Lecture 1 - Laying the Groundwork

David M Kahn *

Summer 2022

1 What is Functional Programming?

It is natural to ask the question: What is functional progamming? It turns out that this is not such an easy question to answer. A programming paradigm is not so much a strict definition, but vague collection of features that are emphasized. Any given "functional" language is likely to satisfy most of these features, but perhaps not all, and not everyone always agrees on which features are important. However, we can hazard a suggestion that the following are the features of a functional programming language:

1. Functions as Data: functions can be passed around and manipulated just as easily as other kinds of data, like integers
2. Lack of Side-Effects: the language avoids leaving any trace of its computation aside from the result; it will avoid, e.g., mutating some memory location[1]

As a result of these properties, a functional language typically favors features like recursion over while-loops, has a special focus on the manipulation of data structures, and often can get particularly good mileage out of a type system to track its transformations. The lack of side effects also usually allows for parallelism to be naturally incorporated, as well as an easier time proving program correctness.

A language like Prolog certainly avoids most side effects like mutation, but favors more general relations over functions - it is usually considered to be logic programming, not functional. A language like Scheme is, however, considered functional, and it doesn't have a static type system. And a language like Python does treat functions as data, but also actively uses a ton of side effects - is it functional? Regardless, SML, the language you will be learning in this class, is about as functional as it gets.

*Adapted from Michael Erdmann's Spring 2022 notes, in turn adapted from a draft of Frank Pfenning

[1]This is not to say that there are no side effects at all – merely that they are kept to a minimum.

2 A Historical Note

The functional programming paradigm *is not new*. In fact, what you might call the very first programming language ever, Church's lambda calculus, was functional, and it is as old as computing: The lambda calculus was debugged throughout the 1930s[1] and finally was formalized as we know it today in 1936[2], which is the same year the Turing machine was formalized[3].

There is a good reason that functional programming is one of the first programming paradigms. The core features and patterns of functional programming were (and are still) designed to match closely with the mathematical and logical formalisms that we use to structure our human reasoning. Indeed, the lambda calculus was designed to *be* a logical formalism. So if you can formally reason through how to solve a problem, function programming is often the perfect tool to enact that reasoning. This is something we will see more later through the correspondence between proofs and programs.

If functional programming is so good, you might wonder why functional programming is not the default programming paradigm today. We can only speculate, but the answer seems to be the inertia of old technological limitations: Functional programming matches human reasoning, but not machine operations. Back in the day, when computers were slow and compilers were weak, one could only hope to get reasonable performance using a progamming language that better matched what our machines could compute quickly. Working with functions at a high level was simply too memory-hungry and slow to consider. And thus non-functional programming languages were passed down through the generations of programmers.

However, the tides are turning. Compiler technology has gotten stronger, to the point that functional languages like OCaml can compete with C in code shootouts[4]. Further, computers have gotten faster in general, to the point that language speed is less relevant, and even "slow" languages like Python are quite usable. And now that these technological barriers are overcome, we are seeing more and more functional features being added into popular programming languages. Anonymous functions were added to C# in 2005[5], PHP in 2009[6], C++ in 2011[7], Java in 2014[8], and Excel in 2021[9]. And others that already had such functions have quietly upgraded their use, like JavaScript in 2015[10], and there is discussion about having Python follow suit[11].

So it turns out that people actively want to be able to program functionally (even if they do so in a language that is not primarily functional). After this course, perhaps you will feel the same :)

3 Notations, Definitions, and Conventions - Oh my!

To talk about the language SML (or indeed almost any programming language), there are 3 key categories to be aware of:

- Expressions/Declarations: the actual code snippets that tell the computer what to do; they are the *syntax* of the language, i.e., the actual symbols used
- Types: an abstract categorization of expressions; they help us reason about the language
- Reductions: the the actual acts of computing; they describe the *semantics* of the language, i.e., what the syntax means[2]

To reason about programs, you will often need to use and relate all 3. For this reason, we now provide some notation, definitions, and conventions to allow us to talk about them more easily.

