Denotational Semantics

Robert D. Cameron

January 23, 2002

1 Introduction

The denotational approach to semantics is to provide a mathematical interpretation of programming language constructs. This is done by first of all grouping various related types of constructs into *syntactic domains*, e.g., declarations, statements, expressions and numerical constants. The constructs of each domain are then assigned their semantics through a *semantic function*. The semantic function maps the various elements of the syntactic domain into a corresponding *semantic domain*; the elements of this domain comprise the mathematical entities which model the meaning of language constructs. This mapping is defined using *semantic equations* on a case by case basis for each syntactic construct type. Following this basic framework, it is possible to formulate detailed and accurate semantic definitions of programming languages.

The denotational approach can be nicely illustrated using the example syntactic domain of binary numerals. These may be specified syntactically using the following BNF productions.

$$\begin{array}{rcl} \langle \text{numeral} \rangle & ::= & \langle \text{digit} \rangle \mid \langle \text{numeral} \rangle \ \langle \text{digit} \rangle \\ \langle \text{digit} \rangle & ::= & \texttt{0} \mid \texttt{1} \end{array}$$

The semantics of such numerals may be described by a semantic function $\mathcal{N}$ which maps binary numerals into the integer numbers they represent. In this case, then, the semantic domain is very simple, namely I, the set of integers. The $\mathcal{N}$ function is defined by the following semantic equations.

$$\begin{array}{rcl} \mathcal{N}[\![\texttt{0}]\!] & = & 0 \\ \mathcal{N}[\![\texttt{1}]\!] & = & 1 \\ \mathcal{N}[\![\langle \text{numeral} \rangle \ \langle \text{digit} \rangle]\!] & = & 2 \times \mathcal{N}[\![\langle \text{numeral} \rangle]\!] + \mathcal{N}[\![\langle \text{digit} \rangle]\!] \end{array}$$

The special brackets $[\![]\!]$ are used as *meta-syntactic* brackets to enclose syntactic entities (terminal and nonterminal symbols). Thus, in the first equation, the `0` within brackets denotes the terminal symbol 0 as a syntactic element, whereas the 0 outside of the brackets denotes the integer number zero as an element of the semantic domain. In this example, the semantic equations for $\mathcal{N}$ completely specify how binary numerals are interpreted as representing integer numbers.

2 The Denotational Semantics of TINY

We will now use the programming language TINY to further illustrate the use of denotational semantics. TINY is a simple programming language, not intended for any practical programming work. It incorporates only a subset of the useful constructs typically found in any modern programming language; this subset is chosen purely to illustrate the principles of denotational semantics. The following denotational description of TINY is adapted from *The Denotational Description of Programming Languages*, by Michael J.C. Gordon.

2.1 The Syntax of TINY

$$
\begin{array}{rcl}
\langle \text{exp} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{true} \mid \texttt{false} \mid \texttt{read} \mid \langle \text{ide} \rangle \mid \texttt{not}\ \langle \text{exp} \rangle \mid \\
 & & \langle \text{exp} \rangle\ \texttt{=}\ \langle \text{exp} \rangle \mid \langle \text{exp} \rangle\ \texttt{+}\ \langle \text{exp} \rangle \\
\langle \text{cmd} \rangle & ::= & \langle \text{ide} \rangle\ \texttt{:=}\ \langle \text{exp} \rangle \mid \texttt{output}\ \langle \text{exp} \rangle \mid \\
 & & \texttt{if}\ \langle \text{exp} \rangle\ \texttt{then}\ \langle \text{cmd} \rangle\ \texttt{else}\ \langle \text{cmd} \rangle\ \texttt{fi} \mid \\
 & & \texttt{while}\ \langle \text{exp} \rangle\ \texttt{do}\ \langle \text{cmd} \rangle\ \texttt{od} \mid \langle \text{cmd} \rangle\ \texttt{;}\ \langle \text{cmd} \rangle
\end{array}
$$

