Basic Denotational Semantics

Version 3

Handout A

6th June 2007

1 Overview

Unlike Syntax, Semantics remains a topic for which there is no universally accepted description mechanism. Researchers disagree whether language semantics should be described in plain English, in terms of some abstract computer, or in one of three well-understood formal approaches.

One of these three approaches is *denotational semantics*, originally due to Dana Scott and Christopher Strachey (though initial ideas can be traced back to Frege). Compared to its alternatives, denotational semantics is the easiest mechanism for describing the meaning of smaller programs. However, denotational semantics becomes very complex for describing advanced features, such as while loops, recursion, or goto statements.

In this handout, we will have a quick look at the *basics* of denotational semantics, avoiding the more complex structures, to give a flavour of what a formal language specification might look like.

For the following discussion, recall again the definition of *meta-language* from the book: A meta-language is a language that is used to describe other languages. Whenever we use a meta-language to describe a specific language, we refer to that specific language as the *object language*.

For example, BNF is a meta-language. When we use BNF to describe the syntax of C, then C is the object language.

As another example, we will now describe denotational semantics in plain English. Thus, English is our meta-language, and the language of denotational semantics is our object language.

2 Denotations

The very basic idea of denotational semantics is to translate (programming) language constructs into mathematical objects. For example, we might translate a C or Java expression into its mathematical *denotation*:

$$[\![\texttt{1 + (int) 3.2}]\!] = 4$$

Note the use of the semantic braces (also known as the *interpretation function*) $[\![]\!]$ for this process: $[\![\mathsf{X}]\!] = v$ means that the semantics of the program fragment X is precisely the mathematical object v.

Using mathematical objects for our description gives us a wealth of existing formalisms to draw from. For now, let us restrict ourselves to arithmetics.

The first question we are faced with is the translation of numbers. For example, we obviously want

$$[\![\mathtt{17}]\!] = 17$$

i.e, we want to have the number 17 as the denotation of the object-language construct `17`. Achieving this denotationally with full formality is surprisingly tricky. Let's say that we only want to represent the first ten numbers (0 to 9). We can achieve our translation by *case distinction*, i.e., by

$$[\![d]\!] = \begin{cases} 0 & \iff & d = \mathsf{0} \\ 1 & \iff & d = \mathsf{1} \\ 2 & \iff & d = \mathsf{2} \\ 3 & \iff & d = \mathsf{3} \\ 4 & \iff & d = \mathsf{4} \\ 5 & \iff & d = \mathsf{5} \\ 6 & \iff & d = \mathsf{6} \\ 7 & \iff & d = \mathsf{7} \\ 8 & \iff & d = \mathsf{8} \\ 9 & \iff & d = \mathsf{9} \end{cases}$$

This works, but it is not particularly elegant. It requires us to write down one rule for every number we want to support– so, if we want to support numbers up to $2^{63} - 1$, as supported by Java's builtin type `long`, we have quite some writing to do. For simplicity, language designers therefore tend to use informal shortcuts: Here, we will assume a function `rep` as given, where `rep` translates a lexeme describing a natural number into a natural number.

We can then write

$$[\![n]\!] = \mathsf{rep}(n)$$

Of course, all we have done now is to introduce a "magical" solution to our problem: we assumed that there is some external function that happens to do what we want to do. Even though such functions are convenient, we must only introduce them with great care– as we will see in the next section, the meaning of a construct is not always obvious. Fortunately, there is a sensible way of defining the function `rep`, as we will see in one of the exercises.

2.1 A Calculator

Having (slightly informally) defined how we interpret numbers, we are now ready to examine a more interesting structure. Figure 1 contains a simple BNF grammar for a pocket calculator, with addition, negation, multiplication, and division, and parenthesised expressions. Our aim is that this calculator should

$$\begin{array}{rcl} \langle expr \rangle & \rightarrow & \mathsf{number} \\ & | & \texttt{(}\ \langle expr \rangle\ \texttt{)} \\ & | & \langle expr \rangle + \langle expr \rangle \\ & | & \langle expr \rangle - \langle expr \rangle \\ & | & \langle expr \rangle * \langle expr \rangle \\ & | & \langle expr \rangle / \langle expr \rangle \end{array}$$

Figure 1: Simple calculator language. The token *number* describes any integral number.

operate on the integers, i.e., all positive or negative numbers (including zero) without a fractional part.

Let us first consider addition. We can describe the addition of two numbers as

$$[\![n_1\ \texttt{+}\ n_2]\!] = \mathsf{rep}(n_1) + \mathsf{rep}(n_2)$$

where n_1 and n_2 are numbers.

Note how we use the addition operator of the meta-language (which is the entirety of mathematics), "+", to define the meaning of the addition operator of the object language (which is our calculator language), "`+`".

