15-150 Fall 2013
Lecture 2

Stephen Brookes

1 Objectives

We remind you that the main objectives of the course are:

- Learn to write functional programs
- Learn to write specifications, and to use rigorous techniques to prove program correctness
- Learn tools and techniques for analyzing the sequential and parallel running time of your programs
- Develop an appreciation for parallelism, and when to use it safely to improve the efficiency of your code
- Learn to structure your code into modules, with clear and well designed interfaces

Today's material is mostly concerned with the first two of these.

2 Today's lecture

An introduction to:

- Functional programming
- Expressions and types
- Declarations, patterns, bindings and scope
- Specifications

3 Functional Programming

Computation as evaluation

Functional programming is a programming paradigm based on computation as evaluation, as opposed to imperative programming, in which computation causes state change. Programs in a functional programming language are expressions, which denote values, or declarations, which bind names to values. Expressions can be evaluated, and (if evaluation terminates) produce a value. Since evaluation causes no side effects or state change, repeated evaluation of the same expression always produces the same result. This means that it is much more straightforward to reason about functional programs than it tends to be for imperative programs; indeed, this feature is often cited as a key motivation for the development of functional languages. There has been a lot of quasi-religious tub-thumping about the virtues of "pure" functional programming and the perceived sins of "impure" features such as assignment and state. Nevertheless most modern "functional" languages include some impure constructs, for pragmatic reasons to do with efficiency and ease of use. We will begin with pure functional programs and explore later what happens when we allow controlled use of impure features. We will point out advantages of the functional style of programming, but we will also try to give a fair assessment of the alternatives.

Simplicity

A major advantage of functional programming is simplicity (in conceptual terms). Programs behave like mathematical functions, which can be applied to suitable arguments and produce a result. There is a close relationship between functional programs and the mathematical notion of function, and techniques from mathematics and logic are excellent tools for specifying and reasoning about the behavior of functional programs. In particular, principles of mathematical induction, which are used extensively in foundational math and logic, will be crucial for this course, We use induction in one form or another to prove termination of programs, and to prove that programs satisfy their intended specifications.

Referential transparency

A functional language obeys a fundamental principle known as *Referential Transparency*[1]: in any functional program you can replace any expression with another expression that has an "equal" value, without affecting the value of the program. We will clarify what we mean by "equal" shortly, but for the moment just note that integer expressions are equal if they evaluate to the same integer value. So the expressions `21 + 21` and `42` are equal. And you probably would agree that `(21 + 21) * 2` and `42 * 2` are also equal, as predicted by this principle!

Referential transparency is a powerful principle that supports "equational reasoning" about functional programs. Roughly speaking, this is substitution of "equals for equals", a notion so familiar from mathematics that we do 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 us to develop better programs.

It is often said (e.g. in Wikipedia) that imperative languages do not satisfy referential transparency, and that only purely functional languages do. This is inaccurate: we will see later that impure languages also obey a form of referential transparency, but that we need to take account not only of value but also side-effects, in defining what "equal" means for imperative programs.

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

4 Progamming Language

We use **Standard ML**, a "functional" programming language with available implementations for many machine architectures. In lab you will see how to

[1]Sometimes called Frege's Principle, after the German philosopher Gottlob Frege, who is traditionally cited as the originator of the idea that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. Another term associated with this is the Compositionality Principle.

get started using the local implementation. There are also downloadable versions for PCs and Macs.

In addition to being *functional*, ML is a *typed* language: an expression can only be used if it has a "type", and typability is determined statically, from the syntactic structure of the program, without attempting to evaluate. An advantage of this is that a well typed expression never goes wrong when evaluated, in the sense that you ever encounter "stupid type errors" such as an attempt to add 1 to a truth value. Forcing the programmer to pay attention to types prevents an enormous number of common errors. Another advantage is that ML actually uses a sophisticated type inference algorithm, so programmers often need say very little (or nothing explicit) about types and ML infers if the code is typable and if so, what types are plausible.

ML is a *call-by-value* functional language: functions always evaluate their arguments. In contrast, some functional languages are call-by-name, or lazy (e.g. Haskell). We will show that even though ML is call-by-value one can easily program lazily in ML, so this language design choice is not a limitation.