2.2 Syntactic Domains

`Ide`	:	identifiers
`Exp`	:	expressions
`Cmd`	:	commands

2.3 Informal Semantics

Informally, the semantics of TINY commands is that they are executed to make some change to the current computer state. The state has three components, namely, the *memory*, the *input*, and the *output*. The memory models the current contents of the computer memory as a function mapping the various defined identifiers to their current values. The input models that part of the input file that remains to be read in at any given time; abstractly, it is a sequence of values which is initially the sequence of values in the input file. Similarly, the output is a sequence of values modelling that part of an output file which has already been generated; initially, it is the empty sequence.

The semantics of TINY expressions is primarily that they are evaluated to return some value, either a boolean value or an integer. The value returned depends, in general, on the current state since expressions may contain identifiers. In addition, evaluation of expressions may affect the current state, i.e., the `read` expression reads the first value from the input stream and advances the input by one value.

2.4 Basic Semantic Domains

$$\textbf{State} = \textbf{Memory} \times \textbf{Input} \times \textbf{Output}$$

The state is modelled mathematically as the domain of all triples (m, i, o), where m is a memory, i is an input and o is an output. (In general, given domains $D_1, D_2, ...D_n$, the notation $D_1 \times D_2 \times \ldots \times D_n$ stands for the domain of tuples $(d_1, d_2, \ldots, d_n)$ where each d_i is a value in the corresponding domain D_i.)

$$
\begin{aligned}
\textbf{Memory} &= \texttt{Ide} \rightarrow [\textbf{Value} + \{\text{unbound}\}] \\
\textbf{Value} &= \textbf{Num} + \textbf{Bool}
\end{aligned}
$$

The memory is the domain of all functions mapping identifiers into the domain [**Value**+{unbound}]. (In general $D_1 \rightarrow D_2$ stands for the domain of functions taking input values from D_1 and generating output values in D_2.) [**Value** + {unbound}] is the domain of objects which are either members of the domain **Value** or the special value unbound. This special value is used to model the case when an identifier has not actually been given a value through an assignment statement. The domain **Value** is in turn the domain of objects which are either members of the domain **Num** (numbers)

or of the domain **Bool** (boolean values).

$$\begin{aligned} \textbf{Input} &= \textbf{Value}^* \\ \textbf{Output} &= \textbf{Value}^* \end{aligned}$$

For a given domain D, D^* denotes the domain of all sequences of 0 or more elements from D.

2.5 Semantic Functions

The detailed semantics of TINY now requires that we show how TINY expressions and commands are modelled in terms of these basic semantic domains. We need to define two functions:

$$\begin{aligned} \mathcal{E} &: \texttt{Exp} \rightarrow \{\text{denotations of expressions}\} \\ \mathcal{C} &: \texttt{Cmd} \rightarrow \{\text{denotations of commands}\} \end{aligned}$$

The function $\mathcal{E}$ describes the mapping from the syntactic domain of expressions (`Exp`) into the mathematical objects we use to model expressions and the function $\mathcal{C}$ similarly defines the mapping of commands as syntactic objects into their corresponding semantic models.

In general, we model expressions as functions which take as input the current state and generate as output one of the following:

- a special value error, if an error occurs, or
- the value for the expression and a possibly modified state (for expressions with side-effects).

This gives the functionality of $\mathcal{E}$

$$\mathcal{E} : \texttt{Exp} \rightarrow [\textbf{State} \rightarrow [[\textbf{Value} \times \textbf{State}] + \{\text{error}\}]]$$

The domain $[\textbf{State} \rightarrow [[\textbf{Value} \times \textbf{State}] + \{\text{error}\}]]$ is thus the domain of objects which are used to mathematically model the various types of expression.

