## 1 What is Denotational Semantics?

So far we have looked at

- *operational semantics* involving rules for state transitions,
- *definitional semantics* involving translations from a source language to a target language, where the target language is simpler or better understood, and
- *axiomatic semantics* involving logical rules for deriving relations between preconditions and postconditions.

Another approach, known as *denotational semantics*, involves translations to mathematical objects. The objects in question are generally functions or relations with well-defined extensional meaning in terms of sets. The main challenge is getting a precise understanding of the sets over which these function and relations operate.

## 2 Diagonalization

For example, consider the $\lambda$-term $\lambda x.x$. This clearly represents the identity function that takes the input object $x$ to itself. But what is its domain? A 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 the function space $D \to E$. Thus there must be a function $f : D \to (D \to E)$ taking $x \in D$ to the function $f(x) : D \to E$ that it represents.

It is conceivable that $f$ could actually be a bijection between $D$ and the function space $D \to E$. However, this is impossible if $E$ contains more than one element. In fact, $f$ cannot even be onto. This follows by a diagonalization argument.[1] 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 $x$.

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 | |
|---|---|---|---|---|
| $f_0$ | $f_0\,0$ | $f_0\,1$ | $f_0\,2$ | $\ldots$ |
| $f_1$ | $f_1\,0$ | $f_1\,1$ | $f_1\,2$ | $\ldots$ |
| $f_2$ | $f_2\,0$ | $f_2\,1$ | $f_2\,2$ | $\ldots$ |
| $\vdots$ | $\vdots$ | | | |

Thus for any function $f : D \to (D \to E)$, where $E$ contains more than one element, there always exists a function $d : D \to E$ that is not $f\,x$ for any $x \in D$. This says that the cardinality (size) of the set of functions $D \to E$ is strictly larger than the cardinality of $D$, no matter what the sets $D$ and $E$ are, as long as $E$ contains at least two elements.

[1] Due to Georg Cantor (1845–1918), the inventor of set theory.

## 3 Denotational Semantics of 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 ensure that we are precise in identifying the extension of functions.

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

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

The syntactic categories of $a, b, c$ are arithmetic expressions AExp, Boolean expressions BExp, and commands Com, respectively.

Our denotational semantics will be defined in terms of *states*, which are functions $\mathit{Env} = \mathit{Var} \to \mathbb{Z}$. Our semantics will interpret arithmetic expressions as functions from states to integers, Boolean expressions as functions from states to Boolean values $\mathbb{2} = \{\mathit{true}, \mathit{false}\}$, and commands as *partial* functions from states to states.

$$\mathcal{A}[\![a]\!] \in \mathit{Env} \to \mathbb{Z} \qquad \mathcal{B}[\![b]\!] \in \mathit{Env} \to \mathbb{2} \qquad \mathcal{C}[\![c]\!] \in \mathit{Env} \rightharpoonup \mathit{Env}$$

(The notation $\to$ denotes *total* functions, those that produce a value on all inputs in the domain; and $\rightharpoonup$ denotes *partial* functions, those that may not produce a value on all inputs in the domain.) Given an initial state as input, the partial function $\mathcal{C}[\![c]\!]$ produces the final state reached by applying the command if the program $c$ halts; however, there will be no such final state if $c$ does not halt (e.g., while true do skip). Thus the function is partial.

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 arithmetic expressions,

$$\mathcal{A}[\![n]\!]\sigma \triangleq n \qquad \mathcal{A}[\![x]\!]\sigma \triangleq \sigma(x) \qquad \mathcal{A}[\![a_1 + a_2]\!]\sigma \triangleq \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma$$

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

Note that by a slight but convenient abuse, we are overloading the metasymbol $+$ in the definition of $\mathcal{A}[\![a_1 + a_2]\!]$. The $+$ on the left-hand side represents a syntactic object, namely a symbol in the language IMP, whereas the $+$ on the right-hand side represents a semantic object, namely the addition operation in the integers. We can also streamline the definition of $\mathcal{B}[\![\neg b]\!]$ using a semantic *if-then-else*:

$$\mathcal{B}[\![\neg b]\!]\sigma \triangleq \mathit{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathit{then\ false\ else\ true}.$$

For a command $c$, given an initial state as input, the function $\mathcal{C}[\![c]\!]$ should produce the final state reached by applying $c$, provided the computation halts. However, there will be no such final state if $c$ does not halt; for example, while true do skip. This is why the function is 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 we can regard $\mathcal{C}[\![c]\!]$ as a total function $\mathcal{C}[\![c]\!] \in \mathit{Env} \to \mathit{Env}_\perp$, where $\mathcal{C}[\![c]\!]\sigma = \tau$ if $c$ terminates in state $\tau$ on input state $\sigma$, and $\mathcal{C}[\![c]\!]\sigma = \perp$ if $c$ does not terminate on input state $\sigma$.

Using this notation, we can define

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

For the last case involving sequential composition $c_1\ ;\ c_2$, another way of achieving this effect is by defining a *lifting* operator $(\cdot)^\dagger : (D \to E_\bot) \to (D_\bot \to E_\bot)$ on functions:

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

With this notation, we have

$$\mathcal{C}[\![c_1\ ;\ c_2]\!]\sigma \triangleq \mathcal{C}[\![c_2]\!]^\dagger(\mathcal{C}[\![c_1]\!]\sigma)$$

or equivalently,

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

where $\circ$ denotes functional composition $(f \circ g)(\sigma) \triangleq f(g(\sigma))$.