The ML syntax for function definition allows notation very similar to the style of functional description used in mathematics. In particular, one can define a function by giving a series of clauses (or "cases"), each clause defining the function's behavior when applied to arguments whose value matches a simple pattern. For example, a function defined on integers may be defined by giving a clause for 0 and a clause for non-zero arguments. Patterns and pattern matching are very useful for structuring code to enhance readability.

5 Types

ML is a typed language. Types include:

- `int` (integers)
- `real` (real numbers)
- `bool` (truth values)
- Product types (or tuple types), built with `*`, e.g. `int * int * real`
- Function types, built with `->`, e.g. `int -> real`

There are built-in conventions on priority and associativity, e.g. `*` binds more strongly than `->`, and `->` associates to the right. Thus `int * int -> real` is the same type as `(int * int) -> real`, and `int -> int -> int` is the same type as `int -> (int -> int)`.

Note that we gave no association rule for the `*` operation on types. Indeed, `(int * int) * int` is not the same as `int * (int * int)`, and neither is the same as `int * int * int`, These types represent, respectively, pairs of an integer pair and an integer; pairs of an integer and an integer pair; and triples of integers.

Values

Each type represents a set of "values", and the type of an expression serves as a specification of the kind of value it denotes. For example, an expression of type `int -> real` denotes a function from integers to reals, and will (unless the application fails to terminate) produce a real result when applied to an integer-valued argument. Product (or tuple) types include

- `int * int`
  (pairs of integers)
- `int * int * int`
  (triples of integers)
- `real * int`
  (pairs of a real and an integer)
- `int * real`
  (pairs of an integer and a real); not the same as `real * int`

Function types include

- `int -> int`
  (functions from integers to integers)
- `real -> int`
  (functions from reals to integers)
- `int * int -> int * int`
  (functions from pairs of integers to pairs of integers)

There are primitive operations for combining integers and for combining reals, and the syntax echoes conventional math. Addition is `+`, multiplication is `*`, subtraction is `-`, and unary minus is `~`.

Real numerals include `3.0` and `333.999`. Integer numerals include `3` and `42`. You cannot use `3.0` instead of `3` (the type is different). The truth values are written `true` and `false`.

6 Expressions, declarations and patterns

First some arithmetical examples. In these examples, the comments describe the value denoted by the preceding expression; they also resemble the results produced when we evaluate the expressions using the ML interpreter.

```
(3+4)*6;
(*    = 42 : int                                    *)

(3.6 + 3.4) * 6.0;
(*     = 42.0 : real                                *)

42.0 / 7.0;
(*     = 6.0 : real                                 *)

(42 div 5, 42 mod 5);
(*    = (8, 2) : int * int                          *)

5 * (42 div 5) + (42 mod 5) = 42;
(*      = true : bool                               *)
```

Here is what the ML runtime read-eval-print loop said:

```
Standard ML of New Jersey v110.73 [. . .]
- (3+4)*6;
val it = 42 : int

- (3.6 + 3.4) * 6.0;
val it = 42.0 : real

- 42.0 / 7.0;
```

```
val it = 6.0 : real

- (42 div 5, 42 mod 5);
val it = (8,2) : int * int

- 5 * (42 div 5) + (42 mod 5) = 42;
val it = true : bool
```

The last example is an instance of a Fundamental Theorem of arithmetic, that specifies the relationship between $\widehat{\text{div}}$ and `mod`: For all integers m and all non-zero integers n, $n * (m \texttt{ div } n) + (m \texttt{ mod } n) = m$.

ML has many built-in primitive operations, some used as "infix" operators (including addition, multiplication and subtraction). Integer division and remainder `div` and `mod` have type `int * int -> int` and are infix operators. Real division is the infix operator `/` of type `real * real -> real`. There is also a "coercion" function `real` of type `int -> real`.

The ML notation for tuples uses parentheses, e.g. $(1, 42)$ and $(1, (2, 3))$.

The syntax for functions includes `fn p => e` (a "function expression" or "abstraction"), and application, written as `e e'` (`e` applied to `e'`). You may insert parentheses around one or both of the expressions in an application, to emphasize grouping or disambiguate the notation. For example, `e(e')` or `(e e')`. By convention, application associates to the left, so `e e1 e2` is the same as `(e e1) e2`.

