Abstract Machine Semantics

CPSC 509: Programming Language Principles

Ronald Garcia*

12 November 2011

Introduction

In these notes we will be talking about *control* structures in programming languages. To do so, we'll introduce yet another style of *operational semantics*, one that makes program control more apparent. We have already discussed *big-step semantics*, which take the form $t \Downarrow v$, and we've discussed two kinds of *small-step semantics*: *structural operational semantics* which appear in Pierce's textbook, and *reduction semantics*, which appear in Felleisen et al's book and course notes. We'll now talk about a third kind of small-step semantics, called *abstract machine semantics*. Abstract machine semantics operationalize the intuition that I gave you about how to think about running a reduction semantic: Given a program, you have to find an evaluation context and redex, reduce the redex, and then plug the result back into the context. Here we see this in a mechanical form, and it turns out that this semantic approach is a useful guide toward implementing compilers, which must translate your high-level language into low-level instructions that run on some machine with a bunch of registers, some stack memory, and some heap memory.[1]

A Simple Language

Our running example for this introduction is a very simple language of boolean and arithmetic. This language is bare-bones, but has enough content to capture most of the concepts that concern us at this time. We'll first present the language using reduction semantics, and then will return to present it as an abstract machine.

Syntax

$$
\begin{array}{rcl}
n \in \mathbb{Z}, & t \in \textsc{Term}, \quad v \in \textsc{Val}, \quad r \in \textsc{Redex}, \quad E \in \textsc{ECtxt} & \\
t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid n \mid t + t \\
v & ::= & n \mid \mathsf{true} \mid \mathsf{false} \\
r & ::= & v + v \mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \\
E & ::= & \Box \mid E[\Box + t] \mid E[v + \Box] \mid E[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t]
\end{array}
$$

Notions of Reduction

$$
\begin{array}{rcl}
n_1 + n_2 & \rightsquigarrow & n_3 \text{ where } n_3 = n_1 + n_2 \\
\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 & \rightsquigarrow & t_1 \\
\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2 & \rightsquigarrow & t_2
\end{array}
$$

[1] Nowadays, most computers have just plain ole' memory, which you can use to implement a stack and heap, but these concepts are so ingrained in the brains of programmers that it's helpful to at least at first maintain the fiction that they are two truly distinct things.

One-Step Reduction

$$\frac{t \rightsquigarrow t'}{E[t] \longrightarrow E[t']}$$

In the syntax of reduction semantics, we explicitly state the grammar of redexes r, those expressions that are subject to the notions of reduction. Some instances of our redexes, however, are stuck, for instance if 7 then 5 else false $\in$ REDEX. We could have restricted the definition of redexes further to disallow this kind of expression, but we chose not to.

Motivation for Abstract Machine Semantics

In some sense, the reduction semantics style takes the *congruence rules* of structural operational semantics, like the rule for if :

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

and expresses them succinctly as a grammar of *evaluation contexts*.

$$E ::= \ldots \mid E[\text{if } \square \text{ then } t \text{ else } t] \ldots$$

We can then view evaluation as a succession of breaking a program down into a context and redex and applying a notion of reduction:

$$t_0 = E_0[r_0] \longrightarrow E_0[t_0'] = t_1 = E_1[r_1] \longrightarrow E_1[t_1'] = t_2 = E_2[r_2] \longrightarrow \ldots$$

After each reduction step $E_i[r_i] \longrightarrow E_i[t_i'] = t_{i+1}$, it is necessary to re-decompose the program $t_{i+1} = E_{i+1}[r_{i+1}]$ to find the next redex to reduce using $\longrightarrow$. All of this is quite high-level and depends on the truth of a particular theorem.

Theorem 1 (Unique Decomposition)**.** *For any program t, either t is a value v, or there exists a unique context E and unique redex r such that $t = E[r]$.*

Proving this theorem is somewhat involved, but its truth guarantees that evaluation is either finished (when $t = v$), can make progress ($t = E[r]$ and r can be reduced by some notion of reduction), or is stuck ($t = E[r]$ and r is one of those "broken" redexes. The proof guarantees as well that the programming language is deterministic, because if there is an E and r such that $t = E[r]$, then both E and r are unique: there is only one possible step at this point in the program.

