Denotational Semantics

CS 6520, Spring 2000

1 Denotations

So far in class, we have studied *operational semantics* in depth. In operational semantics, we define a language by describing the way that it behaves. In a sense, no attempt is made to attach a "meaning" to terms, outside the way that they are evaluated. For example, the symbol `'elephant` doesn't mean anything in particular within the language; it's up to a programmer to mentally associate meaning to the symbol while defining a program useful for zookeepers. Similarly, the definition of a `sort` function has no inherent meaning in the operational view outside of a particular program. Nevertheless, the programmer knows that `sort` manipulates lists in a useful way: it makes animals in a list easier for a zookeeper to find.

In *denotational semantics*, we define a language by assigning a mathematical meaning to functions; i.e., we say that each expression *denotes* a particular mathematical object. We might say, for example, that a `sort` implementation denotes the mathematical sort function, which has certain properties independent of the programming language used to implement it.

In other words, operational semantics defines evaluation by

$$sourceExpression_1 \longrightarrow sourceExpression_2$$

whereas denotational semantics defines evaluation by

$$sourceExpression_1 \longrightarrow mathematicalEntity_1 = mathematicalEntity_2 \longleftarrow sourceExpression_2$$

One advantage of this approach is that we can take advantage of existing thoeries by mapping source expressions to mathematical objects in the theory.

The denotation of expression in a language is typically defined using a structurally-recursive definition over expressions. By convention, if e is a source expression, then $[\![e]\!]$ means "the denotation of e", or "the mathematical object reprsented by e".

As an example, consider a simple language of arithmetic expressions, a:

$$\begin{array}{rcl} a & = & 0 \mid 1 \mid 2 \mid \ldots \\ & \mid & (a + a) \\ & \mid & (a - a) \end{array}$$

Naturally, the intended meaning of the expression 1 is the mathematical number one, and the intended meaning of the expression $1 + 7$ is the mathematical number eight. We can formalize this meaning through the following denotational definitions:

$$\begin{array}{rcll} [\![0]\!] & = & 0 & \\ [\![1]\!] & = & 1 & \\ & \ldots & & \\ [\![a_1 + a_2]\!] & = & [\![a_1]\!] + [\![a_2]\!] & \\ [\![a_1 - a_2]\!] & = & [\![a_1]\!] - [\![a_2]\!] & \text{if } [\![a_1]\!] > [\![a_2]\!] \\ [\![a_1 - a_2]\!] & = & 0 & \text{if } [\![a_1]\!] \leq [\![a_2]\!] \end{array}$$

In the above definitions, digits, $+$, $-$, $>$, and $=$ appearing outside $[\![\,]\!]$ are mathematical numbers, operators, and relations (the ones that we've known and loved for years). From this definition, it's clear that the meaning of the expression $(1 + 2 + 3) - 4$ is two.

We might have defined the semantics instead as

$$
\begin{array}{rcl}
[\![0]\!] & = & 2 \\
[\![1]\!] & = & 4 \\
& \ldots & \\
[\![a_1 + a_2]\!] & = & 0 \\
[\![a_1 - a_2]\!] & = & [\![a_1]\!] + [\![a_2]\!]
\end{array}
$$

This perverse definition of the language may not be especially useful. But it highlights the difference between source terms, such as 1 and $2+5$, and the denotations we can choose to assign to them. From now on we'll work for the earlier, more useful definition.

2 Functions

If our source language contains lambda abstractions, such as $\lambda x.x$, then we can assign a denotation abstractions by mapping them to mathematical functions. In other words, we will agree that $\lambda x.x$ really *means* the identity function. Furthermore, we would like to say that $\lambda y.(\lambda x.x)y$ also means the identity function, and is just another name for it.

In mathematics, a function is equivalent to a set that pairs inputs and outputs. Thus, the identity function over natural numbers is

$$\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \ldots\}$$

If I is the identity function over numbers, then we read the notation $I(0)$ as finding the pair in the set whose first part is 0, then extracting the second part of the set. The constraint that the set reprsents a function ensures that only one element of the set will start with 0.

To allow our language to represent such functions, we extend the arithmetic language with variables, abstractions, and applications:

$$
\begin{array}{rcl}
a & = & \ldots \\
& | & x \\
& | & (a\ a) \\
& | & (\lambda x.a)
\end{array}
$$

At first glance, the introduction of variables is troubling. What, for example, is the denotation of the expression $x+4$?

One possibility is to say that $x+4$ has no denotation since x is unbound. But this choice would break our natural recursive technique for assigning denotations to expressions.

