15-150 Fall 2013

Stephen Brookes

Lecture 10
Functions as values

Thursday, 26 September

1 Outline

- Functions as values
  - first-order and higher-order types
  - anonymous, non-recursive, function expressions
  - recursive function declarations
  - application and composition
  - extensionality and equality
- Programming with higher-order functions
  - functions that take functions as arguments
  - functions that return functions as results
- Maps and folds on lists
  - higher-order functions in action
  - bulk processing of data
  - potential for parallelism

2 Introduction

So far we have used functions from tuples of values to values, and values have been integers/lists/trees. We call such functions "first-order", a rather old-fashioned designation that persists for historical reasons. For example functions of type `int -> int` are first-order. Actually the term "first-order" really refers to the type, and every function value of a first-order type is deemed to be a first-order function. We can also define functions that take functions as arguments and/or return functions as results. These are called "higher-order" functions. Again this is really a property of types. A type is higher-order if it has form `t1 -> t2` where `t1` or `t2` contains an `->`. For example, functions of type `(int -> int) -> int` are higher-order.

Recall how we defined *equivalence* for first-order functions. For example, two functions of type `int -> int` are equivalent if they produce equal results (or both fail to terminate) when applied to equal integer arguments. This notion generalizes in a straightforward way to higher-order functions. Two functions of type `(int -> int) -> int` are equivalent if they produce equal results (or both fail to terminate) when applied to equivalent functions.

ML has a special syntax for "curried functions" of several arguments, which allows us to define higher-order functions using a convenient notation.

These ideas are very powerful. We can write elegant functional programs to solve interesting problems in very natural ways, and we can reason about functional programs by thinking in terms of mathematical functions.

3 Functions as values

Recall that ML allows recursive function definitions, using the keyword `fun`. Function definitions (or declarations) give names to function values. If we want to apply the function to a variety of different arguments, it's obviously convenient to use the name over and over, rather than having to write out the function code every time.

But we can also build "anonymous" , non-recursive, function expressions, using `fn`. This can also be convenient, especially when we want to supply the function as an argument to another function! For example, the expression

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

is well-typed, has type `int -> int`. An expression like this (beginning with `fn`) is called an *abstraction*.[1]

We can use it to build other expressions, by applying it to an expression of the correct argument type, as in

```
(fn x:int => x+1) (20+21),
```

and this particular expression evaluates to 42.

We can also build other expressions out of abstractions. For example,

```
(fn x:int => x+1, fn y:real => 2.0 * y)
```

is an expression of type `(int -> int) * (real -> real)`.

If we so desire, we can even give a name to a non-recursive function, using a `val` declaration, as in the declaration

[1]Also known as a λ-expression for historical reasons: `fn x => e` is the ML analogue to $\lambda x.e$ from the λ-calculus.

```
val (inc, dbl) = (fn x:int => x+1, fn y:real => 2.0 * y)
```

which binds `inc` to the function value `fn x:int => x+1` and `dbl` to the function value `fn y:real => 2.0*y`.

The ML interpreter treats functions as values: evaluation of an expression of type `t->t'` stops as soon as it reaches an abstraction, i.e. a `fn`-expression or a function name that has been declared (and hence bound to) a function value.

ML (like all functional programming languages) makes no restriction on function values just because they are functions; you can use functions either as arguments to other functions, or as results to be returned by functions. Thus functions are values just like values of other types.

Composition

Function composition is an infix operator written as `o` and can also be used (in applicative form) as a function

```
(op o) : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b).
```

For all types `t1, t2, t3`, and values `f:t2 -> t3` and `g:t1 -> t2`, `(f o g)` has type `t1 -> t3`, evaluates as follows:

```
(f o g) =>* (fn x => f(g x))
```

and satisfies the equation

```
(f o g)(x) = f(g x)
```

for all values `x` of type `t1`. Moreover, `(op o)(f, g) = f o g`.
Note that `f o g` evaluates to a function that first applies `g` to its argument then applies `f` to the result. [2]
For example

```
(fn (x:int) => x+1) o (fn (y:int) => y+1)
```

evaluates to a function of type `int -> int` that adds 2 to its argument. In fact, writing = for extensional equivalence, we have

