Lecture Notes on Dynamic Semantics

15-411: Compiler Design
Frank Pfenning

Lecture 13
October 7, 2014

1 Introduction

In the previous lecture we have specified the *static semantics* of a small imperative language. In this lecture we proceed to discuss its *dynamic semantics*, that is, how programs execute. The relationship between the dynamic semantics for a language and what a compiler actually implements is usually much less direct than for the static semantics. That's because a compiler doesn't actually run the program. Instead, it translates it from some source language to a target language, and then the program in the target language is actually executed.

In our context, the purpose of writing down the dynamic semantics is therefore primarily to precisely specify how programs are supposed to execute. Just the exercise of writing this down formally should help us think about the special cases and make sure our implementation is correct.

Another important purpose is to verify properties of the language itself in a formal (mathematical) way. Much of the theory of programming languages is concerned with just that and therefore requires an operational semantics. A third purpose is to actually *prove* that a compiler is correct. That requires at least two operational specifications: one for the source language and one for the target language. To date, this still requires a major research effort (and is, in any case, out of the scope of this course).

2 Evaluating Expressions

When trying to specify the operational semantics of a programming language, there are a bewildering array of choices regarding the *style* of presentation. Some choices are natural semantics, structural operational semantics, abstract machines, substructural operational semantics, and many more. Having developed *substructural operational semantics* (SSOS) myself, I have a natural bias towards that style of specification. It has the great virtue that in many cases one can extend the language with new constructs without having to rewrite the rules already in place. However, it requires some machinery, namely *substructural logic*, which is a little more extensive than what I would like to introduce in this course. So instead I am using an *abstract machine*, despite some of its shortcomings.

An *abstract*, which is a form of so-called *small-step operational semantics*, we step through the evaluation of an expression e until we have reached a value v. So the basic judgment might be written $e \quad \longrightarrow \quad e'$. However, this is much to simplistic. For example, it does not represent the call stack, or the current value of the variables that are recorded in an *environment*, or what to do with the eventual value. We will introduce such semantic artefacts one by one, as they are needed.

Consider the expression $e_1 + e_2$. By the left-to-right evaluation rule, we first have to evaluate e_1 and then e_2. So why we evaluate e_1 we have to "remember" that we still have to evaluate e_2 then sum up the value. The information on what we still have to do is collected in a so-called *continuation* K. We write the judgment as

$$e \rhd K$$

which we read as "*evaluate expression* e *and pass the result to the continuation* K". In the continuation there is a "hole" (written as an underscore character _) in which we plug in the value passed to it. So:

$$e_1 + e_2 \rhd K \qquad \longrightarrow \qquad e_1 \rhd (\_ + e_2 \; , \; K)$$

When e_1 has been reduced to a value c_1, we plug it into the hole and evaluate e_2 next;

$$c_1 \rhd (\_ + e_2 \; , \; K) \qquad \longrightarrow \qquad e_2 \rhd (c_1 + \_ \; , \; K)$$

Finally, when e_2 has been reduced to a value c_2 we perform the actual addition and pass the result to K.

$$c_2 \rhd (c_1 + \_ \; , \; K) \qquad \longrightarrow \qquad c \rhd K \qquad (c = c_1 + c_2 \bmod 2^{32})$$

In the last rule we appeal to the mathematical operation of addition modulo 2^{32} on two given integers modulo 2^{32}. Since we describe C0, we assume that constants are 32-bit words in two's complement representation. All other binary modular arithmetic operations $\oplus$ are handled in a similar way, so we summarize them as

$$\begin{array}{lcll} e_1 \oplus e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \oplus e_2 \; , \; K) & \\ c_1 \rhd (\_ \oplus e_2 \; , \; K) & \longrightarrow & e_2 \rhd (c_1 \oplus \_ \; , \; K) & \\ c_2 \rhd (c_1 \oplus \_ \; , \; K) & \longrightarrow & c \rhd K & (c = c_1 \oplus c_2 \bmod 2^{32}) \end{array}$$

