CS256: Programming Languages and Semantics

http://www.eecs.harvard.edu/~greg/cs256sp2006/

Greg Morrisett
greg@eecs.harvard.edu
327 Maxwell Dworkin

February 1, 2006

1 What is this course really about?

This is really a course about *modeling* and *proving* properties about programming systems. The systems in question could be a programming language such as Java, or it could be a procotol such as TCP, or it could be a banking web service.

It's pretty easy to construct models: we simply write an interpreter. Of course, to understand the model, we have to understand the semantics of the *meta-language* that we used to express the model. And while it can be easy to write down some model of a system, it's often difficult to use the model in any productive way, other than to serve as a sort of reference specification. There's a real *art* to constructing useful models so that they make the proving or analyzing bits easy.

2 What will I learn?

This course is structured into two major parts. The first part is a rigorous introduction to three kinds of models: operational, denotational, and axiomatic models. Operational models are the easiest to write, require no hairy math, and as a result, scale up to realistic languages, protocols, and systems. However, it can be relatively difficult to analyze an operational model because they typically aren't very abstract. Denotational and axiomatic models require a bit more math and logic (at least for interesting systems), and are in some sense, a bit harder to construct. However, they tend to make certain reasoning tasks easier and more uniform. For instance, establishing the input-output equivalence of two programs is usually much easier in a denotational setting than in an operational setting.

The second part of the course is focused on programming language mechanisms. We'll start at the very beginning with just the (simply-typed) lambda calculus and work our way up through advanced features including control operators (e.g., `callcc`, threads); advanced typing features (parametric, subtype, and bounded polymorphism); encapsulation mechanisms (types, modules, classes, etc.). We'll study how these mechanisms work, how to easily model them, how they interact with each other, and how we can establish properties and avoid pitfalls when integrating them in languages. We'll also see how particular linguistic mechanisms help or hinder analysis and efficient implementation of languages.

Along the way, we're going to experiment with *mechanical proof development environments*.

3 What are the logistics?

We'll meet twice a week (Wed. and Fri. 2:30-4:00pm in MD 221).

There are no required texts for this course. I will provide lecture notes (like this). However, I can recommend the following texts (which are on reserve in the library):

- *The Formal Semantics of Programming Languages*, Glynn Winskel, MIT Press.
- *Types and Programming Languages*, Benjamin C. Pierce, MIT Press.
- *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce, ed., MIT Press.

You'll have a take-home assignment about once a week. Some of the assignments will involve coding interpreters. Some of the assignments will involve hand-writing proofs. Some of the assignments will involve writing interpreters and proofs within a mechanized theorem proving environment.

You'll also have a final project (in lieu of an exam). The final project can be a survey paper on a research topic, or perhaps a mechanical model and proof of something bigger than what we do in class. Alternatively, the final project can be some sort of implementation or the development of some research idea. At the end of each month, I'll ask that you send me a progress report on your project.

The homeworks will make up 70% of your grade, and the final project will make up 20% of your grade. The last 10% of your grade will be determined by class participation. So speak up!

4 Intro to Operational Semantics

We'll start by modeling an extremely simple imperative programming language called IMP. The *abstract syntax* for IMP is given below by the following BNF definitions:

$$
\begin{array}{rcll}
x & \in & \text{Id} & \\
i & \in & \text{Integer} & \\
\oplus & \in & \text{Binop} & ::= \quad + \mid * \mid - \\
e & \in & \text{Exp} & ::= \quad x \mid i \mid e_1 \oplus e_2 \\
c & \in & \text{Com} & ::= \quad x\texttt{:=}e \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c
\end{array}
$$

Note that we're using abstract syntax here, not concrete syntax. Formally, a command, or expression is a (finite) tree, but for convenience sake, I'm writing these as strings. We'll formalize these BNF definitions a bit later.

Identifiers in IMP are globally scoped, and range over integers (bignums). Intuitively, a program is a command which we run in the context of some store (or memory) to produce a new store. We model stores as functions from identifiers to integers:

$$s \in \text{Store} = \text{Id} \to \text{Integer}$$

To make the meaning of commands (and expressions) precise, we'll construct a small-step, structured operational semantics. What we're essentially going to do is formally describe an abstract machine that takes a configuration and steps to a new configuration, representing one "instruction" being executed in the command.

