Lecture Notes on Dynamic Semantics

15-411: Compiler Design
Frank Pfenning

Lecture 13
October 8, 2013

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 *natural semantics*, despite some of its shortcomings.

In *natural semantics*, which is a form of so-called *big-step operational semantics*, we relate an expression e directly to its value v. So the basic judgment might be written $\mathsf{eval}(e, v)$. While accurate, this can be a bit lengthy, so we write $e \downarrow v$ instead. Here, e are expression in our (elaborated) abstract syntax, and v are 32 bit integers, interpreted in two's complement representation.

We begin with straightforward rule. The rules of natural semantics are intended to be read bottom-up, from the conclusion to the premise.

$$\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad v = v_1 + v_2 \ (\mathsf{mod}\, 2^{32})}{e_1 \mathbin{+} e_2 \downarrow v}$$

We read this as follows:

> *To evaluate e_1 + e_2 we evaluate e_1 to some value v_1, then e_2 to some value v_2 and return the sum $v_1 + v_2$ in arithmetic modulo 2^{32}.*

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 rule above now carries along η, and the case of a variable looks it up.

$$\frac{\eta \vdash e_1 \downarrow v_1 \quad \eta \vdash e_2 \downarrow v_2 \quad v = v_1 + v_2 \ (\mathsf{mod}\, 2^{32})}{\eta \vdash e_1 \mathbin{+} e_2 \downarrow v} \qquad \frac{\eta(x) = v}{\eta \vdash x \downarrow v}$$

The next problem is posed by operations that may raise an exception. We write $\eta \vdash e \uparrow \mathit{exn}$ if e raises the exception exn. We use a only a few predefined exceptions, and the language provides no way to handle such exceptions, greatly simplifying its semantics. We obtain four rules to specify the behavior of division. We write

$\mathsf{trunc}(x)$ for truncation of x towards 0.

$$\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad \begin{array}{c} -2^{31} \le v_1/v_2 < 2^{31} \\ v = \mathsf{trunc}(v_1/v_2) \end{array}}{e_1 \mathbin{/} e_2 \downarrow v} \qquad \frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad \begin{array}{c} v_2 = 0 \\ \text{or } (v_1 = -2^{31} \text{ and } v_2 = -1) \end{array}}{e_1 \mathbin{/} e_2 \uparrow \mathsf{arith}}$$

$$\frac{e_1 \uparrow exn}{e_1 \mathbin{/} e_2 \uparrow exn} \qquad \frac{e_1 \downarrow v_1 \quad e_2 \uparrow exn}{e_1 \mathbin{/} e_2 \uparrow exn}$$

The last two rules follow from the general convention that we perform left-to-right evaluation of subexpressions. This leads to an unfortunate proliferation of rules. We follow the convention of annotating a rule with $^{(+LR)}$ to indicate that we have omitted additional versions of the rule which can be obtained by replacing premises that converge ($\downarrow v$) by premises that raise an exception ($\uparrow exn$), from left to right, and propagating the exception in the conclusion.

The remaining kind of expressions are fairly straightforward, but we have to remember that some boolean operators shirtcircuit evaluation. We also fix the interpretation of true as 0 and false as 1.

$$\frac{}{\eta \vdash \mathsf{false} \downarrow 0} \qquad \frac{}{\eta \vdash \mathsf{true} \downarrow 1}$$

$$\frac{\eta \vdash e_1 \downarrow 0}{\eta \vdash e_1 \mathbin{\&\&} e_2 \downarrow 0} \qquad \frac{\eta \vdash e_1 \downarrow 1 \quad \eta \vdash e_2 \downarrow v_2}{\eta \vdash e_1 \mathbin{\&\&} e_2 \downarrow v_2}\ {}^{(+LR)}$$

The $^{(+LR)}$ annotation on the second rule means that the following two rules are implied.

