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

Michael Erdmann*

Spring 2020

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 representations 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. When we are mainly concerned with proving correctness but not complexity of an implementation, the number of steps is largely irrelevant and we will frequently simply 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.

### Extensional Equivalence

We say that two expressions $e$ and $e'$ of the same (nonfunction) type 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, defined for pairs of well-typed expressions of the *same type*.

Two functions `f` and `g` of type `t -> t'` are said to be extensionally equivalent precisely when `f(v)` and `g(u)` are extensionally equivalent for all extensionally equivalent values `v` and `u` of type `t`. Formally, `f` $\cong$ `g` if and only if `f(v)` $\cong$ `g(u)` for all values `v:t` and `u:t` with `v` $\cong$ `u`.

### 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 purely 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

**Type:** `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:** Sequential evaluation of arithmetic expressions proceeds from left to right, until we have obtained values (which are always representations of integers). More formally:

| | | | |
|---|---|---|---|
| $e_1$ `+` $e_2$ | $\stackrel{1}{\Longrightarrow}$ | $e_1'$ `+` $e_2$ | if $e_1 \stackrel{1}{\Longrightarrow} e_1'$ |
| $n_1$ `+` $e_2$ | $\stackrel{1}{\Longrightarrow}$ | $n_1$ `+` $e_2'$ | if $e_2 \stackrel{1}{\Longrightarrow} e_2'$, and with $n_1$ an integer value |
| $n_1$ `+` $n_2$ | $\stackrel{1}{\Longrightarrow}$ | $n$ | with $n$ the integer value representing the sum of the integer values $n_1$ and $n_2$ |

We ignore any limitations imposed by particular implementations, such as restrictions on the number of bits in the representation of integers. Note that some well-typed expressions have no values. For example, `(3 div 0):int`, but 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.

(One can compare two real numbers using the function `Real.==` but not with `=`. However, this is dangerous: Due to floating point arithmetic, two mathematically equal real numbers, such as `a*b` and `b*a`, may actually turn out not be equal in the computer.)

# 4 Booleans

**Type:** `bool`.
**Values:** `true` and `false`.
**Operations:** $e_1$ `orelse` $e_2$, $e_1$ `andalso` $e_2$, `if` $e_1$ `then` $e_2$ `else` $e_3$, $e_1$ `<` $e_2$, etc.

**Typing Rules:** $e_1$ `orelse` $e_2$ : `bool` if $e_1$ : `bool` and $e_2$ : `bool`. (Similarly for other operations.)

`if` $e_1$ `then` $e_2$ `else` $e_3$ : $t$
if $e_1$ : `bool`
and $e_2 : t$
and $e_3 : t$

Observe that the last rule applies for any type $t$ and forces both branches of the conditional to have the same type.

**Evaluation:** The sequential evaluation rules for basic Boolean operations are much like they are for integer operations, left to right, etc. So we focus here on evaluating the `if-then-else` expression. First we evaluate the condition and then one of the branches of the conditional, depending on its value:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 \qquad (\text{if } e_1 \overset{1}{\Longrightarrow} e_1')$$

$$\texttt{if true then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} e_2$$

$$\texttt{if false then } e_2 \texttt{ else } e_3 \overset{1}{\Longrightarrow} e_3$$

Consequently, observe that the expression

```
if 3 > 4 then 10 div 0 else 17
```

has type `int` and value `17` whereas the expression

```
if 3 < 4 then 10 div 0 else 17
```

also has type `int` but has no value.

**CAUTION:** It is very easy to abuse `if-then-else` expressions. NEVER EVER write something like `if n=0 then true else false`. Just write `n=0`. After all, `n=0` is an expression of type `bool`. Think types!

Moreover, SML's powerful pattern matching facility makes many `if-then-else` expressions unnecessary. When testing for a constant like 0, it is better to use pattern matching, either via a case expression or function clauses. Similarly, rather than nest multiple Boolean tests, frequently it is better to case on a tuple. See page 6.

# 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 page 5).
**Typing Rules:**

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

**Evaluation:** Sequential evaluation of tuples is 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:`$t_1$`) =>` $e_b$`)` for any type $t_1$ and expression $e_b$
(the *formal parameter* of the function is $x$ and the *body* of the function is $e_b$).
**Operations:** The only operation is application $e_2\ e_1$, written as juxtaposition.
**Typing Rules:**