Our configurations will be a pair $\langle c, s\rangle$ of a command and a store. Our step relation will take one configuration to another written as:

$$\langle c_1, s_1\rangle \longmapsto \langle c_2, s_2\rangle$$

A terminal configuration will be of the form $\langle \texttt{skip}, s\rangle$ representing a program that has completed execution. We'll write

$$\langle c_1, s_1\rangle \longmapsto^* s$$

if there exists a sequence of zero or more steps that produce a configuration $\langle \text{skip}, s\rangle$.

We'll define the step relation using inference rules and an auxiliary relation for evaluating expressions.

$$(C1)\quad \langle x \texttt{ := } i, s\rangle \longmapsto \langle \texttt{skip}, s[x \mapsto i]\rangle$$

$$(C2)\quad \frac{\langle e, s\rangle \longmapsto \langle e', s\rangle}{\langle x \texttt{ := } e, s\rangle \longmapsto \langle x \texttt{:=} e', s'\rangle}$$

$$(C3)\quad \langle \texttt{skip}; c, s\rangle \longmapsto \langle c, s\rangle$$

$$(C4)\quad \frac{\langle c_1, s\rangle \longmapsto \langle c_1', s'\rangle}{\langle c_1; c_2, s\rangle \longmapsto \langle c_1'; c_2, s'\rangle}$$

$$(C5)\quad \langle \texttt{if i then } c_1 \texttt{ else } c_2, s\rangle \longmapsto \langle c_1, s\rangle \qquad (i \neq 0)$$

$$(C6)\quad \langle \texttt{if 0 then } c_1 \texttt{ else } c_2, s\rangle \longmapsto \langle c_2, s\rangle$$

$$(C7)\quad \frac{\langle e, s\rangle \longmapsto \langle e', s'\rangle}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, s\rangle \longmapsto \langle \texttt{if } e' \texttt{ then } c_1 \texttt{ else } c_2, s'\rangle}$$

$$(C8)\quad \langle \texttt{while } e \texttt{ do } c, s\rangle \longmapsto \langle \texttt{if } e \texttt{ then } \{c; \texttt{ while } e \texttt{ do } c\} \texttt{ else skip}, s\rangle$$

Here are the auxiliary relations for expressions:

$$(E1)\quad \langle x, s\rangle \longmapsto \langle s(x), s\rangle$$

$$(E2)\quad \langle i_1 \oplus i_2, s\rangle \longmapsto \langle i, s\rangle \qquad (i == i_1 \oplus i_2)$$

$$(E3)\quad \frac{\langle e_1, s\rangle \longmapsto \langle e_1', s'\rangle}{\langle e_1 \oplus e_2, s\rangle \longmapsto \langle e_1' \oplus e_2, s'\rangle}$$

$$(E4)\quad \frac{\langle e_2, s\rangle \longmapsto \langle e_2', s'\rangle}{\langle i \oplus e_2, s\rangle \longmapsto \langle i \oplus e_2, s'\rangle}$$

I claim (and we'll later prove) that for any configuration $\langle c, s\rangle$ there is at most one rule whose conclusion the configuration matches. That is, at most one rule can fire for a given configuration. In other words, $\longmapsto$ is a partial function from configurations to configurations.

For example, consider the configuration:

$$\langle x \texttt{ := } 3 + 4, s\rangle$$

Rule C1 doesn't apply because 3+4 is not an integer – it's a compound expression. So only rule C2 can possibly apply. It only applies if we can prove $\langle 3 + 4, s\rangle \longmapsto \langle i, s\rangle$ for some integer i. That follows from rule E2. But does E3 also apply? No, because that would require us to find a rule that matches $\langle 3, s\rangle \longmapsto ?$ but no such rule exists.

Formally, when we're proving that a configuration steps, we need to produce a derivation. In the case of the example above, the derivation would look like this:

$$(C1)\ \frac{(E2)\ \dfrac{}{\langle 3 + 4, s\rangle \longmapsto \langle 7, s\rangle}}{\langle x \texttt{ := } 3 + 4, s\rangle \longmapsto \langle x \texttt{ := } 7, s\rangle}$$

And note that:

$$(C1)\frac{}{\langle x \texttt{ := } 7, s\rangle \longmapsto \langle \texttt{skip}, s[x \mapsto 7]\rangle}$$

So we can claim that $\langle x \texttt{ := } 3+4, s\rangle \longmapsto^* \langle \texttt{skip}, s[x \mapsto 7]\rangle$. In other words, we can consider $\longmapsto^*$ as also being a partial function from configurations to configurations. It's partial because not every command terminates.

