1 Denotational Semantics

1.1 Introduction

So far we've been looking at translations where the target is a simpler or better understood language. These are called **definitional translations**. A closely related approach semantics, **denotational semantics**, is translations to mathematical objects. The objects in question will be functions with well-defined extensional meaning, i.e. in terms of sets. The main challenge will be getting a precise understanding of what sets these function operate over. For example, consider the function $\lambda x.x$, which is the identity function, but over which domain? Another even more problematic example is the function $\lambda x.(x\ x)$. In this case, $x \in D$ (x is an element of the domain) *and* $x \in D \to E$ (x is a function from the domain to the set of expressions, since we apply x to an element of the domain). How can both of these statements about x be true? Maybe $D = D \to E$? This is a fairly rigid requirement, so maybe it suffices to allow isomorphic sets: $D \cong D \to E$. However, a set cannot be isomorphic to the set of functions from itself to E, unless $|E| = 1$. This follows by a diagonalization argument (if the sets are countable, and the proof generalizes for uncountable sets as well). To carry out the proof, we list

$$\begin{array}{cc} & d_1\ d_2\ \ d_3 \\ f_1 & e_{11}\ e_{12}\ e_{13}\ \ldots \\ f_2 & e_{21}\ e_{22}\ e_{23}\ \ldots \\ \vdots & \vdots \end{array}$$

where $f_i(d_j) = e_{ij}$. Then a bijection would associate exactly one d_i with each function in the list. But, consider the function

$$\{d_1 \mapsto \neg e_{11}, d_2 \mapsto \neg e_{22}, \ldots\}.$$

This function is in $D \to E$ but is not in the list, thereby contradicting the existence of a bijection. Similarly, functions like the Y combinator are problematic because they require an isomorphism between two sets of apparently different cardinality.
This means that we need to take more care in which functions we write down. The solution to the cardinality 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

We will use the notation $\lambda x \in D.e$ when writing down denotational semantics. The addition of the domain will make sure we are precise in identifying the extension of functions.

Aside: this is not a type declaration. Later, we will introduce types and write them as $\lambda x : \tau.e$. The reason for differentiating these is that types are pieces of language syntax whereas sets are mathematical objects which are referred to with mathematical notation.
Recall:

$$\begin{array}{rcl} a & ::= & n \mid x \mid (\mathsf{a}_0 \oplus \mathsf{a}_1) \\ b & ::= & \textbf{true} \mid \textbf{false} \mid (\neg\ \mathsf{b}) \mid (\mathsf{b}_0 \wedge \mathsf{b}_1) \mid (\mathsf{a}_0 = \mathsf{a}_1) \mid \cdots \\ c & ::= & \textbf{skip} \mid x := a \mid (\mathsf{c}_0\ ;\ \mathsf{c}_1) \mid (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \mid (\textbf{while}\ b\ \textbf{do}\ c\ ) \end{array}$$

To define the denotational semantics, we will make reference to states, which are functions $\Sigma = Var \to \mathbb{Z}$, in order to define the following functions.

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

Intuitively, we'd like the meaning of commands to be functions from states to states: given an initial state, produce the final state reached by applying the command. But, this becomes problematic if there is no such final state, i.e. if the program resulting from issuing the command doesn't terminate (e.g. while **true** DO **skip**).

Definition. For any set S, let $S_\perp = S \cup \{\perp\}$ where $\perp$ is a bottom element (more on this later).

So, define $\mathcal{C}[\![c]\!] \in \Sigma \to \Sigma_\perp$, where we will use $\perp$ to indicate non-termination.
As usual, we define denotational semantics by structural induction. Note that 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 sub-expressions in the definitions.

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

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

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

Note that we can express the translation of negation more compactly with a mathematical *if then else* expression:

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

Alternatively, we can write down the function extensionally,

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

For the commands:

$$\begin{array}{rcl} \mathcal{C}[\![\textbf{skip}]\!]\sigma & = & \lfloor\sigma\rfloor \\ \mathcal{C}[\![x := a]\!]\sigma & = & \lfloor\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]\rfloor \\ \mathcal{C}[\![\textbf{if b then } \mathsf{c}_1 \textbf{ else } \mathsf{c}_2]\!]\sigma & = & if\ \mathcal{B}[\![b]\!]\sigma\ then\ \mathcal{C}[\![\mathsf{c}_1]\!]\sigma\ else\ \mathcal{C}[\![\mathsf{c}_2]\!]\sigma \end{array}$$

where we use the **lift** function, $\lfloor\ \rfloor : S \to S_\perp$ (for any set S), to distinguish between elements of these sets.

