CSE341, Fall 2011, Lecture 7 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

Lectures 7, 8, and 9 will focus on first-class functions and function closures. By "first-class" we mean that functions can be computed and passed *wherever* other values can in ML. As examples, we can pass them to functions, return them from functions, put them in pairs, have them be part of the data a datatype constructor carries, etc. Function closures refer to functions that use variables defined outside of them, which we will start seeing next lecture. The term *higher-order function* just refers to a function that takes or returns other functions.

Taking functions as arguments

The most common use of first-class functions is passing them as arguments to other functions, so we motivate this use first.

Here is a first example of a function that takes another function:

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

We can tell the argument `f` is a function because the last line calls `f` with an argument. What `n_times` does is compute `f(f(...(f(x))))` where the number of calls to `f` is `n`. That is a genuinely useful helper function to have around. For example, here are 3 different uses of it:

```
fun double x = x+x
val x1 = n_times(double,4,7) (* answer: 112 *)

fun increment x = x+1
val x2 = n_times(increment,4,7) (* answer: 11 *)

val x3 = n_times(tl,2,[4,8,12,16]) (* answer: [12,16] *)
```

Like any helper function, `n_times` lets us *abstract* the common parts of multiple computations so we can *reuse* some code in different ways by passing in different arguments. The main novelty is making one of those arguments a function, which is a powerful and flexible programming idiom. It also makes perfect sense — we are not introducing any new language constructs here, just using ones we already know in ways you may not have thought of.

Once we define such abstractions, we can find additional uses for them. For example, even if our program today does not need to triple any values n times, maybe tomorrow it will, in which case we can just define the function `triple_n_times` using `n_times`:

```
fun triple x = 3*x

fun triple_n_times (n,x) = n_times(triple,n,x)
```

Polymorphic Types

Let us now consider the type of `n_times`, which is `('a -> 'a) * int * 'a -> 'a`. It might be simpler at first to consider the type `(int -> int) * int * int -> int`, which is how `n_times` is used for `x1` and `x2` above: It takes 3 arguments, the first of which is itself a function that takes and returns an `int`. Similarly, for `x3` we use `n_times` as though it has type `(int list -> int list) * int * int list -> int list`. But choosing either one of these types for `n_times` would make it less useful because only some of our example uses would type-check. The type `('a -> 'a) * int * 'a -> 'a` says the third argument and result can be any type, but they have to be the *same* type, as does the argument and return type for the first argument. When types can be any type and do not have to be the same as other types, we use different letters (`'b`, `'c`, etc.)

This is called *parametric polymorphism*, or perhaps more commonly *generic types*. It lets functions take arguments of any type. It is technically a separate issue from first-class functions: there are functions that take functions and do not have polymorphic types and there are functions with polymorphic types that do not take functions. However, many of our examples with first-class functions will have polymorphic types. That is a good thing because it makes our code more reusable.

It is quite essential to ML. For example, without parametric polymorphism, we would have to redefine lists for every type of element that a list might have. Instead, we can have functions that work for any kind of list, like `length`, which has type `'a list -> int` even though it does not use any function arguments. Conversely, here is a higher-order function that is not polymorphic: it has type `(int->int) * int -> int`:[1]

```
fun times_until_zero (f,x) =
    if x = 0 then 0 else 1 + times_until_zero(f, f x)
```

Anonymous functions

There is no reason that a function like `triple` that is passed to another function like `n_times` needs to be defined at top-level. As usual, it is better style to define such functions locally if they are needed only locally. So we could write:

```
fun triple_n_times (n,x) =
  let fun triple x = 3*x in n_times(triple,n,x) end
```

In fact, we could give the `triple` function an even smaller scope: we need it only as the first argument to `n_times`, so we could have a let-expression there that evaluates to the triple function:

```
fun triple_n_times3 (n,x) = n_times((let fun triple y = 3*y in triple end), n, x)
```

Notice that in this example, which is actually poor style, we need to have the let-expression "return" `triple` since, as always, a let-expression produces the result of the expression between `in` and `end`. In this case, we simply look up `triple` in the environment, and the resulting function is the value that we then pass as the first argument to `n_times`.

ML has a much more concise way to define functions right where you use them, as in this final, best version:

```
fun triple_n_times3 (n,x) = n_times((fn y => 3*y), n, x)
```

This code defines an *anonymous function* `fn y => 3*y`. It is a function that takes an argument `y` and has body `3*y`. The `fn` is a keyword and `=>` (not `=`) is also part of the syntax. We never gave the function a name (it's *anonymous*, see?), which is convenient because we did not need one. We just wanted to pass a function to `n_times`, and in the body of `n_times`, this function is bound to `f`.

[1]It would be better to make this function tail-recursive using an accumulator; see Lecture 6.

