1 Denotational Semantics

1.1 Introduction

So far we have been looking at translations from one language to another, where the target language is simpler or better understood. These are called emphdefinitional translations. Another approach to semantics, *denotational semantics*, involves translations to mathematical objects. The objects in question will be functions with well-defined extensional meaning in terms of sets. The main challenge will be getting a precise understanding of what sets these function operate over.

For example, consider the identity function $\lambda x.x$. This clearly represents some kind of function that takes any input object x to itself. But what is its domain? An even more interesting example is the function $\lambda x.\, xx$. Let's say that the domain of this function is D. Then x represents some element of D, since x is an input to the function. But in the body, x is applied to x, so x must also represent some function $D \to E$. For this to make sense, it must be possible to interpret every element of D as an element of $D \to E$. Thus there must be a function $D \to (D \to E)$.

It is conceivable that D could actually be isomorphic to the function space $D \to E$. However, this is impossible if E contains more than one element. This follows by a diagonalization argument. Let $e_0, e_1 \in E$, $e_0 \neq e_1$. For any function $f : D \to (D \to E)$, we can define $d : D \to E$ by $d = \lambda x.\, \mathsf{if}\ f\ x\ x = e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0$. Then for all x, $d\ x \neq f\ x\ x$, so $d \neq f\ x$ for any x, thus f cannot be onto.

This type of argument is called *diagonalization* because for countable sets D, the function d is constructed by arranging the values $f\ x\ y$ for $x, y \in D$ in a countable matrix and going down the diagonal, creating a function that is different from every $f\ x$ on at least one input (namely x).

	0	1	2	
f_0	$f_0\ 0$	$f_0\ 1$	$f_0\ 2$	$\ldots$
f_1	$f_1\ 0$	$f_1\ 1$	$f_1\ 2$	$\ldots$
f_2	$f_2\ 0$	$f_2\ 1$	$f_2\ 2$	$\ldots$
$\vdots$	$\vdots$			

The solution to this conundrum is that the set of *computable* functions is smaller than the set of all functions—almost all functions are not computable.

1.2 Denotational Semantics for IMP

When defining denotational semantics, we will use the notation $\lambda x \in D.e$ to indicate that the domain of the function is the set D. This will make sure we are precise in identifying the extension of functions.

Note that this is not really a type declaration. Later, we will introduce types and write them as $\lambda x : \tau.\, e$. The distinction is that types are pieces of language syntax, whereas sets are semantic objects.

The syntax of IMP was

$$\begin{array}{lcl} a & ::= & n \mid x \mid a_0 \oplus a_1 \\ b & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b_0 \wedge b_1 \mid a_0 = a_1 \mid \cdots \\ c & ::= & \mathsf{skip} \mid x := a \mid c_0; c_1 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \end{array}$$

The syntactic categories a, b, c are arithmetic expressions, Boolean expressions, and commands, respectively.

To define the denotational semantics, we will refer to *states*, which are functions $\Sigma = \mathbf{Var} \to \mathbb{Z}$.

$$\begin{array}{lcll} \mathcal{A}[\![a]\!] & \in & \Sigma \to \mathbb{Z} & \\ \mathcal{B}[\![b]\!] & \in & \Sigma \to \mathbf{2} & \text{where } \mathbf{2} = \{TRUE, FALSE\} \\ \mathcal{C}[\![c]\!] & \in & \Sigma \to ? & \end{array}$$

Intuitively, we would like the meaning of commands to be functions from states to states. Given an initial state, the function produces the final state reached by applying the command. However, there will be no such final state if the program does not terminate (e.g., $\mathsf{while\ true\ do\ skip}$). Thus the function would have to be partial. However, we can make it a total function by including a special element $\perp$ (called *bottom*) denoting nontermination. For any set S, let $S_\perp \triangleq S \cup \{\perp\}$. Then $\mathcal{C}[\![c]\!] \in \Sigma \to \Sigma_\perp$, where $\mathcal{C}[\![c]\!](\sigma) = \tau$ if c terminates in state τ on input state σ, and $\mathcal{C}[\![c]\!](\sigma) = \perp$ if c does not terminate on input state σ.

