = 3 \wedge \texttt{Y} > 0$ to assert that the current value of the store contains 3 in `X`'s location and a positive number in `Y`'s location. We write a configuration $\{P\}\mathrm{C}\{Q\}$, to assert that, if predicate P holds true prior to evaluation of command C, then predicate Q holds upon termination of C (if C does indeed terminate). For example, we can write $\{\texttt{X} = 3 \wedge \texttt{Y} > 0\}\texttt{Y:=X+Y}\{\texttt{X} = 3 \wedge \texttt{Y} > 3\}$, and indeed this holds true.

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.
- 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 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.*

It is traditional to study relational semantics first, so we focus upon it here.

If the intended behavior of a program, C, is written as a pair of predicates, P, Q, a relational semantics can be used to verify that $\{P\}\mathrm{C}\{Q\}$ holds. For example, we might wish to show that an integer division subroutine, `DIV`, that takes inputs `NUM` and `DEN` and produces outputs `QUO` and `REM`, has this behavior:

$$\{\neg(\texttt{DEN} = 0)\}\texttt{DIV}\{\texttt{QUO} \times \texttt{DEN} + \texttt{REM} = \texttt{NUM}\}$$

A proof of the above claim is a derivation built with the rules in Figure 7.

$$\begin{array}{l}
\text{let } P_0 \text{ be } X = Y + Z \\
\quad P_1 \text{ be } X = Y + (Z+1), \quad P_2 \text{ be } X = (Y-1) + (Z+1) \\
\quad \mathrm{E}_0 = \texttt{Y not=0}, \quad \mathrm{C}_0 = \texttt{Y:=Y-1;Z:=Z+1}
\end{array}$$

$$\dfrac{\dfrac{(P_0 \wedge \mathrm{E}_0) \supset P_2 \quad \dfrac{\{P_2\}\texttt{Y:=Y-1}\{P_1\} \quad \{P_1\}\texttt{Z:=Z+1}\{P_0\}}{\{P_2\}\mathrm{C}_0\{P_0\}} \quad P_0 \supset P_0}{\{P_0 \wedge \mathrm{E}_0\}\mathrm{C}_0\{P_0\}}}{\{P_0\}\texttt{while}\ \mathrm{E}_0\ \texttt{do}\ \mathrm{C}_0\ \texttt{od}\{P_0 \wedge \neg \mathrm{E}_0\}}$$

Figure 8: Axiomatic Semantics Derivation

Figure 7 displays the rules for the primary command constructions. The rule for I := E states that a property, P, about I will hold upon completion of the assignment if $[\mathrm{E}/\mathrm{I}]P$ (that is, P restated in terms of E) holds beforehand. $[\mathrm{E}/\mathrm{I}]P$ stands for the substitution of phrase E for all free occurrences of I in P. For example, $\{\texttt{X} = 3 \wedge \texttt{X+Y} > 3\}\texttt{Y:=X+Y}\{\texttt{X} = 3 \wedge \texttt{Y} > 3\}$ holds because $[\texttt{X+Y}/\texttt{Y}](\texttt{X} = 3 \wedge \texttt{Y} > 3)$ is $\texttt{X} = 3 \wedge \texttt{X+Y} > 3$.

The second rule lets us weaken a result. For example, since $(\texttt{X} = 3 \wedge \texttt{Y} > 0) \supset (\texttt{X} = 3 \wedge \texttt{X+Y} > 3)$ holds, we deduce that $\{\texttt{X} = 3 \wedge \texttt{Y} > 0\}\texttt{Y:=X+Y}\{\texttt{X} = 3 \wedge \texttt{Y} > 3\}$ holds.

The properties of command composition are defined in the expected way, by the third rule. The fourth rule, for the if-command, makes a property, Q, hold upon termination if Q holds regardless of which arm of the conditional is evaluated. Note that each arm of the conditional uses information about the result of the conditional's test.

The most fascinating rule is the last one, for the while-loop. If we can show that a property, P, is preserved by the body of the loop, then we can assert that no matter how long the loop iterates, P must hold upon termination. P is called the *loop invariant*. The rule is an encoding of a mathematical induction proof: to show that P holds upon completion of the loop, we must prove *(i)* the basis case: P holds upon loop entry (that is, after zero iterations), and *(ii)* the induction case: if P holds after i iterations, then P holds after $i+1$ iterations as well. Therefore, if the loop terminates after some number, k, of iterations, the induction proof ensures that P holds.