For an effectful operation such as division, the last step could also raise an arithmetic exception arith (`SIGFPE` on the x86 architecture family, numbered 8). How do we represent that? We would like to abort the computation entirely and go to a state where the final outcome is regorded as an arithmetic exception. We describe this as follows:

$$\begin{array}{lcll} e_1 \oslash e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \oslash e_2 \ , K) & \\ c_1 \rhd (\_ \oslash e_2 \ , K) & \longrightarrow & e_2 \rhd (c_1 \oslash \_ \ , K) & \\ c_2 \rhd (c_1 \oslash \_ \ , K) & \longrightarrow & c \rhd K & (c = c_1 \oslash c_2) \\ c_2 \rhd (c_1 \oslash \_ \ , K) & \longrightarrow & \mathsf{exception}(\mathsf{arith}) & (c_1 \oslash c_2 \text{ undefined}) \end{array}$$

Here, some care must be taken to define the value $c_1 \oslash c_2$ correctly in the cases of division and modulus, and the conditions under which the result is mathematically "undefined" (like division by zero) and therefore must raise an excaption. We have specified this in previous lectures and assignments, so we won't detail the conditions here.

What happens when evaluation finishes normally? In the case of the empty continuation we stop the abstract machine and return $\mathsf{value}(c)$

$$c \rhd \cdot \quad \longrightarrow \quad \mathsf{value}(c)$$

Boolean expression work similarly. We show three examples: constants true and false (which are represented as 1 and 0), and the short-circuiting conjunction (&&).

$$\begin{array}{lcl} \mathsf{true} \rhd K & \longrightarrow & 1 \rhd K \\ \mathsf{false} \rhd K & \longrightarrow & 0 \rhd K \\ e_1 \ \&\& \ e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \ \&\& \ e_2 \ , K) \\ 0 \rhd (\_ \ \&\& \ e_2 \ , K) & \longrightarrow & 0 \rhd K \\ 1 \rhd (\_ \ \&\& \ e_2 \ , K) & \longrightarrow & e_2 \rhd K \end{array}$$

Notice how e_2 is ignored in case 0 is returned to the continuation $(\_ \ \&\& \ e_2 \ , K)$, which encodes the short-circuiting behavior of the conjunction.

3 Variables

We can continue along this line, be we get stuck for variables. Where do their values come from? We need to add an *environment* η that maps variables to their values. We write

$$\eta ::= \cdot \mid \eta, x \mapsto v$$

and $\eta[x \mapsto v]$ for either adding $x \mapsto v$ to η or overwriting the current value of x by v (if $\eta(x)$ is already defined). The state of the abstract machine now contains the environment η. We separate by a turnstile ($\vdash$) from the expression to evaluate and its continuation.

$$\eta \vdash e \rhd K$$

The rules so far just carry this along. For example:

$$\begin{array}{llll}
\eta \vdash e_1 \oplus e_2 \rhd K & \longrightarrow & \eta \vdash e_1 \rhd (\_ \oplus e_2 \ , K) & \\
\eta \vdash c_1 \rhd (\_ \oplus e_2 \ , K) & \longrightarrow & \eta \vdash e_2 \rhd (c_1 \oplus \_ \ , K) & \\
\eta \vdash c_2 \rhd (c_1 \oplus \_ \ , K) & \longrightarrow & \eta \vdash c \rhd K & (c = c_1 \oplus c_2 \bmod 2^{32})
\end{array}$$

Variables are just looked up in the environment.

$$\eta \vdash x \rhd K \quad \longrightarrow \quad \eta \vdash \eta(x) \rhd K$$

Because we are interested in evaluating only expression that have already passed all static semantic checks of the language, we know that $\eta(x)$ will be defined (all variables must be initialized before they are used).

4 Executing Statements

Executing statements in L3, the fragment of C0 we have considered so far, can either complete normally, return from the current function with a return statement, or raise an exception. The "normal" execution of a statement does not pass a value to its continuation; instead it has an effect on its environment by assigning to its existing variables or declaring new ones. We write this as

$$\eta \vdash s \blacktriangleright K$$

where the continuation K should start with a statement. We think of a statement s as poassing on a void value to K, which we write as $(\,)$. For example:

$$\begin{array}{lll}
\eta \vdash \mathsf{nop} \blacktriangleright K & \longrightarrow & \eta \vdash (\,) \blacktriangleright K \\
\eta \vdash \mathsf{seq}(s_1, s_2) \blacktriangleright K & \longrightarrow & \eta \vdash s_1 \blacktriangleright (s_2 \ , K) \\
\eta \vdash (\,) \blacktriangleright (s \ , K) & \longrightarrow & \eta \vdash s \blacktriangleright K
\end{array}$$

