Semantics of Programming Languages

Waldemar Horwat

Presented at MacHack '90

Abstract

This paper is an introduction to the field of semantics of programming languages, describing three kinds of semantics—operational, denotational, and axiomatic—and a most primitive programing language called the lambda calculus.

Not only is it possible to reason about programs and programming languages formally, doing so carries very important benefits of eliminating ambiguities, acquiring a greater understanding of the issues addressed by the language, and the possibility of verification. Such formal reasoning and verification, already in use in designing hardware, will become much more important in the years and decades ahead as software verification systems become usable and widely available. When that time comes, the art of computer programming will change forever.

1. Introduction

Semantics

Semantics is the study of meaning. A computer program in isolation is just a sequence of syntactic characters. The semantics or meaning of a program is generally defined to be its behavior. The semantics usually become apparent when the program is run on a computer, although this is not necessary—one can understand the semantics of a program by, say, reading the source code. For example, anyone familiar with the C language can describe the behavior of the following program:

```
#include <stdio.h>

void main()
 {
 printf("Hello, world!\n");
 }
```

There is nothing inherent in this program's text or its syntax that would lead it to print the string “Hello, world!” on the screen when run. After all, `printf` could have been defined as a function which returns the length of a string, in which case the above program would do nothing.

Nevertheless, in every valid C implementation the above program will print the string “Hello, world!” on the standard output stream (usually the screen). The fact that this is the defined action of the program follows from the *semantics* of the C language, which is informally specified in [KR].

I emphasize the word “informally” in the above paragraph. Whereas formal methods are widely used to specify the syntax of a program (e.g. BNF grammars and syntax flowcharts), much of the specification of semantics is still informal. Nevertheless, significant formal systems for specifying semantics have existed for the past decade or so, and they are gaining wider acceptance. In this paper I will present three ways of formally specifying semantics of programming languages: operational semantics, direct denotational semantics, and axiomatic semantics. I will also discuss the utility and scope of formal semantics.

Problems with Informal Semantics

Current Problems

Although informal specifications of programming languages' semantics will always be useful for teaching programming, they are not without problems. The largest problem is ambiguity. The semantics of the Common Lisp language were defined informally in an excellent book by Guy Steele [5]. Despite great efforts to eliminate them, ambiguities in Common Lisp's specification persisted, leading to a gradual divergence of implementations of the language. One of the tasks before a current standards subcommittee is to resolve the ambiguities and clean up the language.

The C language is another one for which the semantics were defined informally, and new ambiguities keep being discovered. The outcome of resolving all of the ambiguities is a large ANSI standard for the C language. Despite the size of the standard, there are no guarantees that more ambiguities will not be found in the language.

The ambiguities are not limited to programming languages. For example, if you want to delete a number of files in a Macintosh directory, do you know what the combined effect of using `PBGetCatInfo` and `PBDelete` is? Might not deleting files have adverse effects on `PBGetCatInfo`'s numbering?

Current Solutions

The usual solution to ambiguities such as the one above is to try a sample program on a real system. This is an approach frequently taken by Macintosh developers when trying to figure out what something in *Inside Macintosh* means. Unfortunately, the system designer might have had a different idea about what the semantics were, and programs relying on information obtained this way might break on future revisions of the system. Someone who tried the above `PBGetCatInfo` experiment would have noticed that the files are indexed in an alphabetical order (thus the file index should be decremented by one every time a file is deleted), but is this guaranteed to hold for future systems (or external file systems)?

Future Problems

The problems with precise definition of semantics are only going to get worse. Object-oriented programming, concurrency, and exception processing all open large Pandora's boxes of semantic problems. What happens in Object Pascal if an object is disposed while a method defined on it is still executing? In languages such as Common Lisp that permit the types of objects to be changed at run time, what happens if the type of an object is changed while it is used by some system service or while one of its methods is running? In a C implementation that supports concurrent processes, if process A stores the value `1` in a variable or a structure field and process B later reads that value, is it guaranteed to get `1`? Enforcing a strict guarantee such as this one might restrict the compiler or the run-time system too much, but some guarantee must be made. In languages supporting exception processing, what happens when an exception occurs in the exception processing code? What if the language supports both the restart and resume models, as Common Lisp does? The combinations of things that can go wrong quickly become mind-boggling, and in most cases the eventual response of system designers is to give up and rely on users not to attempt to do weird things. There are just too many rules to spell out precisely.

Formal Semantics to the Rescue

Fortunately, many of the problems from the previous section can be addressed by formally specifying the behavior of languages and libraries. A formal description is unambiguous—barring mistakes, everyone reading a formal specification should come to the same conclusions about the behavior of a system; thus, there is at least hope that a language standard is enforceable. It is not necessary that a formal specification define everything; in fact, many formal specifications define the results of certain actions as *undefined* in order to give system designers some implementation latitude.

Other Advantages

Formal semantics have other advantages as well. Writing the semantics of a program forces the designer to think clearly about the problems he is addressing. Since formal semantics are mathematically precise, developing them will often reveal subtle flaws in the design. Furthermore, as will be discussed in the conclusion, developing the semantics for an implementation brings at least the theoretical possibility of *proving* that the implementation is correct. Computer program debugging and testing will be revolutionalized forever once this theoretical possibility becomes practical.