Here is an example that shows the rules in action. We wish to verify that

$$\{\texttt{X} = \texttt{Y} \wedge \texttt{Z} = 0\}\ \texttt{while Y not=0 do Y:=Y-1; Z:=Z+1 od}\ \{\texttt{X} = \texttt{Z}\}$$

holds true. The key to the proof is determining a loop invariant; here, a useful invariant is $\texttt{X} = \texttt{Y} + \texttt{Z}$, because $\texttt{X} = \texttt{Y} + \texttt{Z} \wedge \neg(\texttt{Y not=0})$ implies $\texttt{X} = \texttt{Z}$. This leaves us $\{\texttt{X} = \texttt{Y} + \texttt{Z} \wedge \texttt{Y not=0}\}\ \texttt{Y:=Y-1; Z:=Z+1}\{\texttt{X} = \texttt{Y} + \texttt{Z}\}$ to prove. We work backwards: the rule for assignment gives us: $\{\texttt{X} = \texttt{Y} + (\texttt{Z} + 1)\}\texttt{Z:=Z+1}\{\texttt{X} = \texttt{Y} + \texttt{Z}\}$, and we can also deduce that $\{\texttt{X} = (\texttt{Y} - 1) + (\texttt{Z} + 1)\}\texttt{Y:=Y-1}\{\texttt{X} = \texttt{Y} + (\texttt{Z} + 1)\}$ holds. Since $\texttt{X} = \texttt{Y} + \texttt{Z} \wedge \texttt{Y not=0}$ implies $\texttt{X} = (\texttt{Y} - 1) + (\texttt{Z} + 1)$, we can assemble a complete derivation; it is given in Figure 8.

3 Applications of Semantics

Increasingly, language designers are using semantics definitions to formalize their creations. An early example was the formalization of a large subset of Ada in denotational semantics [9]. The semantics definition was then prototyped using Mosses's SIS compiler generating system [22]. Scheme is another widely-used language which has been given a standardized denotational semantics [31].

Another notable example is the formalization of the complete Standard ML language in structural operational semantics [20].

Perhaps the most significant application of semantics definitions has been to rapid prototyping—the synthesis of an implementation for a newly defined language. Some prototyping systems are SIS [22], PSI [26], MESS [19], Actress [4], and Typol [7]. The first two process denotational semantics, the second two process action semantics, and the last handles natural semantics. SIS and Typol are interpreter generators, that is, they interpret a source program with the semantics definition, and PSI, MESS, and Actress are compiler generators, that is, compilers for the source language are synthesized.

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* [1, 6, 25], supplies precise techniques for analyzing semantics definitions, extracting properties from the definitions, applying the properties to data flow and type inference, and proving the soundness of the code-improvement transformations that result. Abstract interpretation provides the theory that allows a compiler writer to prove the correctness of compilers.

Finally, axiomatic semantics is a long-standing fundamental technique for validating the correctness of computer code. Recent emphasis on large-scale and safety-critical systems has again placed the spotlight on this technique. Current research on data type theory [5] suggests that a marriage between the techniques of data-type checking and axiomatic semantics is not far in the future.

4 Research Issues in Semantics

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

In the functional programming paradigm, a higher-order functional language can use functions as arguments to other functions. This makes the language's domains more complex than those in Figure 2. Denotational semantics can be used to understand these complexities; an applied mathematics called *domain theory* [12, 32] is used to formalize the domains with algebraic equations. For example, the *Value* domain for a higher-order, Scheme-like language takes the form

$$Value \ = \ Nat \ + \ (Value \ \rightarrow \ Value)$$

That is, legal values are numbers or functions on values. Of course, Cantor's theorem makes it impossible to find a set that satisfies this equation, but domain theory uses the concept of continuity from topology to restrict the size of *Value* so that a solution can be found like that in Section 2.4:

$$Value = lim_{i \geq 0}\, V_i, \text{ where } \begin{array}{l} V_0 = \{\bot\} \\ V_{i+1} = Nat \uplus (V_i \rightarrow^{ctn} V_i) \end{array}$$

where $V_i \rightarrow^{ctn} V_i$ denotes the topologically continuous functions on V_i.