[2]Please remember the order in which the functions get composed! Sometimes in math texts function composition is defined so that $f \circ g$ applies f first, then g.

```
(fn (x:int) => x+1) o (fn (y:int) => y+1) = (fn (z:int) => z+2)
```

Some examples of function values built using composition:

```
val inc = fn (x:int) => x+1;
val double = fn (x:int) => 2*x;
val add2 = inc o inc;
val add4 = add2 o add2;
val shrink =
     fn (x:int) => case compare(x, 0) of
                      |   EQUAL => 0
                      |   LESS  => x+1
                      |   _     => x-1;
```

Each of the function expressions in this code has type `int -> int`.
In the scope of these declarations, the following equations hold:

```
inc 2 = 3
(double o inc) 4 = 10
(inc o double) 4 = 9
add2 2 = 4
add4 2 = 6
shrink 2 = 1
shrink (~2) = ~1
(fn y:int => add4(add4 y)) (add2 15 + add4 13) = 42
```

Each of the expressions in the equations (I mean the expressions being "equated") has type `int`. In our earlier discussion of *referential transparency* we introduced our mathematical notion of equality for expressions of type `int`: two expressions of type `int` are equal if and only if they evaluate to the same integer numeral, or they both fail to terminate. That's how we interpret these equations: in each case the equation says that the expression on the left evaluates to the numeral on the right.

Recall that we said that two function values `f` and `g` of type `int -> int` are *extensionally* equal if and only if, for all integer values `n`, `f(n)` and `g(n)` are equal. That is, `f = g` iff for all `n:int`, `f(n) = g(n)`.

Some examples of equality for function expressions of type `int -> int`:

```
(inc o double) = fn x:int => (2*x)+1
(double o inc) = fn x:int => 2*(x+1)
```

```
add2 = (fn y:int => inc(inc y))
add4 = (fn z:int => add2(add2 z))
(fn x:int => x+1) = (fn w:int => w+1)
```

4 Higher-order functions

In the previous section we presented some functions from integers to integers. These are usually classified as *first-order* functions, because their arguments and results are primitive data values, in this case integers, not functions.

In contrast, the composition function, also discussed above as

```
(op o) : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)
```

has a (most general) type of form `t -> t'`, where the argument type `t` is

```
('a -> 'b) * ('c -> 'a),
```

built from function types. A function type like this in which the "argument type" is itself a function type, is classified as a *higher-order* type. Note that every *instance* of a higher-order type is also a higher-order type, since the arrows don't go away!

The significance of higher-order types *versus* the first-order case is that a function with a higher-order type may take a *function* as argument. Applying a higher-order function of type `t -> t'` to a function of the required argument type `t` will then produce an application expression of the corresponding result type `t'`, just as with first-order functions. There is nothing special here, as far as ML is concerned. The reason: in ML functions are values, and as such are capable of being used as arguments of other functions, or being returned as results.

Equality and referential transparency

The notion of *extensional equality* extends to higher-order function types in a natural way. Intuitively, two functions are equal iff they map equal arguments to equal results.

For example, consider the type `(int -> int) -> int`. We already know what it means to say that two functions of type `int -> int` are equal. We say that functions `F` and `G` of type `(int -> int) -> int` are *equal*, `F = G`, if and only if, for all functions `f` and `g` of type `int -> int`, if `f = g` then `F(f) = G(f)`.

This idea generalizes in the obvious way. Assume that we already have a definition of equality for expressions of type `t` and for expressions of type `t'`. Consider two functions `F` and `G` of type `t -> t'`. We say that `F = G` if and only if, for all values `x` and `y` of type `t`, if `x = y` then `F(x) = G(y)`.

You should check that this definition coincides with the notion of equality already introduced (less formally) earlier for first-order types like `int -> int` and `int list -> int list`.

