Formal Programming Language Semantics note 10

Introduction to denotational semantics

We now turn our attention to *denotational* descriptions of the semantics of programming languages. We will start by considering the our old friend **IMP**.

Meanings for program phrases. The idea of denotational semantics is to assign a "meaning" to each program (or bit of program) in the language, which encapsulates all the information we need to determine how the program will behave in any context. The meaning of a whole program will depend in a systematic way on the meaning of its various parts.

For example, in operational semantics we have considered relations such as

$$\langle e, \sigma \rangle \ \Downarrow \ v,$$

which says that the expression e, in the particular state σ, will evaluate to the value v. Of course, different states σ will give different values v here. So we might say that the *meaning* of the expression e is encapsulated by a certain function

$$[\![e]\!] : \text{States} \longrightarrow \text{Values}.$$

The above relation can then be rephrased as

$$[\![e]\!](\sigma) \ = \ v.$$

Likewise, the meaning of a command c could be captured by a partial function

$$[\![c]\!] : \text{States} \rightharpoonup \text{States}.$$

(It had better be a partial function, because there are non-terminating commands. The "harpoon" symbol here is used for partial functions.)

We will refer to $[\![e]\!]$ and $[\![c]\!]$ here as the meanings or *denotations* of e and c. There isn't really a single God-given notion of what "meaning" means for programming languages, so the denotational meaning of programs — the definition of the operation $[\![-]\!]$ — is something we have to choose for ourselves. This means that there may well be more than one denotational semantics for a given language. (We will see some other kinds later.)

A denotational semantics of IMP. Let us be more precise about the definition of our denotational semantics. We have already the set $\mathbb{S}$ of possible states for **IMP**, so in accordance with the above intuition, let us define sets $\mathcal{D}^u$ for each

phrase category u as follows. We will write $A \longrightarrow B$ here for the set of all (total) functions from A to B, and $A \rightharpoonup B$ for the set of all partial functions.

$$\begin{aligned} \mathcal{D}^{\texttt{Aexp}} &= \mathbb{S} \longrightarrow \mathbb{Z}, \\ \mathcal{D}^{\texttt{Bexp}} &= \mathbb{S} \longrightarrow \mathbb{T}, \\ \mathcal{D}^{\texttt{Com}} &= \mathbb{S} \rightharpoonup \mathbb{S}. \end{aligned}$$

The idea is that these sets serve as our "universes of meanings" or *semantic domains*: if P is a phrase of syntactic category u, then $[\![P]\!]$ will be an element of the domain $\mathcal{D}^u$.

We now define the denotations $[\![P]\!]$ themselves. The important point is that the definition of $[\![-]\!]$ is *compositional*: it is given by induction on the syntactic structure of phrases, and the meaning of a phrase is determined by the meaning of its immediate subphrases. We will use the notation $\Lambda\sigma.\cdots$ to mean "the (total or partial) function that maps σ to $\cdots$", so that "$f = \Lambda\sigma.\cdots$" amounts to the same as saying "for all $\sigma \in \mathbb{S}$, $f(\sigma) = \cdots$". The convention is that if the value of the mathematical expression represented by $\cdots$ is undefined for a particular σ, then the partial function $\Lambda\sigma.\cdots$ is undefined on this σ.

The clauses for arithmetic and boolean expressions are obvious enough, and closely parallel the evaluation rules of Note 4. (We number the semantic clauses in a manner consistent with Note 4.)

For Aexp:

$$\begin{aligned} (1) \quad [\![n]\!] &= \Lambda\sigma.\ n \\ (2) \quad [\![X]\!] &= \Lambda\sigma.\ \sigma(X) \\ (3) \quad [\![a_0 \texttt{ - } a_1]\!] &= \Lambda\sigma.\ [\![a_0]\!](\sigma) - [\![a_1]\!](\sigma) \\ (4) \quad [\![a_0 \texttt{ * } a_1]\!] &= \Lambda\sigma.\ [\![a_0]\!](\sigma) \times [\![a_1]\!](\sigma) \end{aligned}$$

