Computer Science 15-312
Evaluation Semantics and Dynamic Typing

Robert Harper

Spring, 1999

1 Overview

The dynamic semantics of MinML is given by a transition relation $e \mapsto e'$ defined using Plotkin's method of Structured Operational Semantics (SOS). A significant advantage of SOS is that it admits a crisp statement and proof of type safety, in the form of the Progress and Preservation theorems. A disadvantage of SOS is that it is rather detailed in that it defines the evaluation of expressions step-by-step.

Here we consider another formalism for giving dynamic semantics, called *evaluation semantics (ES)*, in which we define directly the evaluation relation, $e \Downarrow v$, by a set of inference rules. These rules define the unique value v (if such exists) for every expression e. In doing so they specify the evaluation order only implicitly, but in the process achieve an appealing simplicity that is lacking in the SOS formulation. On the other hand, evaluation semantics does not admit as a clean formulation of type safety as does SOS, so it is a question of intent as to which is the preferable formalism. Generally speaking, SOS is preferable for proving properties of languages and ES is preferable for describing languages to programmers.

2 Evaluation Semantics

The evaluation relation, $e \Downarrow v$, for MinML is inductively defined by the following inference rules.

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v}$$

$$\frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v}$$

$$\frac{}{\texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end} \Downarrow \texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/f, x]e \Downarrow v}{\texttt{apply(}e_1, e_2\texttt{)} \Downarrow v}$$

(where $v_1 = \texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}$.)

This concludes the definition of the evaluation semantics of MinML. As you can see, the specification is quite small and is very intuitively appealing.

3 Correspondences

The precise relationship between SOS and ES is given by the following theorem.

Theorem 1

1. *If $e \Downarrow v$, then $e \mapsto^* v$.*

2. *If $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. Consequently, if $e \mapsto^* v$, then $e \Downarrow v$.*

Proof:

1. By induction on the rules defining the evaluation relation. The result is clearly true for values, since trivially $v \mapsto^* v$. Suppose that $e = \texttt{apply(}e_1, e_2\texttt{)}$ and assume that $e \Downarrow v$. Then $e_1 \Downarrow v_1$, where $v_1 = \texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}$, $e_2 \Downarrow v_2$, and $[v_1, v_2/f, x]e \Downarrow v$. By induction we have that $e_1 \mapsto^* v_1$, $e_2 \mapsto^* v_2$ and $[v_1, v_2/f, x]e \mapsto^* v$. It follows that $\texttt{apply(}e_1, e_2\texttt{)} \mapsto^* \texttt{apply(}v_1, e_2\texttt{)} \mapsto^* \texttt{apply(}v_1, v_2\texttt{)} \mapsto [v_1, v_2/f, x]e \mapsto^* v$, as required. The other cases are handled similarly.

2. By induction on the rules defining single-step transition. Suppose that $e = \texttt{apply(}v_1, v_2\texttt{)}$, where $v_1 = \texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}$, and $e' = [v_1, v_2/f, x]e$. Suppose further that $e' \Downarrow v$; we are to show that $e \Downarrow v$. Since $v_1 \Downarrow v_1$ and $v_2 \Downarrow v_2$, the result follows immediately from the assumption that $e' \Downarrow v$. Now suppose that $e = \texttt{apply(}e_1, e_2\texttt{)}$ and

$e' = \texttt{apply}(e_1', e_2)$, where $e_1 \mapsto e_1'$. Assume that $e' \Downarrow v$; we are to show that $e \Downarrow v$. It follows that $e_1' \Downarrow v_1$, $e_2 \Downarrow v_2$, and $[v_1, v_2/f, x]e \Downarrow v$. By induction $e_1 \Downarrow v_1$, and hence $e \Downarrow v$. The remaining cases are handled similarly. It follows by induction on the rules defining multi-step evaluation that if $e \mapsto^* v$, then $e \Downarrow v$. The base case, $v \mapsto^* v$, follows from the fact that $v \Downarrow v$. Now suppose that $e \mapsto e' \mapsto^* v$. By induction $e' \Downarrow v$, and hence $e \Downarrow v$ by what we have just proved.

■

4 Type Safety For Evaluation

One disadvantage of evaluation semantics is that it is not sufficiently detailed as to admit a precise definition of type safety. Recall that in the SOS framework we defined type safety as the conjunction of preservation and progress. Preservation states that the type of an expression doesn't change during evaluation; progress states that well-typed programs cannot "get stuck". Evaluation semantics admits an analogue of type preservation (if $e : \tau$ and $e \Downarrow v$, then $v : \tau$), but there appears to be no way to capture the notion of progress (what would it mean to "get stuck"?).

Exercise 2
Prove type preservation for evaluation semantics by induction on the rules defining evaluation.

A standard technique for expressing progress in the framework of evaluation semantics is to reduce progress to preservation for an extended evaluation relation. Specifically, we extend the evaluation relation with rules governing "stuck" expressions stating that the value of such an expression is a special token wrong that is, by definition, ill-typed. Since preservation ensures that the value of a well-typed expression is well-typed, the value cannot be wrong. This leads to the slogan "well-typed programs cannot go wrong."