Functions can use patterns `p` to match against values of their argument type. Patterns include variables, constants (like `0` and `true`), tuples, and lists. All variables used in a pattern must be different, so for example `(x,x)` is not a legal pattern. The pair pattern `(x,y)` matches pair values of form `(v1,v2)`, where `v1` and `v2` are values. In particular, this pattern matches the pair `(1,2)` of type `int * int`, and matches the pair `(true, true)` of type `bool * bool`. When a pattern is used to match against a value, if the match succeeds it produces *value bindings*, of the variables occurring in the pattern.

Matching `(x,y)` against the value `(1,2)` succeeds, and binds `x` to 1, `y` to 2; these bindings are available for use throughout the *scope* of the pattern. As an example, the scope of the pattern `(x,y)` in the expression

```
((fn (x,y) => x+y+3) (1, 2)) + 4
```

is the function body, i.e. the sub-expression `x+y+3`. The value of this whole expression is the same as the value of `(1+2+3)+4`.

You can, if desired (or required by us!), put type annotations in function expressions. This may help to guide the ML interpreter, or aid in debugging code. For example, `fn x => x+1` is an abstraction of type `int -> int`. We could have used any of the following alternatives:

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

In the first few weeks of class, we require you to annotate functions with argument and result types, so that you get used to using types. Later we will see that ML can automatically infer types using a syntax-directed algorithm, so that many of these annotations may safely be omitted.

Here is a simple function, which uses a tuple pattern to match against a pair of integers. When applied to an expression it evaluates that expression to obtain a pair of integers, binds `x` and `y` to the components, then returns a pair consisting of the quotient and remainder of these two values.

```
fn (x:int, y:int):int*int  => (x div y, x mod y);
(*        : int * int -> int * int                    *)

(fn (x:int, y:int):int*int => (x div y, x mod y)) (42, 5);
(*      = (8, 2) : int * int                          *)
```

Above, we used an "anonymous" function expression. You don't always have to give a function a name. However, if you plan to use it many times, naming it is a good idea, since you can use the name every time you want to apply the function without having to write the entire abstraction. We use a declaration to bind an expression value to a name. For a simple (non-recursive) declaration the syntax is `let val p = e in e' end`. For a simple recursive function definition, the syntax is `fun f p = e`.

Here are two examples. We attach a comment giving each function's name and type. After, we give another comment describing an example of the function's use, and a specification of the function's behavior. Every function should be accompanied by comments giving its name and type, and a specification that states clearly what assumptions you make about the arguments to which the function will be applied, and what properties the

value returned will have. Later we will introduce a more formal format for presenting specifications, which will help us to remember the key ingredients.

We can use a function name throughout the scope of its declaration. The scope of a declaration (at the top level of the ML interactive window, like this) begins at the declaration and continues unless another declaration for the same name is given later. The second declaration is thus allowed to use the first function. Note the use of `=` in the second function's body, at type `int * int -> bool`. The second function's body also uses a `let` expression that binds `q` and `r` to the components of (the value of) `divmod(x, y)` in the expression `x = q*y + r`. The scope of these bindings is local, only as far as the matching `end`.

```
fun divmod(x:int, y:int):int*int = (x div y, x mod y);

(*   divmod : int * int -> int * int                          *)
(*   Specification: if x:int, y:int, and y<>0,                 *)
(*     divmod(x,y) returns the pair (q, r),                    *)
(*     where q is the quotient and r is the remainder          *)
(*     of x divided by y.                                      *)

(*   Example: divmod(42, 5) = (8, 2) : int * int               *)

fun check (x:int, y:int):bool =
let
   val (q, r) = divmod(x, y)
in
   x = q*y + r
end;

(* check : int * int -> bool *)
(* Specification: For all x:int and all y>0: check(x,y) = true.  *)
```

This specification is valid, by the Fundamental Theorem of arithmetic.

The spec for `divmod` carefully requires that the `y` argument is non-zero. There is a good reason for this! Evaluating `divmod(42, 0)` is "exceptional", because you can't divide an integer by zero. ML detects this at runtime and reports the error as `exception Div`. Later we will discuss in more detail the ML facilities for dealing with runtime errors.

Evaluation

As we said earlier, ML is a *call-by-value* language: functions evaluate their arguments. For example, evaluation of the application

```
check(2+2, 5)
```

begins by evaluating `2+2` (result is `4`, obviously!), then `5` (already a value); then evaluates the body of `check` with `x` bound to `4`, `y` bound to `5`; this will evaluate `divmod`(4, 5), which returns the pair `(0, 4)`, then bind `q` to `0` and `r` to `4`, so the expression `x=q*y+r` gets evaluated with `x` bound to `4`, `y` to `5`, `q` to `0` and `r` to `4`. Because $4 = 0 * 5 + 4$, the result is `true`.

