15-150: Principles of Functional Programming

Values, Expressions, Types, and Declarations*

1 Computation as Evaluation

These notes introduce the basic vocabulary used throughout the course for talking about Standard ML (SML) programs: values, expressions, types, and declarations, restricted for now to the primitive types (`int`, `real`, `bool`) and to tuples built from them.

Many familiar programming languages, for example, C or Java, are based on an *imperative* model of computation: a program is a sequence of commands that modify the memory of the machine, and each step of execution inspects the current contents of memory, performs some simple computation, updates memory, and moves on to the next instruction.

For computation in SML, the central notion is *evaluation of expressions*, not execution of commands. This is much closer to the experience of evaluating an expression in ordinary algebra: given a polynomial in a variable x, we substitute a particular value for x and use the rules of arithmetic to compute the result. SML generalizes this idea to a much richer collection of values and operations, but the basic picture is the same: a program is something we *evaluate*, and its meaning is the value (if any) that evaluation produces.

This evaluation-based view has real advantages. Because it is so close to ordinary mathematics, it supports precise reasoning about what programs do and why they are correct – reasoning that goes well beyond what testing alone can offer.

2 Values, Expressions, and Types

Every SML expression has three aspects, and these three aspects are all we need to reason about it:

- it may or may not have a *type*,
- it may or may not have a *value*, and
- it may or may not produce an *effect*.

For now we set effects aside entirely and consider only *pure* expressions, ones whose evaluation neither raises an exception nor causes any observable side effect. (Exceptions are the one exception we allow ourselves to mention informally in passing; a full treatment comes later.)

*Compiled and adapted by Dilsun Kaynar from materials developed by several former instructors of 15-150, most notably Stephen Brookes, Michael Erdmann, Robert Harper, and Frank Pfenning, whose lecture notes and textbook chapters form the basis of much of what follows.

2.1 Types

A *type* is a description of the value an expression will produce, if it produces one at all. When we know an expression has type `int`, we know that, should evaluation terminate normally, the result will be an integer, and nothing else. SML checks, before ever attempting to evaluate an expression, whether the expression is well-formed according to a set of syntax-directed typing rules. An expression that passes this check is *well-typed*; one that fails is *ill-typed*, and SML rejects it with a type error. SML never attempts to evaluate (run) an ill-typed expression.

We write a *typing assertion* as

$$e : t$$

meaning that expression e has type t. Typing assertions are justified by typing rules that express the type of a compound expression in terms of the types of its parts. For example, the rule for addition of integers says that $e_1 + e_2$: `int` provided e_1 : `int` and e_2 : `int`. Chaining such rules together gives a *typing derivation*; for instance, the assertion $(3 + 4) + 9$: `int` is justified because 3 : `int`, 4 : `int`, and 9 : `int` are all axioms, and addition of two `int`s is again an `int`.

Typing is entirely static: it is determined by the syntactic structure of an expression, without evaluating anything. This is what makes it possible for SML to reject an ill-typed expression *before* evaluation is ever attempted.

2.2 Values and Evaluation

A *value* is a special kind of expression: one that is already fully reduced, and so evaluates to itself. Every value that has a type does so as an axiom, rather than via some derivation.

Once SML has determined that an expression is well-typed, it attempts to *evaluate* it. We write

$$e \hookrightarrow v$$

to mean that expression e evaluates to value v. Evaluation proceeds by a sequence of *reductions*, each of which rewrites the expression to another expression. We write

$$e \overset{1}{\Longrightarrow} e' \qquad \text{and} \qquad e \overset{k}{\Longrightarrow} e'$$

for "e reduces to e' in one step" and "in exactly k steps," respectively, and we write

$$e \Longrightarrow e'$$

for "e reduces to e' in zero or more steps." When we care only about correctness and not about the number of steps taken, we will simply write $e \Longrightarrow e'$ and not worry further about how many steps were involved. These two relations are closely tied together: $e \hookrightarrow v$ exactly when either e already is v, or $e \overset{1}{\Longrightarrow} e_1 \overset{1}{\Longrightarrow} e_2 \overset{1}{\Longrightarrow} \cdots \overset{1}{\Longrightarrow} v$ for some sequence of one-step reductions.