The last line codifies that if there no further statement to execution, we grab the first statement from the continuation. When executing an assignment we first have to evaluate the assignment, then change the variable value in the environment.

$$\begin{array}{lll}
\eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{assign}(x, \_) \ , K) \\
\eta \vdash c \blacktriangleright (\mathsf{assign}(x, \_) \ , K) & \longrightarrow & \eta[x \mapsto c] \vdash (\,) \blacktriangleright K
\end{array}$$

Conditionals follow the pattern of the short-circuiting conjunction.

$$\begin{array}{lll}
\eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{if}(\_, s_1, s_2) \ , K) \\
\eta \vdash 1 \rhd (\mathsf{if}(\_, s_1, s_2)) & \longrightarrow & \eta \vdash s_1 \blacktriangleright K \\
\eta \vdash 0 \rhd (\mathsf{if}(\_, s_1, s_2)) & \longrightarrow & \eta \vdash s_2 \blacktriangleright K
\end{array}$$

While loops are a bit more complicated. We take a slight shortcut by using the identity

$$\mathsf{while}(e, s) \equiv \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop})$$

to avoid writing out several rules implementing the right-hand side of this identity directly.

$$\eta \vdash \mathsf{while}(e, s) \blacktriangleright K \quad \longrightarrow \quad \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K$$

Loops bring up the question of nontermination, which is modeled naturally: we just have abstract machine transitions $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$ without ever arriving at a final state. The final states are just $\mathsf{value}(c)$ and $\mathsf{exception}(\mathsf{arith})$.

Declarations are also pretty straightforward, since they just add a new variable with undefined value to the environment.

$$\eta \vdash \mathsf{decl}(x, \tau, s) \blacktriangleright K \quad \longrightarrow \quad \eta[x \mapsto \_] \vdash s \blacktriangleright K$$

In a language that permits shadowing of variables, we would have to save the current value of x and restore it after s has finished executing. Or we would think of η as a list where $\eta(x)$ refers to the value of the rightmost occurrence, which would be removed when we leave the scoope of declaration. Or we could statically rename the variables during elaboration so that no shadowing can occur during execution.

At this point we have handled all salient statements except return statements that are tied to function calls. We discuss them in the next section.

5 Function Calls

A function call first has to evaluate the function arguments, from left to right. Then we invoke the function, whose body starts to execute in an environment that maps that formal parameters of the function to the argument values. But meanwhile we have to save the current environment of the caller somewhere. Similarly, we also have to save the continuation of the caller, so that when the callee returns we pass it the return value. So a stack frame $\langle \eta, K \rangle$ consists of an environment η and a continuation K.

$$\text{Call stack} \quad S \quad ::= \quad \cdot \mid S \,,\, \langle \eta, K \rangle$$

Now states representing evaluation of expression and execution of statements have the forms

$$\begin{array}{ll} \text{Evaluation} & S \,;\, \eta \vdash e \rhd K \\ \text{Execution} & S \,;\, \eta \vdash s \blacktriangleright K \end{array}$$

We only show the special case of evaluation function calls with two and zero arguments, for simplicity.

$$\begin{array}{lcl}
S\ ;\eta \vdash f(e_1, e_2) \rhd K & \longrightarrow & S\ ;\eta \vdash e_1 \rhd (f(\_, e_2)\ ,\ K) \\
S\ ;\eta \vdash c_1 \rhd (f(\_, e_2)\ ,\ K) & \longrightarrow & S\ ;\eta \vdash e_2 \rhd (f(c_1, \_)\ ,\ K) \\
S\ ;\eta \vdash c_2 \rhd (f(c_1, \_)\ ,\ K) & \longrightarrow & (S\ ,\ \langle \eta, K\rangle)\ ;[x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot \\
& & \quad (f(x_1, x_2)\{s\}) \\
S\ ;\eta \vdash f(\,) \rhd K & \longrightarrow & (S\ ,\ \langle \eta, K\rangle)\ ;\cdot \vdash s \blacktriangleright \cdot \qquad (f(\,)\{s\})
\end{array}$$

In the last state we see a new environment with values for x_1 and x_2 and a new stack frame save the caller's environment η and continuation K. We indicate the definition of f with formal parameters x_1, x_2 and body s. The empty continuation "·" will never be reached since the static semantics checks that every control flow path through the statement s terminates in a return statement. In order to support functions returning void, we would add an empty return to the end of the function body during elaboration and add a few additional rules similar to the given ones.

