# 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 translations are called *definitional translations*. The target of the translation is a target-language AST. Another translation-based approach is *denotational semantics*, in which the target of the translation is mathematical objects that more directly capture the computational behavior. The objects in question will be functions with well-defined extensional meaning in terms of sets. Recall that a mathematical function $f$ can be viewed as a set of pairs $(x, y)$ in which each $x$ occurs only once and for each such pair $y = f(x)$. A central challenge will be to understand precisely 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 self-application function $\lambda x.\, x\ x$. 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 $d = \lambda x.$ if $f\ x\ x = e_0$ then $e_1$ 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

Therefore, in constructing our denotational semantics, we will be careful only to write down functions that are well-defined, operating over well-defined domains and codomains. We will write $\lambda x \in D\,.\,e$ to represent a function from domain $D$ to the codomain of $e$. Then we can be sure that the function has an extensional representation as a set of pairs.

Note that this notation functions somewhat like a type declaration, but types are language syntax, whereas $D$ here denotes a set, a mathematical object. When we later introduce types and write type declarations like $\lambda x{:}\,\tau.\,e$, we view this as just syntax rather than defining a mathematical function.

Recall that the syntax of IMP is:

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

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 stores, functions $\Sigma = \mathbf{Var} \to \mathbb{Z}$.

$$\begin{array}{rcll} \mathcal{A}[\![a]\!] & \in & \Sigma \to \mathbb{Z} & \\ \mathcal{B}[\![b]\!] & \in & \Sigma \to \mathbb{B} & \text{where } \mathbb{B} = \{true, false\} \\ \mathcal{C}[\![c]\!] & \in & \Sigma \to ? & \end{array}$$

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., **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 $S_\perp \stackrel{\triangle}{=} \{\lfloor x \rfloor \mid x \in S\} \cup \{\perp\}$. The element $\lfloor x \rfloor$ is the injection of $x$ into $S_\perp$. It is useful to distinguish between $x$ and $\lfloor x \rfloor$, for example if $x = \perp$. Then $\mathcal{C}[\![c]\!] \in \Sigma \to \Sigma_\perp$, where $\mathcal{C}[\![c]\!](\sigma) = \lfloor \sigma' \rfloor$ if $c$ terminates in state $\sigma'$ on input state $\sigma$, and $\mathcal{C}[\![c]\!](\sigma) = \perp$ if $c$ does not terminate on input state $\sigma$.

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,

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

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

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

We can express negation more compactly with a conditional expression:

$$\mathcal{B}[\![\neg b]\!]\sigma \quad = \quad \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } false \text{ else } true.$$

Alternatively, we can write down the function extensionally:

$$\{(\sigma, true) \mid \sigma \in \Sigma \wedge \neg\mathcal{B}[\![b]\!]\sigma\} \quad \cup \quad \{(\sigma, false) \mid \sigma \in \Sigma \wedge \mathcal{B}[\![b]\!]\sigma\}.$$

For the commands, we can define

$$\begin{array}{rcl} \mathcal{C}[\![\mathsf{skip}]\!]\sigma & = & \lfloor \sigma \rfloor \\ \mathcal{C}[\![x := a]\!]\sigma & = & \lfloor \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \rfloor \\ \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\sigma & = & \begin{cases} \mathcal{C}[\![c_1]\!]\sigma, & \text{if } \mathcal{B}[\![b]\!]\sigma = true, \\ \mathcal{C}[\![c_2]\!]\sigma, & \text{if } \mathcal{B}[\![b]\!]\sigma = false. \end{cases} \end{array}$$

For sequential composition,

$$\mathcal{C}[\![c_1; c_2]\!]\sigma \quad = \quad \begin{cases} \mathcal{C}[\![c_2]\!]\sigma', & \text{if } \mathcal{C}[\![c_1]\!] = \lfloor \sigma' \rfloor \\ \perp, & \text{if } \mathcal{C}[\![c_1]\!]\sigma = \perp. \end{cases}$$

