Computer Science 15-312
MinML, A MiniMaL Functional Language
Part III: Dynamic Semantics

Robert Harper

Spring, 2000

1 Introduction

In this section we consider the *dynamic semantics*, or *rules of evaluation*, for MinML. We will use a technique called *structured operational semantics (SOS)* to specify the evaluation rules for MinML.

The dynamic semantics of MinML is an example of what is known as a *language-based* model of computation, as opposed to a more familiar *machine-based* model. Before giving details, let us first discuss the distinction between these two approaches.

In a machine-based model the dynamic semantics of a language is specified by describing how the language is compiled into code for an (abstract or concrete) machine. That is, we define the language by prescribing how it is to be implemented. This is common practice for languages such as C, which are designed for writing low-level systems code where access to the "bare metal" is essential. However, such code is notoriously non-portable (because real machines differ in critical details such as byte-order, memory model, or data representation). Worse, machine models of computation are inherently "indirect" in the sense that the meaning of a program is defined only in terms of the meaning of another program (the code to which it compiles), rather than in terms of the language itself. This means that programs can exhibit behavior (such as "bus errors" or "segmentation faults") that can only be understood in terms of the implementation, rather than in terms of the language itself. While this can be important for certain systems applications, the vast majority of applications do not (and, arguably, should not) rely on such details.

In contrast, a language-based model is one for which the semantics of a program is given entirely in terms of the constructs of the language itself. This means that no mysterious implementation-specific problems can arise, greatly enhancing portability. Moreover, language-based models support reasoning about program behavior entirely at the level of the program itself, avoiding the need to understand how the language is compiled on a given platform. The overall idea of a language-based model is to treat computation as a form of calculation. In high school you learned to calculate with polynomials. Given a polynomial, you can readily "plug in" (substitute) a value for one or more variables, then carry out the arithmetic steps to determine the final result. You can think of a polynomial as a kind of program, and you can think of the calculation steps you used as a kind of "dynamic semantics" for "executing" polynomials. To put it another way, you can think of a program as a kind of "generalized polynomial" in which we compute by substituting values for variables and executing primitive instructions generalizing the arithmetic operations.

Before going into details, it is important to dispel a possible misunderstanding. The choice of a language-based *vs.* a machine-based model of computation is not independent of the language itself. That is, we cannot expect to use either approach with any given language. Languages that admit language-based models are sometimes called *abstract* languages because they abstract away from the details of the implementation. MinML is an abstract language in this sense, and we shall give it a language-based semantics. Languages that do not admit language-based models are sometimes called *concrete* languages since they are closely tied to actual machines. Nearly all assembly languages, and early dialects of the C language[1], are concrete languages, and are usually given a machine-based semantics. Recent languages, such as Java, occupy a middle ground. The semantics of Java is given by a mapping to an abstract machine, the Java Virtual Machine (JVM). Thus Java is, in a sense, a concrete language, but relative to an abstract machine! The JVM is given a semantics entirely in its own terms, without reference to a lower-level platform.

2 The Dynamic Semantics of **MinML**

The dynamic semantics of MinML is given by (what else?) an inductive definition of the *one-step evaluation* relation, $e \mapsto e'$, between *closed* ex-

[1] Subsequent standardization efforts have attempted to give a more abstract semantics to C.

pressions. Recall that we are modelling computation in MinML as a form of "in place" calculation; the relation $e \mapsto e'$ means that e' is the result of performing a single step of computation starting with e. To calculate the value of an expression e, we repeatedly perform single calculation steps until we reach a *value*, v: a number, a boolean constant, or a function.

