15–212: Fundamental Structures of Computer Science II

Some Notes on Evaluation

Frank Pfenning

Draft of September 4, 1997

These notes provide a brief introduction to evaluation the way it is used for proving properties of ML programs. We assume that the reader is already familiar with ML.

When proving the correctness of a concrete program (when compared to the correctness of an abstract algorithm), it is paramount to refer to an underlying definition of the programming language. For our purposes, it is most convenient if this definition is *operational*, that is, we describe how expressions evaluate.

For simplicity we deal only with pure ML programs, that is, the only effects we allow are non-termination and exceptions, which are modelled by allowing an expression not to have a value.

As the language is organized around its types, so will the definition of the operational semantics. This definition is not complete or fully formalized—for such a definition the interested and intrepid reader is referred to the *Definition of Standard ML (Revised)*.

1 Notation

It will be critical for an understanding of the definitions and proof that we distinguish between a mathematical entity (such as an integer or a real number) and its representation as an object in ML. Again, for the sake of simplicity, our formal proofs will ignore limits of the machines realizing ML. For example, we assume that there are ML representation of all integers and real numbers. Given a mathematical object o, we write $\overline{o}$ for the representation of o in ML. We use a `typewriter font` for expressions in ML and *italics* for mathematical expressions.

We write e for arbitrary expressions in ML and v for values, which are a special kind of expression. We write

$e \hookrightarrow v$	expression e *evaluates to* value v
$e \stackrel{1}{\Longrightarrow} e'$	expression e *reduces to* e' in 1 step
$e \stackrel{k}{\Longrightarrow} e'$	expression e *reduces to* e' in k steps
$e \Longrightarrow e'$	expression e *reduces to* e' in 0 or more steps

Our notion of *step* in the operational semantics is defined abstractly and will not coincide with the actual operations performed in an implementation of ML. Since we will be mainly concerned with proving correctness, but not complexity of implementation, the number of steps is largely irrelevant and we will write $e \Longrightarrow e'$ for reduction.

Evaluation and reduction are related in the sense that if $e \hookrightarrow v$ then $e \stackrel{1}{\Longrightarrow} e_1 \stackrel{1}{\Longrightarrow} \cdots \stackrel{1}{\Longrightarrow} v$ and *vice versa*.

Note that values evaluate to themselves "in 0 steps". In particular, for a value v there is no expression e such that $v \stackrel{1}{\Longrightarrow} e$.

2 Integers

Types. `int`.
Values. For every integer n there is an ML object $\overline{n}$.
Operations. e_1 `+` e_2, e_1 `-` e_2, e_1 `*` e_2, e_1 `div` e_2, e_1 `mod` e_2, and others which we omit here.
Typing Rules. e_1 `+` e_2 : `int` if e_1 : `int` and e_2 : `int` and similarly for the other operations.
Evaluation. Evaluation of arithmetic expressions proceeds from left to right, until we have obtained values (which are always representation of integers). More formally:

$$\begin{array}{rcll} e_1 \texttt{ + } e_2 & \stackrel{1}{\Longrightarrow} & e_1' \texttt{ + } e_2 & \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ \overline{n_1} \texttt{ + } e_2 & \stackrel{1}{\Longrightarrow} & \overline{n} \texttt{ + } e_2' & \text{if } e_2 \stackrel{1}{\Longrightarrow} e_2' \\ \overline{n_1} \texttt{ + } \overline{n_2} & \stackrel{1}{\Longrightarrow} & \overline{n_1 + n_2} & \end{array}$$

3 Real Numbers

Analogous to integers. The only new phenomenon is division by zero, which does not have a value. The evaluation rule for division is then qualified:

$$\overline{x}\texttt{/}\overline{y} \quad \stackrel{1}{\Longrightarrow} \quad \overline{(\tfrac{x}{y})} \quad \text{for } y \neq 0$$

4 Booleans

Types. `bool`.
Values. `true` and `false`.
Operations. `if` e_1 `then` e_2 `else` e_3.
Typing Rules.
`if` e_1 `then` e_2 `else` $e_3 : t$
if e_1 : `bool`
and $e_2 : t$
and $e_3 : t$
Note that this rule applies for any type t and forces both branches of the conditional to have the same type.
Evaluation. First we evaluate the condition and then one of the branches of the conditional, depending on its value.

$$\begin{array}{rcll} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 & \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ \texttt{if true then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & e_2 & \\ \texttt{if false then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & e_3 & \end{array}$$

5 Products

We only show the situation for pairs; arbitrary tuples are analogous.
Types. t_1 `*` t_2 for any type t_1 and t_2.
Values. `(`v_1`,` v_2`)` for values v_1 and v_2.
Operations. One can define projections, but in practice one mostly uses pattern matching (see below).
Typing Rules.