Another approach, and the one that we will take here, is to fundamentally change our denotational mapping. Instead of mapping digits (e.g., 1) to mathemetical numbers (e.g., one), we will map them to mathematical functions that take an *environment* an produce a number. So the denotation of 1 will be a function that takes an environment and produces 1. Equivalently, the denotation of 1 will be a set of pairs, where each pair maps an environment to 1:

$$[\![1]\!] \quad = \quad \{\langle e, 1\rangle \mid \forall e \in \mathit{Environments}\}$$

Then, the denotation of x can be a function that takes an environment and returns the binding of x in the environment. Similarly, the denotation of $x+4$ will be the function that takes an environment and returns four more than the value associated with x in the environment.

$$
\begin{array}{rcl}
[\![0]\!] & = & \{\langle e, 0\rangle \mid \forall e\} \\
[\![1]\!] & = & \{\langle e, 1\rangle \mid \forall e\} \\
& \ldots & \\
[\![a_1 + a_2]\!] & = & \{\langle e, [\![a_1]\!]e + [\![a_2]\!]e\rangle \mid \forall e\} \\
[\![a_1 - a_2]\!] & = & \{\langle e, [\![a_1]\!]e \ominus [\![a_2]\!]e\rangle \mid \forall e\} \\
& \text{where} & n \ominus m = n - m \text{ if } n > m \\
& \text{and} & n \ominus m = 0 \text{ if } n \leq m \\
[\![(\lambda x.a)]\!] & = & \{\langle e, f\rangle \mid \forall e, f = \{\langle v, [\![a]\!]e'\rangle \mid \forall v, e' = e \cup \{(x, v)\}\}\} \\
[\![(a_1\ a_2)]\!] & = & \{\langle e, ([\![a_1]\!]e)([\![a_2]\!]e)\rangle \mid \forall e\} \\
[\![x]\!] & = & \{\langle e, v\rangle \mid \forall e, v = e(x)\}
\end{array}
$$

In the definition of $(\lambda x.a)$ we assume that a type is implicitly given for x, and the $\forall v$ on the right-hand side means $\forall v$ in the specified type. Note that this view of types is *different* from our view of types as syntactic restructions on programs! In this view, a type corresponds to a set of mathematical objects.

We can simplify the notation in our semantics, somewhat. Keeping in mind that mathematical functions are sets, we can nevertheless use λ notation to describe mathematical functions. (Thus, λ will appear both in source and mathematical worlds, just like 1 is used in both worlds.)

The above semantics, with the revised notation is

$$
\begin{array}{rcll}
[\![0]\!] & = & \lambda e.0 & \\
[\![1]\!] & = & \lambda e.1 & \\
 & \ldots & & \\
[\![a_1 + a_2]\!] & = & \lambda e.[\![a_1]\!]e + [\![a_2]\!]e & \\
[\![a_1 - a_2]\!] & = & \lambda e.[\![a_1]\!]e \ominus [\![a_2]\!]e & \\
 & \text{where} & n \ominus m = n - m \text{ if } n > m & \\
 & \text{and} & n \ominus m = 0 \text{ if } n \leq m & \\
[\![(\lambda x.a)]\!] & = & \lambda e.\lambda v.[\![a]\!]e' & \text{where } e' = e \cup \{(x,v)\} \\
[\![(a_1\ a_2)]\!] & = & \lambda e.([\![a_1]\!]e)([\![a_2]\!]e) & \\
[\![x]\!] & = & \lambda e.e(x) &
\end{array}
$$

3 Errors and Continuation Semantics

Now that we have both numbers and functions, some expressions have no denotation. For example, $(2 + (\lambda x.x))$ has no detonation by the above rules. Put another way, $[\![\,]\!]$ is a partial function.

One way to make $[\![\,]\!]$ a complete function is by adding a special error object into our mathematical set of denotations. Then, we redefine the semantics of our language as follows:

$$
\begin{array}{rcll}
[\![0]\!] & = & \lambda e.0 & \\
[\![1]\!] & = & \lambda e.1 & \\
 & \ldots & & \\
[\![a_1 + a_2]\!] & = & \lambda e.[\![a_1]\!]e \oplus [\![a_2]\!]e & \\
 & \text{where} & n \oplus m = n + m \text{ if } n \text{ and } m \text{ are numbers} & \\
 & \text{and} & n \oplus m = \mathsf{error} \text{ otherwise} & \\
[\![a_1 - a_2]\!] & = & \lambda e.[\![a_1]\!]e \ominus [\![a_2]\!]e & \\
 & \text{where} & n \ominus m = n - m \text{ if } n \text{ and } m \text{ are numbers where } n > m & \\
 & \text{and} & n \ominus m = 0 \text{ if } n \text{ and } m \text{ are numbers where } n \leq m & \\
 & \text{and} & n \ominus m = \mathsf{error} \text{ otherwise} & \\
[\![(\lambda x.a)]\!] & = & \lambda e.\lambda v.[\![a]\!]e' & \text{where } e' = e \cup \{(x,v)\} \\
[\![(a_1\ a_2)]\!] & = & \lambda e.([\![a_1]\!]e) \odot ([\![a_2]\!]e) & \\
 & \text{where} & f \odot v = f(v) \text{ if } f \text{ is a function} & \\
 & \text{and} & f \odot v = \mathsf{error} \text{ otherwise} & \\
[\![x]\!] & = & \lambda e.e \odot x & \\
 & \text{where} & e \odot x = e(x) \text{ if } x \in \mathrm{dom}(e) & \\
 & \text{and} & e \odot x = \mathsf{error} \text{ otherwise} &
\end{array}
$$

