15–150: Principles of Functional Programming
Some Notes on Evaluation

Michael Erdmann*

Spring 2013

These notes provide a brief introduction to evaluation the way it is used for proving properties of SML programs. We assume that the reader is already familiar with SML. We deal here only with pure functional SML programs that may raise exceptions (no other side-effects).

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.

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 SML (Revised)*.

1 Notation

For the sake of simplicity, we generally will not distinguish between a mathematical entity (such as an integer or a real number) and its representation as an object in SML. Similarly, for simplicity, our formal proofs will ignore limits of the machines realizing SML. For example, we assume that there are SML representation of all integers and real numbers. We use a `typewriter font` for actual SML code and *italics* more generally for mathematical or SML expressions and values.

We write e for arbitrary expressions in SML 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 SML. 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 $v \stackrel{1}{\Longrightarrow} e$.

*Modified from a draft by Frank Pfenning for 15-212, 1997.

Extensional Equivalence

We say that two expressions e and e' are *extensionally equivalent*, and write $e \cong e'$, whenever one of the following is true: (i) evaluation of e produces the same value as does evaluation of e', or (ii) evaluation of e raises the same exception as does evaluation of e', or (iii) evaluation of e and evaluation of e' both loop forever. In other words, evaluation of e appears to behave just as does evaluation of e'. NOTE: Extensional equivalence is an equivalence relation on well-typed SML expressions.

Referential Transparency

A functional language obeys a fundamental principle known as *Referential Transparency*: in any functional program one may replace any expression with any other extensionally equivalent expression without affecting the value of the program.

Referential transparency is a powerful principle that supports reasoning about functional programs. Roughly speaking, this is substitution of "equals for equals", a notion so familiar from mathematics that one does it all the time without making a fuss. While this may sound obvious, in fact this principle is extremely useful in practice, and it can lend support to program optimization or simplification steps that help develop better programs.

Aside: It is often said that imperative languages do not satisfy referential transparency, and that only purely functional languages do. This is inaccurate: imperative languages also obey a form of referential transparency, but one needs to take account not only of values but also of side-effects, in defining what "equivalent" means for imperative programs.

For functional programs, because evaluation causes no side-effects, if ones evaluates an expression twice, one obtains the same result. And the relative order in which one evaluates (non-overlapping) sub-expressions of a program makes no difference to the value of the program, so one may in principle use parallel evaluation strategies to speed up code while being sure that this does not affect the final value.

2 Integers

Types. `int`.
Values. All the integers (given our assumptions on page 1).
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:

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

We ignore any limitations imposed by particular implementations, such as restrictions on the number of bits in the representation of integers. Note that some expressions have no values. For example, there is no value v such that `3 div 0` $\Longrightarrow v$.

3 Real Numbers

Analogous to integers. Of course, in the implementation these are represented as floating point values with limited precision. As a result it is almost never appropriate to compare values of type `real` for equality (which can be done with the function `Real.==`).

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.

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 \quad \text{if } e_1 \overset{1}{\Longrightarrow} e_1'$$
$$\texttt{if true then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} e_2$$
$$\texttt{if false then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} e_3$$

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`,` e_2`)` : t_1 `*` t_2
if $e_1 : t_1$
and $e_2 : t_2$.

Evaluation. Tuples are evaluated from left to right.

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

6 Functions

We start with simple functions and later extend this 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

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

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`. [Why is that? Because `x+1` and `y-1`, and thus `x` and `y`, must each have the same type as `1`, namely `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) ⟹ 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)`.