Reduction semantics are nice and abstract, but for some purposes, they can be seen as too abstract. Just knowing that unique decomposition holds isn't enough: we want to *find* that unique decomposition. A language implementation modeled after reduction semantics would have to do the work of decomposing and then making a reduction step. Naturally a sufficient proof of unique decomposition contains within it an algorithm to find unique context-redex pairs, but the most straightforward proof technique implies a rather inefficient (but informative) algorithm. Even solving this problem is not quite enough to be satisfying. Both reduction semantics and structural operational semantics are somewhat unsatisfying in that at each step of program reduction, each rebuilds the entire program and then decomposes it again (which manifests in the reduction example above as $E_0[t_1'] = t_1 = E_1[r_1]$). Real language implementations are typically not so inefficient.

Abstract Machine Semantics

We now show how a reduction semantics can be elaborated into a more realistic model of program execution, one that does not keep decomposing and rebuilding a program after each reduction step. An *abstract machine semantics* presents a high-level model of how a computer might actually go about running a program. Historically, abstract machines were the first kind of operational semantics, introduced by Peter Landin in the highly influential 1964 paper "The Mechanical Evaluation of Expressions." However, abstract machines were considered too low-level by many researchers, and this led to the development of structural

operational semantics (by Plotkin) and reduction semantics (by Felleisen). However, it was observed that a reduction semantics is just an abstract representation of an abstract machine.

In this section we expand the reduction semantics above to form an abstract machine. This machine implements the deterministic search for a context-redex pair that can be reduced. Then, after a reduction, the machine proceeds directly from the current decomposition $E_n[t'_n]$ to find the next redex, without completely plugging the term and starting over from scratch. This approach is much more efficient in practice.

The abstract machine is defined by a set of *configurations*, which for our simple language are comprised of evaluation context-term pairs. The machine operates in four different *modes*, which are reflected in a grammar of configurations:

$$\begin{array}{lll} C \in \textsc{Cfg} \\ C & ::= & \langle E, t\rangle_{\text{focus}} \mid \langle E, r\rangle_{\text{reduce}} \mid \langle E, v\rangle_{\text{return}} \mid v \end{array}$$

The *focus* mode represents when the abstract machine is searching downward into a term for a redex. The *reduce* mode represents when the abstract machine has found a redex and is ready to reduce it. The *return* mode represents when the machine has found or produced a value, and is returning that value to the current context in order to find the next piece of work to do. The final mode, a standalone value v, denotes that reduction has concluded with a final value.

The heart of the abstract machine semantics is the single step relation

$$\longrightarrow \subseteq \text{CFG} \times \text{CFG}$$

which captures how the machine configurations evolve over time. Since our language happens to be deterministic, we can consider $\longrightarrow$ to be a partial function, but in general that is not the case, so we simply consider it to be a binary relation.

We break the step relation down based on which mode it is in. The *focus* steps transition based on the top-level structure of the term position.

$$\begin{array}{rcl} \langle E, t_1 + t_2\rangle_{\text{focus}} & \longrightarrow & \langle E[\Box + t_2], t_1\rangle_{\text{focus}} \\ \langle E, \textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3\rangle_{\text{focus}} & \longrightarrow & \langle E[\textsf{if } \Box \textsf{ then } t_2 \textsf{ else } t_3], t_1\rangle_{\text{focus}} \\ \langle E, v\rangle_{\text{focus}} & \longrightarrow & \langle E, v\rangle_{\text{return}} \end{array}$$

Each of these steps only looks at the top-level structure of a term. For instance:
$\langle E, 5 + 6\rangle_{\text{focus}} \longrightarrow \langle E[\Box + 6], 5\rangle_{\text{focus}}$, since it only checked that the expression was a plus. It naively examines the left-hand term of the addition expression to continue evaluation. However, when a focus configuration is focused upon a value, it transitions to a *return* configuration with the same context and term (which we know to be a value). Thus, $\langle E[\Box + 6], 5\rangle_{\text{focus}} \longrightarrow \langle E[\Box + 6], 5\rangle_{\text{return}}$

The *return* steps examine the innermost piece of the current context to find out what the next step of work should be.