For example, the expression `(3 + 4)+ 9` has type `int`. Its evaluation is

$$(3 + 4) + 9 \overset{1}{\Longrightarrow} 7 + 9 \overset{1}{\Longrightarrow} 16,$$

so $(3 + 4) + 9 \hookrightarrow 16$.

Not every well-typed expression has a value. The expression `3 div 0` is perfectly well-typed (it has type `int`), but its evaluation raises an exception rather than producing an integer; there is no value v with `3 div 0` $\hookrightarrow v$. Similarly, some well-typed expressions simply fail to terminate. Having a type is a promise about *what kind of value* an expression will produce *if* it produces one – it is not a promise that evaluation will yield a value.

2.3 Extensional Equivalence and Referential Transparency

Two expressions e_1 and e_2 of the same type are *extensionally equivalent*, written $e_1 \cong e_2$, when one of the following holds:

- evaluation of e_1 and of e_2 produce the same value, or
- evaluation of e_1 and of e_2 both raise the same exception, or
- evaluation of e_1 and of e_2 both fail to terminate.

Informally: e_1 and e_2 are extensionally equivalent when they *behave the same way*, as far as anyone using them could tell. Extensional equivalence is an equivalence relation, defined only between well-typed expressions of the same type. It is the appropriate notion of equivalence for the primitive types and for tuples built from them; a more general version of the definition (needed once we have functions) refines conditions like these to allow the resulting values themselves to be merely equivalent, rather than identical.

Extensional equivalence matters because SML enjoys a fundamental property known as *referential transparency*: any expression in a program may be replaced by an extensionally equivalent expression without changing the observable behavior of the program. This is exactly the familiar mathematical practice of substituting "equals for equals,". For example, we can state that $(3 + 4) * 6 \cong 7 * 6$ because $3 + 4 \cong 7$.

Referential transparency is what makes equational, algebra-style reasoning about SML programs legitimate: because a pure expression produces no side effects, evaluating it twice gives the same result both times, and the order in which independent subexpressions are evaluated cannot affect the final value of the program.

3 Primitive Types

We now look in more detail at the primitive types integers, reals, and booleans. For each of these types we describe its values, and describe the operations available on those values.

3.1 Integers

Type: `int`.

Values: the integers $0, 1, \tilde{}1, 2, \tilde{}2, \ldots$. (Negative integer literals are written with a leading tilde, `~`, rather than a minus sign, to distinguish a negative numeral from the subtraction operator.) We treat SML as capable of representing any integer, ignoring machine limits such as a fixed number of bits.

Operations: `+`, `-`, `*`, `div`, `mod`, among others.

Typing rules: $e_1 + e_2$: `int` provided e_1 : `int` and e_2 : `int`, and similarly for the other arithmetic operations.

Evaluation: arithmetic expressions are evaluated left to right, reducing each argument to a value

before combining them:

$$\begin{aligned} e_1 + e_2 &\overset{1}{\Longrightarrow} e_1' + e_2 && \text{if } e_1 \overset{1}{\Longrightarrow} e_1', \\ n_1 + e_2 &\overset{1}{\Longrightarrow} n_1 + e_2' && \text{if } e_2 \overset{1}{\Longrightarrow} e_2' \text{ (with } n_1 \text{ already a value)}, \\ n_1 + n_2 &\overset{1}{\Longrightarrow} n && \text{where } n \text{ is the integer sum of } n_1 \text{ and } n_2. \end{aligned}$$

and similarly for the other integer operations. As noted above, not every well-typed arithmetic expression has a value: `3 div 0` type-checks as `int` but has no value, since evaluating it raises an exception.

The relationship between `div` and `mod` is exactly the familiar division algorithm from arithmetic: for all integers m and all nonzero integers n,

$$n * (m \texttt{ div } n) + (m \texttt{ mod } n) = m.$$

For instance, `(42 div 5)` and `(42 mod 5)` evaluate, respectively, to `8` and `2`, and indeed $5*8+2 = 42$.

