CS 4110 – Programming Languages and Logics
Lecture #8: Denotational Semantics

We have now seen two operational models for programming languages: small-step and large-step. In this lecture, we 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 an IMP 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 will 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.

1 A Denotational Semantics for IMP

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

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

Note that $\mathcal{C}[\![c]\!]$ is actually a partial function (as opposed to a total function), both because the store may not be defined on the free variables of the program and because program may not terminate for certain input stores. The function $\mathcal{C}[\![c]\!]$ is not defined for non-terminating programs as they have no corresponding output stores.

We will 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 that $\mathcal{C}[\![c]\!]$ denotes, 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.

$$\mathcal{A}[\![a]\!] : \mathbf{Store} \rightharpoonup \mathbf{Int}$$
$$\mathcal{B}[\![b]\!] : \mathbf{Store} \rightharpoonup \{\textsf{true}, \textsf{false}\}$$

Now we want to define these functions. To make it easier to write down these definitions, we will describe (partial) functions using 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:

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

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

The denotations for commands are as follows:

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

Note that $\mathcal{C}[\![c_1; c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$, where $\circ$ is the composition of relations, 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.

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

But now we have a problem: the last "definition" is not really a definition because it expresses $\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!]$ in terms of itself! This is not a definition but a recursive equation. What we want is the solution to this equation.

2 Fixed points

We gave a recursive equation that the function $\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!]$ must satisfy. To understand some of the issues involved, let's consider a simpler example. Consider the following equation for a function $f : \mathbb{N} \to \mathbb{N}$.

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

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 $g : \mathbb{N} \to \mathbb{N}$ that satisfy the recursive equation $g(x) = g(x) + 1$), or multiple solutions (e.g., find two functions $g : \mathbb{R} \to \mathbb{R}$ that satisfy $g(x) = 4 \times g(\frac{x}{2})$).

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 it's 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 takes one approximation f_k and returns the next approximation f_{k+1}:

$$F : (\mathbb{N} \rightharpoonup \mathbb{N}) \to (\mathbb{N} \rightharpoonup \mathbb{N})$$

where

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

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:

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