Formality and Completeness A complete formal definition of all the bits of SML can be found in [12][3]. Here, however, we will work slightly completely and less formally. In particular, we will assume that all SML code is functional, and the only possible side-effects are exceptions and non-termination. We will also ignore pesky practical limitations and assume that the SML types like *int* and *real* correspond exactly with the mathematical entities they intend to represent (the integers and reals). Thus, for instance, we will not think about problems like integer overflow.

Font Firstly, as a convention, we write actual concrete code in a `typewriter font`. This includes concrete expressions and types. Then for more general mathematical terms, we will use *italics*. This includes mathematical variables we may write in our reasoning that stand in for expressions and types, but does not include variables present in the code itself. In particular, we usually write e as a mathematical variable for expressions, v for values (a special kind of expression), and t for types.

Value We use the term *value* to refer a special kind of expression. Intuitively, they are the expressions that are "done" computing, and are the actual objects we like to think of our program manipulating. For instance, `2` is a value, while `1+1` is a mere expression since its addition is still left to compute.

Reduction To indicate how expressions compute, we use the following notation:

$e \stackrel{1}{\Longrightarrow} e'$	e reduces to e' in 1 step
$e \stackrel{k}{\Longrightarrow} e'$	e reduces to e' in k steps
$e \Longrightarrow e'$	e reduces to e' in 0 or more steps

[2]In particular, reduction gives us an *operational* semantics. However, other forms of semantics are possible.

[3]For a simpler language reference, see `https://smlhelp.github.io/book/`

Note that these "steps" are an abstract notion - a computer might perform many operations to simulate one of our "steps". However, these "steps" do provide an exact number from which to reason about a program's complexity.

Evaluation and Valuability Evaluation is the attempt to reduce an expression e all the way to a value. If it does reduce a to a value v, we can use the special reduction notation $e \hookrightarrow v$, and may call that expression e *valuable*. But beware: Not all expressions are valuable - some might loop forever, or throw an exception.

Typing To indicate that an expression e has can be categorized with type t, we write $e : t$. This can be done in code as well, when writes, e.g., `2:int`. In SML, the meaning of $e : t$ is roughly that e looks like it would evaluate to a value of type t[4]. But beware: Being well-typed does not mean that e is valuable in the first place[5]!

Totality We say that a function f`:t -> t'` is *total* if, for every input value v`:t`, applying f to v will evaluate all the way to a value. But beware: Not every function is total - some might loop forever, or throw an exception. If you ever apply a function in a proof and expect a value to come out of it, you should probably make sure that function is total!

Extensional Equality We write $e \cong e'$ to mean that the expressions e and e' are extensionally equivalent. Intuitively, two expressions e and e' *of the same type* are extensionally equivalent whenever they both behave in the same way during evaluation. That means:

- they both reduce to the same value
- they both throw the same exception, and any data carried on those exceptions is extensionally equivalent
- they both do not terminate
- in the case that e and e' are functions of type `t -> t'`, given any extensionally equivalent input values v and v' of type `t`, the applications $e\ v$ and $e'\ v'$ are extensionally equivalent [6]

Substitution We write $[e/x]e'$ to refer to the expression e' where all free instances of the variable x are replaced with the expression e'. (A variable is *free* whenever there is no place it would get bound.) For example, $[2/x]$`1+x` refers to `1+2`. Substitutions for distinct variables can be combined into one simultaneous substitution like $[e/x, e'/y]$.

[4] We can separately define when a value is of a given type.
[5] At least this is the case in SML and most common languages.
[6] Or alternatively, the simpler definition that $e\ v \cong e'\ v$ for any value v will also suffice.

Referential Transparency Referential transparency is a property of a language that means substitution respects exensional equivalence. That is, if you replace any subexpression in a program with an extensionally equivalent expression, the resulting program will be extensionally equivalent to the original. One way of stating this is that $[e_1/x]e \cong [e_2/x]e$ whenever $e_1 \cong e_2$. This holds in the part of SML we are working with because it lacks side effects like mutation[7].

Environment The collection of variable bindings present at some point in a program, is called the *environment*. By convention, we write the environement as a (potentially large) substitution of each variable for the value bound to it. Because of referential transparency, variables can be replaced with their definitions without changing the code's meaning, so it is ok to reuse substitution for this notation.