Commands are modelled as functions which change the state if the command succeeds or generate the value error if an unexpected situation arises.

$$\mathcal{C} : \texttt{Cmd} \rightarrow [\textbf{State} \rightarrow [\textbf{State} + \{\text{error}\}]]$$

2.6 Semantic Equations

For each syntactic clause we want a semantic equation of the form, e.g.,

$$\mathcal{E}[\![\text{syntactic clause}]\!] = \text{denotation of syntactic clause}$$

For both expressions and commands, these R.H.S. denotations are functions, i.e., functions in the domain

$$[\textbf{State} \rightarrow [[\textbf{Value} \times \textbf{State}] + \{\text{error}\}]$$

for expressions and functions in the domain

$$[\textbf{State} \rightarrow [\textbf{State} + \{\text{error}\}]]$$

for commands.

In order to denote these functions, we will use *lambda notation*, following the lambda calculus of Alonzo Church. In the lambda notation, functions are described by *lambda expressions* which have the following syntax.

$$\langle\text{lambda-exp}\rangle \quad ::= \quad \lambda\ \langle\text{parameters}\rangle\ .\ \langle\text{body}\rangle$$

The ⟨parameters⟩ are single letter names for the arguments of the function, typically x, y, and z. The body is an expression defining the value of the function in terms of the values of the parameters. For example, the successor function, which, for any given integer returns the next one in sequence, can be represented using the following lambda expression.

$$\lambda x.x + 1$$

The absolute value function can be written as

$$\lambda x.\ \text{if } x \geq 0 \text{ then } x \text{ else } -x$$

using an extension of lambda notation to include conditional expressions. A simple example of a two argument function is

$$\lambda xy.\ \text{if } x \geq y \text{ then } x \text{ else } y$$

which returns the maximum of two numbers.

The semantic equations for expressions can now be given as follows.

$$\mathcal{E}[\![\texttt{0}]\!] \quad = \quad \lambda s.(0, s) \tag{1}$$

For the numeral `0` in TINY, the denotation is the function which returns the number 0 as its value and does not change the current state. Note that the notation $(0, s)$ is being used to denote the two-tuple (pair) which is a member of the domain [**Value** × **State**].

$$\mathcal{E}[\![\texttt{1}]\!] \quad = \quad \lambda s.(1, s) \tag{2}$$

$$\mathcal{E}[\![\texttt{true}]\!] \quad = \quad \lambda s.(\text{true}, s) \tag{3}$$

$$\mathcal{E}[\![\texttt{false}]\!] \quad = \quad \lambda s.(\text{false}, s) \tag{4}$$

$$\mathcal{E}[\![\texttt{read}]\!] \quad = \quad \lambda(m, i, o).\ \text{if } i = \{\} \text{ then error else } (\text{head}(i), (m, \text{tail}(i), o)) \tag{5}$$

Here (m, i, o) replaces s as the name of the state parameter, so that the components of the state can be referred to individually. This clause defines the semantics of the `read` expression as follows. The test $i = \{\}$ checks if the current input is the empty sequence, if so the special value error is returned. Otherwise, the value of the expression is the first element in the input sequence (denoted by head(i)) and the state is modified by advancing the input stream (tail(i) denotes the remaining elements after the first one in the old input sequence).

$$\mathcal{E}[\![\langle\text{ide}\rangle]\!] \quad = \quad \lambda(m, i, o).\ \text{if } m[\![\langle\text{ide}\rangle]\!] = \text{unbound} \text{ then error else } (m[\![\langle\text{ide}\rangle]\!], (m, i, o)) \tag{6}$$

Recall that the memory is a function mapping identifiers to values, so $m[\![\langle\text{ide}\rangle]\!]$ is just the value corresponding to identifier ⟨ide⟩. If, in fact, there is no proper value for this identifier, then the value unbound is returned by the memory when it is applied to that identifier. In such a case, the result of evaluating the identifier as an expression is to be the special value error, otherwise the result of evaluation should be the actual value of the identifier together with an unchanged state.