The denotational semantics of sequences of commands is a little more challenging since we need to **reverse lift**:

$$\mathcal{C}[\![\mathsf{c}_1 \ ; \ \mathsf{c}_2]\!]\sigma = (if\ \mathcal{C}[\![\mathsf{c}_1]\!]\sigma = \perp\ then \perp\ else\ \mathcal{C}[\![\mathsf{c}_2]\!]\,(\mathcal{C}[\![\mathsf{c}_1]\!]\sigma))\,.$$

This isn't quite correct since $\mathcal{C}[\![\mathsf{c_1}]\!] \in \Sigma_\perp$ but the domain of $\mathcal{C}[\![\mathsf{c_2}]\!]$ is Σ. To fix this, we introduce a piece of metalanguage:

$$\mathsf{let}\ \sigma' = \mathcal{C}[\![\mathsf{c_1}]\!]\sigma\ \mathsf{in}\ \mathcal{C}[\![c_2]\!]\sigma'$$

where we define the semantics of the let statement such that if it is applies to $\perp$ it returns $\perp$ and otherwise it uses the reverse lifted object. In the example above, this does what we want: if $\mathcal{C}[\![\mathsf{c_1}]\!] = \perp$ then the whole thing returns $\perp$, and otherwise σ' gets the element of Σ which maps to $\mathcal{C}[\![\mathsf{c_1}]\!]$ under the lifting map.
Another way of achieving this effect is by defining a lift operator on functions, $(\ )^* : (D \to E_\perp) \to (D_\perp \to E_\perp)$ such that

$$(\ )^* = \lambda f \in (D \to E_\perp).\, (\lambda x.\ if\ x = \perp\ then \perp\ else\ f(x))\,.$$

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

$$\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] = \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then\ let}\ \sigma' = \mathcal{C}[\![c]\!]\sigma\ \mathsf{in}\ \mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma'\ \mathsf{else}\ \sigma$$

However, there is a problem here: this is not well-founded. To fix this, we need to think differently about the meaning of a while statement. Suppose:

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

Then

$$\mathcal{W} = \lambda\sigma \in \Sigma.\mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then\ let}\ \sigma' = \mathcal{C}[\![c]\!]\sigma\ \mathsf{in}\ \mathcal{W}\sigma'\ \mathsf{else}\ \lfloor\sigma\rfloor$$

But note that this is an equation, and not a definition. If we define $\mathcal{F}$ as:

$$\mathcal{F} = \lambda w \in \Sigma \to \Sigma_\perp.\lambda\sigma \in \Sigma\ \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then\ let}\ \sigma' = \mathcal{C}[\![c]\!]\sigma\mathsf{in}\ w\sigma'\ \mathsf{else}\lfloor\sigma\rfloor$$

Then $\mathcal{W} = \mathcal{F}\mathcal{W}$, i.e., we are looking for a fixed point of $\mathcal{F}$. But how do we take fixed points without using the dreaded Y combinator? Eventually we will have $\mathcal{W} = \mathrm{fix}\ \mathcal{F}$, 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 functions. Intuitively, by running through the loop more and more times, we will get better and better approximations. Consider

$$\mathsf{while}\ b\ \mathsf{do}\ c \cong_0 \mathsf{if}\ b\ \mathsf{then}\ c; \perp\mathsf{else\ skip}$$

where $\perp$ means diverge. This simulates 0 iterations of the loop. We could then simulate 1 iteration of the loop by:

$$\mathsf{if}\ b\ \mathsf{then}\ c; (\mathsf{if}\ b\ \mathsf{then}\ c; \perp\mathsf{else\ skip})\mathsf{else\ skip}$$

It should now be clear how to simulate n iterations of the while loop. We now define a sequence of functions:
$d_0 = \lambda\sigma \in \Sigma.\perp$
$\mathcal{F}(d_0)$ = the denotation of if b then c; $\perp$ else skip
Then $\mathcal{F}(\mathcal{F}(d_0))$ corresponds to unrolling the loop once. Inductively, we define $d_n = \mathcal{F}^n(d_0)$, which describes the loop up to $n-1$ iterations. So, then denotation of the while statement should be:

$$\mathit{fix}\mathcal{F} = \text{`limit'}_{n\to\infty}\mathcal{F}^n(d_0)$$

If we want to take the limit of a sequence of functions, we need some structure in the space of functions. We will define an ordering on these functions, $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ then find the least upper bound of this sequence and define fix F = l.u.b. $\{d_n\}$. We first introduce some general notions.

Recall that a *partial order* is a pair consisting of a set S (called the carrier set) and a relation $\sqsubseteq$ on S such that:
$\sqsubseteq$ is reflexive (for all $d \in S, d \sqsubseteq d$)
$\sqsubseteq$ is transitive (for all $d, e, f \in S$: if $d \sqsubseteq e$ and $e \sqsubseteq f$, then $d \sqsubseteq f$)

$\sqsubseteq$ is antisymmetric (for all $d, e \in S : d \sqsubseteq e \wedge e \sqsubseteq d \Rightarrow d = e$)

Examples include $(\mathbb{Z}, \leq)$, $(\mathbb{Z}, =)$ (known as a discrete partial order because the relation is equality), $(\mathbb{Z}, \geq)$, $(\{\mathsf{true}, \mathsf{false}\}, \rightarrow)$, if $(S, \sqsubseteq)$ is a partial order then so is $(S, \sqsupseteq)$, and $(2^S, \subseteq)$.

We can represent a partial order visually by drawing a *Hasse diagram*. Draw each element as a point, with the point representing d_2 drawn above the point representing 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.

Now, 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 $d_1 \sqsubseteq d_2$, and $\perp \sqsubseteq_\perp d$ for all $d \in S_\perp$.

So, if S is the space of functions we were working with earlier, $S_\perp$ is that space with a least element $\perp$ added. Intuitively, non-termination is less than (contains less information than) any function.