Now we can define the denotational semantics of expressions by structural induction. This induction is a little more complicated since we are defining all three functions at once. However, it is still well-founded because we only use the function value on subexpressions in the definitions. For numbers,

$$\mathcal{A}[\![n]\!] \triangleq \lambda\sigma \in \Sigma.n = \{(\sigma, n) \mid \sigma \in \Sigma\}.$$

For the remaining definitions, we use the shorthand of defining the value of the function given some $\sigma \in \Sigma$.

$$\begin{aligned}
\mathcal{A}[\![x]\!]\sigma &\triangleq \sigma(x) \\
\mathcal{A}[\![a_1 \oplus a_2]\!]\sigma &\triangleq \mathcal{A}[\![a_1]\!]\sigma \oplus \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![\mathsf{true}]\!]\sigma &\triangleq \mathit{TRUE} \\
\mathcal{B}[\![\mathsf{false}]\!]\sigma &\triangleq \mathit{FALSE} \\
\mathcal{B}[\![\neg b]\!]\sigma &\triangleq \begin{cases} \mathit{TRUE}, & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{FALSE}, \\ \mathit{FALSE}, & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{TRUE}. \end{cases}
\end{aligned}$$

We can express negation more compactly with a conditional expression:

$$\mathcal{B}[\![\neg b]\!]\sigma \triangleq \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \mathit{FALSE}\ \mathsf{else}\ \mathit{TRUE}.$$

Alternatively, we can write down the function extensionally:

$$\{(\sigma, \mathit{TRUE}) \mid \sigma \in \Sigma \wedge \neg\mathcal{B}[\![b]\!]\sigma\} \cup \{(\sigma, \mathit{FALSE}) \mid \sigma \in \Sigma \wedge \mathcal{B}[\![b]\!]\sigma\}.$$

For the commands, we can define

$$\begin{aligned}
\mathcal{C}[\![\mathsf{skip}]\!]\sigma &\triangleq \sigma \\
\mathcal{C}[\![x := a]\!]\sigma &\triangleq \sigma[\mathcal{A}[\![a]\!]\sigma/x] \\
\mathcal{C}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\sigma &\triangleq \begin{cases} \mathcal{C}[\![c_1]\!]\sigma, & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{TRUE}, \\ \mathcal{C}[\![c_2]\!]\sigma, & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{FALSE}. \end{cases}
\end{aligned}$$

For sequential composition,

$$\mathcal{C}[\![c_1; c_2]\!]\sigma \triangleq \begin{cases} \mathcal{C}[\![c_2]\!]\left(\mathcal{C}[\![c_1]\!]\sigma\right), & \text{if } \mathcal{C}[\![c_1]\!]\sigma \neq \perp, \\ \perp, & \text{if } \mathcal{C}[\![c_1]\!]\sigma = \perp. \end{cases}$$

Another way of achieving this effect is by defining a *lift* operator on functions:

$$\begin{aligned}
(\cdot)^* &: (D \to E_\perp) \to (D_\perp \to E_\perp) \\
(f)^* &\triangleq \lambda x.\mathsf{if}\ x = \perp\ \mathsf{then}\ \perp\ \mathsf{else}\ f(x).
\end{aligned}$$

With this notation, we have

$$\mathcal{C}[\![c_1; c_2]\!]\sigma \triangleq \left(\mathcal{C}[\![c_2]\!]\right)^*\left(\mathcal{C}[\![c_1]\!]\sigma\right).$$

We have one command left: $\mathsf{while}\ b\ \mathsf{do}\ c$. This is equivalent to $\mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}$, so a first guess at a denotation might be:

$$\begin{aligned}\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma &\triangleq \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \mathcal{C}[\![c; \mathbf{while}\ b\ \mathbf{do}\ c]\!]\sigma\ \mathsf{else}\ \sigma \\ &= \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!])^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma,\end{aligned}$$

but this appears to be a circular definition. However, we can fix this by taking a least fixpoint in some domain. Define

$$\mathcal{W} \triangleq \mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!].$$

