

Denotational semantics

Lecture 6

Thursdays, February 11, 2016

1 Denotational semantics

We have seen two operational models for programming languages: small-step and large-step. We now consider a different semantic model, called *denotational semantics*.

The idea in denotational semantics is to express the meaning of a program as the mathematical function that expresses what the program computes. We can think of a program c as a function from stores to stores: given an initial store, the program produces a final store. For example, the program `foo := bar + 1` can be thought of as a function that when given an input store σ , produces a final store σ' that is identical to σ except that it maps `foo` to the integer $\sigma(\text{bar}) + 1$; that is, $\sigma' = \sigma[\text{foo} \mapsto \sigma(\text{bar}) + 1]$.

We are going to model programs as functions from input stores to output stores. As opposed to operational models, which tell us *how* programs execute, the denotational model shows us *what* programs compute.

For a program c (a piece of syntax), we write $\mathcal{C}\llbracket c \rrbracket$ for the *denotation* of c , that is, the mathematical function that c represents:

$$\mathcal{C}\llbracket c \rrbracket : \text{Store} \rightarrow \text{Store}.$$

Note that $\mathcal{C}\llbracket c \rrbracket$ is actually a partial function (as opposed to a total function), because the program may not terminate for certain input stores; $\mathcal{C}\llbracket c \rrbracket$ is not defined for those inputs, since they have no corresponding output stores.

We write $\mathcal{C}\llbracket c \rrbracket \sigma$ for the result of applying the function $\mathcal{C}\llbracket c \rrbracket$ to the store σ . That is, if f is the function $\mathcal{C}\llbracket c \rrbracket$, then we write $\mathcal{C}\llbracket c \rrbracket \sigma$ to mean the same thing as $f(\sigma)$.

We must also model expressions as functions, this time from stores to the values they represent. We will write $\mathcal{A}\llbracket a \rrbracket$ for the denotation of arithmetic expression a , and $\mathcal{B}\llbracket b \rrbracket$ for the denotation of boolean expression b . Note that $\mathcal{A}\llbracket a \rrbracket$ and $\mathcal{B}\llbracket b \rrbracket$ are total functions.

$$\mathcal{A}\llbracket a \rrbracket : \text{Store} \rightarrow \text{Int}$$

$$\mathcal{B}\llbracket b \rrbracket : \text{Store} \rightarrow \{\text{true}, \text{false}\}$$

Now we want to define these functions. To make it easier to write down these definitions, we will express (partial) functions as sets of pairs. More precisely, we will represent a partial map $f : A \rightarrow B$ as a set of pairs $F = \{(a, b) \mid a \in A \text{ and } b = f(a) \in B\}$ such that, for each $a \in A$, there is at most one pair of the form $(a, _)$ in the set. Hence $(a, b) \in F$ is the same as $b = f(a)$.

We can now define denotations for IMP. We start with the denotations of expressions:

$$\mathcal{A}\llbracket n \rrbracket = \{(\sigma, n)\}$$

$$\mathcal{A}\llbracket x \rrbracket = \{(\sigma, \sigma(x))\}$$

$$\mathcal{A}\llbracket a_1 + a_2 \rrbracket = \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}\llbracket a_1 \rrbracket \wedge (\sigma, n_2) \in \mathcal{A}\llbracket a_2 \rrbracket \wedge n = n_1 + n_2\}$$

$$\mathcal{A}\llbracket a_1 \times a_2 \rrbracket = \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}\llbracket a_1 \rrbracket \wedge (\sigma, n_2) \in \mathcal{A}\llbracket a_2 \rrbracket \wedge n = n_1 \times n_2\}$$

$$\mathcal{B}\llbracket \text{true} \rrbracket = \{(\sigma, \text{true})\}$$

$$\mathcal{B}\llbracket \text{false} \rrbracket = \{(\sigma, \text{false})\}$$

$$\mathcal{B}\llbracket a_1 < a_2 \rrbracket = \{(\sigma, \text{true}) \mid (\sigma, n_1) \in \mathcal{A}\llbracket a_1 \rrbracket \wedge (\sigma, n_2) \in \mathcal{A}\llbracket a_2 \rrbracket \wedge n_1 < n_2\} \cup \\ \{(\sigma, \text{false}) \mid (\sigma, n_1) \in \mathcal{A}\llbracket a_1 \rrbracket \wedge (\sigma, n_2) \in \mathcal{A}\llbracket a_2 \rrbracket \wedge n_1 \geq n_2\}$$