$$\begin{array}{rcl} \langle \Box, v\rangle_{\text{return}} & \longrightarrow & v \\ \langle E[\Box + t], v\rangle_{\text{return}} & \longrightarrow & \langle E[v + \Box], t\rangle_{\text{focus}} \\ \langle E[v_1 + \Box], v_2\rangle_{\text{return}} & \longrightarrow & \langle E, v_1 + v_2\rangle_{\text{reduce}} \\ \langle E[\textsf{if } \Box \textsf{ then } t \textsf{ else } t], v\rangle_{\text{return}} & \longrightarrow & \langle E, \textsf{if } v \textsf{ then } t \textsf{ else } t\rangle_{\text{reduce}} \end{array}$$

Here is where using inside-out contexts helps us: they are naturally suited for easily examining the innermost *frame* of the evaluation context. If one has arrived at a value, and the evaluation context has been exhausted, then there is no more work to do: reduction concludes by producing that value. In the case of our running example, the next step is to focus on the right-hand side of the addition expression, which is also a value, so the machine transitions back to return, but at that point, the machine has accumulated enough information to know that it is safe to reduce. The corresponding reduction sequence is:

$$\langle E[\Box + 6], 5\rangle_{\text{return}} \longrightarrow \langle E[5 + \Box], 6\rangle_{\text{focus}} \longrightarrow \langle E[5 + \Box], 6\rangle_{\text{return}} \longrightarrow \langle E, 5 + 6\rangle_{\text{reduce}}\,.$$

The *reduce* steps arise when the machine has finally found a redex and is ready to perform a "real" reduction step. Up to now all the machine has been doing is searching for a context-redex pair. The reduce

steps correspond directly to our notions of reduction.

$$\begin{array}{rcl} \langle E, n_1 \texttt{ + } n_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, n_3\rangle_{\text{focus}} \text{ where } n_3 = n_1 + n_2 \\ \langle E, \texttt{if true then } t_1 \texttt{ else } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_1\rangle_{\text{focus}} \\ \langle E, \texttt{if false then } t_1 \texttt{ else } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \end{array}$$

In each case, the reduce configuration transitions to a focus transition. It's clear to see that in the first case (addition) the machine could immediately transition to a return configuration, but that's simply an optimization. So our running example proceeds as follows.

$$\langle E, 5 \texttt{ + } 6\rangle_{\text{reduce}} \longrightarrow \langle E, 11\rangle_{\text{focus}} \longrightarrow \langle E, 11\rangle_{\text{return}} \longrightarrow \ldots.$$

In essence, each reduction step is followed by examining the result of the reduction, thereby taking advantage of all of the information that has been stored in the evaluation context E.

Now, given the definition of $\longrightarrow$, we can specify the corresponding evaluator for this language. The initial state of the machine is when the context is empty and the machine is focusing on the entire program ($\langle \Box, t\rangle_{\text{focus}}$). The end state of the machine is represented by a final value v, which must have arisen from the machine is returning a final value to an empty evaluation context (i.e., $\langle \Box, v\rangle_{\text{return}}$). Given these, the definition of the evaluator follows:

$$\mathit{eval}(t) = v \text{ iff } \langle \Box, t\rangle_{\text{focus}} \longrightarrow^* v.$$

Notice how the evaluation context literally plays the role of the *runtime stack* that you probably learned about in an early programming class. The basic structure of any computation is that when the machine is in state $\langle E, t\rangle_{\text{focus}}$, it is about to analyze t all the way to some value v, which will be returned to the current stack as $\langle E, v\rangle_{\text{return}}$. In big-step semantics, we can see the part of the same phenomenon by following the big-step reduction tree upwards. A rule of the form

$$\frac{t_1 \Downarrow n_1 \qquad t_2 \Downarrow n_2}{t_1 \texttt{ + } t_2 \Downarrow n_3} \text{ where } n_3 = n_1 + n_2$$

first evaluates t_1 all the way down to n_1, "remembering" that the next thing to do is evaluate t_2, before summing the results. This is made much more explicit in the abstract machine semantics, since the evaluation context is explicitly carrying this memory. And this is why abstract machine semantics are a clear way of expressing programming language control-constructs. Language features that implement nontraditional control patterns can be expressed by manipulating the evaluation context. We'll see examples of this in the near future.