The rules defining the dynamic semantics of MinML may be classified into two categories: rules defining the fundamental computation steps (or, *instructions*) of the language, and rules for determining where the next instruction is to be executed. The purpose of the search rules is to ensure that the dynamic semantics is *deterministic*, which means that for any expression there is at most one "next instruction" to be executed.[2]

First the instructions governing the primitive operations. We assume that each primitive operation o defines a total function — given values v_1, $\ldots$, v_n of appropriate type for the arguments, there is a unique value v that is the result of performing operation o on $v_1, \ldots, v_n$. For example, for addition we have the following primitive instruction:

$$\frac{}{\texttt{+}(m, n) \mapsto m + n} \;[\textbf{PlusNums}]$$

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

$$\frac{}{\texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_1} \;[\textbf{IfTrue}]$$

$$\frac{}{\texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_2} \;[\textbf{IfFalse}]$$

The primitive instruction for application is as follows:

$$\frac{(v = \texttt{fun}\,f\;\texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\;\texttt{is}\,e\,\texttt{end})}{\texttt{apply(}v, v_1\texttt{)} \mapsto [v, v_1/f, x]e} \;[\textbf{CallFun}]$$

To apply the function $v = \texttt{fun}\,f\;\texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\;\texttt{is}\,e\,\texttt{end}$ to an argument v_1 (which must be a value!), we substitute the function itself, v, for f, and the argument value, v_1, for x in the body, e, of the function. By substituting v for f we are "unrolling" the recursive function as we go along.

This completes the primitive instructions of MinML. The "search" rules, which determine which instruction to execute next, follow.

[2] Some languages are, by contrast, *non-determinstic*, notably those involving concurrent interaction. We'll come back to those later.

For the primitive operations, we specify a left-to-right evaluation order. For example, we have the following two rules for addition:

$$\frac{e_1 \mapsto e_1'}{\texttt{+}(e_1, e_2) \mapsto \texttt{+}(e_1', e_2)}\ [\textbf{PlusArg}_1] \qquad \frac{e_2 \mapsto e_2'}{\texttt{+}(v_1, e_2) \mapsto \texttt{+}(v_1, e_2')}\ [\textbf{PlusArg}_2]$$

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

$$\frac{e \mapsto e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} \mapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi}}\ [\textbf{IfTest}]$$

For applications, we first evaluate the function position; once that is complete, we evaluate the argument position.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply(}e_1, e_2\texttt{)} \mapsto \texttt{apply(}e_1', e_2\texttt{)}}\ [\textbf{AppFun}] \qquad \frac{e_2 \mapsto e_2'}{\texttt{apply(}v_1, e_2\texttt{)} \mapsto \texttt{apply(}v_1, e_2'\texttt{)}}\ [\textbf{AppArg}]$$

This completes the definition of the MinML one-step evaluation relation.

The *multi-step evaluation relation*, $e \mapsto^* e'$, is inductively defined by the following rules:

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''}$$

In words: $e \mapsto^* e'$ iff performing zero or more steps of evaluation starting from the expression e yields the expression e'. The relation $\mapsto^*$ is sometimes called the *Kleene closure*, or *reflexive-transitive closure*, of the relation $\mapsto$.

3 Properties of the Dynamic Semantics

Let us demonstrate that the dynamic semantics of MinML is well-defined in the sense that it assigns at most one value to each expression. (We should be suspicious if this weren't true of the semantics, for it would mean that programs have no definite meaning.)

First, observe that if v is a value, then there is no e (value or otherwise) such that $v \mapsto e$. Second, observe that the evaluation rules are arranged so that at most one rule applies to any given form of expression, *even though* there are, for example, $n + 1$ rules governing each n-argument primitive operation. These two observations are summarized in the following lemma.

Lemma 1
For every closed expression e, there exists at most one e' such that $e \mapsto e'$. In other words, the relation $\mapsto$ is a partial function.

Proof: By induction on the structure of e. We leave the proof as an exercise to the reader. Be sure to consider *all* rules that apply to a given expression e! ■

It follows that evaluation to a value is deterministic:

Lemma 2
For every closed expression e, there exists at most one value v such that $e \mapsto^* v$.

Proof: Follows immediately from the preceding lemma, together with the observation that there is no transition from a value. ■

4 Summary

This completes the dynamic semantics of MinML. The main points are:

1. It is an example of *language-based* model of computation. The semantics is specified entirely in terms of the language itself, without reference to an implementation on an abstract or concrete machine.

2. It is specified using the technique of *structured operational semantics*, a collection of rules that determine the "next instruction" to be executed in a program, and for determining the result of that execution step.