Formal semantics are not a cure-all, though. They are hard to read for the uninitiated, and they do not solve all of the problems associated with good language design. They may also be overkill for many simple language design jobs.

Kinds of Formal Semantics

There are three main kinds of formal semantics, differing by their levels of abstraction:

- *Operational semantics* describes the meaning of a program by defining a simple interpreter and translating (compiling) the program into code that can be "executed" on the interpreter. This form of semantics can lead to a simple interpretation of the language—the interpreter can be implemented as a real program—but operational semantics are often difficult to analyze theoretically.

- *Denotational semantics* describes the meaning of a program by using a mathematical function to map a program to its meaning, which is represented as a mathematical value. The behavior of the program can be read directly from its meaning, and the meaning can be analyzed mathematically to yield further information.

- *Axiomatic semantics* describes the meaning of a program by listing its properties. Axiomatic semantics descriptions are even more theoretical than those of denotational semantics and may not fully define the meaning of a program. Nevertheless, axiomatic semantics is useful for verification.

2. Operational Semantics

Operational semantics describes the meaning of a program by, in essence, interpreting it. For complicated languages a procedure is also given to translate the program text into a simpler intermediate form, in essence compiling it into a form that is easier to interpret.

To illustrate the use of operational semantics, a small language is defined below. The syntax is as follows:

```
Stmt ::=  BEGIN Stmt* END |
          Var := Exp |
          IF Exp THEN Stmt ELSE Stmt |
          ε

Exp ::=   Const |
          Var |
          Var ++ |
          Exp + Exp |
          Exp * Exp

Var ::=   identifier
Const ::= TRUE |
          FALSE |
          integer
```

It is easy to guess what the semantics of this programming language might be, but I will not describe them informally here. Instead, I will immediately give a formal definition of the language's semantics and use them to describe the behavior of a sample program.

We can represent the *state* of an executing program written in the above language by an ordered pair $<p,\sigma>$, where $p$is a Stmt, or $[e,\sigma]$, where e is an Expr and σ is a representation of the contents of memory. Formally, σ is a function σ:Var→Const that returns the value currently stored in a given variable. The domain of σ is the set of defined variable names.

To determine the meaning of a program p, we attempt to *reduce* the initial state $<p,\sigma>$, where σ represents the initial values of p's variables. If it is an error to access a variable before it is set, we just make σ the null function with the empty domain.

We reduce a state $<p,\sigma>$ or $[e,\sigma]$ according to the rules given below, which represent our semantics of the sample programming language. Here $a \Rightarrow b$ denotes "a reduces to b." The reduction rules are transitive; *i.e.*, if $a \Rightarrow b$ and $b \Rightarrow c$, then $a \Rightarrow c$ can be inferred. Some of the reduction rules have preconditions and are listed in the form \F(*Condition*,*Reduction*). This means that *Reduction* is true if *Condition* is true. *Condition* may be another reduction, which has to be derived before *Reduction* can be used. Rules without preconditions can be used anytime.

The notation $\sigma[v\leftarrow c]$ denotes a new function σ' which is identical to σ except that $\sigma'(v)=c$. In other words, $\sigma'(w)=\sigma(w)$ when $w\neq v$ and $\sigma'(w)=c$ when $w=v$.

\F(\L(<Stmt_0,σ> ⇒ <Stmt_0',σ'>),\L(<`BEGIN` Stmt_0 Stmt_1 … Stmt_n `END`,σ> ⇒ <`BEGIN` Stmt_0' Stmt_1 … Stmt_n `END`,σ'>)) (A)

<`BEGIN` ε Stmt_1 … Stmt_n `END`,σ> ⇒ <`BEGIN` Stmt_1 … Stmt_n `END`,σ> (B)

<`BEGIN END`,σ> ⇒ <ε,σ> (C)

\F(\L([Exp,σ] ⇒ [Exp',σ']),\L(<Var := Exp,σ> ⇒ <Var := Exp',σ'>)) (D)

<Var := Const,σ> ⇒ <ε,σ[Var←Const]> (E)

\F(\L([Exp,σ] ⇒ [Exp',σ']),\L(<`IF` Exp `THEN` Stmt_1 `ELSE` Stmt_2,σ> ⇒ <`IF` Exp' `THEN` Stmt_1 `ELSE` Stmt_2,σ'>)) (F)

<`IF TRUE THEN` Stmt_1 `ELSE` Stmt_2,σ> ⇒ <Stmt_1,σ> (G)

<`IF FALSE THEN` Stmt_1 `ELSE` Stmt_2,σ> ⇒ <Stmt_2,σ> (H)

\F(σ(Var) is defined,\L([Var,σ] ⇒ [σ(Var),σ])) (I)

\F(\L(σ(Var) is defined, and it is an integer),\L([Var ++,σ] ⇒ [σ(Var),σ[Var←σ(Var)+1]])) (J)

\F(\L([Exp_1,σ] ⇒ [Exp_1',σ']),\L([Exp_1 + Exp_2,σ] ⇒ [Exp_1' + Exp_2,σ'])) (K)

