CS 411 Lecture 1 Introduction and Small-step Semantics 1 Sep 2003
Lecturer: Greg Morrisett

Three basic techniques used to specify the semantics of programming languages.

1. operational semantics: think interpreters
2. denotational semantics: think compilers
3. axiomatic semantics: think query engines

Each has its strength and weakness. We'll primarily focus on operational semantics because (a) it doesn't require any complicated math, (b) it's very easy to make operational semantics concrete (i.e., write them as code), and (c) they scale to realistic, modern languages. The primary drawback is that they don't work so well for proving certain properties about programs (e.g., sub-program equivalence) when compared to denotational approaches.

We can specify an operational semantics for a language by first defining the abstract syntax for the language, then defining abstract machine states, and then defining a transition relation between machine states.

Abstract syntax: think syntax trees. As opposed to concrete syntax which specifies how a string (i.e., a linear sequence of characters) can be parsed unambiguously into program phrases.

Example of concrete syntax: 2 * 3 + 4

Abstract syntax: Add(Multiply(Int(2),Int(3)),Int(4))

(draw tree here)

Alternative abstract syntax: Multiply(Int(2),Add(Int(3),Int(4)))

(draw tree here)

Rules of the surface language specify which abstract syntax tree we pick when given concrete syntax. In this case, the precedence of * is higher, so the first tree is the result of parsing.

We usually specify abstract syntax using BNF (Backus-Naur Form) notation which is remarkably similar to ML datatype definitions:

BNF:

$$x, y, z \in \mathsf{Var}$$
$$i, j, k \in \mathsf{Int}$$
$$e \in \mathsf{Exp} ::= x \mid i \mid e_1 + e_2 \mid e_1 * e_2$$

SML:

```
type var = string
datatype exp = Var of string | Int of int | Add of exp * exp | Multiply of exp * exp
```

Abstract machine states (aka configurations): essentially, the memory of a machine that is executing a program. This can take many, many forms depending upon the language, the level of abstraction, and what details we wish to model.

A typical thing is for the abstract machine state to contain a way to lookup the value of a variable (sometimes called an environment or store). The abstract machine state also includes the program (or expression) that we're evaluating.

$$\begin{aligned} \mathsf{Store} &= \mathsf{Var} \to \mathsf{Int} \\ \mathsf{MachineState} &= \mathsf{Exp} \times \mathsf{Store} \end{aligned}$$

The last piece of the specification is the transition relation which tells us, if the abstract machine is in a given state, what state(s) it can move to:

$$\mathsf{Transition} = \mathsf{MachineState} \times \mathsf{MachineState}$$

We usually write $\mathsf{MS} \to \mathsf{MS}'$ to indicate that machine state MS can step to MS'. Formally, the pair $(\mathsf{MS}, \mathsf{MS}')$ is in the transition relation $\to$.

For example:

$$\begin{aligned} (3 + 4, s) &\to (7, s) \\ (2 * 6, s) &\to (12, s) \\ (x, s) &\to (s(x), s) \end{aligned}$$

In general, there are an infinite number of possible transitions for a given machine state. In our simple language above, this is the case because we have an infinite set of numbers and variables, and more generally, expressions. Not to mention how many different memories we could have.

So, we need some way to generate the set of all possible transitions from a finite description. This is most easily accomplished when the abstract syntax of the language can be decomposed in such a way that, for each kind of expression, there is at most one way for that kind of expression to transition.

We use inference rules to define the transition relation. Here are some inference rules for our little language:

$$\textsf{var}\quad \frac{s(x) = k}{(x, s) \to (k, s)}$$

$$\textsf{plus}\quad \frac{}{(i + j, s) \to (k, s)}\ \text{(where } k \text{ is the sum of } i \text{ and } j\text{)}$$

$$\textsf{times}\quad \frac{}{(i * j, s) \to (k, s)}\ \text{(where } k \text{ is the product of } i \text{ and } j\text{)}$$

$$\textsf{left-plus}\quad \frac{(e_1, s) \to (e_1', s)}{(e_1 + e_2, s) \to (e_1' + e_2, s)}$$

$$\textsf{left-times}\quad \frac{(e_1, s) \to (e_1', s)}{(e_1 * e_2, s) \to (e_1' * e_2, s)}$$

$$\textsf{right-plus}\quad \frac{(e_2, s) \to (e_2', s)}{(i + e_2, s) \to (i + e_2', s)}$$

$$\textsf{right-times}\quad \frac{(e_2, s) \to (e_2', s)}{(i * e_2, s) \to (i * e_2', s)}$$

You should read the rules as "if the stuff above the line is true, then the stuff below the line is true". The cases where there is nothing above the line are always true. These cases are called axioms. For example, the plus and times rules above are axioms.

