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 Structure of Denotational Semantics

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.

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

For each category of term, we will define a separate denotational semantics that maps that term into a mathematical function representing its meaning. The notation $[\![\cdot]\!]$ is common for denotational semantics, so we will use $\mathcal{A}[\![\cdot]\!]$, $\mathcal{B}[\![\cdot]\!]$, and $C[\![\cdot]\!]$ for our three functions. Since the meaning of an Imp program is dependent on the environment in which it is run (here the store), each will be a function from **Store** to something.

To simplify presentation, we change the type of stores to map variables to mathematical integers, rather than integer symbols. That is, $\textbf{Store} \triangleq \textbf{Var} \to \mathbb{Z}$ for this semantics. We also let $\mathbb{2} = \{\mathit{true}, \mathit{false}\}$ be the set of mathematical boolean values, distinct from the symbols true and false, just as $\mathbb{Z}$ is the mathematical counterpart to **Int**. The denotation functions then have the following types.

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

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.

2 Arithmetic and Boolean Expressions

We can define the denotational semantics for arithmetic and boolean expressions by structural recursion. To avoid repeating nearly-identical rules, we use the metasymbols $\otimes \in \{+, *, -\}$, $\sim \in \{=, \leq\}$, and $\odot \in \{\wedge, \vee\}$.

$$\begin{aligned}
\mathcal{A}[\![\overline{n}]\!]\,\sigma &\triangleq n & \qquad \mathcal{B}[\![\mathsf{true}]\!]\,\sigma &\triangleq \mathit{true} \\
\mathcal{A}[\![x]\!]\,\sigma &\triangleq \sigma(x) & \qquad \mathcal{B}[\![\mathsf{false}]\!]\,\sigma &\triangleq \mathit{false} \\
\mathcal{A}[\![a_1 \otimes a_2]\!]\,\sigma &\triangleq \mathcal{A}[\![a_1]\!]\,\sigma \otimes \mathcal{A}[\![a_2]\!]\,\sigma & \qquad \mathcal{B}[\![\neg b]\!]\,\sigma &\triangleq \neg(\mathcal{B}[\![b]\!]\,\sigma) \\
& & \qquad \mathcal{B}[\![a_1 \sim a_2]\!]\,\sigma &\triangleq (\mathcal{A}[\![a_1]\!]\,\sigma) \sim (\mathcal{A}[\![a_2]\!]\,\sigma) \\
& & \qquad \mathcal{B}[\![b_1 \odot b_2]\!]\,\sigma &\triangleq (\mathcal{B}[\![b_1]\!]\,\sigma) \odot (\mathcal{B}[\![b_1]\!]\,\sigma)
\end{aligned}$$

As in previous semantics, we allow the slight, but convenient, abuse of notation in a few cases and overload the metasymbols $\otimes$, $\sim$ and $\odot$ as well as the symbol $\neg$. 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.

3 Commands

For a command c, the function $C[\![c]\!]$ 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. To simplify some of the analysis, it is easier to work with total functions, so we will 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*. Now we can regard $C[\![c]\!]$ as a total function $C[\![c]\!] : \mathbf{Store} \to \mathbf{Store}_\bot$ where $C[\![c]\!]\,\sigma = \sigma'$ if c terminates with final store σ' on input σ, and $C[\![c]\!]\,\sigma = \bot$ if c diverges with initial store σ. Using this pointed set of stores, we can define most of the rules recursively.

Non-Looping Commands. For non-looping commands, the denotational semantics is defined by a straightforward recursion on the command as follows.

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

Note that for conditionals and sequencing, the italic *if-then-else* is not the Imp symbols, but the mathematical construct equivalently specified using a piecewise function.

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 $C[\![c_1\ ;\ c_2]\!]\,\sigma \triangleq C[\![c_2]\!]^\dagger\,(C[\![c_1]\!]\,\sigma)$. Or, equivalently,

$$C[\![c_1\ ;\ c_2]\!] \triangleq C[\![c_2]\!]^\dagger \circ C[\![c_1]\!]$$

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

While Loops. 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

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

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 $C[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$ in terms of $C[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$, 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.

To untangle this knot, we can take (1) as an *equation* that the function must satisfy, rather than a definition. That is, to define $C[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$, we need to build some function W such that, for every store σ,

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

To find such a W, we first define another function $\mathcal{F} : (\mathbf{Store} \to \mathbf{Store}_{\perp}) \to (\mathbf{Store} \to \mathbf{Store}_{\perp})$ that transforms denotations and loosely represents "one iteration" of the loop:

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

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:

$$W_0\ \sigma\ \triangleq\ \perp.$$

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,

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

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.

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

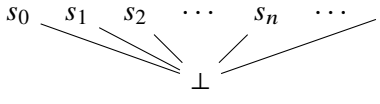
This approximation will also be correct for a program that evaluates the guard at most twice before terminating. In general, we can define

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

and know that W_n will provide the correct answer for both nonterminating programs (it will return $\perp$), and for any loop that checks its guard 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).

Defining the ordering $\sqsubseteq$ is where our use of $\perp$ and pointed sets comes in. The ordering we use is known as the *flat ordering* on a pointed set $S_{\perp}$. The flat ordering is a reflexive ordering ($\forall s \in S_{\perp}.\ s \sqsubseteq s$) that says $\perp$ is less than everything ($\forall s \in S_{\perp}.\ \perp \sqsubseteq s$), but all other elements are independent (if $s_1 \neq s_2$, then they are unrelated). Visually, the ordering appears as follows.



We can extend $\sqsubseteq$ to function point-wise. That is, for functions $f, g : D \to S_\perp$,

$$f \sqsubseteq g \iff \forall d \in D.\, f(d) \sqsubseteq g(d).$$

This ordering on the function space forms something called a *chain-complete partial order* (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 $W_n(\sigma) = \sigma' \neq \perp$, then $W_m(\sigma) = \sigma'$ for all $m \geq n$. This ordering property means the W_n's form a chain, so the Knaster–Tarski theorem—mentioned in Lecture 3 but not proven in this course—applies and proves that $W = \bigsqcup_{n=0}^{\infty} W_n$ is the least fixed point of $\mathcal{F}$. The resulting function is equivalently defined by

$$W\ \sigma = \begin{cases} \perp & \text{if } \forall n.\, W_n\ \sigma = \perp \\ \sigma' & \text{if } \exists n.\, W_n\ \sigma = \sigma'. \end{cases}$$

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

Note that this entire construction is very similar to the "bottom-up" approach to building a fixed point of the rule operator R in Lecture 3. These are, indeed, deeply connected to each other and both are special cases of the Knaster–Tarski theorem.