$$\frac{\eta \vdash e_1 \uparrow exn}{\eta \vdash e_1 \mathbin{\&\&} e_2 \uparrow exn} \qquad \frac{\eta \vdash e_1 \downarrow 1 \quad \eta \vdash e_2 \uparrow exn}{\eta \vdash e_1 \mathbin{\&\&} e_2 \uparrow exn}$$

3 Relating Static and Dynamic Semantics

The judgments in the static and dynamic semantics are designed to be closely related. We will not prove any of these relationships, but they might help us consider the correctness and completeness of our rules. Here are some of the relationships for expressions.

$$\begin{array}{ccccc} \Gamma & \vdash & e & : & \tau \\ : & & & & : \\ \eta & \vdash & e & \downarrow & v \end{array}$$

This picture expresses that the environment η should match the context Γ and that, furthermore, v should have type τ. We say that η *matches* Γ ($\eta : \Gamma$) if for every

declaration $\Gamma, x{:}\tau$ we have a definition $\eta(x) = v$ with $v : \tau$. The typing for values here is a bit degenerate, but it should stipulate, for example, that only $0 : \mathsf{bool}$ and $1 : \mathsf{bool}$. Note that values are typed without an environment because they are just 32 bit words and cannot contain variables.

The above relationship does not quite hold in our semantics, because not all variables in Γ may have been initialized. But we will have checked statically that

$$\delta \vdash e$$

where $\delta \subseteq \mathsf{dom}(\Gamma)$. So we can refine the above by restricting Γ to the defined variables in δ.

$$\begin{array}{ccccc} \delta & \vdash & e & & \\ \Gamma|_\delta & \vdash & e & : & \tau \\ : & & & & : \\ \eta & \vdash & e & \downarrow & v \end{array}$$

Going back to the earlier rules, we can see the significance of these relationships. For example, we see that in the rules for variables, we can never encounter an uninitialized variable. In the rules for logical conjunction, we see that the two cases for the value of e_1 in e_1 `&&` e_2, namely 0 and 1, capture all possibilities because the value v_1 such that $\eta \vdash e_1 \downarrow v_1$ must be of type `bool`.

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.

- $\eta \vdash s \to \eta'$: executing s in environment e completes normally with environment η'.
- $\eta \vdash s \downarrow [v]$: executing s in environment η does not complete normally, but instead returns value v.
- $\eta \vdash s \uparrow exn$: executing s in environment η raises exception exn.

We start with some simple cases:

$$\frac{}{\eta \vdash \mathsf{nop} \to \eta} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \longrightarrow \eta_2}{\eta \vdash \mathsf{seq}(s_1, s_2) \to \eta_2}\ (+LR)$$

The second rule highlights that sequences of statements are executed left to right. We extend our convention regarding the propogation of exception, where any exception by s_1 is propagated, and an exception in s_2 is propagated if s_1 completes normally. So the annotation $^{(+LR)}$ implies the following two rules:

$$\frac{\eta \vdash s_1 \uparrow exn}{\eta \vdash \mathsf{seq}(s_1, s_2) \uparrow exn} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \uparrow exn}{\eta \vdash \mathsf{seq}(s_1, s_2) \uparrow exn}$$

Because s_1 or s_2 may also execute a return statement, we also need the following additional rules:

$$\frac{\eta \vdash s_1 \downarrow [v]}{\eta \vdash \mathsf{seq}(s_1, s_2) \downarrow [v]} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \downarrow [v]}{\eta \vdash \mathsf{seq}(s_1, s_2) \downarrow [v]}$$

How do these judgments line up with our static semantics?

$$\begin{array}{ccccc} \Gamma|_\delta & \vdash & s & : & [\tau] \\ \delta & \vdash & s & \Rightarrow & \delta' \\ : & & & & : \\ \eta & \vdash & s & \to & \eta' \end{array}$$

The diagram is trying to express that if η matches $\Gamma|_\delta$ then η' matches $\Gamma|_{\delta'}$. In other words, if s completes normally then it will define exactly those variables that the static semantics claimed it must, namely those in δ'. Moreover, all the values have the right type.