3.2 Real Numbers

Type: `real`.

Values: real number literals such as `3.14`, `333.999`, and values written in scientific notation, such as `2E10` or `2.0E~3`.

Operations: `+`, `-`, `*`, `/` (real division; there is no `mod` for reals).

Typing and evaluation follow the same pattern as for integers. One important difference in practice: because of limited floating-point precision, it is almost never appropriate to test two `real` values for equality with `=`. (SML does provide `Real.==` for this purpose, but even mathematically equal real-valued expressions, such as `a*b` and `b*a`, can fail to produce identical floating-point representations.)

No implicit conversion. SML never silently converts between `int` and `real`. The expression `3 + 3.14` is rejected outright as ill-typed, because `+` expects both arguments to have the same numeric type. To mix integers and reals, use the built-in conversion functions: `real`, of type `int -> real`, and `floor` (or `round`), of type `real -> int`. So `real(3)+ 3.14` performs the addition in `real`, while `3 + round(3.14)` performs it in `int`.

An arithmetic operator such as `+` is said to be *overloaded*: the same symbol denotes integer addition at type `int * int -> int` and floating-point addition at type `real * real -> real`, and the type checker resolves which one is meant from the types of the arguments.

3.3 Booleans and Conditional Expressions

Type: `bool`.

Values: `true` and `false`.

Operations: `not`, `andalso`, `orelse`, among others

Typing rules:

$$e_1 \texttt{ andalso } e_2 : \texttt{bool} \quad \text{if } e_1 : \texttt{bool} \text{ and } e_2 : \texttt{bool},$$

and similarly for `orelse` and the comparison operators. For the conditional,

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t \quad \text{if } e_1 : \texttt{bool},\ e_2 : t, \text{ and } e_3 : t.$$

This rule holds for *any* type t, but it insists that both branches have the *same* type t – this is what makes it possible to assign the conditional a single, definite type without first knowing which branch will be taken.

Evaluation: the condition is evaluated first; then, depending on its value, exactly one branch is evaluated and the other is simply discarded:

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

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

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

Because only one branch is ever evaluated, the expression

```
if 1 < 2 then 0 else (1 div 0)
```

evaluates to `0`, even though the `else` branch, taken on its own, would raise an exception: that branch is simply never evaluated.

A style note. It is easy to write conditionals that say more than they need to. Beginners often write

```
if e = true then e1 else e2
```

which is just a way of writing `if e then e1 else e2`. Likewise, `if e=false then e1 else e2` is better written as `if not e then e1 else e2`, or better still, as `if e then e2 else e1`. More generally: if an expression already has type `bool`, there is no need to compare it against `true` or `false` at all – just use it directly as the condition.

4 Tuples

SML lets us combine values into *tuples*. We describe the case of pairs; larger tuples work the same way.

Types: $t_1 * t_2$, for any types t_1 and t_2. More generally, $t_1 * t_2 * \cdots * t_n$ for tuples of n components. The type constructor $*$ does not associate: $(t_1 * t_2) * t_3$, $t_1 * (t_2 * t_3)$, and $t_1 * t_2 * t_3$ are three genuinely different types – the first is a pair whose first component is itself a pair, the second a pair whose second component is a pair, and the third a triple.

Values: (v_1, v_2), for values v_1 and v_2; more generally $(v_1, \ldots, v_n)$.

Typing rule:

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

Evaluation: the components of a tuple are evaluated in order, left to right:

$$(e_1, e_2) \overset{1}{\Longrightarrow} (e_1', e_2) \quad \text{if } e_1 \overset{1}{\Longrightarrow} e_1',$$

$$(v_1, e_2) \stackrel{1}{\Longrightarrow} (v_1, e_2') \quad \text{if } e_2 \stackrel{1}{\Longrightarrow} e_2'.$$

For example, `(42 div 5, 42 mod 5)` has type `int * int` and evaluates to `(8, 2)`. As another example, the type `int * int * real` is the type of triples whose first two components are integers and whose third is a real; a value of this type is, e.g., `(3, 4, 3.14)`.