More precisely, we introduce a special syntactic category of *answers* defined by the following grammar:

$$\textit{Answers} \quad a \quad ::= \quad \mathsf{ok}(v) \mid \mathsf{wrong}$$

Here v is a syntactic value, as defined earlier.

Typing for answers is inductively defined by the following single rule:

$$\frac{\vdash v : \tau}{\vdash \mathsf{ok}(v) : \tau}$$

Observe that we consider only closed answers to be well-typed, and that wrong is ill-typed because there is no typing rule governing it.

The evaluation relation is defined as a binary relation $e \Downarrow a$ between expressions e and answers a defined inductively as follows:

$$\frac{}{n \Downarrow \mathsf{ok}(n)}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(n_1) \quad e_2 \Downarrow \mathsf{ok}(n_2)}{+(e_1, e_2) \Downarrow \mathsf{ok}(n_1 + n_2)}$$

$$\frac{e_1 \Downarrow \mathsf{wrong}}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{wrong}}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq n_1}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(n_1) \quad e_2 \Downarrow \mathsf{ok}(v_2) \quad v_2 \neq n_2}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

(The other primitive operations are handled similarly.)

$$\frac{}{\texttt{true} \Downarrow \mathsf{ok}(\texttt{true})} \qquad \frac{}{\texttt{false} \Downarrow \mathsf{ok}(\texttt{false})}$$

$$\frac{e \Downarrow \mathsf{ok}(\texttt{true}) \quad e_1 \Downarrow a}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow a}$$

$$\frac{e \Downarrow \mathsf{ok}(\texttt{false}) \quad e_2 \Downarrow a}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow a}$$

$$\frac{e \Downarrow \mathsf{wrong}}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow \mathsf{wrong}}$$

$$\frac{e \Downarrow \mathsf{ok}(v) \quad v \notin \{\, \texttt{true}, \texttt{false} \,\}}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow \mathsf{wrong}}$$

$$\frac{}{\texttt{fun}\, f\ (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end} \Downarrow \mathsf{ok}(\texttt{fun}\, f\ (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end})}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{ok}(v_2) \quad [v_1, v_2/f, x]e \Downarrow a}{\texttt{apply}(e_1, e_2) \Downarrow a}$$

(where $v_1 = \texttt{fun}\, f\ (x\,{:}\,\tau_1)\,{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end}$.)

$$\frac{e_1 \Downarrow \mathsf{wrong}}{\texttt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{wrong}}{\texttt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq \texttt{fun}\, f\ (x\,{:}\,\tau_1)\,{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end}}{\texttt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

The changes we have made are these:

1. Introduce rules that check for malformed arguments such as a non-numeral value as argument to a numeric primitive operation, a non-boolean value as a test in a conditonal, or a non-function value being applied to an argument.

2. Introduce rules that propagate wrong from sub-expressions. Once evaluation goes wrong it stays wrong!

Why does this work? We have the following correspondence:

Theorem 3

1. *$e \Downarrow \mathsf{ok}(v)$ iff $e \mapsto^* v$.*

2. *$e \Downarrow \mathsf{wrong}$ iff $e \mapsto^* e' \not\mapsto$, where e' is not a value.*

Thus evaluation to wrong corresponds exactly to getting stuck, and evaluation to $\mathsf{ok}(v)$ corresponds exactly to successful evaluation to a value.

Exercise 4
Re-prove preservation for the extended evaluation semantics by induction on evaluation. For each rule yielding wrong as answer, you must argue that the rule cannot apply, given that the expression is well-typed. For example, if $+(e_1, e_2)$ is well-typed, then by inversion it must have type $\texttt{int}$, as must both e_1 and e_2. By the canonical forms lemma and induction, the evaluation of e_1 and e_2 must yield well-typed answers of the form $\mathsf{ok}(n_1)$ and $\mathsf{ok}(n_2)$, respectively. Therefore none of the three rules yielding wrong can apply.

It is important that you understand how the proof of progress is embedded (contrapositively) in the proof of preservation for the extended semantics.

5 Dynamic Typing

The introduction of an instrumented semantics that checks for type errors during execution suggests another method for ensuring safety. The key observation is that the instrumented semantics makes it possible to execute a program regardless of whether it is well-typed — "stuck" expressions are simply those that evaluate to wrong, rather than those for which evaluation is undefined. This suggests that we can dispense with the static type system entirely, relying instead on the instrumentation to catch type faults during execution. Such languages are said to be *dynamically typed*. The best-known (and first) dynamically-typed language is Lisp.

To understand what is at stake, it is helpful to consider what it would mean to implement the instrumented semantics for MinML given above. This question doesn't arise as long as we restrict our attention to (statically) well-typed programs — the extended preservation theorem ensures that none of the additional rules can ever be executed. But if we are to dispense with static typing we must, of course, implement them.