The denotations for commands are as follows:

$$\begin{aligned}\mathcal{C}[\mathbf{skip}] &= \{(\sigma, \sigma)\} \\ \mathcal{C}[x := a] &= \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[a]\} \\ \mathcal{C}[c_1; c_2] &= \{(\sigma, \sigma') \mid \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c_1] \wedge (\sigma'', \sigma') \in \mathcal{C}[c_2])\}\end{aligned}$$

Note that $\mathcal{C}[c_1; c_2] = \mathcal{C}[c_2] \circ \mathcal{C}[c_1]$, where \circ is the composition of relations. (Composition of relations is defined as follows: if $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$ then $R_2 \circ R_1 \subseteq A \times C$ is $R_2 \circ R_1 = \{(a, c) \mid \exists b \in B. (a, b) \in R_1 \wedge (b, c) \in R_2\}$.) If $\mathcal{C}[c_1]$ and $\mathcal{C}[c_2]$ are total functions, then \circ is function composition.

$$\begin{aligned}\mathcal{C}[\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2] &= \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_2]\} \\ \mathcal{C}[\mathbf{while } b \mathbf{ do } c] &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \wedge (\sigma'', \sigma') \in \mathcal{C}[\mathbf{while } b \mathbf{ do } c])\}\end{aligned}$$

But now we've got a problem: the last "definition" is not really a definition, it expresses $\mathcal{C}[\mathbf{while } b \mathbf{ do } c]$ in terms of itself! It is not a definition, but a recursive equation. What we want is the solution to this equation, i.e., we want to find a function f , such that f satisfies the following equation, and we will take the semantics of a while loop to be that function f .

$$\begin{aligned}f &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \wedge (\sigma'', \sigma') \in f)\}\end{aligned}$$

1.1 Fixed points

We gave a recursive equation that the function $\mathcal{C}[\mathbf{while } b \mathbf{ do } c]$ must satisfy.

To understand some of the issues involved, let's consider a simpler example. Consider the following equation for a function $f : \mathbb{N} \rightarrow \mathbb{N}$.

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases} \quad (1)$$

This is not a definition for f , but rather an equation that we want f to satisfy. What function, or functions, satisfy this equation for f ? The only solution to this equation is the function $f(x) = x^2$.

In general, there may be no solutions for a recursive equation (e.g., there are no functions $g : \mathbb{N} \rightarrow \mathbb{N}$ that satisfy the recursive equation $g(x) = g(x) + 1$), or multiple solutions (e.g., find two functions $g : \mathbb{R} \rightarrow \mathbb{R}$ that satisfy $g(x) = 4g(\frac{1}{2}x)$).

We can compute solutions to such equations by building successive approximations. Each approximation is closer and closer to the solution. To solve the recursive equation for f , we start with the partial function $f_0 = \emptyset$ (i.e., f_0 is the empty relation; it is a partial function with the empty set for its domain). We compute successive approximations using the recursive equation.

$$\begin{aligned}
f_0 &= \emptyset \\
f_1 &= \begin{cases} 0 & \text{if } x = 0 \\ f_0(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0, 0)\} \\
f_2 &= \begin{cases} 0 & \text{if } x = 0 \\ f_1(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0, 0), (1, 1)\} \\
f_3 &= \begin{cases} 0 & \text{if } x = 0 \\ f_2(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0, 0), (1, 1), (2, 4)\} \\
&\vdots
\end{aligned}$$

This sequence of successive approximations f_i gradually builds the function $f(x) = x^2$.

We can model this process of successive approximations using a higher-order function F that take one approximation f_k and returns the next approximation f_{k+1} :

$$\begin{aligned}
F &: (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \\
F(f) &= f'
\end{aligned}$$

where

$$f'(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$

A solution to the recursive equation (1) is a function f such that $f = F(f)$. In general, given a function $F : A \rightarrow A$, we have that $a \in A$ is a *fixed point* of F if $F(a) = a$. We also write $a = \text{fix}(F)$ to indicate that a is a fixed point of F .