Scope The variable bindings in an environment are only live for a certain *scope*. SML uses static, lexical scoping, so variables are only live for the clearly-delineated sections of the source code where they get defined. (This is opposed to dynamic scoping, where vairble liveness is only determined at runtime.) For example, in `let val x = 5 in` e `end`, `x` is in scope only in e.

Shadowing In SML, if you redefine a variable while it is live, the old definition gets *shadowed*. This means the newer definition takes priority *while the new definition is in scope*; when the newer definition drops out of scope, the older definition is used again. For example, `let val x = 5 val x = 4 in` e `end`, the value of `x` used in e is 4, since that definition was the latest live one.

Top Level This refers to the outermost scope in an SML file. Special notation is allowed at this level so that variables can be declared without using let-bindings.

4 Base Type Features

Here we list language features of a selection of the main base types. This is not exhaustive, but is certainly a good start.

4.1 Integers

Type `int`

Values the usual integers of $\mathbb{Z}$, like $-2, -1, 0, 1, 2$, etc.

[7] Referential transparency holds in most normal functional settings. It is also possible to define a notion of referential transparency in other settings like imperative programming, but such a notion will need to take into account the effects of side effects.

Expressions one can work with integers by writing them out (`0`,`1`,`2`, etc.), and one can also use the usual operations like `+`, `-`, `*`, `mod`, and `div`; also note that unary negation (which is needed to write negative numbers) is given by `~`(the tilde symbol), rather than `-`

Typing

- n`:int` if n is an `int` value (`0`,`1`,`2`, etc.)
- for unary operators like `~`, we have `~`e `:int` if e`:int`
- for binary operators like `+`, we have e_1`+`e_2`:int` if both e_1`:int` and e_2`:int`

Reduction

- for unary operators like `~`, we have `~`$e \stackrel{1}{\Longrightarrow}$ `~`e' if $e \stackrel{1}{\Longrightarrow} e'$
- for binary operators like `+`, evaluation proceeds left to right

$$e_1 + e_2 \stackrel{1}{\Longrightarrow} e_1' + e_2 \qquad \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1'$$

$$v + e \stackrel{1}{\Longrightarrow} v + e' \qquad \text{if } v \text{ is a value and } e \stackrel{1}{\Longrightarrow} e'$$

$$v_1 + v_2 \stackrel{1}{\Longrightarrow} v_3 \qquad \text{if } v_3 \text{ is the value equal to the sum of values } v_1 \text{ and } v_2$$

4.2 Reals

Type `real`

Values the usual decimal numbers we can write down, like 0.5, 3.14159, 100.00, etc. – note that these do *not* have infinite precision, and do *not* contain any values of type `int` (2 and 2.0 are distinct)

Expressions one can work with reals by writing out numbers with a decimal point, or using the usual arithmetic operations like with `int` [8] (with the exception that \ is used for `real` division)

Typing the same as `int` with `real` in place of `int`

Reduction the same as `int`

4.3 Booleans

Type `bool`

Values `true` and `false`

[8] These operations are *overloaded*, which is unusual for SML.

Expressions one can work with Booleans by writing them out, by using if-then-else statements like `if` e_1 `then` e_2 `else` e_3, using logical operations like `not`, `andalso`, or `orelse`, or using comparisons like `<`, `>=`, or `=`[9] - note that the inequality operator is `<>`.

Typing

- `true:bool` and `false:bool`
- `if` e_1 `then` e_2 `else` e_3`:`t if e_1`:bool` and both e_2`:`t and e_3`:`t – note both branches *must* be the same type
- `not` e`:bool` if e`:bool`
- for binary logical operations like `andalso`, we have e_1 `andalso` e_2`:bool` if both e_1`:bool` and e_2`:bool`
- for comparisons like `<`, we have e_1`<` e_2`:bool` if both e_1 and e_2 are type `int`, or both of them are type `real` [10]