$$\begin{aligned}\mathcal{E}[\![\texttt{not } \langle\text{exp}\rangle]\!] \quad = \quad \lambda s.\ &\text{if } \mathcal{E}[\![\langle\text{exp}\rangle]\!](s) = \text{error then error}\\ &\text{else if } \mathcal{E}[\![\langle\text{exp}\rangle]\!](s) = (\text{true}, s_1) \text{ then } (\text{false}, s_1)\\ &\text{else if } \mathcal{E}[\![\langle\text{exp}\rangle]\!](s) = (\text{false}, s_1) \text{ then } (\text{true}, s_1)\\ &\text{else error}\end{aligned} \tag{7}$$

The meaning of a `not` expression is defined in terms of the meaning of its component expression. The notation $\mathcal{E}[\![\langle \text{exp} \rangle]\!](s)$ means the value of the function denoted by $\mathcal{E}[\![\langle \text{exp} \rangle]\!]$ applied to the state s. If this value is the error value then the value of the `not` expression is also error. If the evaluation of the component expression returns a value of true and some new state s_1 (which may be different from s), then the evaluation of the `not` expression yields the value false and the new state s_1.

$$\begin{aligned}\mathcal{E}[\![\langle \text{exp1} \rangle \texttt{ = } \langle \text{exp2} \rangle]\!] \;=\; \lambda s.\ &\text{if } \mathcal{E}[\![\langle \text{exp1} \rangle]\!](s) = (v_1, s_1) \\ &\quad \text{then if } \mathcal{E}[\![\langle \text{exp2} \rangle]\!](s_1) = (v_2, s_2) \\ &\qquad \text{then if } v_1 = v_2 \text{ then } (\text{true}, s_2) \\ &\qquad\qquad \text{else } (\text{false}, s_2) \\ &\qquad \text{else error} \\ &\quad \text{else error}\end{aligned} \tag{8}$$

Note that $\langle \text{exp2} \rangle$ is evaluated in the state s_1 which results from the evaluation of $\langle \text{exp1} \rangle$.

$$\begin{aligned}\mathcal{E}[\![\langle \text{exp1} \rangle \texttt{ + } \langle \text{exp2} \rangle]\!] \;=\; \lambda s.\ &\text{if } \mathcal{E}[\![\langle \text{exp1} \rangle]\!](s) = (v_1, s_1) \\ &\quad \text{then if } \mathcal{E}[\![\langle \text{exp2} \rangle]\!](s_1) = (v_2, s_2) \\ &\qquad \text{then if } \text{isNum}(v_1) \wedge \text{isNum}(v_2) \text{ then } (v_1 + v_2, s_2) \\ &\qquad\qquad \text{else error} \\ &\quad \text{else error} \\ &\text{else error}\end{aligned} \tag{9}$$

The function $\text{isNum}(v_1)$ denotes a check that the value v_1 is a number rather than a boolean value; if either v_1 or v_2 is boolean then the result is an error.

The semantic equations for commands are developed similarly as follows.

$$\begin{aligned}\mathcal{C}[\![\langle \text{ide} \rangle \texttt{ := } \langle \text{exp} \rangle]\!] \;=\; \lambda(m, i, o).\ &\text{if } \mathcal{E}[\![\langle \text{exp} \rangle]\!](m, i, o) = (v, (m_1, i_1, o_1)) \\ &\quad \text{then } (m_1[\![v/\langle \text{ide} \rangle]\!], i_1, o_1) \\ &\quad \text{else error}\end{aligned} \tag{10}$$

Here, the notation $m_1[\![v/\langle \text{ide} \rangle]\!]$ denotes a new memory which defines the same mapping of identifiers to values that the memory m_1 does, except that the given identifier $\langle \text{ide} \rangle$ maps to the value v.