We will see later (once we have introduced patterns) how to conveniently take a tuple apart into its components; for now it is enough to know how tuples are built and typed.

5 Declarations and Scope

A *declaration* introduces one or more *bindings*, which means associations between a name and either a type or a value that can be used in the expressions that follow it.

5.1 Value and Type Bindings

A *value binding* associates a variable with a value (and, in the first few weeks of the course, we generally require that its type also be stated explicitly):

```
val m : int = 3 + 2;
val pi : real = 3.14 and e : real = 2.17;
```

The first binding introduces the variable `m`, records that it has type `int`, and evaluates `3+2` to obtain the value `5`, which is what `m` is bound to. The second binding introduces two variables, `pi` and `e`, simultaneously.

In general, a value binding

```
val var : t = exp
```

is accepted only if `exp` has type `t`; if so, `exp` is evaluated to obtain a value, and that value – not the expression itself – is what becomes bound to `var`.

A *type binding* instead associates a name with a type, and has no effect on values at all:

```
type float = real;
```

After this declaration, `float` may be used anywhere `real` could be used; the two names simply denote the same type.

A crucial fact about SML: **variables do not vary**. Once a value binding associates a value with a name, that particular binding never changes. This makes variables in SML much closer to variables in mathematics than to variables in languages like C, where assignment can overwrite a variable's contents.

5.2 A First Look at Function Declarations

Everything said so far about declarations applies to `val` bindings. SML also lets us declare *functions*

A function declaration has the form

```
fun f (x : t1) : t2 = exp
```

and does two things at once: it introduces a name `f`, and it gives a rule for producing a result of type `t2` from an argument `x` of type `t1`, by evaluating `exp` with `x` standing for whatever value `f` is eventually applied to. Like a `val` binding, a function declaration introduces a name whose scope extends over the rest of the program.

For example,

```
val pi : real = 3.14;

fun circ (r : real) : real = 2.0 * pi * r;
(* circ : real -> real *)
(* Example: circ 1.0 = 6.28 *)

fun area (r : real) : real = pi * r * r;
(* area : real -> real *)
(* Example: area 1.0 = 3.14 *)
```

Both `circ` and `area` are written in terms of a parameter `r`, standing for whatever real number the function is eventually applied to, and both refer to the previously declared `pi`. [1]

5.3 Compound Declarations

Declarations can be sequenced, one after another:

```
val m : int = 3 + 2;
val n : int = m * m;
```

Here `m` is bound to `5`, and then `n`, whose definition may refer to `m`, is bound to `25`. Scopes nest: the scope of the binding for `m` includes the entire rest of the declaration, including the binding for `n`.

A careful reader might now ask: since a binding, once made, cannot change, what happens if we declare *another* value for a name that is already bound? Nothing new – a second declaration is just an ordinary binding whose scope begins where it appears in the text, exactly like any other. Occurrences written before it keep referring to the first binding; occurrences written after it refer to the new one:

```
val m : int = 3 + 2;
val n : int = m * m;
val m : int = 0;
```

The declaration of `n` is unaffected by the third line: `n` is still bound to `25`, because the occurrence of `m` inside `n`'s definition was already fixed to the *first* binding of `m` before the third line ever appeared. Only a new occurrence of `m`, written after the third line, would see `m = 0`. We say the third declaration *shadows* the first – a name for this situation, not a separate mechanism. ("Out of scope" does not mean "gone," though: as we will see below, a function that refers to a name keeps using the binding that was in scope when *the function* was declared, even if that name is later shadowed.)

[1] We will see *how* a call such as `circ 1.0` is evaluated when we look at functions more closely as a separate topic.