Challenging issues also arise in the object-oriented programming paradigm. Not only can objects be parameters ("messages") to other objects' procedures ("methods"), but coercion laws based on *inheritance hierarchies* allow controlled mismatches between actual and formal parameters. Just like an integer actual parameter might be coerced to a floating-point formal parameter, we might coerce a object that contains methods for addition, subtraction, and multiplication to be an actual parameter to a method that expects a formal-parameter object with just addition and subtraction methods. Carelessly defined coercions lead to unsound programs, so denotational and natural semantics have been used to formalize domain hierarchies and safe coercions for the inheritance hierarchies [13].

Yet another challenging topic is parallelism and communication as it arises in the distributed programming paradigm. Here, multiple processes run in parallel and synchronize through communication. Structural operational semantics has been adapted to formalize systems of processes and to study the varieties of communication the processes might undertake. Indeed, new notational systems have been developed specifically for this subject area [29].

Finally, a longstanding research topic is the relationship between the different forms of semantic definitions. If one has, say, both a denotational semantics and an axiomatic semantics for a 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. 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 [12, 28].

A standard example is proving the soundness of a structural operational semantics to a denotational semantics: for program, P, and input, v, $(P, v) \Rightarrow v'$ in the operational semantics implies $\mathcal{P}[\![P]\!](v) = v'$ in the denotational semantics. Adequacy is a form of inverse: if $\mathcal{P}[\![P]\!](v) = v'$, and v' is a primitive value (e.g., an integer or boolean), then $(P, v) \Rightarrow v'$. There is a stronger form of adequacy, called *full abstraction* [35], which has proved difficult to achieve for realistic languages, although recent progress has been made [2].

5 Defining Terms

Action semantics: A variation of denotational semantics where low-level details are hidden by use of modularized sets of operators and combinators.

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 and C. Hankin, editors. *Abstract interpretation of declarative languages.* Ellis Horwood, Chichester, 1987.

[2] 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.

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

[4] 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.

[5] L. Cardelli. Type theory. In Allen Tucker, editor, *CRC Handbook of Computer Science and Engineering.* CRC Press, Boca Raton, FL, 1996.

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] C. Gunter. *Foundations of Programming Languages.* MIT Press, Cambridge, MA, 1992.

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

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

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

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

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

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

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

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

[21] C. Morgan. *Programming from specifications, 2d. Ed.* Prentice Hall, 1994.

[22] 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.

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

[24] P.D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, England, 1992.

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

[26] F. Nielson and H. Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

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

[28] 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.

[29] 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.

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

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

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

[33] D.A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, 1994.

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

[35] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman/Wiley, 1988.

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

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

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

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

6 Further Information

The best starting point for further reading is the comparative semantics text of Nielson and Nielson [27], which thoroughly develops the topics in this chapter. See also the texts by Slonneger and Kurtz[34] and Watt[38].

Operational semantics has a long history, but good, modern introductions are Hennessey's text [14] and Plotkin's report on structural operational semantics [30]. The principles of natural semantics are documented by Kahn [18].

Mosses's paper [23] is a useful introduction to denotational semantics; textbook-length treatments include those by Schmidt [32], Stoy [36], Tennent [37], and Winskel [39]. Gunter's text [12] uses denotational-semantics-based mathematics to compare several of the semantics approaches, and Schmidt's text [33] shows the recent influences of data-type theory on denotational semantics. Action semantics is surveyed by Watt [38] and defined by Mosses [24].

Of the many textbooks on axiomatic semantics, one might start with books by Dromey [10] or Gries [11]; both emphasize precondition semantics, which is most effective at deriving correct code. Apt's paper [3] is an excellent description of the formal properties of relational semantics, and Dikstra's text [8] is the standard reference on precondition semantics. Hoare's landmark papers on relational semantics [15, 16] are worth reading as well. Many texts have been written on the application of axiomatic semantics to systems development; two samples are by Jones [17] and Morgan [21].

7 Acknowledgement

Brian Howard and Anindya Banerjee provided helpful criticism.