It may not be obvious, but equality satisfies the usual mathematical laws that we expect for a sensible notion of "equality". In particular, for every type `t`, equality for expressions of type `t` is an *equivalence relation*:

- For all expressions `e:t`, `e = e`.
- For all expressions $\mathtt{e}_1, \mathtt{e}_2 : \mathtt{t}$, if $\mathtt{e}_1 = \mathtt{e}_2$, then $\mathtt{e}_2 = \mathtt{e}_1$.
- For all expressions $\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3 : \mathtt{t}$, if $\mathtt{e}_1 = \mathtt{e}_2$ and $\mathtt{e}_2 = \mathtt{e}_3$, then $\mathtt{e}_1 = \mathtt{e}_3$.

Now that we've spoken more generally about "equality", we remind you of the fundamental and important property that functional programs are *referentially transparent*. We can now state a more precisely formulated version of this property, as follows. Here we use the notation $E[e]$ for an expression E having a sub-expression e; then $E[e']$ for the expression obtained by replacing the sub-expression e by e'; and we implicitly refer to equality on type `t` (in $e = e'$) and on type `T` (in $E[e] = E[e']$).

> (Referential Transparency)
> If $E[e]$ is a well-typed expression of type `T`, with a sub-expression e of type `t`, and e' is another expression of type `t` such that $e = e'$, then $E[e] = E[e']$.

Another way to say the same thing is that *equality is a congruence*. That's a fancy way to say that "Every syntactic construct in ML *respects* equality".

For example, to say that

```
(op +) : int * int -> int
```

respects equality means that for all well-typed expressions $\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}'_1, \mathtt{e}'_2$ of type `int`, if $\mathtt{e}_1 = \mathtt{e}'_1$ and $\mathtt{e}_2 = \mathtt{e}'_2$ then $\mathtt{e}_1 + \mathtt{e}_2 = \mathtt{e}'_1 + \mathtt{e}'_2$.

Here are some simple examples, to help make these ideas clear.

- Consider the function `checkzero:(int -> int) -> int` given by:

  ```
  fun checkzero (f:int -> int):int = f(0);
  ```

  It's easy to check that for all `f,g:int->int`, if `f = g` we get `f(0)=g(0)` and thus `checkzero(f) = checkzero(g)`. Hence, according to the definition of equality for type `(int -> int) -> int`, we have

  ```
  checkzero = checkzero.
  ```

  Although you might think this is trivial, this equation implies the non-trivial fact that for all `f` and `g` of type `int -> int`, `checkzero(f) = checkzero(g)`.

- Recall the composition operator, from above. It is easy to check that, for all types `t1, t2, t3`, and all function values `f,f':t2 -> t3` and `g,g':t1 -> t2`, and `f = f'` and `g = g'`, then `f o g = f' o g'`. This is the same as saying that `(op o)` respects equality.

- We saw that expressions `fn x:int => x+x` and `fn y:int => 2*y` are equal. It follows by referential transparency that

  ```
  40 + (fn x:int => x+x) 1 = 40 + (fn y:int => 2*y) 1
  ```

  Although in this particular example we can easily see that the value of the overall expression is 42 in both cases, in a more complex expression we could make the analogous deduction even without knowing what the value of the entire expression is!

Before moving on, notice that *equality* on function expressions is *NOT* the same as "both evaluate to the same value, or both fail to terminate", which was how we characterized equality at simple types like `int` and `int list`. Instead, two function expressions `e` and `e'` are equal iff they both fail to terminate or they evaluate to function values `f` and `f'` that are extensionally equal.

For example, the expressions `fn x:int => 2*x` and `fn y:int => y+y` are (already) function values (of type `int -> int`) that are extensionally equal. But they don't evaluate to the *same* function value: each evaluates to itself and the expressions are syntactically different. Nevertheless, each of these expressions can be said to *compute* the same function from integers to integers, and that's why we are comfortable saying that they are "equal".

In summary, *functions are values*, as far as the ML intepreter is concerned; evaluation of an expression of a function type stops as soon as it reaches an abstraction. But function types are not ML "equality types", so you can't test function values for equality. (We already discussed why this would be infeasible – remember the Halting Problem!). And the appropriate notion of equality for functions is our mathematically based notion, as defined here. Using extensionality and referential transparency, we can think of ML function expressions as mathematical objects, subject to mathematical analysis and logical reasoning.