It is common to use anonymous functions as arguments to other functions. Moreover, you can put an anonymous function anywhere you can put an expression — it simply is a value, the function itself. The only thing you cannot do with an anonymous function is recursion, exactly because you have no name to use for the recursive call. In such cases, you need to use a `fun` binding as before, and `fun` bindings must be in let-expressions or at top-level.

For non-recursive functions, you could use anonymous functions with `val` bindings instead of a `fun` binding. For example, these two bindings are exactly the same thing:

```
fun increment x = x + 1
val increment = fn x => x+1
```

They both bind `increment` to a value that is a function that returns its argument plus 1. So function-bindings are *almost* syntactic sugar, but they support recursion, which is essential.

Unnecessary Function Wrapping

While anonymous functions are incredibly convenient, there is one poor idiom where they get used for no good reason. Consider:

```
fun nth_tail_lame (n,x) = n_times((fn y => tl y), n, x)
```

What is `fn y => tl y`? It is a function that returns the list-tail of its argument. But there is already a variable bound to a function that does the exact same thing: `tl`! In general, there is no reason to write `fn x => f x` when we can just use `f`. This is analogous to the beginner's habit of writing `if x then true else false` instead of `x`. Just do this:

```
fun nth_tail (n,x) = n_times(tl, n, x)
```

Maps and filters

We now consider a very useful higher-order function over lists:

```
fun map (f,lst) =
    case lst of
        [] => []
      | fst::rest => (f fst)::(map(f,rest))
```

The `map` function takes a list and a function `f` and produces a new list by applying `f` to each element of the list. Here are two example uses:

```
val x4 = map (increment, [4,8,12,16]) (* answer: [5,9,13,17] *)
val x5 = map (hd, [[1,2],[3,4],[5,6,7]]) (* answer: [1,3,5] *)
```

The type of `map` is illuminating: `('a -> 'b) * 'a list -> 'b list`. You can pass `map` any kind of list you want, but the argument type of `f` must be the element type of the list (they are both `'a`). But the return type of `f` can be a different type `'b`. The resulting list is a `'b list`. For `x4`, both `'a` and `'b` are *instantiated* with `int`. For `x5`, `'a` is `int list` and `'b` is `int`.

The ML standard library provides a very similar function `List.map`, but it is defined in a curried form, a topic we will discuss in an upcoming lecture.

The definition and use of map is an incredibly important idiom even though our particular example is simple. We could have easily written a recursive function over lists of integers that incremented all the elements, but instead we divided the work into two parts: The map *implementer* knew how to traverse a recursive data structure, in this case a list. The map *client* knew what to do with the data there, in this case increment each number. You could imagine either of these tasks — traversing a complicated piece of data or doing some calculation for each of the pieces — being vastly more complicated and best done by different developers without making assumptions about the other task. That's exactly what writing `map` as a helper function that takes a function lets us do.

Here is a second very useful higher-order function for lists. It takes a function of type `'a -> bool` and an `'a list` and returns the `'a list` containing only the elements of the input list for which the function returns true:

```
fun filter (f,lst) =
    case lst of
        [] => []
      | fst::rest => if f fst
                     then fst::(filter (f,rest))
                     else filter (f,rest)
```

Here is an example use that assumes the list elements are pairs with second component of type `int`; it returns the list elements where the second component is even:

```
fun get_all_even_snd lst = filter((fn (_,v) => v mod 2 = 0), lst)
```

(Notice how we are using a pattern for the argument to our anonymous function.)

Returning functions

Functions can also return functions. Here is an example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

The type of `double_or_triple` is `(int -> bool) -> (int -> int)`: The if-test makes the type of `f` clear and as usual the two branches of the if must have the same type, in this case `int->int`. However, ML will print the type as `(int -> bool) -> int -> int`, which is the same thing. The parentheses are unnecessary because the `->` "associates to the right", i.e., `t1 -> t2 -> t3 -> t4` is `t1 -> (t2 -> (t3 -> t4))`.

Not just for numbers and lists

Because ML programs tend to use lists a lot, you might forget that higher-order functions are not useful only for lists. Some of our first examples just used integers. But higher-order functions also are great for our own data structures that we define with datatype bindings. Here we reuse our `is_even` function from before to see if all the constants in an arithmetic expression are even. We could easily reuse `true_of_all_constants` for any other property we wanted to check.

```
datatype exp = Constant of int | Negate of exp | Add of exp * exp | Multiply of exp * exp

fun true_of_all_constants(f,e) =
    case e of
        Constant i => f i
      | Negate e1 => true_of_all_constants(f,e1)
      | Add(e1,e2) => true_of_all_constants(f,e1)
                      andalso true_of_all_constants(f,e2)
      | Multiply(e1,e2) => true_of_all_constants(f,e1)
                           andalso true_of_all_constants(f,e2)

fun all_even e = true_of_all_constants(is_even,e)
```