\F(\L([Exp_2,σ] ⇒ [Exp_2',σ']),\L([Const_1 + Exp_2,σ] ⇒ [Const_1 + Exp_2',σ'])) (L)

$$\F(\text{Const}_1 \text{ and } \text{Const}_2 \text{ are integers},\L([\text{Const}_1 + \text{Const}_2,\sigma] \Rightarrow [\text{Const}_1+\text{Const}_2,\sigma])) \quad \text{(M)}$$

$$\F(\L([\text{Exp}_1,\sigma] \Rightarrow [\text{Exp}_1',\sigma']),\L([\text{Exp}_1 * \text{Exp}_2,\sigma] \Rightarrow [\text{Exp}_1' * \text{Exp}_2,\sigma'])) \quad \text{(N)}$$

$$\F(\L([\text{Exp}_2,\sigma] \Rightarrow [\text{Exp}_2',\sigma']),\L([\text{Const}_1 * \text{Exp}_2,\sigma] \Rightarrow [\text{Const}_1 * \text{Exp}_2',\sigma'])) \quad \text{(O)}$$

$$\F(\text{Const}_1 \text{ and } \text{Const}_2 \text{ are integers},\L([\text{Const}_1 * \text{Const}_2,\sigma] \Rightarrow [\text{Const}_1 \cdot \text{Const}_2,\sigma])) \quad \text{(P)}$$

A Derivation

As an example of the use of the above rules, let us derive the meaning of the following program p:

```
BEGIN
x:=3
IF TRUE THEN y:=x++*x
ELSE y:=17
END
```

The initial σ is the empty function {}[1]. The only rule that can be applied to reduce p is (A), hoping that we can later apply rule (B); however, to use rule (A) we must first show its precondition <x:=3,{}> ⇒ <ε,σ'>. In order to do this, we use rule (E) to obtain <x:=3,{}> ⇒ <ε,{x←3}>. Substituting this as rule (A)'s precondition, we get <p, {}> ⇒ <BEGIN ε IF TRUE THEN y:=x++*x ELSE y:=17 END,{x←3}>. Applying rule (B) we get <p,{}> ⇒ <BEGIN IF TRUE THEN y:=x++*x ELSE y:=17 END,{x←3}>.

Now we have to satisfy rule (A) again; to do this we must reduce the IF statement. Rule (G) reduces <IF TRUE THEN y:=x++*x ELSE y:=17,{x←3}> to <y:=x++*x,{x←3}>. Since x++*x is not a constant, we must apply rule (D) to reduce it to a constant, which we do as follows. We want to reduce the multiplication operator, but to do that using rule (P), both operands must be constants. Rules (N) and (O) reduce expressions to constants; note that rule (O) states that the left expression must be reduced before the right one can be. To reduce x++ and x to constants, we use rules (J) and (I) to get [x++,{x←3}] ⇒ [3,{x←4}] and [x,{x←4}] ⇒ [4,{x←4}]. Substituting using rules (N) and (O), we get [x++*x,{x←3}] ⇒ [3*4,{x←4}], and then [3*4,{x←4}] ⇒ [12,{x←4}] using rule (P). By transitivity, [x++*x,{x←3}] ⇒ [12,{x←4}], which can be substituted using rules (D) and (E) into <y:=x++*x, {x←3}> to get <y:=x++*x,{x←3}> ⇒ <ε,{x←4,y←12}>.

This reduction is then used to reduce the remainder of the BEGIN ... END statement to <BEGIN ε END, {x←4,y←12}>, then to <BEGIN END,{x←4,y←12}>, and finally to <ε,{x←4,y←12}>. Thus, we obtain <p,{}> ⇒ <ε, {x←4,y←12}>, which indicates that the program will terminate without errors and store 4 in x and 12 in y.

The entire derivation is summarized below:

(1) <x:=3,{}> ⇒ <ε,{x←3}> (E)

(2) <p,{}> ⇒ <BEGIN ε IF TRUE THEN y:=x++*x ELSE y:=17 END,{x←3}> (A) using (1)

(3) <BEGIN ε IF ... END,{x←3}> ⇒ <BEGIN IF ... END,{x←3}> (2)

(4) <p,{}> ⇒ <BEGIN IF TRUE THEN y:=x++*x ELSE y:=17 END,{x←3}>Transitivity (2) and (3)

[1]In this example, set notation is used for functions. {a←b, c←d} represents a function f such that $f(a)=b$, $f(c)=d$, and f is undefined for every other input.

(5) <IF TRUE THEN y:=x++*x ELSE y:=17,{x←3}> ⇒ <y:=x++*x,{x←3}> (G)

(6) [x++,{x←3}] ⇒ [3,{x←4}] (J)

(7) [x++*x,{x←3}] ⇒ [3*x,{x←4}] (N) using (6)

(8) [x,{x←4}] ⇒ [4,{x←4}] (I)

(9) [3*x,{x←4}] ⇒ [3*4,{x←4}] (O) using (8)

(10) [3*4,{x←4}] ⇒ [12,{x←4}] (P)

(11) [x++*x,{x←3}] ⇒ [12,{x←4}] Transitivity (7), (9), and (10)

(12) <y:=x++*x,{x←3}> ⇒ <y:=12,{x←4}> (D) using (11)

(13) <y:=12,{x←4}> ⇒ <ε,{x←4,y←12}> (E)

(14) <IF TRUE THEN y:=x++*x ELSE y:=17,{x←3}> ⇒ <ε,{x←4,y←12}> Transitivity (5), (12), (13)

(15) <BEGIN IF ... END,{x←3}> ⇒ <BEGIN ε END,{x←4,y←12}> (A) using (14)

(16) <BEGIN ε END,{x←4,y←12}> ⇒ <BEGIN END,{x←4,y←12}> (B)

(17) <BEGIN END,{x←4,y←12}> ⇒ <ε,{x←4,y←12}> (C)

(18) <*p*,{}> ⇒ <ε,{x←4,y←12}> Transitivity (4), (15), (16), and (17)

Discussion

The above reduction proceeded without a hitch. However, sometimes problems arise when performing reductions. Two possibilities are stuck states—states where no rules apply—and states where several different rules can be applied.

