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

The idea of denotational semantics is to translate the program to a mathematical object that represents what it computes. The objects are generally functions are relations with well-defined extensional meanings in terms of sets. That is, we are taking the *intensional* representation of a computation that is a program in the language, and we are giving it an *extensional* meaning as a mathematical function. The main challenge is getting a precise understanding of the meaning of the sets over which these functions or relations operate.

1 Denotational Semantics for Imp

To define a denotational semantics for Imp, we are faced with the same situation as with an operational semantics: there are three different categories of Imp terms (**AExp**, **BExp**, and **Com**), and we need a different semantics for each. As a reminder, the BNF grammar for Imp is as follows.

$$\begin{array}{rrcl}
\textbf{AExp}: & a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
\textbf{BExp}: & b & ::= & \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \\
\textbf{Com}: & c & ::= & \mathsf{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c
\end{array}$$

For each category of term, we will define a separate denotational semantics that maps that term into a mathematical function representing its meaning. Since the meaning of an Imp program is dependent on the environment in which it is run (here the store), these functions will have the following types.

$$\mathcal{A}[\![a]\!] : \textbf{Store} \to \mathbb{Z} \qquad \mathcal{B}[\![b]\!] : \textbf{Store} \to \mathbb{2} \qquad \mathcal{C}[\![c]\!] : \textbf{Store} \rightharpoonup \textbf{Store}$$

Note that, as with the big-step semantics, the denotational semantics for arithmetic and boolean expressions both produce total functions, while the semantics for commands produces a partial function.

1.1 Arithmetic and Boolean Expressions

We can define the denotational semantics for arithmetic and boolean expressions by structural induction as follows.

$$\mathcal{A}[\![n]\!]\,\sigma \triangleq n \qquad \mathcal{A}[\![x]\!]\,\sigma \triangleq \sigma(x) \qquad \mathcal{A}[\![a_1 \otimes a_2]\!]\,\sigma \triangleq \mathcal{A}[\![a_1]\!]\,\sigma \otimes \mathcal{A}[\![a_2]\!]\,\sigma \text{ (where } \otimes \in \{+, *, -\})$$

$$\mathcal{B}[\![\mathsf{true}]\!]\,\sigma \triangleq \mathit{true} \qquad \mathcal{B}[\![\mathsf{false}]\!]\,\sigma \triangleq \mathit{false} \qquad \mathcal{B}[\![\neg b]\!]\,\sigma \triangleq \begin{cases} \mathit{true} & \text{if } \mathcal{B}[\![b]\!]\,\sigma = \mathit{false} \\ \mathit{false} & \text{if } \mathcal{B}[\![b]\!]\,\sigma = \mathit{true} \end{cases}$$

$$\mathcal{B}[\![a_1 \sim a_2]\!]\,\sigma \triangleq (\mathcal{A}[\![a_1]\!]\,\sigma) \sim (\mathcal{A}[\![a_2]\!]\,\sigma) \text{ (where } \sim \in \{=, \leq\})$$

$$\mathcal{B}[\![b_1 \odot b_2]\!]\,\sigma \triangleq (\mathcal{B}[\![b_1]\!]\,\sigma) \odot (\mathcal{B}[\![b_1]\!]\,\sigma) \text{ (where } \odot \in \{\wedge, \vee\})$$

Note that by a slight but convenient abuse, we are overloading the metasymbols in $\otimes$, $\sim$, and $\odot$ for three of the rules. The symbol on the left side represents the syntactic object in the Imp language, while the symbol on the right side represents a semantic object, namely a mathematical operation on integers or booleans. We could also streamline several of the boolean rules in a similar way. For instance, we could define $\mathcal{B}[\![\neg b]\!]$ by

$$\mathcal{B}[\![\neg b]\!]\,\sigma \ \triangleq\ \mathit{if}\ (\mathcal{B}[\![b]\!]\,\sigma)\ \mathit{then\ false\ else\ true} = \neg(\mathcal{B}[\![b]\!]\,\sigma)$$

1.2 Commands

For a command c, the function $\mathcal{C}[\![c]\!]$ should take an initial state and produce the final state reached by applying c. However, if the computation does not halt, there is no final state! This is why the function is partial. If we want to make it total, we can add a special element $\bot$ (called "bottom") to the codomain that indicates nontermination. For any set S, let $S_\bot = S \cup \{\bot\}$. This is called a *pointed set*.

Then we can regard $\mathcal{C}[\![c]\!]$ as a total function $\mathcal{C}[\![c]\!] : \textbf{Store} \to \textbf{Store}_\bot$ where $\mathcal{C}[\![c]\!]\,\sigma = \sigma'$ if c terminates with final store σ' on input σ, and $\mathcal{C}[\![c]\!]\,\sigma = \bot$ if c diverges with initial store σ.