Reduction

- for branching,
  - `if true then` e_1 `else` $e_2 \stackrel{1}{\Longrightarrow} e_1$
  - `if false then` e_1 `else` $e_2 \stackrel{1}{\Longrightarrow} e_2$
- for negation,
  - `not` $e \stackrel{1}{\Longrightarrow}$ `not` e' if $e \stackrel{1}{\Longrightarrow} e'$
  - `not true` $\stackrel{1}{\Longrightarrow}$ `false`
  - `not false` $\stackrel{1}{\Longrightarrow}$ `true`
- for binary logical operations like `andalso`, the evaluation is *short-circuiting*
  - `true andalso` $e \stackrel{1}{\Longrightarrow} e$
  - `false andalso` $e \stackrel{1}{\Longrightarrow}$ `false` (note e is not evaluated in this case)
- comparisons evaluate left-to-right like the arithmetic binary operators

[9] The operator `=` is *very* special. As you find in many programming languages, it is overloaded across many types, but you should only really use it on certain ones. In particular, SML will actually prevent you from using `=` on functions or reals, since there is no way to actually tell if two of them are logically the same. However, it works fine on simple types integers, booleans, and strings. You probably want to define auxilliary functions for equality on more complicated data types. These same notes also apply to the inequality operator `<>`.

[10] These comparisons are therefore also overloaded.

4.4 Strings

Type `string`

Values the usual text (and escape characters) within quotations, like `"foo"`, `"bar"`, and `"\n"`

Expressions one can work with strings by writing them out, or by concatenating with the caret symbol `^`

Typing

- `"`s`":string` for ASCII strings s
- e`^`e'`:string` if e`:string` and e'`:string`

Reduction e_1`^`$e_2 \stackrel{1}{\Longrightarrow} e_3$ where e_3 is e_2 concatenated onto the end of e_1

4.5 Unit

This type just exists to denote a token data object that can be passed around.

Type `unit`

Values only `()`

Expressions only `()`

Typing `() : unit`

Reduction nothing to reduce

5 Compound Type Features

Here we list the language features of a selection of some initial compound types (types made of other types). This list is not close to exhaustive.

These are where the power of the SML type system starts to kick off. Knowing the type of some data tells you exactly how to work with it. Usually there is one sort of way to *make* something of a given type, and one sort of way to *use* it. This means you can use types to guide your programming!

5.1 Tuples

Type t_1 `*` t_2 `*` ... `*` t_n

Values tuples like `(1, 7)`, `("one plus one", 2)`, and `(true, false, true)`, where every element is itself a value.

Expressions

- Make: `(` e_1`,` e_2`, ...,` e_n`)`
- Use: `case` e `of (a,b) =>` e' and similar for larger tuples. This binds each piece of the tuple e to a variable (in this case `a` and `b`), and gives those variables the scope of e'.[11]

Typing

- Make: `(` e_1`,` e_2`, ...,` e_n`)`$:t_1$ `*` t_2 `* ...` `*` t_n if $e_i : t_i$ for each i
- Use: `case` e `of (a,b) =>` e' `:`t if $e : t_1$ `*` t_2 and, given that $a : t_1$ and $b : t_2$, it is the case that $e' : t$ - this extends similarly for larger tuples

Reduction

- Make: evaluation occurs left to right
  - `(`e_1`,` e_2`)` $\overset{1}{\Longrightarrow}$ `(`e_1'`,` e_2`)` if $e_1 \overset{1}{\Longrightarrow} e_1'$
  - `(`v`,` e`)` $\overset{1}{\Longrightarrow}$ `(`v`,` e'`)` if $e \overset{1}{\Longrightarrow} e'$ and v is a value
- Use: First the tuple is evaluated, then the clause
  - `case` e_1 `of (a,b) =>` $e_2 \overset{1}{\Longrightarrow}$ `case` e_1' `of (a,b) =>` e_2 if $e_1 \overset{1}{\Longrightarrow} e_1'$
  - `case (`v_1, v_2`) of (a,b) =>` $e \overset{1}{\Longrightarrow} [v_1/a, v_2/b]e$

Special Notes This type is also called a "product", and 2-tuples may be called "pairs". Also, parenthesis have actual meaning in the interpretation of SML tuples: `(1,2,3)`, `((1,2),3)`, and `(1,(2,3))` are all distinct.