$(e_1\texttt{,}\ e_2) : t_1 \texttt{ * } t_2$
if $e_1 : t_1$
and $e_2 : t_2$.

Evaluation. Tuples are evaluated from left to right.

$$\begin{array}{lcll} (e_1\texttt{,}\ e_2) & \stackrel{1}{\Longrightarrow} & (e_1'\texttt{,}\ e_2) & \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ (v_1\texttt{,}\ e_2) & \stackrel{1}{\Longrightarrow} & (v_1\texttt{,}\ e_2') & \text{if } e_2 \stackrel{1}{\Longrightarrow} e_2' \end{array}$$

6 Functions

We start with simple functions and later extend it to clausal function definitions.

Types. t_1 `->` t_2 for any type t_1 and t_2.

Values. `(fn (x:`t_1`) =>` e_2`)` for any type t_1 and expression e_2.

Operations. The only operation is application $e_1\ \ e_2$, written as juxtaposition.

Typing Rules.

`(fn (x:`t_1`) =>` e_2`)` $: t_1$ `->` t_2
if $e_2 : t_2$ assuming `x` $: t_1$.

$e_2\ \ e_1 : t_2$
if $e_2 : t_1$ `->` t_2
and $e_1 : t_1$.

Evaluation. Applications are evaluated by first evaluating the function, then the argument, and then substituting the actual parameter (= argument) for the formal parameter (= variable) in the body of the function.

$$\begin{array}{rcll} e_1\ \ e_2 & \stackrel{1}{\Longrightarrow} & e_1'\ \ e_2 & \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ v_1\ \ e_2 & \stackrel{1}{\Longrightarrow} & v_1\ \ e_2' & \text{if } e_2 \stackrel{1}{\Longrightarrow} e_2' \\ \texttt{(fn (x:}t_1\texttt{) => } e_2\texttt{)}\ v_1 & \stackrel{1}{\Longrightarrow} & [v_1/\texttt{x}]e_2 & \end{array}$$

where $[v_1/\texttt{x}]e_2$ is the notation for substituting v_1 for occurrences of the parameter `x` in e_2. This substitution must respect the rules of scope for variables.

In presentation of proofs, identifiers bound to functions (and sometimes other values) are not expanded into their corresponding value in order to shorten the presentation. In other words, we do not consider looking up the value of an identifier in the environment as an explicit step in evaluation.

7 Patterns

Patterns p, which can be used in clausal function definitions, are either variables, constants, or tuples of patterns. Patterns must be *linear*, that is, each variable may occur at most once. With datatype declarations, we will later see one other case, namely a value constructor applied to an argument.

The general form of a function definition is then

$$\begin{array}{l} \texttt{(fn } p_1 \texttt{ => } e_1 \\ \ \ \texttt{| } p_2 \texttt{ => } e_2 \\ \ \ \ldots \\ \ \ \texttt{| } p_n \texttt{ => } e_n\texttt{)} \end{array}$$

Such a function will have type t `->` s if every pattern p_i has type t and every expression e_i has type s. When we check if pattern p_i has type t, we have to assign appropriate types to the variables in p_i. We may assume the types of these variables when checking e_i. For example:

`(fn (x,y) => (x+1) * (y-1)) : (int * int) -> int`

since `(x+1) * (y-1) : int` assuming `x : int` and `y : int`. These assumptions arise, since the pattern `(x,y)` must have type `int * int`.

To evaluate an application we proceed as before: we first evaluate the function then the argument part. The resulting expression

```
(fn p1 => e1
  | p2 => e2
  ...
  | pn => en) v
```

is evaluated by *matching* the value v against each pattern in turn, starting with p_1. If the value matches a pattern p_i, it will provide a *substitution* for the variables in the pattern. These substitutions are applied to e_i and the resulting expression is evaluated. For example, given the definition

```
fun fact' (0, k) = k
  | fact' (n, k) = fact' (n-1, n*k)
```

we have

`fact' (3,1)` $\Longrightarrow$ `fact' (3-1, 3*1)`

since

1. matching the value `(3,1)` against the pattern `(0,k)` fails,
2. matching the value `(3,1)` against the pattern `(n,k)` succeeds with the substitution of `3` for `n` and `1` for `k`,
3. substituting `3` for `n` and `1` for `k` in `fact' (n-1, n*k)` yields `fact' (3-1, 3*1)`.