CS611 Lecture 17 Denotational Semantics of IMP October 10, 2001

Scribe: Rohit Ananthakrishna and Manpreet Singh (based on earlier notes from Cristina Patron and Suzanne Shontz) Lecturer: Andrew Myers

1 Operational Semantics vs. Denotational Semantics

We have described the behaviour of programs in IMP in an operational manner by inductively defining transition relations to express evaluation and execution. The style of semantics we used is often called *structural operational semantics* because of the syntax-directed way in which the rules are presented. It is also called *natural semantics* because of the way derivations resemble proofs in natural deduction.

It is fairly easy to to turn the description of the semantics into a recursive-descent interpreter for IMP. On the other hand, it is hard to compare two programs written in different programming languages. This suggests we should define the meaning of a program in terms of the underlying semantic domain. We will have a semantic function (called denotation) which maps pieces of syntax to meanings. This new style of semantics, called denotational semantics, can be thought of as describing a *compiler* for IMP that converts syntax into an extensional representation of that syntax.

2 Semantic Functions

Denotational semantics operates on expressions to produce mathematical objects that are the meaning of the expression. The mechanism for this is a *semantic function*, which is usually a mathematical function.

We now give an example.

Example: $\mathcal{C}[\![(\lambda\ x\ x)]\!] = \lambda x \in D.x$

Here $\mathcal{C}$ is a semantic function that represents the action of a compiler. It takes as input $(\lambda\ x\ x)$, which is a piece of abstract syntax. (Note: $(\lambda\ x\ x)$ should really be thought of as an abstract syntax tree.) It then gives meaning to the expression. The brackets $[\![\ ]\!]$ are used to represent this. The meaning of $(\lambda\ x\ x)$ is given mathematically on the right-hand side of the equation. The meaning is that λ takes $x \in D$ and returns $\lambda(x)$. This can be thought of in terms of function extensions. In this case, the extension of λ is given by $\{(a, a) \mid a \in D\}$.

Now that we are using $\lambda-$expressions to describe mathematical functions, we need to determine how to parse them.

We start off by giving the convention for a mathematical function of several variables. $\lambda xyz.e$ means that λ takes the three variables x, y, and z as input and gives e as output. Because we apply λ left-to-right, this is the same as $\lambda x \lambda y \lambda z.e$.

The next rule for $\lambda-$expressions is that they extend as far to the right as possible.
Thus, $\lambda x \lambda y \lambda z.x \lambda w.w = \lambda x(\lambda y(\lambda z.(x(\lambda w.w))))$. Be careful with those parentheses!

The final rule is that application is left associative. Thus, $xyz = (xy)z = x(y, z)$.

So, an example of a function taking many variables as inputs is $f = \lambda xyz.e = \lambda x \lambda y \lambda z.e$.

Typed functions are simply functions and their types. We will usually write the types of the arguments for mathematical functions unless the name of the argument makes it obvious. Suppose we want to add two integers x and y. This can be achieved using the following PLUS function:

$$PLUS = \lambda x \in Z.\lambda y \in Z.x + y.$$

This function takes $x \in Z$ and $y \in Z$ (one argument at a time) and yields $x + y \in Z$. Thus, the domain is $Z \times Z$ and the codomain is Z.

Mathematically, this is written as:

$$PLUS : (Z \times Z) \to Z;\ (x, y) \longmapsto x + y.$$

Because $(Z \times Z) \to Z$ is isomorphic to $Z \to (Z \to Z)$ and because $PLUS$ operates on one input argument at a time, this is the same as:

$$PLUS \in Z \to (Z \to Z).$$

Note that while application associates to the left, the constructor $(\to)$ associates to the right, i.e., $Z \to Z \to Z = Z \to (Z \to Z)$.

3 Semantic Functions for IMP

Recall now the three kinds of expressions in IMP:

- arithmetic expressions

  $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$

- boolean expressions

  $b ::= a_0 \leq a_1 \mid a_0 = a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$

- commands

  $X := a_0 \mid \mathbf{skip} \mid \mathbf{if}\, b_0\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mid \mathbf{while}\, b_0\, \mathbf{do}\, c_0$

What is the intrinsic meaning of these syntactic categories?