As an aside, note that nothing prevents one from introducing true but useless reductions while constructing a proof. For example, rule (I) could have been applied at step (8) to obtain, say, [x,{x←7}] ⇒ [7,{x←7}]. While this is valid, it would not help in reducing <*p*,{}> because transitivity could not have been applied in step (11), as [3*x,{x←7}] from step (9) would not match [3*x,{x←4}] from step (7).

Errors

A stuck state would be reached if, say, we were to try to reduce IF 7 THEN ε ELSE ε. There is no reduction rule for conditionals on non-boolean conditions. Such a stuck state indicates that the program is in error.

Ambiguity and Determinism

The above language is deterministic—no matter in what order the rules are applied, the result will always be the same. The fact that the language is deterministic can be proved by induction.

Nevertheless, the language can be made nondeterministic, by, say, changing rule (O) to:

$$\F(\L([Exp_2,\sigma] \Rightarrow [Exp_2',\sigma']),\L([Exp_1 * Exp_2,\sigma] \Rightarrow [Exp_1 * Exp_2',\sigma'])) \quad (O')$$

Now there are two orders in which the arguments to * can be reduced, and they will lead to different answers. If the right argument x is evaluated before x++, y will be assigned 9 instead of 12.

This kind of nondeterminism can be useful to give language implementations some latitude. Keep in mind that nondeterminism is qualitatively different from ambiguity as described in the first section. Nondeterminism means that a program can have one of several specific behaviors depending on the implementor's choice; ambiguity means that the language implementor and the program author may not agree on the language's semantics. The order of evaluation of expressions in C is nondeterministic, but not necessarily ambiguous; it is stated in the language specification that programs must not rely on the order of evaluation of expressions.

Summary

Operational semantics provides one way to formally describe the meaning of a programming language. Its advantages are relative simplicity and the possibility of writing an interpreter to run the program by just coding the semantic rules. As with other formal semantic systems, operational semantics provides a way to unambiguously settle most language meaning questions[2], just as BNF and other syntax notations settle most language syntax questions. All correct implementations of the language will have the same behavior, and any correct, deterministic program will yield the same results on any implementation.

3. The Lambda Calculus

The subject of the next section, denotational semantics, is not easy to understand, and to demonstrate some of the ideas involved in a simpler context, I am introducing the *lambda calculus* in this section. The lambda calculus [3] is perhaps the simplest programming language, but it has been shown to be universal—any program written in any other known programming language can be expressed as a lambda calculus program. The lambda calculus is the precursor of the modern programming language LISP.

Syntax

The syntax of a lambda calculus expression is extremely simple:

```
Exp ::=  Var |
         (Exp Exp) |
         λVar . Exp

Var ::=  identifier
```

Identifiers are used to represent symbols, also called variables. The $(\text{Exp}_1\ \text{Exp}_2)$ expression is called an application; Exp_1 is generally a function and Exp_2 its argument. The λVar . Exp expression is a function that takes one argument Var and returns the value specified by Exp.

Parentheses may be dropped if conventions to group applications left-to-right and assign λ-expressions a lower precedence than applications are adopted, so `λa.bac` ≡ `λa.((ba)c)`.

Reduction Rules

There are three rules for reducing lambda expressions, as listed below. The notation $E[a/b]$ means replacing all occurrences of b with a in E. All of the rules below are bidirectional. When several rules apply to a given lambda expression, they can be applied in any order. Two lambda expressions are equal if one can be reduced to the other.

$\lambda x.E \Leftrightarrow \lambda y.E[y/x]$, as long as y is not free in E (α)

[2]Some issues, especially dealing with input and output, will probably never be settled adequately and unambiguously.

$(\lambda x.E)\,F \Leftrightarrow E[F/x]$ (β)

$\lambda x.Ex \Leftrightarrow E$, as long as x is not free in E (η)

The technical restrictions on "free variables" are necessary to prevent inappropriate "capturing" of scoped variables. As long as all variables have different names, the restrictions are not important.

Here are some examples of reductions:

(λa.λb.ba)cd ⇒ (λb.bc)d ⇒ dc (β, β)

(λa.(λb.b)a)cd ⇒ (λb.b)cd ⇒ cd (η, β) or (β, β)

(λa.aa)(λx.x) ⇒ (λx.x)(λx.x) ⇒ (λx.x) (β, β)

λa.ab ⇔ λc.cb, but *not* λa.ab ⇔ λb.bb! (The latter would "capture" the free variable b.)(α)

(λx.xx)(λx.xx) ⇒ (λx.xx)(λx.xx) ⇒ (λx.xx)(λx.xx) ⇒ … (β, β, …)

The last expression cannot be reduced to a form where no more reductions are possible.

