Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Denotational semantics

Lecture 6 Tuesday, February 18, 2020

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 an initial store, the program produces a final store. For example, the program $\mathsf{foo} := \mathsf{bar} + 1$ 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 $\sigma(\mathsf{bar}) + 1$; that is, $\sigma' = \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{bar}) + 1]$.

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 $\mathcal{C}[\![c]\!]$ for the *denotation* of c, that is, the mathematical function that c represents:

$$\mathcal{C}[\![c]\!] : \textbf{Store} \rightharpoonup \textbf{Store}.$$

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

We write $\mathcal{C}[\![c]\!]\sigma$ for the result of applying the function $\mathcal{C}[\![c]\!]$ to the store σ. That is, if f is the function $\mathcal{C}[\![c]\!]$, then we write $\mathcal{C}[\![c]\!]\sigma$ 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 $\mathcal{A}[\![a]\!]$ for the denotation of arithmetic expression a, and $\mathcal{B}[\![b]\!]$ for the denotation of boolean expression b. Note that $\mathcal{A}[\![a]\!]$ and $\mathcal{B}[\![b]\!]$ are total functions.

$$\mathcal{A}[\![a]\!] : \textbf{Store} \to \textbf{Int}$$

$$\mathcal{B}[\![b]\!] : \textbf{Store} \to \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\}$$

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 $f : A \rightharpoonup B$ 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:

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n)\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x))\} \\
\mathcal{A}[\![a_1 + a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n = n_1 + n_2\} \\
\mathcal{A}[\![a_1 \times a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n = n_1 \times n_2\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![\textbf{true}]\!] &= \{(\sigma, \textbf{true})\} \\
\mathcal{B}[\![\textbf{false}]\!] &= \{(\sigma, \textbf{false})\} \\
\mathcal{B}[\![a_1 < a_2]\!] &= \{(\sigma, \textbf{true}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 < n_2\} \;\cup \\
&\qquad \{(\sigma, \textbf{false}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 \geq n_2\}
\end{aligned}$$

The denotations for commands are as follows:

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

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.

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

But now we've got a problem: the last "definition" is not really a definition, it expresses $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ 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.

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

1.1 Fixed points

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

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

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

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}:

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

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 $F : A \to A$, 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:

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

1.2 Fixed-point semantics for loops

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

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

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

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:

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

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

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

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

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

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

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

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

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

$$\mathcal{C}[\![\textbf{while}\ \mathsf{foo} < \mathsf{bar}\ \textbf{do}\ \mathsf{foo} := \mathsf{foo} + 1]\!] = \{(\sigma, \sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\}\ \cup$$
$$\{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + n]) \mid \sigma(\mathsf{foo}) + n = \sigma(\mathsf{bar}) \wedge n \geq 1\}$$

The last equality is a condensation of the union. Now we can rewrite this in the closed form:

$$\mathcal{C}[\![\textbf{while}\ \mathsf{foo} < \mathsf{bar}\ \textbf{do}\ \mathsf{foo} := \mathsf{foo} + 1]\!] = \{(\sigma, \sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\}\ \cup$$
$$\{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + n]) \mid n = \sigma(\mathsf{bar}) - \sigma(\mathsf{foo}) \wedge n \geq 1\}$$

2 Going deeper: introduction to domain theory

As we saw denotational semantics is an alternative to operational semantics, where we go straight to the mathematics (the *what*) instead of going through the operations (the *how*). The advantage is that we can discuss the meaning of programs directly as mathematical objects. As an example, we can use mathematical equality to compare denotations.

In order to answer questions such as "When does the fixed point exist?", "Is it unique if it exists?", it is helpful to introduce some basic definitions from domain theory.

A partial order is a *complete partial order* if it has a least upper bound $\bigsqcup_{n \in \omega} d_n$ in D of any ω-chain (infinite increasing chain) $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ of elements of D. This means that if we take any chain in D, there is an element d of D that is greater than any other element in the chain, and if another element c is greater than any other element in the chain, then $d \sqsubseteq c$. Some examples of cpo's are

1. A *discrete* cpo is any set ordered by the identity relation. Any chain is of the form $a \sqsubseteq a$, thus any chain has a least upper bound a.

2. The powerset $\mathcal{P}(X)$ of any set X, ordered by $\subseteq$, or by $\supseteq$, forms a cpo.

3. The set of partial functions $X \rightharpoonup Y$ ordered by inclusion, between sets X and Y, is a cpo.

4. Extending the non-negative integers ω by ∞ and ordering them in a chain

$$0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq n \sqsubseteq \ldots \infty$$

yields a cpo, called Ω.

A function $f : D \to E$ between cpo's D and E is monotonic iff

$$\forall d, d' \in D. d \sqsubseteq d' \longrightarrow f(d) \sqsubseteq f(d').$$

Such a function is *continuous* iff it is monotonic and for all chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ in D we have

$$\bigsqcup_{n \in \omega} f(d_n) = f(\bigsqcup_{n \in \omega} d_n).$$

A cpo *with bottom* is a cpo which has a least element $\bot$, that is an element which is less than every other element.

Theorem 1 (Fixed Point theorem)**.** *Let $f : D \to D$ be a continuous function on D a cpo with bottom $\bot$. Define*

$$fix(f) = \bigsqcup_{n \in \omega} f^n(\bot).$$

Then $f(fix(f)) = fix(f)$ and, if $f(d) \sqsubseteq d$ then $fix(f) \sqsubseteq d$. Consequently $fix(f)$ is the least fixed point of f.

This theorem helps us find the fixed points $0, 1$ of the squaring function $f : \omega \to \omega$, $f(n) = n^2$. We can extend f to a function on a cpo; set $h : \Omega \to \Omega$ defined by $h(n) = n^2$ on the integers and $h(\infty) = \infty$. As an exercise, try proving that this function is monotonic and continuous. The bottom $\bot$ of Ω is 0, and we have that $0 = \bigsqcup_{n \in \omega} h^n(\bot)$ is a fixed point of h, and of f. Moreover, it is the least fixed point. This makes sense because $0 \sqsubseteq 1 \sqsubseteq \infty$. If we take Ω' to be all the elements of Ω except 0, with the same ordering, then 1 becomes the bottom, and we get that $1 = \bigsqcup_{n \in \Omega} h^n(1)$ is a fixed point. Similarly, if we take Ω'' to be all elements of Ω except 0 and 1, with the same ordering, then 2 becomes the bottom, and we see that $\infty = \bigsqcup_{n \in \omega} f^n(2)$ is a fixed point. (Note that it's the least fixed point among $2, 3, 4 \ldots \in \Omega''$, but not among the elements of Ω.)

In denotational semantics, we take the union all the F^i's to get a fixed point of higher order function F precisely due to this theorem, where we set $d_0 = F^0$, $d_1 = F^1$, $d_2 = F^2$, and so on. In other words, our cpo consists of the elements F^i, and an additional element, their union. The union is then both the least fixed point (due to the theorem) and the unique fixed point (due to the fact any other candidate for a fixed point fails to be greater than the union).