Let us take for example the arithmetic expressions. Our first thought, when we see an expression is to evaluate it and thus the meaning of an arithmetic expression is an integer. Notice though that this evaluation depends on the particular store we have: given store σ, each expression a denotes a unique integer, so the meaning of an arithmetic expression is really a function from stores to integers. Hence we can define a function $\mathcal{A}$ which translates the syntax of the arithmetic expressions into their meaning:

$$\mathcal{A}[\![a]\!]\sigma = n \Leftrightarrow \langle a, \sigma\rangle \Downarrow n$$

Similarly, given a particular store σ boolean expressions b denotes a unique truth value. We can define:

$$\mathcal{B}[\![b]\!]\sigma = t \Leftrightarrow \langle b, \sigma\rangle \Downarrow t$$

Since a command c maps one store into another, we define:

$$\mathcal{C}[\![c]\!]\sigma = \sigma' \Leftrightarrow \langle c, \sigma\rangle \Downarrow \sigma'$$

In conclusion, we have the meaning functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

- $\mathcal{A} \in \mathbf{Aexp} \to (\Sigma \to N)$
- $\mathcal{B} \in \mathbf{Bexp} \to (\Sigma \to T)$
- $\mathcal{C} \in \mathbf{Com} \to (\Sigma \to \Sigma)$

We say that the arithmetic expression a denotes $\mathcal{A}[\![a]\!]$ and $\mathcal{A}[\![a]\!]$ is a denotation of a. Similarly, $\mathcal{B}[\![b]\!]$ is a denotation of the boolean b and $\mathcal{C}[\![c]\!]$ is a denotation of command c. Each denotation is in fact a function:

- $\mathcal{A}[\![a]\!] : \Sigma \to N$

- $\mathcal{B}[\![b]\!] : \Sigma \to T$
- $\mathcal{C}[\![c]\!] : \Sigma \to \Sigma$

This signature for $\mathcal{C}$ won't quite work, however, because of the possibility of non-termination. We'll see how to fix it shortly.

The functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are defined by structural induction.

3.1 Arithmetic Denotations

Firstly, we define the denotation of arithmetic expressions $\mathcal{A} \in \mathbf{Aexp} \to (\Sigma \to N)$ using structural induction:

- $\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.\, n$

This means that the denotation of n is a function which associates the natural number n to any state σ. Similarly,

- $\mathcal{A}[\![X]\!] = \lambda\sigma \in \Sigma.\, \sigma X$
- $\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$
- $\mathcal{A}[\![a_0 - a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma$
- $\mathcal{A}[\![a_0 \times a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma$

Notice that the signs $+, -, \times$ on the left-hand sides represent syntactic signs in IMP, whereas the signs on the right represent operations on numbers.
We can write the last three definitions as inductive definitions, similar to the inference rules in the operational semantics:

$$\frac{\mathcal{A}[\![a_0]\!] = f_0 \quad \mathcal{A}[\![a_1]\!] = f_1}{\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.\, f_0\sigma + f_1\sigma}$$

Instead of using the λ-notation, we can present the definition of the semantics as a relation between states and numbers:

- $\mathcal{A}[\![n]\!] = \{(\sigma, n) | \sigma \in \Sigma\}$
- $\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X) | \sigma \in \Sigma\}$
- $\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) | (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) | (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) | (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$

3.2 Boolean Denotations

As for the arithmetic expressions, the function $\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to T)$ is defined using induction on the structure of expressions.

We start by applying $\mathcal{B}$ to the booleans with no subexpressions as follows:

- $\mathcal{B}[\![true]\!] = \lambda\sigma \in \Sigma.true$
- $\mathcal{B}[\![false]\!] = \lambda\sigma \in \Sigma.false.$

After applying both sides of the first function to σ, we get $\mathcal{B}[\![true]\!]\sigma = true$ by a β-reduction. Not only is this notation more compact, it makes the meaning more clear.

The rest of the rules for boolean denotations are as follows:

- $\mathcal{B}[\![a_0 = a_1]\!]\sigma = \textbf{if}\ \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\ \textbf{then}\ true\ \textbf{else}\ false$
- $\mathcal{B}[\![a_0 \leq a_1]\!]\sigma = \textbf{if}\ \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\ \textbf{then}\ true\ \textbf{else}\ false$
- $\mathcal{B}[\![b_0 \wedge b_1]\!]\sigma = \textbf{if}\ \mathcal{B}[\![b_0]\!]\sigma\ \wedge\ \mathcal{B}[\![b_1]\!]\sigma\ \textbf{then}\ true\ \textbf{else}\ false$
- $\mathcal{B}[\![b_0 \vee b_1]\!]\sigma = \textbf{if}\ \mathcal{B}[\![b_0]\!]\sigma\ \vee\ \mathcal{B}[\![b_1]\!]\sigma\ \textbf{then}\ true\ \textbf{else}\ false$

3.3 Command Denotations

In order to derive the rules for command denotations, we first note that some commands do not terminate. For example,

$$(\neg\exists\sigma'.\langle c, \sigma\rangle \Downarrow \sigma')$$

does not terminate.

Instead, commands are partial functions (i.e., they are not total) from states to states ($\Sigma \rightharpoonup \Sigma$). For example, consider (**while** $x = 0$ **do skip**). Its denotation is given by $\{(\sigma, \sigma) \mid \sigma(x) = 0\}$, which is not defined for $\sigma(x) \neq 0$. Thus, the corresponding command is not total.