Universality

Although I will not prove that the lambda calculus is universal here, I will try to motivate how that might be true. While the lambda calculus does not define numbers, mutable variables, records, arrays, lists, loops, multi-parameter functions, *etc.*, it turns out that all of these capabilities can be emulated! Below I sketch one well-known way of emulating the natural numbers.

Define the macros below. The macros are purely for readability; they should be expanded before the lambda calculus reduction rules are used.

`0` ≡ λa.λb.b
`1` ≡ λa.λb.ab
`2` ≡ λa.λb.a(ab)
`3` ≡ λa.λb.a(a(ab))
`4` ≡ λa.λb.a(a(a(ab))), and so on.

`Succ` ≡ λn.λa.λb.a($n$ab) is the successor function;
`+` ≡ λn.λm.λa.λb.(na)(ma)b denotes addition.

Notice that two-argument addition was defined despite the fact that lambda calculus functions admit only one argument: + is a function that takes the first number n and returns a function that takes the second number m and returns n+m. Thus, `+3` is a *function* that adds `3` to whatever argument it gets, while `+3 2` ≡ `(+3)2` is `5`, which is verified below. This idea of using one-argument functions to emulate multiple-argument functions will be important in the next section on denotational semantics.

`+3 2` ≡ (λn.λm.λa.λb.(na)(ma)b) (λa.λb.a(a(ab))) (λa.λb.a(ab))
⇒ (λm.λa.λb.((λa.λb.a(a(ab)))a)(ma)b) (λa.λb.a(ab)) (β)
⇒ λa.λb.((λa.λb.a(a(ab)))a) ((λa.λb.a(ab))a)b (β)
⇒ λa.λb.(λb.a(a(ab))) ((λa.λb.a(ab))a)b (β)
⇒ λa.λb.(λb.a(a(ab))) (λb.a(ab))b (β)
⇒ λa.λb.(λb.a(a(ab))) (a(ab)) (β)
⇒ λa.λb.a(a(a(ab)))) (β)

$\equiv 5$

Although I will not do it here, it is not difficult to define other macros such as `*`, `Pred`, and `Zero?`. `Zero?` $n\ E\ F$ will return the value of E if n is zero and F otherwise. Using these macros it is almost possible to define factorial:

$$\texttt{Fact} = \lambda n.(\texttt{Zero?}\ n\ \texttt{1}\ (\texttt{*}n\ (\texttt{Fact}\ (\texttt{Pred}\ n)))) \qquad (1)$$

Unfortunately, this definition doesn't work, because the `Fact` macro is defined in terms of itself. Nevertheless, it is possible to define a recursive function in the lambda calculus by using a wonderful trick called the *paradoxical combinator* `Y`:

$$\texttt{Y} \equiv \lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx))$$

If f is a function, let g=`Y`f. Then, it can be easily shown that g=f(g):

$$g \equiv \texttt{Y}f \equiv (\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx)))f \Leftrightarrow (\lambda x.f(xx))\ (\lambda x.f(xx)) \Leftrightarrow \qquad (\beta, \beta)$$
$$f((\lambda x.f(xx))\ (\lambda x.f(xx))) \Leftrightarrow f((\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx)))f) \equiv f(\texttt{Y}f) \equiv f(g) \qquad (\beta)$$

Thus, g is a "fixed point" of f—applying f to g returns g itself! The `Y` operator finds such a function g for any given f.

This is exactly what is needed to define a recursive function. We rewrite equation (1) above to `Fact`=f(`Fact`), where $f \equiv \lambda h.\lambda n.$(`Zero?` n `1` (`*`n (h (`Pred` n)))). Now we just find the fixed point of f and call it `Fact`:

$$\texttt{Fact} \equiv \texttt{Y}f \equiv \texttt{Y}\ \lambda h.\lambda n.(\texttt{Zero?}\ n\ \texttt{1}\ (\texttt{*}n\ (h\ (\texttt{Pred}\ n))))$$

Summary

The lambda calculus is a very simple programming language, comprised only of symbols, one-argument functions, and one-argument function applications. Yet, it can be shown that the lambda calculus is universal—it can compute anything that can be computed with any other programming language known.

The lambda calculus will be used in the next section as a target language for describing the meanings of programs.

4. Denotational Semantics

Denotational semantics [6] [4] describes the meaning of a program by mapping it to a lambda expression. The meaning of a programming language consists of a description of the mapping procedure from the program text to lambda expressions. The lambda calculus is usually enhanced by adding numbers, types, conditionals, other objects, and sometimes nondeterminism, but this does not fundamentally change the concepts behind denotational semantics.

Concepts

Valuation Functions

The program is converted to a lambda calculus expression by using a *valuation function*. I will use the sample language from section 2 to illustrate the use of valuation functions. That language consists of three main types of syntactic entities, constants, expressions, and statements, each with its own valuation function. The valuation functions have the following "signatures" (types):

K: Const → *Value*
E: Exp → *ExpContinuation* → *StmtContinuation*

Γ: Stmt → *StmtContinuation*→ *StmtContinuation*

The same trick is used for E and Γ as for $+$ in the previous section—a multi-argument function is constructed out of single-argument ones. E is a function which, when given an expression, returns a function which, when given an *ExpContinuation* returns a *StmtContinuation*.