However, we only defined what it means for two numbers to be added together– this is much less than what we want addition to be capable of. Looking at our syntax, we see that "`+`" may have arbitrary expressions to its left and right. For example, the expression `(2 - 1) + (5 + 1)` is syntactically allowed.

Fortunately, denotational semantics allows us to define the semantic function *recursively*. We can thus express our semantics as

$$[\![e_1\texttt{+}e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

Here, our e_1 and e_2 may be arbitrary expressions. Similarly, we can define the remaining operators:

$$\begin{array}{rcl} [\![e_1\texttt{-}e_2]\!] & = & [\![e_1]\!] - [\![e_2]\!] \\ [\![e_1\texttt{*}e_2]\!] & = & [\![e_1]\!] \cdot [\![e_2]\!] \\ [\![e_1\texttt{/}e_2]\!] & = & \lfloor \dfrac{[\![e_1]\!]}{[\![e_2]\!]} \rfloor \end{array}$$

Note the use of floating point truncation ($\lfloor\ \rfloor$) on the division result to ensure that the result is still an integer.

We have almost succeeded in giving a full denotational definition of our calculator. However, we accidently forgot to give a definition of the semantics of parentheses. This omission meant that that our semantics was partially *undefined*: There was no definition that would tell us the meaning of, for example,

`(1)`. Fortunately, our omission is easy to fix. We just add the following definition:

$$[\![\texttt{(}e\texttt{)}]\!] \quad = \quad [\![e]\!]$$

Alas, we are still not done. Another part of our language is undefined, though in a more subtle fashion. Using the rules we have written down, we get

$$[\![\texttt{1 / 0}]\!] = \lfloor \frac{1}{0} \rfloor$$

However, the meaning of $\frac{1}{0}$ is undefined in the integers. Thus, there is still one hole in our definition that we need to close.

2.1.1 Handling Errors

Since we cannot compute a proper integer number that represents $\frac{1}{0}$ in a meaningful fashion, we can choose what we want our language/pocket calculator to do. An easy way out would be e.g. to define

$$[\![e_1\texttt{/}e_2]\!] = \begin{cases} 0 & \iff [\![e_2]\!] = 0 \\ \lfloor \frac{[\![e_1]\!]}{[\![e_2]\!]} \rfloor & \text{otherwise} \end{cases}$$

Then, our calculator would always compute 0 whenever we tried to divide by zero. In practice, language designers rarely take this approach: just returning 0 "because we have no idea what else to do" is likely to mask an error or special case. Thus, it is better to trigger an error condition:

$$[\![e_1\texttt{/}e_2]\!] = \begin{cases} \textsc{Error} & \iff [\![e_2]\!] = 0 \\ \lfloor \frac{[\![e_1]\!]}{[\![e_2]\!]} \rfloor & \text{otherwise} \end{cases}$$

Thus, we now have e.g.

$$[\![(1+2)/(1-1)]\!] = \textsc{Error}$$

This error condition has a wide-reaching effect on our semantics. Previously, our function $[\![]\!]$ always returned an integer (i.e., a number out of $\mathbb{Z}$). Now, $[\![]\!]$ can return either a number out of $\mathbb{Z}$ or the value Error.

However, all of our previous function definitions, such as

$$[\![e_1\texttt{+}e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

assumed that $[\![e_1]\!]$ and $[\![e_2]\!]$ would be a number from $\mathbb{Z}$, on which addition is defined; addition involving Error, however, is not defined. Consequently, $[\![(1/0)+1]\!]$ is now undefined.

To address this problem, we must update all of our previous definitions, e.g.

$$[\![e_1\texttt{+}e_2]\!] = \begin{cases} \textsc{Error} & \iff [\![e_1]\!] = \textsc{Error} \text{ or } [\![e_2]\!] = \textsc{Error} \\ [\![e_1]\!] + [\![e_2]\!] & \text{otherwise} \end{cases}$$

$$
\begin{array}{lcl}
\langle \mathit{program} \rangle & \rightarrow & \langle \mathit{stmtlist} \rangle \langle \mathit{expr} \rangle \\
\langle \mathit{expr} \rangle & \rightarrow & \mathsf{number} \\
& | & \mathsf{(}\ \langle \mathit{expr} \rangle\ \mathsf{)} \\
& | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle - \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle * \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle / \langle \mathit{expr} \rangle \\
& | & \mathsf{id} \\
\langle \mathit{stmt} \rangle & \rightarrow & \mathsf{id}\ \mathsf{:=}\ \langle \mathit{expr} \rangle \\
\langle \mathit{stmtlist} \rangle & \rightarrow & \varepsilon \\
& | & \langle \mathit{stmt} \rangle \mathsf{;} \langle \mathit{stmtlist} \rangle
\end{array}
$$

Figure 2: Simple calculator language. The token *number* describes any integral number.