For Bexp:

$$\begin{aligned} (5) \quad [\![t]\!] &= \Lambda\sigma.\ t \\ (6/7) \quad [\![a_0 \texttt{ = } a_1]\!] &= \Lambda\sigma.\ \begin{cases} \textit{true} & \text{if } [\![a_0]\!](\sigma) = [\![a_1]\!](\sigma) \\ \textit{false} & \text{if } [\![a_0]\!](\sigma) \neq [\![a_1]\!](\sigma) \end{cases} \\ (8/9) \quad [\![a_0 \texttt{ <= } a_1]\!] &= \Lambda\sigma.\ \begin{cases} \textit{true} & \text{if } [\![a_0]\!](\sigma) \leq [\![a_1]\!](\sigma) \\ \textit{false} & \text{if } [\![a_0]\!](\sigma) > [\![a_1]\!](\sigma) \end{cases} \\ (10) \quad [\![\texttt{not } b]\!] &= \Lambda\sigma.\ \neg([\![b]\!](\sigma)) \\ (11/12) \quad [\![b_0 \texttt{ and } b_1]\!] &= \Lambda\sigma.\ \mathit{And}([\![b_0]\!](\sigma), [\![b_1]\!](\sigma)) \end{aligned}$$

Clearly these clauses define *total* functions $\mathbb{S} \to \mathbb{Z}$ or $\mathbb{S} \to \mathbb{T}$ as required.

The clauses for commands are a bit more interesting. The denotation of a command is essentially the transformation on states that is induced by executing the command. First, `skip` does nothing to the state, so its denotation is the identity function:

$$(13) \quad [\![\texttt{skip}]\!] = \Lambda\sigma.\ \sigma$$

The denotation of an assignment is the function that updates the state in the obvious way. This clause shows that the denotations of commands can depend on the denotations of expressions.

$$(14) \quad [\![X \texttt{ := } a]\!] = \Lambda\sigma.\ \sigma[X \mapsto [\![a]\!](\sigma)]$$

Sequencing of commands is interpreted by composition of state transformations:

$$(15)\quad [\![c_0 \texttt{ ; } c_1]\!] = \Lambda\sigma.\, [\![c_1]\!]([\![c_0]\!](\sigma)) \qquad (= [\![c_1]\!] \circ [\![c_0]\!])$$

The interpretation of `if` statements is fairly obvious:

$$(16/17)\quad [\![\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1]\!] = \Lambda\sigma.\, \begin{cases} [\![c_0]\!](\sigma) & \text{if } [\![b]\!](\sigma) = \textit{true} \\ [\![c_1]\!](\sigma) & \text{if } [\![b]\!](\sigma) = \textit{false} \end{cases}$$

All of the above clauses are in some sense just repackagings of the corresponding operational rules, so you may be wondering what all the fuss is about. However, we have saved the best clause till last: the clause for `while`. This involves an important new idea — that of *least fixed points* — and it is here that the real difference between operational and denotational semantics starts to emerge.

Denotations of while commands. Suppose that $[\![b]\!]$ and $[\![c]\!]$ have already been defined. Our goal is to define a function $h = [\![\texttt{while } b \texttt{ do } c]\!]$. As a first attempt, we might try a straightforward adaptation of the operational rules (18) and (19), in the spirit of the clauses above. This might lead us to try something like:

$$h \;=\; \Lambda\sigma.\, \begin{cases} h([\![c]\!](\sigma)) & \text{if } [\![b]\!](\sigma) = \textit{true} \\ \sigma & \text{if } [\![b]\!](\sigma) = \textit{false}. \end{cases}$$

However, this is not a definition by itself, since it "defines" h in terms of itself. Rather, it is a *property* which we would expect h to possess if we have defined it correctly. So in order to define h, let us try a slightly different approach.

We cannot directly define the semantics of `while` commands, so instead let us approach the problem by introducing into our language an infinite sequence of auxiliary constructs $\texttt{while}_0, \texttt{while}_1, \texttt{while}_2, \ldots$ whose meaning we *can* define, and which provide successively approximations to the original `while`.

The idea is that $\texttt{while}_k$ will behave like `while` for up to k loop iterations, but after that it will simply diverge and not yield a final state. The construct $\texttt{while}_0$ will be dreadful: by definition, commands of the form $\texttt{while}_0\ b\ \texttt{do}\ c$ will always diverge! We may capture this operationally simply by giving *no evaluation rules* for $\texttt{while}_0$, so that it will be impossible to derive any assertions of the form

The remaining constructs are defined inductively: given $\texttt{while}_k$, we define $\texttt{while}_{k+1}$ to be a construct which is able to do one more loop iteration than $\texttt{while}_k$. More specifically, the effect of $\texttt{while}_{k+1}\ b\ \texttt{do}\ c$ will be to evaluate b, quit the loop if the result is *false*, otherwise execute c and then perform $\texttt{while}_k\ b\ \texttt{do}\ c$. We may give operational rules for this as follows:

$$\frac{\langle b, \sigma\rangle \Downarrow \textit{false}}{\langle \texttt{while}_{k+1}\ b\ \texttt{do}\ c, \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle b, \sigma\rangle \Downarrow \textit{true} \quad \langle c, \sigma\rangle \Downarrow \sigma' \quad \langle \texttt{while}_k\ b\ \texttt{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{while}_{k+1}\ b\ \texttt{do}\ c, \sigma\rangle \Downarrow \sigma''}$$