Using that notation, we can define most of the rules recursively as follows.

$$\begin{aligned}
\mathcal{C}[\![\mathsf{skip}]\!]\,\sigma &\triangleq \sigma \\
\mathcal{C}[\![x := a]\!]\,\sigma &\triangleq \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma] \\
\mathcal{C}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\,\sigma &\triangleq \begin{cases} \mathcal{C}[\![c_1]\!]\,\sigma & \text{if } \mathcal{B}[\![b]\!]\,\sigma = \mathit{true} \\ \mathcal{C}[\![c_2]\!]\,\sigma & \text{if } \mathcal{B}[\![b]\!]\,\sigma = \mathit{false} \end{cases} \\
&= \mathit{if}\ \mathcal{B}[\![b]\!]\,\sigma\ \mathit{then}\ \mathcal{C}[\![c_1]\!]\,\sigma\ \mathit{else}\ \mathcal{C}[\![c_2]\!]\,\sigma \\
\mathcal{C}[\![c_1\,;\,c_2]\!]\,\sigma &\triangleq \begin{cases} \mathcal{C}[\![c_2]\!]\,(\mathcal{C}[\![c_1]\!]\,\sigma) & \text{if } \mathcal{C}[\![c_1]\!]\,\sigma \neq \bot \\ \bot & \text{if } \mathcal{C}[\![c_1]\!]\,\sigma = \bot \end{cases} \\
&= \mathit{if}\ \mathcal{C}[\![c_1]\!]\,\sigma = \bot\ \mathit{then}\ \bot\ \mathit{else}\ \mathcal{C}[\![c_2]\!]\,(\mathcal{C}[\![c_1]\!]\,\sigma)
\end{aligned}$$

For the last case involving sequential composition $c_1\,;\,c_2$, another way to achieve this effect is by defining a *lifting* operator $(\cdot)^\dagger : (D \to E_\bot) \to (D_\bot \to E_\bot)$ on functions that maps $\bot$ to bot and otherwise applies the original function. That is,

$$f^\dagger(x) \triangleq \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f(x).$$

This notation allows us to simplify the definition of $\mathcal{C}[\![c_1\,;\,c_2]\!]\,\sigma \triangleq \mathcal{C}[\![c_2]\!]^\dagger\,(\mathcal{C}[\![c_1]\!]\,\sigma)$. Or, equivalently,

$$\mathcal{C}[\![c_1\,;\,c_2]\!] \triangleq \mathcal{C}[\![c_2]\!]^\dagger \circ \mathcal{C}[\![c_1]\!]$$

where $f \circ g$ is standard function composition.

We have one command left: $\mathsf{while}\ b\ \mathsf{do}\ c$. Recalling the small-step operational semantics from before, this is semantically equivalent to $\mathsf{if}\ b\ \mathsf{then}\ (c\,;\,\mathsf{while}\ b\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}$, so we might hope the definition would be

$$\begin{aligned}
\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\,\sigma &= \mathit{if}\ \mathcal{B}[\![b]\!]\,\sigma\ \mathit{then}\ \mathcal{C}[\![c\,;\,\mathsf{while}\ b\ \mathsf{do}\ c]\!]\,\sigma\ \mathit{else}\ \sigma \\
&= \mathit{if}\ \mathcal{B}[\![b]\!]\,\sigma\ \mathit{then}\ \mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]^\dagger(\mathcal{C}[\![c]\!]\,\sigma)\ \mathit{else}\ \sigma.
\end{aligned} \tag{1}$$

Unfortunately, this definition is circular. It isn't merely recursive—defining the semantics of a command with respect to its subterms—it attempts to define the semantics of $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$ in terms of $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$, which is not valid. The big-step semantics in the previous lecture did not suffer from this problem because it was an inductively-defined relation, and we relied on the well-founded nature of the derivation trees. Here we are trying to define a function, so we need another way to solve the circularity.

If we take (1) as an equation, rather than a definition, however, then what this says is we need to find a function W such that, for every store σ,

$$W\,\sigma = \mathit{if}\ \mathcal{B}[\![b]\!]\,\sigma\ \mathit{then}\ W^\dagger(\mathcal{C}[\![c]\!]\,\sigma)\ \mathit{else}\ \sigma. \tag{2}$$

To find such a function, let us define a function $\mathcal{F} : (\textbf{Store} \to \textbf{Store}_\bot) \to (\textbf{Store} \to \textbf{Store}_\bot)$ that loosely represents "one iteration" of the loop:

$$\mathcal{F}\ w\ \sigma \triangleq \mathit{if}\ \mathcal{B}[\![b]\!]\,\sigma\ \mathit{then}\ w^\dagger(\mathcal{C}[\![c]\!]\,\sigma)\ \mathit{else}\ \sigma$$

Now we can simply say that we need to find a W such that $\mathcal{F}\ W = W$. That is, we are looking for a *fixed point* of $\mathcal{F}$. By how do we take a fixed point of $\mathcal{F}$? The solution is 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 get better and better approximations.