When executing a return statement we simply have to restore the caller's environment and continuation from the stack and pass the return value to the caller's continuation.

$$\begin{array}{lcl}
S\ ;\eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow & S\ ;\eta \vdash e \rhd (\mathsf{return}(\_)\ ,\ K) \\
S\ ,\ \langle \eta', K'\rangle\ ;\eta \vdash c \rhd (\mathsf{return}(\_)\ ,\ K) & \longrightarrow & S\ ;\eta' \vdash c \rhd K'
\end{array}$$

We start the machine initially in a state where we call the main function, and we stop the abstract machine if we reach this continuation.

$$\begin{array}{lcll}
\cdot\ ;\cdot \vdash \mathsf{main}(\,) \rhd \cdot & & & \text{(initial state)} \\
\cdot\ ;\eta \vdash c \rhd \cdot & \longrightarrow & \mathsf{value}(c) & \text{final state}
\end{array}$$

which will eventually step to $\mathsf{value}(c)$, where c is returned by the main function.

6 Summary

We use $\odot$ to stand for either a pure operation $\oplus$, or a potentially effectful operation $\oslash$ as well as shift, comparison, and bitwise operators.

Expressions	e	$::=$	$c \mid e_1 \odot e_2 \mid \mathsf{true} \mid \mathsf{false} \mid e_1\ \texttt{\&\&}\ e_2 \mid x \mid f(e_1, e_2) \mid f(\,)$
Statements	s	$::=$	$\mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{assign}(x, e) \mid \mathsf{decl}(x, \tau, s)$
		$\mid$	$\mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \mid \mathsf{return}(e)$
Values	v	$::=$	$c \mid (\,)$
Environments	η	$::=$	$\cdot \mid \eta, x \mapsto c$
Stacks	S	$::=$	$\cdot \mid S\ ,\ \langle \eta, K \rangle$
Cont. frames	ϕ	$::=$	$\_ \odot e \mid c \odot \_ \mid \_\ \texttt{\&\&}\ e \mid f(\_, e) \mid f(c, \_)$
		$\mid$	$s \mid \mathsf{assign}(x, \_) \mid \mathsf{if}(\_, s_1, s_2) \mid \mathsf{return}(\_)$
Continuations	K	$::=$	$\cdot \mid \phi\ ,\ K$
Machine states	s	$::=$	$S\ ;\ \eta \vdash e \rhd K$
		$\mid$	$S\ ;\ \eta \vdash s \blacktriangleright K$
		$\mid$	$\mathsf{value}(c)$
		$\mid$	$\mathsf{exception}(\mathsf{arith})$

A computation is a sequence of machine states which could be infinite, terminate in state $\mathsf{value}(c)$, or raise $\mathsf{exception}(\mathsf{arith})$. The initial state, by convention, is $\cdot\ ;\ \cdot \vdash \mathsf{main}(\,) \rhd \cdot$.