Let us write W for the command `while` b `do` c, and W_k for $\texttt{while}_k\ b\ \texttt{do}\ c$. Notice the following two important facts:

- If $\langle W_k, \sigma\rangle \Downarrow \sigma'$ then $\langle W_{k+1}, \sigma\rangle \Downarrow \sigma'$ (and indeed $\langle W, \sigma\rangle \Downarrow \sigma'$). In other words, each approximation to W is at least as good as the previous one.
- If $\langle W, \sigma\rangle \Downarrow \sigma'$ then $\langle W_k, \sigma\rangle \Downarrow \sigma'$ must hold for some large enough k, since the execution of W in state σ must quit the loop after some finite number of iterations. (Note that k may depend on σ — there will not usually be a single k which works uniformly for all σ).

Let us now turn to the denotational semantics. We are going to creep up on the sought-after denotation $h = [\![W]\!]$ by defining denotations h_k for each W_k. It is clear what the denotation of W_0 ought to be: since W_0 always diverges, we take h_0 to be the *empty* or *everywhere undefined* partial function $\bot : \mathbb{S} \rightharpoonup \mathbb{S}$. And given h_k, it is apparent from the above operational rules that we ought to define

$$h_{k+1} \;=\; \Lambda\sigma.\; \begin{cases} h_k([\![c]\!](\sigma)) & \text{if } [\![b]\!](\sigma) = \mathit{true} \\ \sigma & \text{if } [\![b]\!](\sigma) = \mathit{false}. \end{cases}$$

Let us write $\subseteq$ for the inclusion ordering on partial functions $\mathbb{S} \rightharpoonup \mathbb{S}$: $f \subseteq g$ if whenever $f(x)$ is defined, so is $g(x)$ and they are equal. Clearly $h_0 \subseteq h_1$, since h_0 is the smallest partial function; and from the above equation it follows (by induction) that $h_k \subseteq h_{k+1}$ for all k. So we have an increasing chain of partial functions $h_0 \subseteq h_1 \subseteq h_2 \subseteq \cdots$.

Now, from the observations above we see that $\langle W, \sigma\rangle \Downarrow \sigma'$ if and only if some $\langle W_k, \sigma\rangle \Downarrow \sigma'$. This suggests that we should define h so that $h(\sigma) = \sigma'$ if and only if some $h_k(\sigma) = \sigma'$. In other words, we let h be the *union* of the partial functions h_k:

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!] \;=\; h \;=\; \bigcup_{k\geq 0} h_k$$

This completes the definition of the denotational semantics of **IMP**^{b}.

Extending to IMP^e, IMP^s, IMP^b. One can also give denotational semantics for **IMP**^{e} and **IMP**^{s} via a straightforward adaptation of the above definitions, by modifying our semantic domains so as to take account of exceptions and side-effects. We leave the details to the avid reader.

A more substantial modification is needed for **IMP**^{b}, since here the notion of state varies according to the set of identifiers in scope. We therefore have to treat many of the above notions as defined relative to a given static environment Γ. Let us first define $[\![\texttt{int}]\!] = \mathbb{Z}$ and $[\![\texttt{bool}]\!] = \mathbb{T}$. If $\Gamma = [(X_1, u_1, d_1), \ldots, (X_r, u_r, d_r)]$, we may define the set $\mathbb{S}_\Gamma$ of states for Γ to be the set of all lists $[(X_1, v_1), \ldots, (X_r, v_r)]$ where $v_i \in [\![u_i]\!]$ for each i. (Note that $\mathbb{S}_\Gamma$ is isomorphic to $[\![u_1]\!] \times \cdots \times [\![u_i]\!]$.) We next define semantic domains $\mathcal{D}^u_\Gamma$ (for $u = \mathsf{Aexp}, \mathsf{Bexp}, \mathsf{Com}$) as before, but using $\mathbb{S}_\Gamma$ instead of $\mathbb{S}$. Finally, if P is a well-typed phrase of type u in environment Γ (that is, $\Gamma \vdash P : u$), we define its denotation $[\![P]\!]_\Gamma$ to be a certain element of $\mathcal{D}^u_\Gamma$. (It is typical of most denotational interpretations that one defines the meaning of terms relative to some environment.) The definition of $[\![P]\!]_\Gamma$ is given by induction on the structure of P — or more accurately, by induction on the typing derivation of $\Gamma \vdash P : u$. Again, the reader may enjoy working out the details.