Then, we can easily compute that

$$[\![(1/0) + 1]\!] = \textsc{Error}$$

3 Denotations and Environments

Denotational semantics can also explain state in programs. Consider the extended calculator in Figure 2. We have updated this calculator to have a new start symbol $\langle \mathit{program} \rangle$, which consists of a statement list $\langle \mathit{stmtlist} \rangle$ followed by a single expression $\langle \mathit{expr} \rangle$. The statement list is straightforward: It may be empty, or it may be a single statement followed (recursively) by another statement list.

We allow only one statement, namely a *variable write*:

$$\mathsf{id:=}\ \langle \mathit{expr} \rangle$$

The intuition behind the variable write is that it should update some variable, represented by "id", to now contain the value represented by $\langle \mathit{expr} \rangle$.

We also extend the definition of expr to allow identifier occurrences, i.e., "id". The intuition behind this construct is simply that we read out the contents of this variable.

We now want to update our interpretation function to handle such constructs, so that we can e.g. compute

$$[\![\texttt{a := 1; a+2}]\!] = 3$$

or even

$$[\![\texttt{a := 6; b := a+1; a*b}]\!] = 42$$

3.1 Environments

However, how are we to define the meaning of a variable, just by looking at it? If we break down an expression such as $\mathsf{a}+2$, we wind up having to compute $[\![\mathsf{a}]\!]$– however, we have no idea what this might mean, since we are considering a outside of any context.

A context, then, is precisely what we need to solve the problem. We introduce an *environment*, usually written E, which is a partial function mapping identifiers to values. Partial functions are well-understood mathematical entities, so it is perfectly acceptable to employ them in our description. We define the following basic operations on them:

- *empty environment*: $[]$ — This describes an environment in which no identifier is mapped to a value.
- *update*: $E, \mathsf{id} \mapsto v$ — this describes an envrionment that behaves like E, except that the identifier "id" is now mapped to v
- *lookup*: $E(\mathsf{id})$ – this looks up "id" in the environment E, and yields whichever value "v" the identifier maps to.

We can define the meaning of the above constructions in clear mathematical language, which formalises what we described above:

$$E(\mathsf{id}) = \begin{cases} v & \iff E = (E', \mathsf{id} \mapsto v) \\ E'(\mathsf{id}) & \iff E = (E', \mathsf{id'} \mapsto v) \text{ and } \mathsf{id'} \neq \mathsf{id} \end{cases}$$

Using environments, we can now write

$$E = [], \mathsf{a} \mapsto 6, \mathsf{b} \mapsto 7$$

to describe an environment that maps "a" to $E(\mathsf{a}) = 6$ and "b" to $E(\mathsf{b}) = 7$. We can update this environment again:

$$E' = E, \mathsf{a} \mapsto 0$$

Now, $E'(\mathsf{b}) = 7$ is unchanged, but $E'(\mathsf{a}) = 0$.

Note that the functions E and E' are only partial; for example, $E(\mathsf{c})$ is undefined.

3.2 Once more, with Environments!

How can we now use environments in our definition of semantics? We simply parameterise our interpretation function by an environment. This requires minor updates to our earlier definitions, e.g. we now replace

$$[\![e_1 + e_2]\!] = \begin{cases} \textsc{Error} & \iff [\![e_1]\!] = \textsc{Error} \text{ or } [\![e_2]\!] = \textsc{Error} \\ [\![e_1]\!] + [\![e_2]\!] & \text{otherwise} \end{cases}$$

by

$$[\![e_1\texttt{+}e_2]\!](E) = \begin{cases} \textsc{Error} & \iff [\![e_1]\!](E) = \textsc{Error} \text{ or } [\![e_2]\!](E) = \textsc{Error} \\ [\![e_1]\!](E) + [\![e_2]\!](E) & \text{otherwise} \end{cases}$$

After transforming the remaining operation definitions as above, we are ready to finish the denotational description of our extended calculator.

First, consider how we now treat the occurrence of an identifier in an expression:

$$[\![\mathsf{id}]\!](E) = \begin{cases} \textsc{Error} & \iff E(\mathsf{id}) \text{ is undefined} \\ E(\mathsf{id}) & \text{otherwise} \end{cases}$$

As you can see, we made sure to handle the error case where the program tries to read a variable that is not defined.

Having handled all of expressions, we move on to statements. Since we define statements over the nonterminal $\langle \textit{stmt} \rangle$, and not over $\langle \textit{expr} \rangle$, we put a small index onto our semantic function so that we can distinguish it from the semantic function for expressions, i.e., we write

$$[\![\mathsf{id}\ \texttt{:=}\ \mathsf{expr}]\!]_s(E) = \ldots$$

But what are the semantics of a statement? For an expression, the semantics are clear: we compute a number (or an error) and return it. But what does a statement compute?

