

CS256: Programming Languages and Semantics

Introduction to Denotational Semantics

<http://www.eecs.harvard.edu/~greg/cs256sp2006/>

Greg Morrisett
greg@eecs.harvard.edu
327 Maxwell Dworkin

February 10, 2006

1 Overview

Thus far, we've seen two different kinds of operational models. The big-step is generally easier to specify and work with, but it doesn't scale to many language features as we'll see later on. I already hinted that some features, such as exceptions, become very painful in a large-step setting, and other features, such as fine-grained interleaving (for modelling concurrent activity) don't work at all in a big step setting. And of course, we can't say anything about non-terminating expressions in the big-step setting. This is in large part because our *meta-language* of inference rules is too weak to accomodate this.

Next, I want to demonstrate the ideas behind denotational semantics. For the most part, it's going to look a lot like our large-step semantics. The key difference is the meta-language. We're going to use (mathematical) sets to describe the semantics of commands. In essence, we're going to compile expressions e and commands c directly into set theory. That will allow us to reason (at the level of sets) about the equivalence of two expressions or two commands.

Recall that s ranges over Stores and that Stores are functions from identifiers to integers:

$$s \in \text{Store} = \text{Id} \rightarrow \text{Integer}$$

We'll begin by giving a translation $\mathcal{E}[e]$ for expressions with type:

$$\mathcal{E} : \text{Exp} \rightarrow \mathcal{P}(\text{Store} \times Z)$$

That is, \mathcal{E} will map expressions to sets of pairs of a store (representing the input) and integer (representing the output). In practice, \mathcal{E} will actually return more than just a relation, but rather a total function from Stores to integers, but for now, let's think of it as returning a set of pairs.

The definition for \mathcal{E} is given inductively based on the abstract syntax for expressions:

$$\begin{aligned}\mathcal{E}[i] &= \{(s, i)\} \\ \mathcal{E}[x] &= \{(s, s(x))\} \\ \mathcal{E}[e_1 \oplus e_2] &= \{(s, i) \mid \exists i_1, i_2. i = i_1 \oplus i_2 \wedge (s, i_1) \in \mathcal{E}[e_1] \wedge (s, i_2) \in \mathcal{E}[e_2]\}\end{aligned}$$

It's relatively easy to establish by induction on e that $\langle e, s \rangle \Downarrow i$ iff $(s, i) \in \mathcal{E}[e]$. That is, the big-step semantics and the denotational semantics line right up.

Now let's move on to commands. The translation of a command, given by \mathcal{C} , will be a relation on stores. Like the translation of expressions, the relation will in fact be a function. For example, we ought to have the property that $\mathcal{C}[\text{skip}]$ is the identity on stores, and $\mathcal{C}[c_1; c_2]$ is something like function composition.

However, there's a problem. What function should we get out of $\mathcal{C}[\text{while } 1 \text{ do skip}]$? Intuitively, we *don't* get a store out when we run this command (in any store) because it runs forever. We can model this

in the set-theoretic setting by treating commands as *partial functions* from stores to stores. That is, we can think of the meaning of a command c as:

$$\{(s, s') \mid \langle c, s \rangle \Downarrow s'\}$$

or equivalently:

$$\{(s, s') \mid \langle c, s \rangle \mapsto^* \langle \mathbf{skip}, s' \rangle\}$$