Recursive function definitions and equality

In the scope of a recursive function definition of the form

```
fun f(x:t):t' = e'
```

the function name `f` satisfies the equation `f = fn x:t => e'`.

There is a corresponding equational law:

If `v` is a value such that `e = v`, then

```
(fn x:t => e') e = [x:v]e',
```

where we write `[x:v]e'` for the expression built by substituting `v` for the free occurrences of `x` in `e'`.

This law is sometimes called the β-value reduction law, for historical reasons having to do with the λ-calculus. Since ML functions are call-by-value, i.e. functions always evaluate their arguments, you cannot substitute into the function body unless the argument expression has been evaluated down to a syntactic value.

Functions evaluate their arguments

Let `(fn x:t => e')` be well-typed expression of type `t->t'`. If `e:t` is a well-typed expression and `v` is an ML value such that `e =>* v`, then

```
(fn x:t => e') e =>* [x:v]e',
```

where again we write `[x:v]e'` for the expression built by substituting `v` for the free occurrences of `x` in `e'`.

Non-termination and equality

We have been very careful in the above account of evaluation and equality to avoid accidentally concluding erroneous "facts" by sloppy reasoning. For example, consider the recursive function given by

```
fun silly (n:int) : int = silly n;
```

This declaration binds `silly` to the function value `fn n:int => (silly n)`, of type `int -> int`. As we said above, in the scope of this declaration, we have the equation

```
(a)  silly = fn n:int => (silly n)
```

What can we prove about the "value" of the expression `silly 0`? (It isn't hard to see that evaluation of this expression doesn't terminate.) We can make a sequence of logical steps, such as:

```
silly 0
   = (fn x:int => (silly x)) 0    by (a) and referential transparency
   = [x:0] (silly x)              by the value-reduction law
   = silly 0                      by definition of substitution
```

but you *cannot* find a value `v` such that `silly 0 = v` is provable!

5 Higher-order functions on lists

Filtering a list

A total function `p` of type `t -> bool` represents a "predicate" on values of type `t`. A value `v:t` such that `p(v) = true` is said to *satisfy* the predicate; when `p(v) = false` the value `v` does not satisfy the predicate. Totality of `p` means that for each value `v` there is a definite answer: `p(v)` evaluates to `true` or `false`.

We can define a recursive ML function with the following specification:

```
(* filter : (a -> bool) -> (a list -> a list)                 *)
(* REQUIRES: p is a total function                            *)
(* ENSURES: filter p L = a list of the values in L that satisfy p *)
```

```
  fun filter p [ ] = [ ]
   |  filter p (x::L) = if p(x) then x::filter p L else filter p L;
```

Actually some of the parentheses that we included in the type specification above for `filter` are superfluous, because the ML function type constructor `->` associates to the right. We could just as well have said

```
  (* filter : (a -> bool) -> 'a list -> a list                    *)
```

The only reason we didn't do that was because we wanted to emphasize the fact that, when applied to a predicate `p`, `filter p` returns a *function*. You don't have to always apply `filter` to a predicate and a list. It's perfectly reasonable to apply it to just a predicate and to use the resulting function as a piece of data to be passed around, used by other functions, or just returned.

Consider for example the following first-order function

```
  (* divides : int * int -> bool *)
  (* REQUIRES: x>0 *)
  (* ENSURES:
      For all y>0, divides(x, y) = true if y mod x = 0, false otherwise *)

   fun divides (x, y) = (y mod x = 0);
```

Because of its type, the only way we can apply this function is to a pair of integers. For example, `divides(2,4) = true`.

But if we introduce a closely related function as follows, with the type `int -> (int -> bool)` we'll be able to "partially apply" it to one integer and get back a predicate that checks for divisibility by that integer.[3]

```
  (* divides' : int -> (int -> bool) *)
  (* REQUIRES: x>0 *)
  (* ENSURES:
      For all y>0, divides' x y = true if y mod x = 0, false otherwise *)

  fun divides' x y = (y mod x = 0);
```

[3]This is an example of "currying" a function of two arguments to obtain a function of one argument that returns another function. We'll return to this idea in a later lecture when we talk about "staging" computation.