For instance there is no state s such that $\langle \texttt{while 1 do } c, s\rangle \longmapsto^* \langle \texttt{skip}, s'\rangle$. That is, we cannot find a *finite* sequence of n configurations $M_1, M_2, \ldots, M_n$ such that

$$\langle \texttt{while 1 do } c, s\rangle \longmapsto M_1 \longmapsto M_2 \longmapsto \cdots \longmapsto M_n \longmapsto \langle \texttt{skip}, s'\rangle.$$

5 Big Step Semantics

The operational semantics for IMP given in the last section was a *small-step*, structured operational semantics. It mirrors the individual steps that a standard computer might make in evaluating the program. Of course, often we wish to abstract away from the details of how many steps a computation takes, or what intermediate states it steps through. For those purposes, it's often useful to define an alternative "big-step" or *natural* semantics where evaluation seems to happen in one big step.

We can define a big-step semantics for IMP in two ways: We can leverage the small-step semantics and simply define $\langle c, s\rangle \Downarrow s'$ to mean $\langle c, s\rangle \longmapsto^* \langle \texttt{skip}, s'\rangle$.

Alternatively, we can specify the big-step semantics directly as follows:

$$(C1') \quad \langle \texttt{skip}, s\rangle \Downarrow s$$

$$(C2') \quad \frac{\langle e, s\rangle \Downarrow i}{\langle x \texttt{ := } e, s\rangle \Downarrow s[x \mapsto i]}$$

$$(C3') \quad \frac{\langle c_1, s_1\rangle \Downarrow s_2 \qquad \langle c_2, s_2\rangle \Downarrow s_3}{\langle c_1; c_2, s_1\rangle \Downarrow s_3}$$

$$(C4') \quad \frac{\langle e, s\rangle \Downarrow 1 \qquad \langle c_1, s\rangle \Downarrow s'}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, s\rangle \Downarrow s'} \qquad (i \neq 0)$$

$$(C5') \quad \frac{\langle e, s\rangle \Downarrow 0 \qquad \langle c_2, s\rangle \Downarrow s'}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, s\rangle \Downarrow s'}$$

$$(C6') \quad \frac{\langle \texttt{if } e \texttt{ then } \{c; \texttt{while } e \texttt{ do } c\} \texttt{ else skip}, s\rangle \Downarrow s'}{\langle \texttt{while } e \texttt{ do } c, s\rangle \Downarrow s'}$$

where the big-step evaluation relation for expressions is given as:

$$(E1') \quad \langle i, s\rangle \Downarrow i$$

$$(E2') \quad \langle x, s\rangle \Downarrow s(x)$$

$$(E3') \quad \frac{\langle e_1, s\rangle \Downarrow i_1 \qquad \langle e_2, s\rangle \Downarrow i_2}{\langle e_1 \oplus e_2 \Downarrow i\rangle} \qquad (i = i_1 \oplus i_2)$$

A big-step semantics is usually much easier to write down and corresponds more closely to what we're used to writing when we build an interpreter in a language such as Prolog or ML. It also has a lot fewer rules (in practice) and as a result, doing any kind of proof is generally easier (since the proofs are often done by case analysis on the rules.)

Nevertheless, big-step semantics can make it difficult to distinguish non-termination from errors. For instance, suppose we assume that one of the binary arithmetic operations is division. Then of course division by zero is undefined. As a result, there's no derivation that allows us to conclude $\langle x \texttt{ := } 3/0, s\rangle \Downarrow s'$. And

we saw earlier that there's no derivation for $\langle \texttt{while true do } c, s\rangle \Downarrow s'$. So at this level of abstraction, we cannot easily distinguish bad terminal states from non-termination.

One way to fix this is to add distinguished "answers." For instance, we might say that answers include states or $\bot$ and define $\langle c, s\rangle \Downarrow \bot$ if there is an infinite sequence of configurations $M_1, M_2, M_3, \ldots$ such that $\langle c, s\rangle \longmapsto M_1 \longmapsto M_2 \longmapsto M_3 \longmapsto \cdots$. Note, however, that such a definition has to appeal to the small-step semantics in order to remain constructive.