With this kind of interpretation, $\mathcal{C}[\mathbf{while} \ 1 \ \mathbf{do} \ \mathbf{skip}]$ will be the empty relation or the no-where defined partial function. Okay, so now we know the type of \mathcal{C} . We can write this as:

$$\mathcal{C} : Exp \rightarrow \mathcal{P}(\text{Store} \times \text{Store})$$

and here's a potential definition:

$$\mathcal{C}[\mathbf{skip}] = \{(s, s)\}$$

$$\mathcal{C}[x := e] = \{(s, s[x \mapsto i]) \mid (s, i) \in \mathcal{E}[e]\}$$

$$\mathcal{C}[c_1; c_2] = \{(s_1, s_2) \mid \exists s. (s_1, s) \in \mathcal{C}[c_1] \wedge (s, s_2) \in \mathcal{C}[c_2]\}$$

$$\mathcal{C}[\mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2] = \{(s_1, s_2) \mid (s_1, i) \in \mathcal{E}[e] \wedge i \neq 0 \wedge (s_1, s_2) \in \mathcal{C}[c_1]\} \\ \cup \{(s_1, s_2) \mid (s_1, 0) \in \mathcal{E}[e] \wedge (s_1, s_2) \in \mathcal{C}[c_2]\}$$

$$\mathcal{C}[\mathbf{while} \ e \ \mathbf{do} \ c] = \mathcal{C}[\mathbf{if} \ e \ \mathbf{then} \ \{c; \mathbf{while} \ e \ \mathbf{do} \ c\} \ \mathbf{else} \ \mathbf{skip}]$$

$$= \{(s_1, s_2) \mid (s_1, i) \in \mathcal{E}[e] \wedge i \neq 0 \wedge (s_1, s_2) \in \mathcal{C}[c; \mathbf{while} \ e \ \mathbf{do} \ c]\} \\ \cup \{(s_1, s_2) \mid (s_1, 0) \in \mathcal{E}[e] \wedge (s_1, s_2) \in \mathcal{C}[\mathbf{skip}]\}$$

$$= \{(s_1, s_2) \mid (s_1, i) \in \mathcal{E}[e] \wedge i \neq 0 \wedge \exists s. (s_1, s) \in \mathcal{C}[c] \wedge (s, s_2) \in \mathcal{C}[\mathbf{while} \ e \ \mathbf{do} \ c]\} \\ \cup \{(s, s) \mid (s, 0) \in \mathcal{E}[e]\}$$

Oops. This last line isn't a definition since the meaning of the loop appears on both the left and the right. Rather, it's an equation that we would like to hold, but we need to turn it into a definition (i.e., solve for the " $\mathbf{while} \ e \ \mathbf{do} \ c$ "). The point is that there are no "recursive" definitions in set theory. We need to construct the sets in some well-founded (e.g., inductive) fashion. And note that given such an equation, it need not be the case that there's a solution, or that if there is a solution, it's unique. (Consider $x = 4/x$.)

One approach to defining the meaning of such loops is to define the meaning of the finite unwindings of the loops, and then take the union of these finite approximations to generate the meaning for the entire loop as in:

$$\mathcal{A}_0[\mathbf{while} \ e \ \mathbf{do} \ c] = \emptyset$$

$$\mathcal{A}_{i+1}[\mathbf{while} \ e \ \mathbf{do} \ c] = \{(s, s) \mid (s, 0) \in \mathcal{E}[e]\} \cup \\ \{(s_1, s_2) \mid (s_1, j) \in \mathcal{E}[e] \wedge j \neq 0 \wedge \exists s. (s_1, s) \in \mathcal{C}[c] \wedge \\ (s, s_2) \in \mathcal{A}_i[\mathbf{while} \ e \ \mathbf{do} \ c]\}$$

Then we can finally define:

$$\mathcal{C}[\mathbf{while} \ e \ \mathbf{do} \ c] = \bigcup_{i \geq 0} \mathcal{A}_i[\mathbf{while} \ e \ \mathbf{do} \ c]$$

2 Homework

These are some paper exercises due next Friday.

1. Prove that $\langle c, s_1 \rangle \Downarrow s_2$ iff $(s_1, s_2) \in \mathcal{C}[c]$.
2. Using the denotational semantics, show that the following programs are equivalent when run in states such that x and y are non-negative.

```
while x do {                while y do {
  s := s + y;                s := s + x;
  x := x - 1                  y := y - 1
}                              }
y := 0;                       x := 0;
```