The mapping of constants to values by K is obvious; when given a constant like `TRUE` or `12345`, K returns some suitable lambda calculus representation of that value. However, E and Γ deserve some explanation.

Meaning

For our sample language, the meaning of a program is the set of its side effects on variables. What, then, is the meaning of a statement? Just listing a statement's side effects is not good enough, because the statement could force some change of control flow (like a `GOTO` statement), or it could cause some exception or error. Modeling these actions as side effects on something like a program counter is difficult and counterintuitive. Instead, we will use the *continuation-passing* style to model meaning. The *meaning* of a statement is a function Γ which takes the meaning of the remainder of the program (the *continuation*, also written as a *StmtContinuation* or an *ExpContinuation*) as an argument and returns the meaning of the statement combined with the remainder of the program. This is why the signature of Γ is Stmt → *StmtContinuation*→ *StmtContinuation*. If the statement terminates normally, its meaning will be comprised of its side effects prepended to the meaning of its continuation (the meaning of the remainder of the program). If, however, the statement causes an error, its meaning will be just that error, regardless of the continuation[3].

Type Definitions

The signatures of the variables to be used in the valuation functions are listed below. The variables will be used consistently; *i.e.*, an *e* will always denote a *Value*, etc.

e∈*Value* = {`TRUE`, `FALSE`} ∪ **Z** (**Z** is the set of integers)

s∈*Store* = Identifier → (*Value* ∪ {`Error`})

c∈*StmtContinuation* = *Store* → *Answer*

k∈*ExpContinuation* = *Value* → *Store* → *Answer* = *Value* → *StmtContinuation*

Answer = *Store* ∪ {`Error`}

A *Value* is either a boolean or an integer. A *Store* is a function mapping variable names to values or `Error` if the corresponding variable has not been defined yet; the *Store* is used for holding the current values of variables and performs a function analogous to σ from operational semantics. *StmtContinuation*s have been described above; *ExpContinuation*s are similar and are used to denote the behavior of the remainder of the program with respect to an expression instead of a statement. For the context of evaluating an expression, the remainder of the program is expecting a *Value*. Thus, an *ExpContinuation* is expecting an additional *Value* argument. An *Answer* is the result of running the program—either a final *Store* or an error[4].

Sample Denotational Semantics

The valuation functions for the sample language from section 2 are as follows:

E[Const] = $\lambda k.\ k(\mathrm{K}$[Const]$)$

E[Var] = $\lambda k.\lambda s.\ isValue(s$[Var]$)\ ?\ k\ (s$[Var]$)\ s$: `Error`

[3]This approach makes modeling languages which support various kinds of exception handling easy; in many languages a statement may choose one of two continuations—a normal one and an exceptional one.

[4]The semantics could be modified to also return the *Store* in the *Answer* even if an error occurs.

$\mathrm{E}[\text{Var++}] = \lambda k.\lambda s.\ \mathit{isInteger}(s[\text{Var}])\ ?\ k\ (s[\text{Var}])\ (\text{bind}\ s\ [\text{Var}]\ (s[\text{Var}]+1))\ :\ \texttt{Error}$

$\mathrm{E}[\text{Exp}_1 + \text{Exp}_2] = \lambda k.\ \mathrm{E}[\text{Exp}_1](\lambda e_1.\ \mathrm{E}[\text{Exp}_2](\lambda e_2.\mathit{isInteger}(e_1)\ ?\ \mathit{isInteger}(e_2)\ ?\ k\ (e_1+e_2)$
$: \lambda s.\texttt{Error}$
$: \lambda s.\texttt{Error}))$

$\mathrm{E}[\text{Exp}_1 * \text{Exp}_2] = \lambda k.\ \mathrm{E}[\text{Exp}_1](\lambda e_1.\ \mathrm{E}[\text{Exp}_2](\lambda e_2.\mathit{isInteger}(e_1)\ ?\ \mathit{isInteger}(e_2)\ ?\ k\ (e_1\cdot e_2)$
$: \lambda s.\texttt{Error}$
$: \lambda s.\texttt{Error}))$

$\Gamma[\text{Var} := \text{Exp}] = \lambda c.\ \mathrm{E}[\text{Exp}_1](\lambda e.\lambda s.\ c\ (\text{bind}\ s\ [\text{Var}]\ e))$

$\Gamma[\texttt{IF}\ \text{Exp}\ \texttt{THEN}\ \text{Stmt}_1\ \texttt{ELSE}\ \text{Stmt}_2] = \lambda c.\ \mathrm{E}[\text{Exp}_1](\lambda e.\mathit{isBoolean}(e)\ ?\ \mathit{isTrue}(e)\ ?\ \Gamma[\text{Stmt}_1]c$
$: \Gamma[\text{Stmt}_2]c$
$: \lambda s.\texttt{Error})$