So the solution to the recursive equation (1) is a fixed-point of the higher-order function F . We can compute this fixed point iteratively, starting with $f_0 = \emptyset$ and at each iteration computing $f_{k+1} = F(f_k)$. The fixed point is the limit of this process:

$$\begin{aligned}
f &= \text{fix}(F) \\
&= f_0 \cup f_1 \cup f_2 \cup f_3 \cup \dots \\
&= \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \dots \\
&= \bigcup_{i \geq 0} F^i(\emptyset)
\end{aligned}$$

1.2 Fixed-point semantics for loops

Returning to our original problem: we want to find $\mathcal{C}[\mathbf{while} \ b \ \mathbf{do} \ c]$, the (partial) function from stores to stores that is the denotation of the loop **while** b **do** c . We will do this by expressing $\mathcal{C}[\mathbf{while} \ b \ \mathbf{do} \ c]$ as the fixed point of a higher-order function $F_{b,c}$.

$$\begin{aligned}
F_{b,c} &: (\mathbf{Store} \rightarrow \mathbf{Store}) \rightarrow (\mathbf{Store} \rightarrow \mathbf{Store}) \\
F_{b,c}(f) &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \wedge (\sigma'', \sigma') \in f)\}
\end{aligned}$$

Compare the definition of our higher-order function $F_{b,c}$ to our recursive equation for $\mathcal{C}[\mathbf{while} \ b \ \mathbf{do} \ c]$.

The higher-order function $F_{b,c}$ takes in the partial function f , and acts like one iteration of the while loop, except that, instead of invoking itself when it needs to go around the loop again, it instead calls f .

We can now define the semantics of the while loop:

$$\begin{aligned} \mathcal{C}[\mathbf{while} \ b \ \mathbf{do} \ c] &= \bigcup_{i \geq 0} F_{b,c}^i(\emptyset) \\ &= \emptyset \cup F_{b,c}(\emptyset) \cup F_{b,c}(F_{b,c}(\emptyset)) \cup F_{b,c}(F_{b,c}(F_{b,c}(\emptyset))) \cup \dots \\ &= \text{fix}(F_{b,c}) \end{aligned}$$

Let's consider an example: **while** foo < bar **do** foo := foo + 1. Here $b = \text{foo} < \text{bar}$ and $c = \text{foo} := \text{foo} + 1$.

$$\begin{aligned} F_{b,c}(\emptyset) &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[\![b]\!]\} \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!]\} \wedge (\sigma'', \sigma') \in \emptyset)\} \\ &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \end{aligned}$$

$$\begin{aligned} F_{b,c}^2(\emptyset) &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[\![b]\!]\} \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!]\} \wedge (\sigma'', \sigma') \in F_{b,c}(\emptyset))\} \\ &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 1]) \mid \sigma(\text{foo}) < \sigma(\text{bar}) \wedge \sigma(\text{foo}) + 1 \geq \sigma(\text{bar})\} \end{aligned}$$

But if $\sigma(\text{foo}) < \sigma(\text{bar}) \wedge \sigma(\text{foo}) + 1 \geq \sigma(\text{bar})$ then $\sigma(\text{foo}) + 1 = \sigma(\text{bar})$, so we can simplify further:

$$\begin{aligned} &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 1]) \mid \sigma(\text{foo}) + 1 = \sigma(\text{bar})\} \end{aligned}$$

$$\begin{aligned} F_{b,c}^3(\emptyset) &= \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[\![b]\!]\} \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!]\} \wedge (\sigma'', \sigma') \in F_{b,c}^2(\emptyset))\} \\ &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 1]) \mid \sigma(\text{foo}) + 1 = \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 2]) \mid \sigma(\text{foo}) + 2 = \sigma(\text{bar})\} \end{aligned}$$

$$\begin{aligned} F_{b,c}^4(\emptyset) &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 1]) \mid \sigma(\text{foo}) + 1 = \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 2]) \mid \sigma(\text{foo}) + 2 = \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + 3]) \mid \sigma(\text{foo}) + 3 = \sigma(\text{bar})\} \end{aligned}$$

If we take the union of all $F_{b,c}^i(\emptyset)$, we get the expected semantics of the loop.

$$\begin{aligned} \mathcal{C}[\mathbf{while} \ \text{foo} < \text{bar} \ \mathbf{do} \ \text{foo} := \text{foo} + 1] &= \{(\sigma, \sigma) \mid \sigma(\text{foo}) \geq \sigma(\text{bar})\} \cup \\ &\quad \{(\sigma, \sigma[\text{foo} \mapsto \sigma(\text{foo}) + n]) \mid \sigma(\text{foo}) + n = \sigma(\text{bar}) \wedge n \geq 1\} \end{aligned}$$