The first, and least accurate, approximations is the totally undefined function:

$$W_0\ \sigma\ \triangleq\ \bot.$$

This function gives the right answer for nonterminating programs, but is wrong for every terminating program.

The next approximation will be to apply $\mathcal{F}$ and "run the loop" once. That is,

$$\begin{aligned} W_1 &\triangleq \mathcal{F}\ W_0 \\ &= \textit{if } \mathcal{B}[\![b]\!]\ \sigma \textit{ then } W_0^\dagger(\mathcal{C}[\![c]\!]\ \sigma) \textit{ else } \sigma \\ &= \textit{if } \mathcal{B}[\![b]\!]\ \sigma \textit{ then } \bot \textit{ else } \sigma. \end{aligned}$$

This improved approximation gives the correct answer both for nonterminating programs and for while loops where the condition is immediately false, so the body of the loop never runs. That is, loops where the guard is evaluated only once. We appear to be getting closer! By applying $\mathcal{F}$ again, we can get closer.

$$W_2\ \triangleq\ \mathcal{F}\ W_1\ =\ \textit{if } \mathcal{B}[\![b]\!]\ \sigma \textit{ then } W_1^\dagger(\mathcal{C}[\![c]\!]\ \sigma) \textit{ else } \sigma$$

This approximation will also be correct for a program that evaluates the guard at most twice before terminating. In general, we can define

$$W_{n+1}\ \triangleq\ \mathcal{F}\ W_n\ =\ \textit{if } \mathcal{B}[\![b]\!]\ \sigma \textit{ then } W_n^\dagger(\mathcal{C}[\![c]\!]\ \sigma) \textit{ else } \sigma$$

and know that W_n will provide the correct answer for both nonterminating programs (it will return $\bot$), and for any loop that check the guard of the loop at most n times before terminating.

The denotation of the while loop is then the limit of this sequence. But how do we take limits on spaces of functions? To do this, we need some structure on the functions. We will define an ordering $\sqsubseteq$ on the functions such that $W_0 \sqsubseteq W_1 \sqsubseteq W_2 \sqsubseteq \cdots$, and then find the *least upper bound* (or *supremum*) of this sequence. That is, the smallest function W—according to our ordering—such that $W_i \sqsubseteq W$ for every $i \geq 0$. That will be the solution to equation (2).

To show that this least upper bound—and therefore a fixed point of $\mathcal{F}$—exists, we need to apply something called the Knaster–Tarski theorem, which we will not have time to cover. For the theorem to apply, we need that the function space $\mathbf{Store} \to \mathbf{Store}_\bot$ has structure that makes it a *chain-complete partial order* (CPO) and that $\mathcal{F}$ is a continuous map on that on this space—it preserves suprema.

To define the ordering $\sqsubseteq$, we first define an ordering on the underlying pointed set of stores. The ordering we use is known as the *flat ordering* on a pointed set $S_\bot$. The flat ordering says $\bot$ is less than everything ($\forall s \in S.\ \bot \sqsubseteq s$), but all other elements are independent (if $s_1 \neq s_2$, then they are unrelated). We can extend this to function point-wise. That is, for $f, g : D \to S_\bot$, we say $f \sqsubseteq g$ if, for all $d \in D$, $f(d) \sqsubseteq g(d)$. This ordering on the function space forms a CPO, and taking D and S to both be **Store**, we can see that $W_n \sqsubseteq W_{n+1}$ for all $n \geq 0$. This notably requires that if $W_n(\sigma) = \sigma' \neq \bot$, then $W_m(\sigma) = \sigma'$ for all $m \geq n$. This ordering property means the W_n's form a chain, so $W = \bigsqcup_{n=0}^{\infty} W_n$ gives the least fixed point of $\mathcal{F}$.

Also note that

$$W(\sigma) = \begin{cases} \bot & \text{if } \forall n.\ W_n(\sigma) = \bot \\ \sigma' & \text{if } \exists n.\ W_n(\sigma) = \sigma'. \end{cases}$$

By the property above about the stability of $W_n(\sigma)$, this is well-defined.