- `(fn (x:`$t_1$`) =>` $e_b$`)` : $t_1$ `->` $t_2$
  if $e_b : t_2$ 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 (i.e., argument) for the formal parameter (i.e., variable) when evaluating the body of the function:

$$\begin{array}{lcll} \text{(evaluate function)} & e_2\ e_1 & \stackrel{1}{\Longrightarrow} & e_2'\ e_1 \quad \text{if } e_2 \stackrel{1}{\Longrightarrow} e_2' \\ \text{(evaluate argument)} & v_2\ e_1 & \stackrel{1}{\Longrightarrow} & v_2\ e_1' \quad \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ \text{(evaluate body)} & \texttt{(fn (x:}t_1\texttt{) =>}\ e_b\texttt{)}\ v_1 & \stackrel{1}{\Longrightarrow} & [v_1/\texttt{x}]e_b \end{array}$$

where $[v_1/\texttt{x}]$ indicates a binding of value $v_1$ to variable `x`. When evaluating $e_b$, we may therefore substitute $v_1$ for occurrences of the parameter `x` throughout $e_b$. This substitution must respect the rules of scope for variables.

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

**Totality:** We say that a function `f` : $t_1$ `->` $t_2$ is *total* if `f(x)` reduces to a value for all values `x` : $t_1$. Implicit in this definition is that `f` itself reduces to a (function) value.

# 7 Patterns

Patterns $p$, which can be used in clausal function definitions, are either constants, variables, or tuples of patterns. Patterns must be *linear*, that is, each variable may occur at most once. One may also use a wildcard, designated by `_`, as a pattern. (Unlike variables, wildcards can appear multiple times as subpatterns in a pattern.)

Comment for the future: Once we cover datatype declarations, we will see one other instance of patterns, namely a value constructor applied to an argument.

The general form of a function definition then is:

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

Such a function will have type $t$ `->` $s$ if every pattern $p_i$ *matches type* $t$ and every expression $e_i$ has type $s$. When we check whether pattern $p_i$ matches type $t$, we have to assign appropriate types to the variables in $p_i$. We may assume the types of these variables when determining the type of $e_i$. 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 match 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 of a function to an argument, we proceed as before: we first evaluate the function and then the argument parts of the application. If both these evaluations produce values, then the resulting expression

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

is evaluated by trying to *match value* $v$ against each pattern in turn, starting with $p_1$. If value $v$ matches some pattern $p_i$ (and no prior pattern), it will provide bindings of values for the variables in the pattern. The resulting expression $e_i$ is then evaluated with those bindings. If value $v$ fails to match any of the patterns $p_1, \ldots, p_n$, then the evaluation results in a runtime error called a "`nonexhaustive match failure`".

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, resulting in bindings of `3` to `n` and `1` to `k`,
3. substituting `3` for `n` and `1` for `k` in `fact' (n-1, n*k)` yields `fact' (3-1, 3*1)`.

# 8 Case

A case expression has the form:

```
(case e of
    p1 => e1
  | p2 => e2
  ...
  | pn => en)
```

**A case expression is an expression, not an imperative statement.** A well-formed case expression has a type and may be evaluated, resulting in either a value, an exception, or infinite looping.

In order for a case expression to be well-typed, the expression $e$ must have some type `t'` and each pattern $p_i$ must match that type `t'`. In addition, all of the expressions $e_i$ must have one and the same type, let's call it `t`. If all of these conditions are satisfied, then the type of the case expression is `t` as well. In other words, the case expression has the same type as do the $e_i$.

For evaluation of a case expression, one first evaluates the expression $e$. If evaluation of $e$ produces a value $v$, then one tries to match value $v$ against patterns $p_1, \ldots, p_n$, in order, just like with patterns in a function application. If value $v$ matches pattern $p_i$ (and no prior pattern), it provides bindings for the variables in the pattern. These bindings are then used to evaluate the expression $e_i$. (Of course, evaluation of $e$ or of the selected $e_i$ might raise an exception or loop forever, in which case the same will be true for evaluation of the overall case expression.) If value $v$ fails to match any of the patterns $p_1, \ldots, p_n$, then the evaluation results in a runtime error.

Observe that expression $e$ can be any SML expression, not merely a Boolean expression as found in `if-then-else` expressions. Consequently, case expressions offer a powerful method for encoding decisions based on general inputs.

For instance, the following case expression assumes that there are three variables `x`, `y`, `z` in the environment (perhaps all integers). It views them as 3D coordinates and classifies the resulting point in a particular way. The type of this case expression is `string`.

```
case (x > y, z) of
    (_, 0) => "in the xy plane"
  | (true, _) => "not in plane, below bisector"
  |     _     => "not in plane, above bisector"
```

- With bindings `[1/x, 2/y, 0/z]`, the case will evaluate to `"in the xy plane"`.
- With bindings `[1/x, 2/y, 7/z]`, the case will evaluate to `"not in plane, above bisector"`.
- With bindings `[5/x, 2/y, 7/z]`, the case will evaluate to `"not in plane, below bisector"`.