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 definition might be:

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

but this is circular. However, we can take a fixpoint in some domain. Replacing $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$ by $W$, we obtain a fixpoint equation

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

or rather

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

and we would like $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$ to be a solution of this equation. Let us define

$$\mathcal{F} \triangleq \lambda w \in \mathit{Env} \to \mathit{Env}_\bot.\,\lambda\sigma \in \mathit{Env}.\,\mathit{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathit{then}\ w^\dagger(\mathcal{C}[\![c]\!]\sigma)\ \mathit{else}\ \sigma.$$

Then we would like $W = \mathcal{F}\,W$; that is, we are looking for a fixpoint of $\mathcal{F}$. But how do we take fixpoints without using the dreaded $Y$ combinator? Eventually we will have a function $\mathsf{fix}$ with $W = \mathsf{fix}\,\mathcal{F}$, where $\mathcal{F} \in (\mathit{Env} \to \mathit{Env}_\bot) \to (\mathit{Env} \to \mathit{Env}_\bot)$. The solution will be to think of a $\mathsf{while}$ 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

$$W_0 \triangleq \lambda\sigma \in \mathit{Env}.\,\bot.$$

The next approximation is

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

This simulates one iteration of the loop. We could then simulate two iterations by:

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

In general,

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

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 $W_0 \sqsubseteq W_1 \sqsubseteq W_2 \sqsubseteq \cdots$, then find the least upper bound of this sequence. This will be the least solution of the equation (2).

In order to show that the least fixpoint of $\mathcal{F}$ exists, we will apply the Knaster–Tarski theorem. However, we only proved the Knaster–Tarski theorem for the partial order of subsets of some universal set ordered by set inclusion $\subseteq$. We need to extend it to the more general case of chain-complete partial orders (CPOs). To apply this theorem, we must know that the function space $\mathit{Env} \to \mathit{Env}_\bot$ is a CPO and that $\mathcal{F}$ is a continuous map on this space. We will develop this theory next time.

## 4 Binary Relation Semantics

An alternative but equivalent approach to the denotational semantics of IMP is to interpret commands as binary relations, or sets of ordered pairs:

$$[\![c]\!] \subseteq \mathit{Env} \times \mathit{Env}.$$

Every partial and total function is extensionally just a binary relation: a total function is a binary relation containing *exactly* one pair with first component $\sigma$ for each $\sigma$ in the domain, and a partial function is a binary relation containing *at most* one pair with first component $\sigma$ for each $\sigma$ in the domain.

In this view, an equivalent definition to the above would be

$$\begin{aligned} [\![\mathsf{skip}]\!] &\triangleq \{(\sigma,\sigma) \mid \sigma \in \mathit{Env}\} \\ [\![x := a]\!] &\triangleq \{(\sigma, \sigma[\mathcal{A}[\![a]\!]\sigma/x]) \mid \sigma \in \mathit{Env}\} \\ [\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &\triangleq ([\![c_1]\!] \cap \{(\sigma,\tau) \mid \mathcal{B}[\![b]\!]\sigma = \mathit{true}\}) \cup ([\![c_2]\!] \cap \{(\sigma,\tau) \mid \mathcal{B}[\![b]\!]\sigma = \mathit{false}\}) \qquad (3) \\ [\![c_1\ ;\ c_2]\!] &\triangleq [\![c_2]\!] \circ [\![c_1]\!], \end{aligned}$$

where $\circ$ denotes relational composition $R \circ S \triangleq \{(\sigma,\tau) \mid \exists\rho\ (\sigma,\rho) \in S \wedge (\rho,\tau) \in R\}$. The definition of the conditional (3) can be further simplified by defining a binary relation

$$[\![b]\!] \triangleq \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathit{true}\}$$

for Boolean expressions $b$; then (3) becomes

$$[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] = ([\![c_1]\!] \circ [\![b]\!]) \cup ([\![c_2]\!] \circ [\![\neg b]\!]). \qquad (4)$$

In this way the binary relation semantics of the conditional is defined in terms of the simpler set-theoretic operations $\cup$ and $\circ$ on relations.

For the while loop, we can take the least fixpoint of a continuous map on binary relations. We can use the set-theoretic version of the Knaster–Tarski theorem for this. Using (4) to rewrite (1) in terms of binary relations, we obtain

$$W \;=\; (W \circ [\![c]\!] \circ [\![b]\!]) \cup [\![\neg b]\!].$$

One can show that the $\subseteq$-least solution of this equation is

$$W \;=\; [\![\neg b]\!] \circ ([\![c]\!] \circ [\![b]\!])^*,$$

where $R^*$ denotes the reflexive-transitive closure of the relation $R$:

$$R^0 = \{(\sigma, \sigma) \mid \sigma \in \mathit{Env}\} \qquad\qquad R^{n+1} = R \circ R^n \qquad\qquad R^* = \bigcup_n R^n,$$

so we can define

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] \;\stackrel{\triangle}{=}\; [\![\neg b]\!] \circ ([\![c]\!] \circ [\![b]\!])^*.$$

One can show

**Theorem 1.** *For all $\sigma, \tau \in \mathit{Env}$, the following are equivalent:*

(i) $\langle c, \sigma\rangle \Downarrow_{\mathrm{c}} \tau$ *in the big-step semantics of Lecture 6;*

(ii) $\mathcal{C}[\![c]\!]\sigma = \tau$ *in the denotational semantics of §3;*

(iii) $(\sigma, \tau) \in [\![c]\!]$ *in the binary relation semantics of this section.*

## References

[1] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.