The key issue is how to implement a rule such as this:

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq \mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}}{\mathtt{apply(}e_1, e_2\mathtt{)} \Downarrow \mathsf{wrong}}$$

in which we check whether the value of the function position of an application is indeed a function. Inside of a machine, all values are simply bit patterns. Functions might be represented as pointers to regions of memory containing machine language instructions. Faced with a word, how can we determine whether or not it is a valid pointer to a valid function? In general we cannot, so we must instead plan to support these checks in our representations of data objects.

Fundamentally, we must be prepared to support the following operations:

1. Testing whether a given value an integer, a boolean, or a function.

2. Extracting from a valid integer its underlying integer value, or from a valid boolean its underlying truth value, or from a valid function its underlying code.

3. Marking a value as an integer, boolean, or function.

One way to implement these operations is via a *tagging* mechanism whereby all values are marked with a tag indicating whether it is an integer, boolean, or function.[1] It is important to distinguish tags from types! The tag on a value only indicates its *shape*, or *outermost form*; it does not indicate anything deeper about the value. For example, all functions would be tagged as functions, without regard to their domain or range types. This is enough to ensure safety; anything more than this is overkill (and would require much more work to implement.)

The main ideas of tag checking can be illustrated by a few rules. Let us define the set of *tagged values* by the following grammar:

$$t \;::=\; \texttt{Int}\,(n) \mid \texttt{Bool}\,(\texttt{true}) \mid \texttt{Bool}\,(\texttt{false}) \mid \texttt{Func}\,(\texttt{fun}\,f\,(x\texttt{:}\tau_1)\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end})$$

The set of answers is re-defined in terms of tagged values:

$$a \;::=\; \mathsf{ok}(t) \mid \mathsf{wrong}$$

The evaluation rules are modified to make explicit the manipulation of tags. For example, the rules for numbers are as follows:

$$\frac{}{n \Downarrow \mathsf{ok}(\texttt{Int}\,(n))}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_2))}{+(e_1, e_2) \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1 + n_2))}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Bool}\,(v_1))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Func}\,(v_1))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Bool}\,(v_2))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Func}\,(v_2))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

The other rules are handled similarly. Observe that evaluation of an integer constant demands work to attach a tag to the value, and that the evaluation of a primitive operation requires that these tags be checked for validity and that the result of a valid addition be tagged as an integer.

If we were assuming a static type discipline, all this machinery would be unfortunate because we are forced to pay the cost of tag manipulation, even

[1] In a richer language more tags would be required, for example to indicate whether a value is a pair or a floating point number, etc.

though the static type checks prove that it is unnecessary. This semantics accrues advantage only in the case that we are interested in running programs that are not statically typed (or even statically typable). Examples include

1. Conditionals with branches of differing type. For example, in a dynamically typed language the expression `if true then 1 else true fi + 7` is safe to execute and will not incur a run-time type error. Yet it is statically ill-typed.

2. Aggregates, such as lists, with values of different type. For example, in a dynamically typed variant of ML we could form the list `[1,"one",true,fn x=>x]`, even though it is ill-typed.

These examples are said to be useful, precisely because they avoid rejecting programs that do not incur a type fault at run-time, even though it looks as though they might when analyzed purely statically (without regard to run-time behavior). In this sense type faults are isolated *as late as possible*, rather than *as early as possible* (and hence conservatively) as in a statically-typed framework.

To make this work well a dynamically typed language must support run-time type checking (really, run-time tag checking). For example, if l is a heterogeneous list and we wish to compute with its third element, we must perform a run-time test to determine whether it is an integer, or boolean, or function. Having made this determination we can use it safely, without fear of incurring a run-time type fault.

Much energy has been expended on debating the relative merits of static and dynamic typing, with the interlocutors in the debate typically advocating one position or the other as if they are in opposition to one another. But this debate misses a fundamental point: *static and dynamic typing are not opposite viewpoints on language design, rather dynamic typing is a special case of static typing (in a sufficiently rich type system)!*

The point can be made quite succinctly by observing the encoding of dynamic typing directly in ML using recursive data types. Consider the following program:

```
(* The type of tagged values. *)
datatype tagged =
  Int of int | Bool of bool | Function of tagged -> tagged
exception TypeError
fun checked_add (m:tagged, n:tagged):tagged =
    case (m,n) of
      (Int a, Int b) => Int (a+b)
    | (_, _) => raise TypeError
fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
      Function g => g a
    | _ => raise TypeError
```

Here we introduce a type of tagged data values, and define primitive operations on it corresponding to the "checked" version of those operations in the instrumented dynamic semantics. This representation of tagging is fully faithful to the checked operational semantics. Moreover, it is that the virtue that we pay for tag checking exactly to the extent that we use it! Addition still corresponds to a single machine instruction, provided that we're adding integers; if, instead, we're adding tagged values, we pay the cost of pattern matching against the tag and tagging the result. If we wish to use dynamic checking, we can, but we are not forced to do so. In contrast a dynamically typed language admits *only* the checked operations; we cannot recover the more efficient behavior of a statically typed language without sacrificing safety.