For example, we can apply `divides'` to 2 and get a function that tests for evenness:

```
(* even : int -> bool)
val even = divides' 2;
```

We can use `divides'` to help us build lists of prime numbers:

```
(* sieve : int list -> int list *)
fun sieve [ ] = [ ]
 |  sieve(x::L) = x::sieve(filter (not o (divides' x)) L);

(* primes : int -> int list *)
fun primes n = sieve (upto 2 n);
```

Here we have used the built-in ML function

```
not : bool -> bool
```

whose behavior is completely determined by the equations

```
not true = false
not false = true
```

For `n>1`, `primes n` returns the list of prime numbers between 2 and `n`, in increasing order. For example, `primes 10 = [2,3,5,7]`.

Now, admittedly, we *could* have developed the primes function and the sieve function differently, without insisting on the use of `divides'` and `filter`. But we feel this is an elegant example that shows the potential advantages of functional programming with higher-order functions.

Transforming the data in a list

A very common task involves taking a list of some type and performing some operation on each item in the list, to produce another list, of a possibly different type. Typically we have some function that we wish to apply to all ties in the list. We can encapsulate this process very naturally as a higher-order function

```
map : ('a -> 'b) -> ('a list -> 'b list)
```

Again, some of the parentheses are redundant here, but we want to emphasize that `map` takes a function as argument and returns a function as its result.

ML has a built-in function like this, and its definition is:

```
fun map f [ ] = [ ]
 |  map f (x::L) = (f x)::(map f L);
```

As a simple example using `map`, the function

```
addtoeach' : int -> (int list -> int list)
```

defined by

```
fun addtoeach' a = map (fn x => x+a);
```

is the "curried" version of the function

```
addtoeach : int * int list -> int list
```

that we dealt with earlier in the semester.

Recall the function `sum:int list -> int` given by:

```
fun sum [ ] = 0
 |  sum (x::L) = x + sum L;
```

We already saw that for an integer list `L`, `sum L` returns the sum of the integers in `L`. We can use `sum` and `map` to obtain a function

```
count : int list list -> int
```

that adds all the integers in a list of integer lists:

```
fun count R = sum (map sum R);
```

In reasoning about code that uses `map`, we can typically use induction on the length of lists.

For example, as an exercise, prove the following properties:

- If `f:t1 -> t2` is total, then for all `L:t1 list`, `map f L` evaluates to a list `R` such that `length(L) = length(R)`.
- If `f:t1 -> t2` is total, then for all lists `A` and `B` of type `t1 list`,

  ```
  map f (A @ B) = (map f A) @ (map f B).
  ```

  The assumption that the function being "mapped along the list" is total was made here to simplify the problem. In fact, you just need the assumption that for all values `x` occurring in the list, `f(x)` terminates.

Combining the data in a list

Another common task involves *combining* the values in a list to obtain some composite value: for example, as we have seen, adding the integers in a list to obtain their sum. Other examples include concatenating a list of lists to get a single list, and counting the number of items in a list (to get its length). Again we can encapsulate the general idea using higher-order functions. Since lists are inherently sequential data structures there are actually two distinct and natural sequential orders in which we might combine items: from head to tail, or *vice versa*. ML has two built-in functions, accordingly:

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

and their definitions are

```
fun foldl g z [ ]    = z
 |  foldl g z (x::L) = foldl g (g(x,z)) L;

fun foldr g z [ ]    = z
 |  foldr g z (x::L) = g(x, foldr g z L);
```

For all types `t1` and `t2`, all functions `g:t1 * t2 -> t2`, and all values `z:t2`,

```
foldl g z : t1 list -> t2
foldr g z : t1 list -> t2
```

are functions that can be applied to a list and will combine (using `g`) the items in the argument list with `z`. Assuming that `g` is total, for all non-negative integers n and all values $\mathtt{x}_1, \ldots, \mathtt{x}_n$ of type `t1`, the equations

$$\begin{aligned}\mathtt{foldl\ g\ z}\,[\mathtt{x}_1, \ldots, \mathtt{x}_n] &= \mathtt{g}(\mathtt{x}_n, \mathtt{g}(\mathtt{x}_{n-1}, \ldots \mathtt{g}(\mathtt{x}_1, \mathtt{z}) \ldots)) \\ \mathtt{foldr\ g\ z}\,[\mathtt{x}_1, \ldots, \mathtt{x}_n] &= \mathtt{g}(\mathtt{x}_1, \mathtt{g}(\mathtt{x}_2, \ldots, \mathtt{g}(\mathtt{x}_n, \mathtt{z}) \ldots))\end{aligned}$$

hold.