5.4 Limiting Scope: `let` and `local`

By default, a top-level binding's scope extends to the rest of the program. Sometimes we want a binding's scope to be limited to a single expression. The `let` expression does exactly this:

```
let dec in exp end
```

Here `dec` is a declaration whose scope is limited to `exp`; once `exp` has been evaluated, the bindings introduced by `dec` are discarded. The value of the whole `let` expression is the value obtained by evaluating `exp` relative to the bindings introduced by `dec`. For example,

```
let
  val m : int = 3
  val n : int = m * m
in
  m * n
end
```

has type `int` and evaluates to `27`: we first work out the bindings for `m` and `n`, then evaluate `m * n` relative to them. These bindings for `m` and `n` are entirely local; nothing outside the `let` expression can see them, and if `m` or `n` was already bound to something outside, that outer binding is only temporarily eclipsed, and is restored once the `let` expression has finished evaluating.

The `local` declaration serves an analogous purpose when we want one declaration's scope limited to a second declaration, rather than to a single expression:

```
local dec in dec' end
```

Here the bindings from `dec` are visible while processing `dec'`, but are discarded once `dec'` has been processed; only the bindings introduced by `dec'` itself survive into the surrounding scope.

Recall `circ`, which recomputes `2.0 * pi` on every call. If we want to factor out that repeated subexpression without polluting the surrounding scope with an extra visible name, a `local` declaration is the right tool:

```
local
  val pi2 : real = 2.0 * pi
in
  fun circ' (r : real) : real = pi2 * r
end;
(* circ' : real -> real *)
(* Example: circ' 1.0 = 6.28 *)

(* pi2 is not in scope here -- it was local to the declaration of circ' *)
```

`circ` and `circ'` are extensionally equivalent: applied to equal arguments, they produce equal results. Unlike a top-level `val pi2 = 2.0 * pi`, which would remain visible after `circ'` was declared, `local` keeps `pi2` out of the surrounding scope entirely.

5.5 Static Scoping, Illustrated

SML uses *static* (or *lexical*) scoping: an occurrence of a name refers to whichever binding was in scope at the point where the occurrence appears in the program text, regardless of the order in which things happen to be evaluated at run time. The declarations of `circ` and `area` from earlier in this section make this concrete:

```
val pi : real = 3.14;
fun circ (r : real) : real = 2.0 * pi * r;
fun area (r : real) : real = pi * r * r;
```

Both occur in the scope of `pi = 3.14`, so `pi` inside each function body refers to that binding – and continues to, no matter when `circ` or `area` is later called.

Now suppose `pi` is declared again, with a more accurate approximation – the same kind of redeclaration discussed above, just with `pi` in place of `m`:

```
val pi : real = 3.14159;
```

This does *not* change the behavior of `circ` or `area` as already declared: `area 1.0` is still `3.14`, since those declarations are already fixed to the binding of `pi` that was in scope when *they* were declared. Only a *new* declaration of `area`, written after the new binding of `pi`, picks up the more accurate value:

```
fun area (r : real) : real = pi * r * r;
(* this new area shadows the old one, and now uses pi = 3.14159 *)
(* Example: area 1.0 = 3.14159 *)
```

This is the essence of static scoping: a free variable in a function declaration is resolved once, at the point of declaration, not at the point where the function is eventually called – which is exactly the promise made above that shadowing does not affect a function already declared.

6 Summary

At this point we have the basic vocabulary needed to talk about SML programs restricted to primitive types and tuples:

- Every expression may have a type, a value, and (which we have set aside for now) an effect; only well-typed expressions are evaluated.
- Evaluation is a sequence of reductions, $e \overset{1}{\Longrightarrow} e_1 \overset{1}{\Longrightarrow} \cdots \overset{1}{\Longrightarrow} v$, culminating (if it terminates without error) in a value: $e \hookrightarrow v$.
- Extensional equivalence, $\cong$, captures the sense in which two expressions "behave the same," and referential transparency licenses replacing an expression by an extensionally equivalent one anywhere in a program.
- The primitive types `int`, `real`, and `bool`, and the tuple types built from them, each come with their own values, operations, typing rules, and evaluation rules.
- Declarations introduce bindings whose scope can be the rest of a program, a single expression (`let`), or another declaration (`local`); SML's scoping is static, determined by program text rather than by the order of evaluation.