Or, as ML says:

```
- check(2+2, 5);
  val it = true : bool
```

The above explanation is awkward and somewhat convoluted, because we tried to use English to summarize a computation and there was a lot of sequencing to describe. The value of the expression `check(2+2,4)` obviously depends on the values of its sub-expressions `2+2` and `5`, but also on the value of `divmod(4,5)`. We will therefore introduce a convenient notation that allows us to be more succinct, and (if necessary) ignore some of the book-keeping. We write

```
e =>* e'
```

to mean that evaluation of `e` reaches `e'` in zero or more steps. When `e` is an expression with free variables, we can indicate the name-value bindings in scope during evaluation, as in:

```
With [x:4, y:5],
   divmod(x,y) =>* (4 div 5, 4 mod 5) =>* (0, 4)
```

Now we can revisit the earlier evaluation example. First note also that

```
[x:4,y:5] divmod(x,y) =>* (x div y, x mod y)
                      =>* (4 div 5, 4 mod 5)
                      =>* (0, 4)
```

It follows that

```
check(2+2, 5) =>* [x:4, y:5] let val (q,r) = divmod(4,5) in x=q*y+r end
              =>* [x:4, y:5] let val (q,r) = (0,4) in x=q*y+r end
              =>* [x:4,y:5,q:0,r:4] (x=q*y+r)
              =>* (4=0*5+4)
              =>* (4=4)
              =>* true
```

See why the first fact above helps to justify the second line in this derivation.

Here we have deliberately skirted around the issue of how to give a precise definition of the one-step evaluation relation `=>` for ML expressions. Even without being precise, by using `=>*` we are able to abstract away from the details and the number of steps. All of the statements that we make above in the example discussion are valid, and you should be able to understand what they say about expression evaluation at an intuitive level.

More examples

Some more examples using declarations, to explain more about bindings and scope. First we bind the name `pi` to the real number `3.14`, a not very accurate approximation to the value of π. Then we define functions `circ` and `area` for calculating the corresponding approximations to the circumference and the area of a circle with a given radius. These function definitions for `circ` and `area` are in the scope of this declaration of `pi`, so the occurrences of `pi` in their declarations get the value `3.14`. The attached comments give some examples to illustrate what happens.

```
val pi:real = 3.14;
(*     pi = 3.14 : real                                *)

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

(*   Example:  circ 1.0 = 6.28 : real          *)

(*    area : real -> real                              *)
fun area (r : real) : real = pi * r * r;

(*    Example: area 1.0 = 3.14                         *)
```

In the scope of these definitions, `pi` evaluates to `3.14`, `area` behaves like the function `fn r => 3.14 * r * r` and `circumference` behaves like the function `fn r => 2.0 * 3.14 * r`.

Now let's re-define `pi`, binding it to a slightly better approximation.

```
val pi:real = 3.14159;
(*     pi = 3.14159 : real                          *)
```

Although this binding "shadows" the earlier one – the current value of `pi` here is `3.14159` – it doesn't affect the behavior of the functions defined above, since the definitions of `area` and `circumference` given above are still in scope: `area 1.0 = 3.14`, still.

If we now redefine `area`, by typing:

```
fun area (r : real) : real = pi * r * r;
```

this introduces a new binding for `area`, shadowing the earlier one. Now we get `area 1.0 = 3.14159` .

To maintain consistency we would probably want to redefine `circ` similarly.

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

(*   Example: circ 1.0 = 6.141318 : real        *)
```

We could have used a `local` declaration, as follows, to emphasize that the sub-expression `2.0 * pi` is needed every time the function gets used:

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

(* Local binding for pi2 not in scope here *)

(*   circ' 1.0 = 6.141318 : real         *)
```

The functions `circ` and `circ'` are "equivalent" in the sense that when applied to equal arguments they produce equal results. For this reason we say that these functions are *extensionally equivalent*, or just equivalent.

7 Lists

ML has a type constructor `list` (used as a postfix operator) and constructs for building and manipulating lists.

For example, `int list` is the type of integer lists, `real list` is the type of lists of real numbers, and `(real * real) list` is the type of lists of pairs of real numbers. You can also have types such as `(int list) list` (lists of lists of integers), `(int -> int) list` (lists of functions from `int` to `int`), and so on.