As examples, we can add the integers in an integer list using either of these fold functions:

```
fun suml L = foldl (op +) 0 L;
fun sumr L = foldr (op +) 0 L;
```

And we'll get, for all $n \geq 0$ and all integer lists $[\mathtt{x}_1, \ldots, \mathtt{x}_n]$,

$$\begin{aligned} \mathtt{suml}\ [\mathtt{x}_1, \ldots, \mathtt{x}_n] &= \mathtt{x}_n + (\mathtt{x}_{n-1} + \ldots + (\mathtt{x}_1 + 0) \ldots) \\ \mathtt{sumr}\ [\mathtt{x}_1, \ldots, \mathtt{x}_n] &= \mathtt{x}_1 + (\mathtt{x}_2 + \ldots + (\mathtt{x}_n + 0) \ldots) \end{aligned}$$

Since addition of integers is an associative and commutative operation, the sums on the right-hand sides of these equations are equal. Since `x+0 = x` for all integers `x`, we therefore have

$$\mathtt{suml}\ [\mathtt{x}_1, \ldots, \mathtt{x}_n] = \mathtt{sumr}\ [\mathtt{x}_1, \ldots, \mathtt{x}_n] = \textstyle\sum_{i=1}^{n} \mathtt{x}_i$$

And since every value of type `int list` is expressible in the form $[\mathtt{x}_1, \ldots, \mathtt{x}_n]$ for some $n \geq 0$ and some integer values $\mathtt{x}_i$ ($i = 1, \ldots, n$), this shows that the functions `suml` and `sumr` are extensionally equal, i.e. `suml = sumr`.

Given our earlier results about the properties of `sum` (as defined above), we have also shown here that `sum = suml = sumr`.

It follows, then, that the functions

```
fun countl R = suml (map suml R);
fun countr R = sumr (map sumr R);
```

are also extensionally equivalent. Moreover, `countl = countr = count`.

We've appealed to *referential transparency* again, implicitly, in the above paragraphs, to justify the argument. Make sure you see where.

Exercise

Suppose `g` is a total function of type `t1 * t2 -> t2`, such that for all values `x:t1, y:t1, z:t2, g(x,(g(y,z)) = g(y,g(x,z))`. Prove that for all values `z:t2` and lists `L:t1 list`,

```
foldr g z L = foldl g z L.
```

Explain why this shows that (with these assumptions), `foldr g = foldl g`.

Insertion sort, revisited

Recall that we defined a function

```
ins : int * int list -> int list
```

with the specification that for all integers `x` and all integer lists `L`, if `L` is sorted, then `ins(x,L)` is equal to a sorted permutation of `x::L`.

Using this function, we can easily obtain an implementation of insertion sort, by defining:

```
(* isortl : int list -> int list *)
val isortl = foldl ins [ ];
```

Indeed, it is easy to prove, by induction on `L`, that for all integer lists `L`,

```
foldl ins [ ] L = a sorted permutation of L.
```

(In this proof the only information about `ins` that you need is the prior result that `ins` satisfies its specification.)

Equally well we could have used the other fold function:

```
(* isortr : int list -> int list *)
val isortr = foldr ins [ ];
```

And we could prove, by induction on `L`, that for all integer lists `L`,

```
foldr ins [ ] L = a sorted permutation of L.
```

Further, since for an integer list `L` there is *exactly one* sorted permutation of `L`, we could then conclude that `isortl = sortr`.

Exercise

If you did the exercise above, you can use its result here.
Using the definition of `ins`, show that for all `x,y:int` and `L:int list`, `ins(x, ins(y, L))=ins(y, ins(x, L))`.
Deduce that `isortl = isortr`.