$$\begin{aligned}\mathcal{C}[\![\texttt{output } \langle \text{exp} \rangle]\!] \;=\; \lambda(m, i, o).\ &\text{if } \mathcal{E}[\![\langle \text{exp} \rangle]\!](m, i, o) = (v, (m_1, i_1, o_1)) \\ &\quad \text{then } (m_1, i_1, \text{append1}(o_1, v)) \\ &\quad \text{else error}\end{aligned} \tag{11}$$

Here $\text{append1}(o_1, v)$ is an operation which adds the element v to the end of the stream of values o_1.

$$\begin{aligned}&\mathcal{C}[\![\texttt{if } \langle \text{exp} \rangle \texttt{ then } \langle \text{cmd1} \rangle \texttt{ else } \langle \text{cmd2} \rangle \texttt{ fi}]\!] = \\ &\quad \lambda s.\ \text{if } \mathcal{E}[\![\langle \text{exp} \rangle]\!](s) = (v, s_1) \\ &\qquad \text{then if } \text{isBool}(v) \\ &\qquad\quad \text{then if } v = \text{true} \\ &\qquad\qquad \text{then } \mathcal{C}[\![\langle \text{cmd1} \rangle]\!](s_1) \\ &\qquad\qquad \text{else } \mathcal{C}[\![\langle \text{cmd2} \rangle]\!](s_1) \\ &\qquad\quad \text{else error} \\ &\qquad \text{else error}\end{aligned} \tag{12}$$

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{while}\ \langle \text{exp} \rangle\ \texttt{do}\ \langle \text{cmd} \rangle\ \texttt{od}]\!] = \\
\quad \lambda s.\ \text{if}\ \mathcal{E}[\![\langle \text{exp} \rangle]\!](s) = (v, s_1) \\
\qquad \text{then if isBool}(v) \\
\qquad\quad \text{then if } v = \text{false} \\
\qquad\qquad \text{then } s_1 \\
\qquad\qquad \text{else if } \mathcal{C}[\![\langle \text{cmd} \rangle]\!](s_1) = s_2 \\
\qquad\qquad\quad \text{then } \mathcal{C}[\![\texttt{while}\ \langle \text{exp} \rangle\ \texttt{do}\ \langle \text{cmd} \rangle\ \texttt{od}]\!](s_2) \\
\qquad\qquad\quad \text{else error} \\
\qquad\quad \text{else error} \\
\qquad \text{else error}
\end{array}
\tag{13}
$$

Here the interpretation of a `while` loop is defined recursively.

$$
\begin{array}{rcl}
\mathcal{C}[\![\langle \text{cmd1} \rangle\ ;\ \langle \text{cmd2} \rangle]\!] & = & \lambda s.\ \text{if}\ \mathcal{C}[\![\langle \text{cmd1} \rangle]\!](s) = s_1 \\
 & & \qquad \text{then}\ \mathcal{C}[\![\langle \text{cmd2} \rangle]\!](s_1) \\
 & & \qquad \text{else error}
\end{array}
\tag{14}
$$

3 Viewpoint on Denotational Semantics

Denotational semantics provides a way of precisely and completely describing the semantics of a programming language. Such descriptions can be useful, in various ways, to language designers, to language implementors and to programmers. Language designers can benefit in two ways from developing denotational descriptions of their languages as they design them. First of all, such descriptions are often useful in pinpointing weaknesses in the design, and secondly, the completeness of these descriptions helps in making sure that every important detail has been taken care of. Denotational descriptions benefit language implementors by giving them complete language descriptions, allowing implementation of systems which exactly conform both to the original definition and to other implementations. Programmers can use denotational descriptions for reference purposes when the informal semantic descriptions are either unclear or incomplete in describing various details.

Unfortunately, the metalanguage of denotational semantics is in itself quite complicated. As a result, it is probably too much to expect of the average programmer to be able to master denotational semantics and be able to use it for reference purposes. However, the importance of having well-defined standards for programming languages should lead to an increasing use of denotational semantics by language designers and implementors in the future.