The syntax for list expressions includes enumeration, such as `[ ], [1], [true, false], [3,1,4,1,5]`; `nil`, `x::L`, `L@R`. Note that `::` is called "cons", and `@` is "append". There is some redundancy in this notation. For instance, `nil = [ ]`, and `[1,2]=1::(2::nil)`. A value of type `int list` is a list of integers.

You can use `nil`, `::` and enumerations to build patterns for matching against list values, but *not* append! For example, `[ ]` is a list pattern matching only an empty list; `x::L` is a list pattern only matching non-empty lists (and it binds `x` to the list's head value, `L` to the list's tail); the pattern `[x,y,z]` matches lists of length 3, and binds `x` to the first item, `y` to the second, `z` to the third. The syntax `L@R` is not a legal pattern; to allow append patterns would make matching a much less well-behaved concept (can you see why?).

Some simple recursive functions that deal with lists of (decimal) digits. In a function that uses pattern-matching clauses, although we are insisting on type annotations, in the interests of brevity we only use them in the first clause.

```
fun eval ([ ]:int list):int = 0
 |  eval (d::L) = d + 10 * eval(L);

(* eval : int list -> int *)
(* For all L:int list, if L is a list of digits,  *)
(*  (eval L) evaluates to a non-negative integer. *)

(* Examples: *)
(* eval [2,4] = 42       *)
(* eval [0,0,2,4] = 4200 *)
```

```
fun decimal (n:int):int list =
  if n<10 then [n] else (n mod 10) :: decimal(n div 10);

(* decimal : int -> int list      *)
(* For n>=0, eval(decimal n) = n. *)

(* Examples: *)
(* decimal 0 = [0]          *)
(* decimal 42 = [2,4]       *)
(* decimal 4200 = [0,0,2,4] *)
(* decimal ~42 = [~42]      *)
```

8 Specifications

The comments in the above examples serve to specify what we expect the `eval` and `decimal` functions to behave like, and how we intend to use them. It is helpful to adopt a standard convention for writing specifications, and we will from now on insist that each function that you define (other than a built-in ML function, or one that we treat as a special case) must be preceded by 3 comments. First, a comment giving the function name and type; then a requires-condition, listing assumptions about the value of the function's argument that you expect to hold whenever you use the function; and finally an ensures-condition that describes the value produced by calling the function on an argument satisfying these assumptions. Since only well-typed expressions ever get evaluated, we interpret the specification as talking about the function's behavior when applied to suitably typed arguments.

Here are the functions from the previous section, along with specifications written in this style.

```
(* eval : int list -> int *)
(* REQUIRES: L is a list of (decimal) digits.              *)
(* ENSURES:  (eval L) evaluates to a non-negative integer. *)

fun eval ([ ]:int list):int = 0
 |  eval (d::L) = d + 10 * eval(L);
```

```
(* decimal : int -> int list    *)
(* REQUIRES: n >= 0             *)
(* ENSURES: eval(decimal n) = n *)

fun decimal (n:int):int list =
   if n<10 then [n] else (n mod 10) :: decimal(n div 10);
```

We could have used the following specification comments for `decimal` instead, and they assert the same properties as the ones above:

```
(* decimal : int -> int list     *)
(* REQUIRES: n >= 0 *)
(* ENSURES: decimal(n) evaluates to a list of decimal digits L *)
(*          such that eval(L) = n.                             *)
```

Check that you understand what it means to say that `eval` and `decimal` satisfy these specs.

Does `decimal` satisfy the following specification?

```
(* decimal : int -> int list *)
(* REQUIRES: n>42 *)
(* ENSURES: decimal(n) evaluates to a list of decimal digits L *)
(*          such that eval(L) = n.                             *)
```

Yes, but this is a *weaker*, less informative spec than the ones we gave before, because it only tells us what happens when the function is applied to integers bigger than 42.

Figure out why `decimal` does *not* satisfy the following specification:

```
(* decimal : int -> int list     *)
(* REQUIRES: true                *)
(* ENSURES: eval(decimal n) = n. *)
```

9 Coming soon

- How to *prove* that a function meets its specification.
- Testing may be helpful to convince you that this is likely true, but testing cannot usually cover *all* cases.
- Proof techniques are based on *induction*.