In order to make denotations total, we add a new state, $\perp$, called *bottom*, to represent non-termination. Thus, we can say that commands are functions from Σ to $\Sigma_\perp$, and $\mathcal{C} \in \textbf{Com} \to (\Sigma \to \Sigma_\perp)$,
where $\Sigma_\perp$ = elements of $\Sigma \cup \{\perp\}$ = (lift of Σ).

Its main advantage over large-step semantics is that we can now specify non-terminating behavior of commands. The function $\mathcal{C} : \textbf{Com} \to \boldsymbol{\Sigma} \to \boldsymbol{\Sigma}_\perp$ is also defined using induction on the structure of expressions as follows:

- $\mathcal{C}[\![\textbf{skip}]\!]\sigma = \sigma$
- $\mathcal{C}[\![X := a]\!]\sigma = \sigma[X \longmapsto \mathcal{A}[\![a]\!]\sigma]$
- $\mathcal{C}[\![c_0; c_1]\!]\sigma = \begin{cases} \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma) & (\text{if } \mathcal{C}[\![c_0]\!]\sigma) \neq \perp) \\ \perp & (\text{otherwise}) \end{cases}$
- $\mathcal{C}[\![\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\sigma = \textbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \textbf{then}\ \mathcal{C}[\![c_0]\!]\sigma\ \textbf{else}\ \mathcal{C}[\![c_1]\!]\sigma$

Note: It's ok to use $\mathcal{B}$ in the definition of the denotations, since it's not circular.

$$\textbf{if}\ \ \sigma = \perp\ \ \textbf{then}\ \ \perp\ \textbf{else}\ \ \ldots$$

Let us now try to define the denotation for **while**. In a similar manner with the above definitions, we would like to write:

$$\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\sigma = \textbf{if}\ \ \neg\mathcal{B}[\![b]\!]\sigma\ \textbf{then}\ \ \sigma\ \textbf{else}\ \ \mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!](\mathcal{C}[\![c]\!]\sigma)$$

Unfortunately, this definition is circular: it involves $\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!]$ on both sides. The above is actually not a definition, but an equation. It is called a recursive equation because the value we wish to know on the left recurs on the right.
We can also write this equation as an equation about sets:

$$\begin{aligned} \mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] = &\{(\sigma, \sigma)\ \mid\ \neg\ \mathcal{B}[\![b]\!]\sigma\} \\ &\cup\ \{(\sigma, \perp)\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ \mathcal{C}[\![c]\!]\sigma = \perp\} \\ &\cup\ \{(\sigma, \sigma')\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ \mathcal{C}[\![c]\!]\sigma = \sigma'' \neq \perp\ \wedge\ \mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\sigma'' = \sigma'\} \end{aligned}$$

We can rewrite this as
$\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] = \{(\sigma, \sigma)\ \mid\ \neg\mathcal{B}[\![b]\!]\sigma\}$

$$\cup\ \{(\sigma,\bot)\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ (\sigma,\bot)\ \in\ \mathcal{C}[\![c]\!]\}$$
$$\cup\ \{(\sigma,\sigma')\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ (\sigma,\sigma'')\ \in\ \mathcal{C}[\![c]\!]\ \wedge\ (\sigma'',\sigma')\ \in\ \mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\}$$

where $\sigma'' \neq \bot$.

We must come up with an alternative definition for **while**. In order to do this, we will construct a function Γ such that the denotation of *while* is a fixed point of Γ.

We define $\Gamma(f)$ (where f is a command denotation) as follows:

$$\Gamma = \lambda f\ \in\ \Sigma \mapsto \Sigma_\bot.\{(\sigma,\sigma)\ \mid\ \neg\mathcal{B}[\![b]\!]\sigma\}$$
$$\cup\ \{(\sigma,\bot)\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ (\sigma,\bot)\ \in\ \mathcal{C}[\![c]\!]\}$$
$$\cup\ \{(\sigma,\sigma')\ \mid\ \mathcal{B}[\![b]\!]\sigma\ \wedge\ (\Sigma,\Sigma'')\ \in\ \mathcal{C}[\![c]\!]\ \wedge\ (\sigma'',\sigma')\ \in\ f\}$$

where $\sigma'' \neq \bot$.

Then, the denotation of *while* is a fixed point of Γ, i.e.,

$$\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = \Gamma(\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]).$$

Mathematically, this can be written as follows:

$$\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma = \mathrm{fix}(\lambda \mathrm{f}.\ \textbf{if}\, \mathcal{B}[\![\mathrm{b}]\!]\sigma\, \textbf{then}\, \sigma\, \textbf{else}\, \mathrm{f}(\mathcal{C}[\![\mathrm{c}]\!]\sigma)).$$

In the next lecture, we will see how to define the least fixed point operator *fix* (in the sense that it terminates as early as possible) for the domain $\Sigma \to \Sigma_\bot$.