I've specified the rules using concrete syntax, but we could spell them out with abstract syntax instead. You should always think in terms of the abstract syntax. It's just easier when we write on paper to use concrete syntax instead of the trees. For real code (e.g., ML code), the distinction should be more apparent. In fact, we can specify the (abstract) syntax using inference rules as well (instead of BNF). For example:

$$\frac{}{x \in \mathsf{Exp}}$$

$$\frac{}{i \in \mathsf{Exp}}$$

$$\frac{e_1 \in \mathsf{Exp} \qquad e_2 \in \mathsf{Exp}}{(e_1 + e_2) \in \mathsf{Exp}}$$

$$\frac{e_1 \in \mathsf{Exp} \qquad e_2 \in \mathsf{Exp}}{(e_1 * e_2) \in \mathsf{Exp}}$$

Let's see what happens with a particular expression such as $3 * 4 + 2 * x$ which is really $(3 * 4) + (2 * x)$, executing in some store s that maps x to 21. We can proceed by looking at the bottoms of the rules and see if any of them match our current configuration. If they do, then we can see whether we can prove the stuff above the line. This is a goal-directed search.

For instance, the machine state $((3 * 4) + (2 * x), s)$ matches the left-hand side of the bottom of rule left-plus where we take $e_1 = (3 * 4)$ and $e_2 = (2 * x)$. So, we can show that:

$$\frac{(3 * 4, s) \rightarrow ???}{((3 * 4) + (2 * x), s) \rightarrow ???}$$

Let us guess that the ??? on top is $(12, s)$ and the ??? on the bottom is $(12 + 2, s)$. That is, we have instantiated the rule left-plus to get:

$$\frac{(3 * 4, s) \rightarrow (12, s)}{((3 * 4) + (2 * x), s) \rightarrow (12 + (2 * x), s)}$$

To prove that this is valid, we need to show that the assumptions are provable. That is, we must show that $(3 * 4, s) \rightarrow (12, s)$. But this follows directly from the times axiom. So, a full proof that $((3 * 4) + 2, s) \rightarrow (12 + 2, s)$ looks like this:

$$\textsf{left-plus}\ \frac{\textsf{times}\ \dfrac{}{(3 * 4, s) \rightarrow (12, s)}}{((3 * 4) + (2 * x), s) \rightarrow (12 + (2 * x), s)}$$

Now we can show that

$$\textsf{right-plus}\ \frac{\textsf{right-times}\ \dfrac{\textsf{var}\ \dfrac{s(x) = 21}{(x, s) \rightarrow (21, s)}}{(2 * x, s) \rightarrow (2 * 21, s)}}{(12 + (2 * x), s) \rightarrow (12 + (2 * 21), s)}$$

Then we can show that:

$$\textsf{right-plus}\ \frac{\textsf{times}\ \dfrac{}{(2 * 21, s) \to (42, s)}}{(12 + (2 * 21), s) \to (12 + 42, s)}$$

Finally, we can show:

$$\textsf{plus}\ \frac{}{(12 + 42, s) \to (54, s)}$$

So, we have formally proven that $((3 * 4) + (2 * x), s) \to^* (54, s)$ where $\to^*$ is the reflexive, transitive closure of the $\to$ relation. In particular, $\to^* = \{(\mathsf{MS}, \mathsf{MS}') \mid \exists n \geq 0.\ \mathsf{MS} \to^n \mathsf{MS}'\}$.

Now what? Well, we're in a final machine state where we can take no more transitions. So the "answer" of the computation is $(54, s)$.

Theorem 1. *Given an e such that $e \in \mathsf{Exp}$ and given a store $s : \mathsf{Var} \to \mathsf{Int}$, either e is an integer, or else there exists at most one e' such that $(e, s) \to (e', s)$.*

That is, evaluation of expressions is completely deterministic and the transition relation $\to$ is a partial function from machine states to machine states.

How do we go about proving this property? Answer: by induction on the abstract syntax e. More formally, by induction on the height of the derivation that allows us to conclude that $e \in \mathsf{Exp}$

Why can we reason this way? Because we interpret the inference rules as generating the *smallest* set or relation. In other words, if there is something in a set (e.g., Exp), then it must have been generated by a derivation tree of finite height. So, to prove some property about all elements of the set/relation, we can simply argue by induction over all such derivation trees.