For return values we have a related diagram:

$$\begin{array}{ccccc} \delta & \vdash & s & \Rightarrow & \delta' \\ \Gamma|_\delta & \vdash & s & : & [\tau] \\ : & & & & : \\ \eta & \vdash & s & \downarrow & [v] \end{array}$$

That is, if s returns a value v, then that must have the type τ. The rule on the right clearly should satisfy this.

$$\frac{\eta \vdash e \downarrow v}{\eta \vdash \mathsf{return}(e) \downarrow [v]} \qquad \frac{\eta \vdash e \uparrow exn}{\eta \vdash \mathsf{return}(e) \uparrow exn}$$

Assignment straightforwardly updates the environment, propagating exceptions.

$$\frac{\eta \vdash e \downarrow v}{\eta \vdash \mathsf{assign}(x, e) \to \eta[x \mapsto v]}\ {}^{(+LR)}$$

For conditionals, we evaluate only the relevant branch.

$$\frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s_1 \to \eta'}{\eta \vdash \mathsf{if}(e, s_1, s_2) \to \eta'}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s_1 \downarrow [v]}{\eta \vdash \mathsf{if}(e, s_1, s_2) \downarrow [v]}$$

$$\frac{\eta \vdash e \downarrow 0 \quad \eta \vdash s_2 \to \eta'}{\eta \vdash \mathsf{if}(e, s_1, s_2) \to \eta'}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 0 \quad \eta \vdash s_2 \downarrow [v]}{\eta \vdash \mathsf{if}(e, s_1, s_2) \downarrow [v]}$$

Loops are somewhat more interesting. If the loop guard is false, we exit the loop. If it is true, we execute the loop body once (obtaining a new environment η') and then repeat *in the new environment* η'.

$$\frac{\eta \vdash e \downarrow 0}{\eta \vdash \mathsf{while}(e, s) \to \eta}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s \to \eta' \quad \eta' \vdash \mathsf{while}(e, s) \to \eta''}{\eta \vdash \mathsf{while}(e, s) \to \eta''}\ (+LR)$$

We omit the additional, obvious rules for dealing with possible returns.

Loops bring up the question of nontermination. Natural semantics is not particularly well-suited to reflect on nontermination. If, say, an expression e does not terminate in environment η, we cannot find any value v such that $\eta \vdash e \downarrow v$ can be proved, nor is there an exception exn such that $\eta \vdash e \uparrow exn$. But nontermination is a bit stronger than that, because, intuitively, we can always continue with our proof construction but never complete it. In the case of the while loop, the third premise of the second rule would again apply the same rule, with again the same while loop in the third premise, and so on without ever completing.

Declarations are not particularly difficult; we just have to be careful to track the scopes of variables correctly during elaboration so that there are no surprises during execution.

$$\frac{\eta \vdash s \to \eta'}{\eta \vdash \mathsf{decl}(x, \tau, s) \to \eta' \setminus [x \mapsto \_]}\ (+LR)$$

Here we just remove whatever definition might have been given to x during the execution of s.

5 Function Calls

Finally, for this lecture, we come to another connection between statements and expressions: *function calls*. We stack th premises on top of each other so the rule

doesn't become too wide.

$$\frac{\begin{array}{l}\eta \vdash e_1 \downarrow v_1 \\ \cdots \\ \eta \vdash e_n \downarrow v_n \\ f(x_1, \ldots, x_n) = s \\ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash s \downarrow [v]\end{array}}{\eta \vdash f(e_1, \ldots, e_n) \downarrow v} \; (+LR)$$

Here, η is entirely ignored in the body of f (called s), because s only has access to the function parameters $x_1, \ldots, x_n$. If all goes well, we know that s must raise an exception or return a value, it cannot complete normally. That's because we have checked in the static semantics that there is a return statement along each control flow path through s. We can provide an additional version of this rule in case we have a function not returning a value (void), or we can elaborate void into return of a distinguished value, say, 0.

Recall that according to our convention, the function call raises an exception if e_1 does, or if e_1 returns a value and e_2 raises an exception, etc. Finally, any exception from s is passed on.