$\Gamma[\varepsilon] = \lambda c.\ c$

$\Gamma[\texttt{BEGIN END}] = \lambda c.\ c$

$\Gamma[\texttt{BEGIN}\ \text{Stmt}_0\ \text{Stmt}_1\ \ldots\ \text{Stmt}_n\ \texttt{END}] = \lambda c.\ \Gamma[\text{Stmt}_0]\ (\Gamma[\texttt{BEGIN}\ \text{Stmt}_1\ \ldots\ \text{Stmt}_n\ \texttt{END}]c)$

Notation

The quasi-lambda-calculus notation $\mathit{isX}(a)\ ?\ B\ :\ C$ means that if a is an object of type X, then B should be evaluated; otherwise, C should be evaluated. (e_1+e_2) and $(e_1\cdot e_2)$ mean the sum and product of e_1 and e_2, respectively. The (bind s id e) lambda calculus function returns a new store that is identical to s except that identifier id is bound to e. It could be defined as:

$\text{bind} = \lambda s.\lambda i.\lambda e.\ \lambda j.\ (i=j)\ ?\ e\ :\ (s\ j)$

Discussion

Let us consider the first two valuation functions in more detail. As indicated by its signature, E[Const] is a function that expects an *ExpContinuation* (the meaning of the remainder of the program) and returns a *StmtContinuation* (the meaning of the remainder of the program, including this expression). E[Const] is pretty simple—it passes the value K[Const] to the *ExpContinuation* it received, thereby passing the constant to the remainder of the program. Since the signature of an *ExpContinuation* is *Value* → *StmtContinuation*, the result is a *StmtContinuation*, which is returned as the meaning of the remainder of the program including the Const expression.

E[Var] is a little more complicated. It too is a function that expects an *ExpContinuation* (the meaning of the remainder of the program) and returns a *StmtContinuation* (the meaning of the remainder of the program, including this expression). E[Var], when given an *ExpContinuation*, returns the following *StmtContinuation* c:

> c first grabs its *Store* argument (remember that a *StmtContinuation*, as indicated by its signature, is a function that takes a *Store* as a parameter and returns an *Answer*). Then, c checks whether Var is bound in the *Store*. If so, c returns the *Answer* obtained by passing the value of Var found in the *Store* and the *Store* itself to the original *ExpContinuation* (an *ExpContinuation*, when passed a *Value* and a *Store*, returns an *Answer*). If Var is not bound in the *Store*, c returns the *Answer* `Error`.

The other valuation functions can be analyzed in a similar fashion.

Example

To find the meaning of the trivial program `x:=17`, we would reduce Γ[`x:=17`]($\lambda s.s$)($\lambda i.$`Error`). The $\lambda s.s$ is a final *StmtContinuation* used to "terminate" the program and return the final *Store* as an *Answer*. The $\lambda i.$`Error` is the initial *Store* which returns an `Error` for every lookup attempt; it indicates that no variables are initially bound. The reduction can proceed as follows:

Γ[`x:=17`]($\lambda s.s$) ($\lambda i.$`Error`) $\equiv$ ($\lambda c.$ E[`17`]($\lambda e.\lambda s.$ c (bind s [`x`] e))) ($\lambda s.s$) ($\lambda i.$`Error`)	(Γ[Var := Exp])
$\equiv$ ($\lambda c.$ ($\lambda k.$ k(K[`17`])) ($\lambda e.\lambda s.$ c (bind s [`x`] e))) ($\lambda s.s$) ($\lambda i.$`Error`)	(E[Var])
$\equiv$ ($\lambda c.$ ($\lambda k.$ k `17`)) ($\lambda e.\lambda s.$ c (bind s `x` e))) ($\lambda s.s$) ($\lambda i.$`Error`)	(K[Const])
$\Rightarrow$ ($\lambda k.$ k `17`)) ($\lambda e.\lambda s.$ ($\lambda s.s$) (bind s `x` e)) ($\lambda i.$`Error`)	(β)
$\Rightarrow$ ($\lambda e.\lambda s.$ ($\lambda s.s$) (bind s `x` e)) `17` ($\lambda i.$`Error`)	(β)
$\Rightarrow$ ($\lambda s.$ ($\lambda s.s$) (bind s `x` `17`)) ($\lambda i.$`Error`)	(β)
$\Rightarrow$ ($\lambda s.s$) (bind ($\lambda i.$`Error`) `x` `17`)	(β)
$\Rightarrow$ (bind ($\lambda i.$`Error`) `x` `17`)	(β)

We get, as expected, a *Store* with a binding of `17` to `x`.

Summary

Denotational semantics ascribes meanings to programs by translating them into lambda calculus expressions. The semantics of a language is a set of valuation functions which describe how programs written in that language should be translated to lambda calculus expressions. Although harder to understand, this translation has advantages over operational semantics:

- The meaning of the lambda calculus is unique, well-defined, and well-known. It might thus be possible to define the meanings of mixing languages using the lambda calculus.

- Lambda calculus reductions can be used to prove properties of the language and prove the validity of compiler optimizations. For simple arithmetic optimizations such as a+b=b+a this is probably overkill, but the provability of optimizations becomes important when exceptions or concurrent processes are present in the language, as it is very easy to make incorrect assumptions when writing a compiler for such a language.