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

$$\begin{array}{rcl}(\cdot)^* & : & (D \to E_\perp) \to (D_\perp \to E_\perp) \\ (f)^* & \triangleq & \lambda x \in \Sigma_\perp. \begin{cases} \perp, & \text{if } x = \perp \\ f(\sigma), & \text{if } x = \lfloor \sigma \rfloor \end{cases}\end{array}$$

With this notation, we have

$$\mathcal{C}[\![c_1; c_2]\!]\sigma \quad = \quad (\mathcal{C}[\![c_2]\!])^* \, (\mathcal{C}[\![c_1]\!]\sigma) \, .$$

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

$$\begin{array}{rcll}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma & = & \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \mathcal{C}[\![c; \textbf{while } b \textbf{ do } c]\!]\sigma \text{ else } \sigma & (1) \\ & = & \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } (\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } \sigma & (2)\end{array}$$

but (2) is a circular definition: an equation that we expect the denotation of **while** to satisfy. We can see this more clearly by defining:

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

Then we can write (2) as follows:

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

Define $\mathcal{F}$ as

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

We can write (2) simply as $W = \mathcal{F}(W)$. In other words, we are looking for a fixed point of $\mathcal{F}$. Our current technology for finding fixed points is to use the $Y$ combinator, but this is not a well-defined function, because it uses self-application.

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 function that never terminates.

$$W_0 \quad \triangleq \quad \lambda\sigma \in \Sigma. \perp.$$

This simulates 0 iterations of the loop. It's the denotation for $\mathcal{C}[\![\textbf{while true do skip}]\!]$, but not for **while** loops that can terminate. To get the next approximation, we apply $\mathcal{F}$ to the previous one:

$$\begin{array}{rcl}W_1 & \triangleq & \mathcal{F}(W_0) \\ & = & \lambda\sigma \in \Sigma. \text{ if } \mathcal{B}[\![b]\!]\sigma \text{ then } W_0^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } \sigma \\ & = & \lambda\sigma \in \Sigma. \text{ if } \mathcal{B}[\![b]\!]\sigma \text{ then } \perp \text{ else } \sigma.\end{array}$$

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

$$W_2 \quad \triangleq \quad \mathcal{F}(W_1) \quad = \quad \lambda\sigma \in \Sigma. \text{ if } \mathcal{B}[\![b]\!]\sigma \text{ then } (W_1)^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } \sigma.$$

In general,

$$W_{n+1} \quad \triangleq \quad \mathcal{F}(W_n) \quad = \quad \lambda\sigma \in \Sigma. \text{ if } \mathcal{B}[\![b]\!]\sigma \text{ then } W_n^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } \sigma.$$

The denotation $W_n$ represents the behavior of the loop correctly as long as the loop guard $b$ is evaluated no more than $n$ times. Intuitively, the denotation of the while statement is a limit of this sequence. But how do we take limits in a space of functions? We need more structure on the space of functions. We will define an ordering $\sqsubseteq$ on these functions such that $W_0 \sqsubseteq W_1 \sqsubseteq W_2 \sqsubseteq \ldots$, then find the least upper bound of this sequence.

### 1.3 Partial Orders

A *partial order* (also known as a *partially ordered set* or *poset*) is a pair $(S, \sqsubseteq)$, where

- $S$ is a set of elements.
- $\sqsubseteq$ is a relation on $S$ which is:
  - *i.* reflexive: $x \sqsubseteq x$
  - *ii.* transitive: $(x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$
  - *iii.* antisymmetric: $(x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$