Consider what we use statements for: in this language, we only use them to update variables. We represent variable assignments in environments. Thus, it is natural to have statements compute environments. With that in mind, we can finally define the semantics of statements:

$$[\![\mathsf{id}\ \texttt{:=}\ \mathsf{expr}]\!]_s(E) \quad = \quad E, \mathsf{id} \mapsto ([\![\mathsf{expr}]\!](E))$$

That is, we compute the meaning of the expression and map the variable to it. For example:

$$\begin{array}{rcll} [\![\texttt{a := 1+2}]\!](E) & = & E, \mathsf{a} \mapsto ([\![\texttt{1+2}]\!](E)) & \\ & = & E, \mathsf{a} \mapsto [\![\texttt{1}]\!](E) + [\![\texttt{2}]\!](E) & \text{(neither is } \textsc{Error}\text{)} \\ & = & E, \mathsf{a} \mapsto 1 + 2 & \\ & = & E, \mathsf{a} \mapsto 3 & \end{array}$$

For the rest, we now only need to put together the pieces. We define the semantics of statement sequences (with another new index) as

$$\begin{array}{rcll} [\![]\!]_{\overline{s}}(E) & = & E & \text{empty statement sequence} \\ [\![\mathsf{stmt}; \mathsf{stmtseq}]\!]_{\overline{s}}(E) & = & [\![\mathsf{stmtseq}]\!]_{\overline{s}}([\![\mathtt{stmt}]\!]_s(E)) & \end{array}$$

The first rule just keeps the environment that was passed in. The second rule first gives the environment to the statement, so that the statement can update the environment, and then processes the rest of the statement sequence with that updated environment.

Finally, we can define the semantics of a program:

$$[\![\mathsf{stmtseq}\ \ \mathsf{expr}]\!]_p = [\![\mathsf{expr}]\!]([\![\mathsf{stmtseq}]\!]_{\overline{s}}([]))$$

First, we compute the semantics of the statement sequence, "$[\![\mathsf{stmtseq}]\!]_{\overline{s}}([])$", with an initially empty environment. The result is some environment E which we pass into the computation of the expression, giving us "$[\![\mathsf{expr}]\!](E)$". Thus, we first evaluate the assignments in the statement sequence to compute our environment, and then we use this environment in our expressions to look up identifiers.

Here is a final example:

$$\begin{aligned}
[\![\texttt{a:=2; b:=a+1; b*b}]\!]_p &= [\![\texttt{b*b}]\!]([\![\texttt{a:=2; b:=a+1;}]\!]_{\overline{s}}([])) \\
&= [\![\texttt{b*b}]\!]([\![\texttt{b:=a+1;}]\!]_{\overline{s}}([\![\texttt{a:=2}]\!]_s([]))) \\
&= [\![\texttt{b*b}]\!]([\![\texttt{b:=a+1;}]\!]_{\overline{s}}([], \texttt{a} \mapsto [\![\texttt{2}]\!]([]))) \\
&= [\![\texttt{b*b}]\!]([\![\texttt{b:=a+1;}]\!]_{\overline{s}}([], \texttt{a} \mapsto 2)) \\
&= [\![\texttt{b*b}]\!]([\![]\!]_{\overline{s}}([\![\texttt{b:=a+1}]\!]_s([], \texttt{a} \mapsto 2))) \\
&= [\![\texttt{b*b}]\!]([\![]\!]_{\overline{s}}([], \texttt{a} \mapsto 2, \texttt{b} \mapsto [\![\texttt{a+1}]\!]([], \texttt{a} \mapsto 2))) \\
&= [\![\texttt{b*b}]\!]([\![]\!]_{\overline{s}}([], \texttt{a} \mapsto 2, \texttt{b} \mapsto [\![\texttt{a}]\!]([], \texttt{a} \mapsto 2) + [\![\texttt{1}]\!]([], \texttt{a} \mapsto 2))) \\
&= [\![\texttt{b*b}]\!]([\![]\!]_{\overline{s}}([], \texttt{a} \mapsto 2, \texttt{b} \mapsto 2 + 1)) \\
&= [\![\texttt{b*b}]\!]([\![]\!]_{\overline{s}}([], \texttt{a} \mapsto 2, \texttt{b} \mapsto 3)) \\
&= [\![\texttt{b*b}]\!]([], \texttt{a} \mapsto 2, \texttt{b} \mapsto 3) \\
&= [\![\texttt{b}]\!]([], \texttt{a} \mapsto 2, \texttt{b} \mapsto 3) * [\![\texttt{b}]\!]([], \texttt{a} \mapsto 2, \texttt{b} \mapsto 3) \\
&= 3 * 3 \\
&= 9
\end{aligned}$$