5. Axiomatic Semantics

Axiomatic semantics describes the meaning of a program by listing its properties. For example, the meaning of a nondestructive integer sort routine could be described as:

Let A be an array of n integers, indexed 0 through n-1. Calling `sort` on A returns a new array B, also of n integers. Array B contains the same elements as A; formally, $\forall v \in Z$: $|\{i|0 \le i < n \wedge A[i]=v\}| = |\{i|0 \le i < n \wedge B[i]=v\}|$ (for every integer value v, the number of elements of A equal to v is equal to the number of elements of B equal to v). Moreover, the elements of B are in increasing order: $\forall i \in \{0 \ldots n\text{-}2\}$: $B[i] \le B[i+1]$. This definition is independent of the algorithm used to implement the `sort` function.

Axiomatic semantics is useful in proving properties of programs. For example, a deduction that the above properties hold for a given function proves that that function sorts an array of integers. For simple applications such as sorting, this proof can, in fact, be carried out. For complicated programs the benefits of axiomatic semantics will not be realized until the process of proving theorems is automated because writing proofs is itself time-consuming and error-prone. Nevertheless, axiomatic semantics are already in use in hardware design for proving that two integrated circuits have compatible interfaces and can, therefore, communicate with each other.

6. Conclusion

Three different kinds of formal semantics—operational, denotational, and axiomatic—were presented in this paper, but the field is by no means limited to these three. All of the kinds of formal semantics have some common advantages and disadvantages: Formal semantics are precise and powerful but tedious and hard to learn. They are not as well-known among users of languages as BNF grammars, but nowadays they are being used fairly frequently in the design of programing languages, and they may be used for serious programming verification tasks in the future.

In addition to the obvious application of formal semantics for unambiguously specifying the meanings of new programming languages, several other applications exist:

Language Design Aids

Denotational semantics have been used to precisely define several existing programming languages, including Pascal and Scheme [2]. They have also been used to assist in designing and implementing more complex languages such as Ada [1]. The process of defining denotational semantics for a language can indicate weak areas where the language is poorly or incorrectly defined. This is especially important for modern programming languages that include polymorphism, concurrency, and exception handling. It is possible for programs written in such languages to get into semantically difficult situations such as when an exception occurs within an exception handler or propagates out of a concurrent task. Properly handling everything that can go wrong requires the discipline imposed by writing semantics for the language. In these situations writing denotational semantics for a language is useful even if no one other than the author ever reads them.

Verification

The holy grail of the study of semantics is the eventual ability to prove programs correct. Such proofs are currently feasible, but only on a small scale (a few procedures), limiting their utility. However, there does not appear to be any fundamental reason why large-scale programs could not be proven correct. Once proving programs' correctness does become practical, the art of computer programming will be changed forever.

In order to prove a program correct, one needs a specification of the program's correct behavior. Without such a specification, every program is correct—it just might not do what the user wants or expects. One might argue that the specification would be at least as large as the program which is being verified and that there are many opportunities for making errors in the specification. This is true, but, nevertheless, writing a specification is a fundamentally easier process than programming. How much easier it is to just describe the behavior of a database than it is to write one! Moreover, it is easy to write a specification that states that a program will never crash. Try writing a Macintosh program for which you would be willing to give the same guarantee! Finally, tools will undoubtedly be developed to ease the process of developing specifications by describing them graphically or by example.

While general verification of programs is still unreachable, useful work proceeds on many fronts. The development of type systems is one attempt to outlaw a class of common programming errors—type mismatches. The initial type systems were too restrictive, but new, polymorphic ones are being developed. Moreover, verifiers are being written for proving that programs written in dynamically typed languages such as LISP will have no type errors when run.

Formal verification has made greater inroads in hardware design than in software design. Due to the high cost of recalling chips and missed deadlines, the silicon manufacturers are putting great emphasis on testing and verifying new chip designs. Axiomatic semantics are used in the design of computer hardware components to ensure that their interfaces match. Various timing and logic analysis tools are used to check the chip specifications against the design. Designing a modern VLSI chip is comprised largely of writing an initial functional specification and successively translating it to more detailed spec

ifications—register transfer, gate, transistor, and layout levels—and verifying that the two specifications match at each step. The verifications are often done by hand, and a lot of time, effort, and money could be saved by automating them.

The Viper microprocessor is the first one to be completely verified by formal means. Formal verification does not guard against fabrication errors, but at least it ensures that the design satisfies the specifications, which is important in life-critical situations. Not too far into the future the hardware formal specifications may be made available to system and software designers to allow entire systems including both software and hardware to be verified. Sometime later software may be built the way hardware is today.

Bibliography

[1] Serafino Amoroso and Giorgio Ingargiola. *Ada: An Introduction to Program Design and Coding*. Howard W. Sams & Co., Marshfield, MA, 1985.

[2] Jonathan Rees and William Clinger, ed. *Revised^3 Report on the Algorithmic Language Scheme*. MIT Artificial Intelligence Laboratory Memo 848a, September 1986.

[3] J. Barkley Rosser. "Highlights of the History of the Lambda-Calculus." *1982 ACM Symposium on Lisp and Functional Programming*.

[4] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, MA, 1986.

[5] Guy L. Steele. *Common Lisp: The Language*. Digital Press, Digital Equipment Corporation, 1984.

[6] R. D. Tennant. "The Denotational Semantics of Programming Languages." *Communications of the ACM*, **19:8**, August 1976.