$$\begin{array}{lcl}
S \,;\, \eta \vdash e_1 \odot e_2 \rhd K & \longrightarrow & S \,;\, \eta \vdash e_1 \rhd (\_ \odot e_2 \,,\, K) \\
S \,;\, \eta \vdash c_1 \rhd (\_ \odot e_2 \,,\, K) & \longrightarrow & S \,;\, \eta \vdash e_2 \rhd (c_1 \odot \_ \,,\, K) \\
S \,;\, \eta \vdash c_2 \rhd (c_1 \odot \_ \,,\, K) & \longrightarrow & S \,;\, \eta \vdash c \rhd K \qquad (c = c_1 \odot c_2) \\
S \,;\, \eta \vdash c_2 \rhd (c_1 \odot \_ \,,\, K) & \longrightarrow & \mathsf{exception}(\mathsf{arith}) \qquad (c_1 \odot c_2 \text{ undefined}) \\
 & & \\
S \,;\, \eta \vdash \mathsf{true} \rhd K & \longrightarrow & S \,;\, \eta \vdash 1 \rhd K \\
S \,;\, \eta \vdash \mathsf{false} \rhd K & \longrightarrow & S \,;\, \eta \vdash 0 \rhd K \\
S \,;\, \eta \vdash e_1 \mathbin{\&\&} e_2 \rhd K & \longrightarrow & S \,;\, \eta \vdash e_1 \rhd (\_ \mathbin{\&\&} e_2 \,,\, K) \\
S \,;\, \eta \vdash 0 \rhd (\_ \mathbin{\&\&} e_2 \,,\, K) & \longrightarrow & S \,;\, \eta \vdash 0 \rhd K \\
S \,;\, \eta \vdash 1 \rhd (\_ \mathbin{\&\&} e_2 \,,\, K) & \longrightarrow & S \,;\, \eta \vdash e_2 \rhd K \\
 & & \\
S \,;\, \eta \vdash x \rhd K & \longrightarrow & S \,;\, \eta \vdash \eta(x) \rhd K \\
 & & \\
S \,;\, \eta \vdash \mathsf{nop} \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash (\,) \blacktriangleright K \\
S \,;\, \eta \vdash \mathsf{seq}(s_1, s_2) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash s_1 \blacktriangleright (s_2 \,,\, K) \\
S \,;\, \eta \vdash (\,) \blacktriangleright (s \,,\, K) & \longrightarrow & S \,;\, \eta \vdash s \blacktriangleright K \\
 & & \\
S \,;\, \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash e \rhd (\mathsf{assign}(x, \_) \,,\, K) \\
S \,;\, \eta \vdash c \blacktriangleright (\mathsf{assign}(x, \_) \,,\, K) & \longrightarrow & S \,;\, \eta[x \mapsto c] \vdash (\,) \blacktriangleright K \\
 & & \\
S \,;\, \eta \vdash \mathsf{decl}(x, \tau, s) \blacktriangleright K & \longrightarrow & S \,;\, \eta[x \mapsto \_] \vdash s \blacktriangleright K \\
 & & \\
S \,;\, \eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash e \rhd (\mathsf{if}(\_, s_1, s_2) \,,\, K) \\
S \,;\, \eta \vdash 1 \rhd (\mathsf{if}(\_, s_1, s_2)) & \longrightarrow & S \,;\, \eta \vdash s_1 \blacktriangleright K \\
S \,;\, \eta \vdash 0 \rhd (\mathsf{if}(\_, s_1, s_2)) & \longrightarrow & S \,;\, \eta \vdash s_2 \blacktriangleright K \\
 & & \\
S \,;\, \eta \vdash \mathsf{while}(e, s) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K \\
 & & \\
S \,;\, \eta \vdash f(e_1, e_2) \rhd K & \longrightarrow & S \,;\, \eta \vdash e_1 \rhd (f(\_, e_2) \,,\, K) \\
S \,;\, \eta \vdash c_1 \rhd (f(\_, e_2) \,,\, K) & \longrightarrow & S \,;\, \eta \vdash e_2 \rhd (f(c_1, \_) \,,\, K) \\
S \,;\, \eta \vdash c_2 \rhd (f(c_1, \_) \,,\, K) & \longrightarrow & (S \,,\, \langle \eta, K \rangle) \,;\, [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot \\
 & & \qquad (f(x_1, x_2)\{s\}) \\
 & & \\
S \,;\, \eta \vdash f(\,) \rhd K & \longrightarrow & (S \,,\, \langle \eta, K \rangle) \,;\, \cdot \vdash s \blacktriangleright \cdot \qquad (f(\,)\{s\}) \\
 & & \\
S \,;\, \eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash e \rhd (\mathsf{return}(\_) \,,\, K) \\
(S \,,\, \langle \eta', K' \rangle) \,;\, \eta \vdash c \rhd (\mathsf{return}(\_) \,,\, K) & \longrightarrow & S \,;\, \eta' \vdash c \rhd K' \\
\cdot \,;\, \eta \vdash c \rhd \mathsf{value}(\_) & \longrightarrow & \mathsf{value}(c)
\end{array}$$