This semantics has a problem that we've seen before: errors may be ignored, due to the lack of value restrictions in our language. For example, with the above definitions, the denotation of $(\lambda y.1)(2 + (\lambda x.x))$ is one, even though the denotation of $(2 + (\lambda x.x))$ is error.

We can make the order of evaluation explicit, and therefore capture errors correctly, by using an explicit *continuation* function. This form of language definition is called a *continuation semantics*.

The basic idea behind continuation semantics has an anologue in explaining order of evaluation and errors in functional languages. For example, the program

```
(define (g y) (* y y))
(define (f x y) (+ (g x) (g y)))
(f 1 2)
```

can be re-written

```
(define (g y k) (k (* y y)))
(define (f x y k)
   (g x (lambda (xv)
          (g y (lambda (yv)
                 (k (+ xv yv)))))))
(f 1 2 (lambda (v) v))
```

Each function has been coverted, so that instead of simply returning a value to its context, it always passes the value to a continuation function. Furthermore, instead of writing nested function calls, the continuation is always made explicit: inside `f`, `g` is first applied to `x`, and once a result is obtained, `g` is applied applied to `y`. Finally, after a value is obtained for `(g y)`, the sum can be passed to the continuation given to `f`. At the very top level, `f` is called with a continuation that simply returns the result.

The importance of the transformation is that all function calls are tail calls; no context is exer accumulated during a call. This property allows `g` to return an `'error` token to the top-level call, wihout an exception system or cooperation from `f`:

```
(define (g y k) 'error)
(define (f x y k)
   (g x (lambda (xv)
          (g y (lambda (yv)
                 (k (+ xv yv)))))))
(f 1 2 (lambda (v) v))
```

In the above revision, `g` does not call the continuation function, but instead returns `'error` directly. Consequently, the application of `g` to `y` in `f` will never occur, and the final result will be `'error`.

To use this idea in our denotational semantics, we must one again change the denotation of simple constant expressions like 1. Now, the denotation of 1 will be a function that takes an environment and a continuation, and applies the continuation, a mathematical function, to the mathematical value 1.

$$
\begin{array}{rcll}
[\![0]\!] & = & \lambda e \lambda k.k(0) & \\
[\![1]\!] & = & \lambda e \lambda k.k(1) & \\
 & \ldots & & \\
[\![a_1 + a_2]\!] & = & \lambda e \lambda k.[\![a_1]\!] e(\lambda v_1.[\![a_2]\!] e(\lambda v_2.v_1 \oplus_k v_2)) & \\
 & \text{where} & n \oplus_k m = k(n+m) \text{ if } n \text{ and } m \text{ are numbers} & \\
 & \text{and} & n \oplus_k m = \mathsf{error}_+ \text{ otherwise} & \\
[\![a_1 - a_2]\!] & = & \lambda e \lambda k.[\![a_1]\!] e(\lambda v_1.[\![a_2]\!] e(\lambda v_2.v_1 \ominus_k v_2)) & \\
 & \text{where} & n \ominus_k m = k(n-m) \text{ if } n \text{ and } m \text{ are numbers where } n > m & \\
 & \text{and} & n \ominus_k m = k(0) \text{ if } n \text{ and } m \text{ are numbers where } n \leq m & \\
 & \text{and} & n \ominus_k m = \mathsf{error}_- \text{ otherwise} & \\
[\![(\lambda x.a)]\!] & = & \lambda e.\lambda k.k(\lambda v.[\![a]\!] e') & \text{where } e' = e \cup \{(x, v)\} \\
[\![(a_1\ a_2)]\!] & = & \lambda e \lambda k.[\![a_1]\!] e(\lambda v_1.[\![a_2]\!] e(\lambda v_2.v_1 \odot_k v_2)) & \\
 & \text{where} & f \odot_k v = f(v)(k) \text{ if } f \text{ is a function} & \\
 & \text{and} & f \odot_k v = \mathsf{error}_a \text{ otherwise} & \\
[\![x]\!] & = & \lambda e.\lambda k.e \odot_k x & \\
 & \text{where} & e \odot_k x = k(e(x)) \text{ if } x \in \mathrm{dom}(e) & \\
 & \text{and} & e \odot_k x = \mathsf{error}_v \text{ otherwise} &
\end{array}
$$

Note that we can also distinguish between different errors, now, with an imposed order of evaluation.