Then

$$\mathcal{W} = \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (\mathcal{W})^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma.$$

Define $\mathcal{F}$ as

$$\mathcal{F} \triangleq \lambda w \in \Sigma \to \Sigma_\perp.\, \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (w)^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma.$$

Then $\mathcal{W} = \mathcal{F}\mathcal{W}$; that is, we are looking for a fixpoint of $\mathcal{F}$. But how do we take fixed points without using the dreaded Y combinator? Eventually we will have a function fix with $\mathcal{W} = \mathsf{fix}\ \mathcal{F}$, where $\mathcal{F} \in (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$. The solution will be to think of a while statement as the limit of a sequence of approximations. Intuitively, by running through the loop more and more times, we will get better and better approximations.

The first and least accurate approximation is the totally undefined function

$$\mathcal{W}_0 \triangleq \lambda\sigma \in \Sigma.\, \perp.$$

This simulates 0 iterations of the loop. The next approximation is

$$\begin{aligned}\mathcal{W}_1 &\triangleq \mathcal{F}(\mathcal{W}_0) \\ &= \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (\mathcal{W}_0)^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma \\ &= \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \perp\ \mathsf{else}\ \sigma.\end{aligned}$$

This simulates 1 iteration of the loop. We could then simulate 2 iterations by:

$$\mathcal{W}_2 \triangleq \mathcal{F}(\mathcal{W}_1) = \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (\mathcal{W}_1)^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma.$$

In general,

$$\mathcal{W}_{n+1} \triangleq \mathcal{F}(\mathcal{W}_n) = \lambda\sigma \in \Sigma.\, \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ (\mathcal{W}_n)^*(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma.$$

Then denotation of the while statement should be the limit of this sequence. But how do we take limits in spaces of functions? To do this, we need some structure on the space of functions. We will define an ordering $\sqsubseteq$ on these functions such that $\mathcal{W}_0 \sqsubseteq \mathcal{W}_1 \sqsubseteq \mathcal{W}_2 \sqsubseteq \cdots$, then find the least upper bound of this sequence.

1.3 Partial Orders on Function Spaces

Recall that a *partial order* consists of a set S and a relation $\sqsubseteq$ on S that is

- *reflexive*: for all $d \in S$, $d \sqsubseteq d$;
- *transitive*: for all $d, e, f \in S$, if $d \sqsubseteq e$ and $e \sqsubseteq f$, then $d \sqsubseteq f$; and
- *antisymmetric*: for all $d, e \in S$, if $d \sqsubseteq e$ and $e \sqsubseteq d$, then $d = e$.

Examples include $(\mathbb{Z}, \leq)$, $(\mathbb{Z}, =)$, $(\mathbb{Z}, \geq)$, $(\{\mathsf{true}, \mathsf{false}\}, \rightarrow)$, and $(2^S, \subseteq)$. If $(S, \sqsubseteq)$ is a partial order then so is $(S, \sqsupseteq)$.

We can represent a finite partial order visually by drawing a *Hasse diagram*. Draw each element as a point, with the point d_2 drawn above the point d_1 iff $d_1 \sqsubseteq d_2$. Finally, draw a line connecting any two elements if the relation between them is not implied by reflexivity or transitivity.

Given any partial order $(S, \sqsubseteq)$, we can define a new partial order $(S_\perp, \sqsubseteq_\perp)$ such that $d_1 \sqsubseteq_\perp d_2$ if $d_1, d_2 \in S$ and $d_1 \sqsubseteq d_2$, and $\perp \sqsubseteq_\perp d$ for all $d \in S_\perp$.

Thus if S is any set, then $S_\perp$ is that set with a new least element $\perp$ added. In our semantic domains, we can think of $\sqsubseteq$ as "less information than". Thus nontermination $\perp$ contains less information than any element of S.