**Examples:**

- $(\mathbb{Z} \leq)$, where $\mathbb{Z}$ is the integers and $\leq$ is the usual ordering.
- $(\mathbb{Z}, =)$ (Note that unequal elements are incomparable in this order. Partial orders ordered by the identity relation, $=$, are called *discrete*.)
- $(2^S, \subseteq)$ (Here, $2^S$ denotes the powerset of $S$, the set of all subsets of $S$, often written $\mathcal{P}(S)$, and in Winskel, $\mathcal{P}ow(S)$.)
- $(2^S, \supseteq)$
- $(S, \sqsupseteq)$, if we are given that $(S, \sqsubseteq)$ is a partial order.
- $(\omega, |)$, where $\omega = \{0, 1, 2, \ldots\}$ and $a|b \Leftrightarrow (a \text{ divides } b) \Leftrightarrow (b = ka \text{ for some } k \in \omega)$. Note that for any $n \in \omega$, we have $n|0$; we call 0 an upper bound for $\omega$ (but only in this ordering, of course!).

**Non-examples:**

- $(\mathbb{Z}, <)$ is not a partial order, because $<$ is not reflexive.
- $(\mathbb{Z}, \sqsubseteq)$, where $m \sqsubseteq n \Leftrightarrow |m| \leq |n|$, is not a partial order because $\sqsubseteq$ is not anti-symmetric: $-1 \sqsubseteq 1$ and $1 \sqsubseteq -1$, but $-1 \neq 1$.

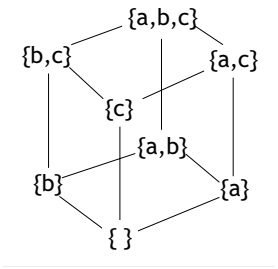
The "partial" in partial order comes from the fact that our definition does not require these orders to be total; e.g., in the partial order $(2^{\{a,b\}}, \subseteq)$, the elements $\{a\}$ and $\{b\}$ are incomparable: neither $\{a\} \subseteq \{b\}$ nor $\{b\} \subseteq \{a\}$ hold.

**Hasse diagrams** Partial orders can be described pictorially using *Hasse diagrams*[1]. In a Hasse diagram, each element of the partial order is displayed as a (possibly labeled) point, and lines are drawn between these points, according to these rules:

1. If $x$ and $y$ are elements of the partial order, and $x \sqsubseteq y$, then the point corresponding to $x$ is drawn lower in the diagram than the point corresponding to $y$.
2. A line is drawn between the points representing two elements $x$ and $y$ *iff* $x \sqsubseteq y$ and $\neg\exists z$ in the partial order, distinct from $x$ and $y$, such that $x \sqsubseteq z$ and $z \sqsubseteq y$ (i.e., the ordering relation between $x$ and $y$ is not due to transitivity).

An example of a Hasse diagram for the partial order on the set $2^{\{a,b,c\}}$ using $\subseteq$ as the binary relation is:

[1]Named after Helmut Hasse, 1898–1979. Hasse published fundamental results in algebraic number theory, including the Hasse (or "local-global") principle. He succeeded Hilbert and Weyl as the chair of the Mathematical Institute at Göttingen.




A partial order like $(\mathbb{Z}, =)$ is called a *discrete* partial order. No elements are related to each other.

Given any partial order $(S, \sqsubseteq)$, we can define a new partial order $(S_\perp, \sqsubseteq_\perp)$ such that $\lfloor d_1 \rfloor \sqsubseteq_\perp \lfloor d_2 \rfloor$ if $d_1, d_2 \in S$ and $d_1 \sqsubseteq d_2$, and $\perp \sqsubseteq_\perp \lfloor d \rfloor$ for all $d \in S$. Thus if $S$ is any set, then $S_\perp$ is that set with a new least element $\perp$ added. In our semantic domains, we think of $\sqsubseteq$ as "contains less information than". Thus nontermination $\perp$ contains less information than any element of $S$.

If we lift a discrete partial order (e.g., $\mathbb{Z}_\perp$), we get a *flat partial order*. The only relationships among different elements are between $\perp$ and each other element. Flat partial orders turn out to be useful.

If a partial order has a least element, that partial order is *pointed*. All lifted partial orders, including flat partial orders, are pointed.