5.2 Functions

Type t_1 `->` t_2

Values A function value is called a *closure*, and is made up of 2 things: an environment of the variables the function uses from the time the function was defined, and an expression representing the actual function operation. For example, when `x` is defined as 2, and we write `fn y => x`, the closure is $\langle [2/x]$, `fn y => x` $\rangle$. However, we may also leave the environement implicit and simply write `fn y => x`.

[11] There are also projection operators like `#1`, `#2`, `#3`, but casing is stylistically favored in SML

Expressions

- Make: the *anonymous* `fn x =>` e
- Use: the application $e_1\ e_2$

Typing

- Make: `fn x =>` e`:`t `->` t' if e`:`t' assuming `x:`t
- Use: $e_1\ e_2$`:`t if e_1`:`t' `->` t and e_2`:`t'

Reduction

- Make: functions don't reduce
- Use: evaluation occurs left to right
  - $e_1\ e_2 \overset{1}{\Longrightarrow} e_1'\ e_2$ if $e_1 \overset{1}{\Longrightarrow} e_1'$
  - $v\ e \overset{1}{\Longrightarrow} v\ e'$ if $e \overset{1}{\Longrightarrow} e'$ and v is a value
  - `(fn x =>` e`)` $v \overset{1}{\Longrightarrow} [v/x]e$

Special Notes By convention this type is *right associative*, which means you should interpret t_1 `->` t_2 `->` t_3 as having parentheses on the right, giving the type t_1 `->` (t_2 `->` t_3). Also, this type is occasionally called an "exponential" or "arrow".

6 Other Features

6.1 Variable Declarations

Declarations are *not* expressions, and do not result in values. Instead, they bind a value to a variable in the environment. We write them as

`val x =` e

where `x` is the variable that will be bound, and the value it will be bound to is whatever e evaluates to (if anything).

6.2 Let Expressions

Expression `let` d `in` e `end` where d is any sequence of declarations

Typing `let val x =` e_1 `in` e_2 `end` :t if e_1:t', and e_2:t given that x:t'

Reduction First any declaration values are evaluated, then the scope expression.

- `let val x =` e_1 `in` e_2 `end` $\overset{1}{\Longrightarrow}$ `let val x =` e_1' `in` e_2 `end`
  if $e_1 \overset{1}{\Longrightarrow} e_1'$
- `let val x =` v `in` e `end` $\overset{1}{\Longrightarrow} [v/x]e$ if v is a value

Note that each declared variable is only live in the the expressions of following declarations, as well as the expression between `in` and `end`.

References

[1] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, pp. 346–366, 1932.

[2] ——, "An unsolvable problem of elementary number theory," *American journal of mathematics*, vol. 58, pp. 345–363, 1936.

[3] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[4] A. Calpini, "Computer language shootout scorecard," 2003. [Online]. Available: https://dada.perl.it/shootout/craps.html

[5] E. Dietrich, T. Brinkley, K. Cenerelli, sheeprock, kevinknowscs, A. A, T. B. G, D. Chalmers, A. De George, N. Turn, and et al., "The history of c# - c# guide," 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history

[6] "Php 5.3.0 release announcement," 2009. [Online]. Available: https://www.php.net/releases/5_3_0.php

[7] D. Kalev, "The biggest changes in c++11 (and why you should care)," Jun 2011. [Online]. Available: https://smartbear.com/blog/the-biggest-changes-in-c11-and-why-you-should-care/

[8] "What's new in jdk 8." [Online]. Available: https://www.oracle.com/java/technologies/javase/8-whats-new.html

[9] A. Gordon and S. P. Jones, "Enriching excel with higher-order functional programming," Jan 2021. [Online]. Available: https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/

[10] R. S. Engelschall, "Ecmascript 6: New features: Overview and comparison," 2015. [Online]. Available: http://es6-features.org/#ExpressionBodies

[11] J. Edge, "Alternative syntax for python's lambda," Mar 2021. [Online]. Available: https://lwn.net/Articles/847960/

[12] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of standard ML: revised*. MIT press, 1997.