1 First-class functions

A value this is first class may be passed as an argument to a function, returned as a value from a function, and bound to a variable—i.e., it is a first-class “citizen” of the language, “afforded all the rights” of other values.

In many languages, functions are not first class. In Java, for example, functions can’t be passed as arguments, returned as results, or assigned to variables. (However, the reflection API can be used to simulate some of these actions.)

But in ML, functions are first class. They can be used anywhere that other values (e.g., integers) can be. Functions can be bound to variables, put in pairs, carried by datatypes, etc. Functions can be passed as arguments to other functions. Functions can return functions. A function that takes or returns a function is itself called a higher-order function.

Here is a first example of a function that takes another function as an argument:

```ml
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:

```ml
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.

Note that we have not introduced any new language constructs here! We’re just using ones we already know in ways you might not have thought of. In particular, much earlier in the course, we decided to have all function be values, and to allow functions to take any values as arguments. As a consequence, functions can take functions as arguments.

2 Functions that take other functions as arguments

Let’s recall our first example of a function that takes another function as an argument:

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

What is the type of \( n\_times \)? It might be simpler at first to consider the type \( n\_times \) as used in \( x1 \) and \( x2 \): It takes 3 arguments, the first of which is itself a function that takes and returns an \( \text{int} \). So, as used in those places, \( n\_times \) has type \( (\text{int} \rightarrow \text{int}) \star \text{int} \star \text{int} \rightarrow \text{int} \). But for \( x3 \) we use \( n\_times \) as though it has type \( (\text{int list} \rightarrow \text{int list}) \star \text{int} \star \text{int list} \rightarrow \text{int list} \). So choosing either one of these types for \( n\_times \) would make it less useful because only some of our example uses would type-check. Therefore, the type actually inferred by ML for \( n\_times \) uses type variables to abstract away from the types. The actual type of \( n\_times \) is \((\text{'a} \rightarrow \text{'a}) \star \text{int} \star \text{'a} \rightarrow \text{'a} \), which means the third argument and result can be any type, but they have to be the same type, as do the argument and return type for the first argument.

Note that, although we just introduced type variables in this lecture, they are not responsible for enabling functions to be first class. Type variables (or parametric polymorphism) are technically a separate feature from first-class functions: there are functions that take functions and do not have polymorphic types—e.g.,

\[
\text{fun } \text{times until zero} (f,x) = \\
\text{if } x = 0 \text{ then } 0 \text{ else } 1 + \text{times until zero}(f, f x)
\]

which\[1\] has type \((\text{int} \rightarrow \text{int}) \star \text{int} \rightarrow \text{int} \)—and there are functions with polymorphic types that do not take functions—e.g., \( \text{length} : \text{'a list} \rightarrow \text{int} \). 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.

Once we define abstractions like \( n\_times \), 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 \( \text{triple_n times} \) using \( n\_times \):

\[
\text{fun } \text{triple} x = 3*x \\
\text{fun } \text{triple_n_times} (n,x) = n\_times(\text{triple},n,x)
\]

Higher-order functions are useful for our own data structures that we define with datatype bindings. Here we use \( \text{is_even} \) to see whether all the constants in an arithmetic expression are even. We could easily reuse \( \text{true of all constants} \) for any other property we wanted to check.

\[
\text{datatype exp} = \text{Constant of int} \mid \text{Negate of exp} \mid \text{Add of exp} \star \text{exp} \mid \text{Multiply of exp} \star \text{exp}
\]

\[
\text{fun } \text{true of all constants}(f,e) = \\
\text{case } e \text{ of } \\
\text{Constant } i \Rightarrow f i \\
| \text{Negate } e1 \Rightarrow \text{true of all constants}(f,e1) \\
| \text{Add}(e1,e2) \Rightarrow \text{true of all constants}(f,e1) \\
| \text{Multiply}(e1,e2) \Rightarrow \text{true of all constants}(f,e1) \\
| \text{andalso } \text{true of all constants}(f,e2) \\
| \text{andalso } \text{true of all constants}(f,e2)
\]

\[
\text{fun } \text{is_even} x = (x \mod 2 = 0) \\
\text{fun } \text{all_even } e = \text{true of all constants}(\text{is_even},e e)
\]

3 \ Anonymous functions\[2\]

There is no reason that a function, like \( \text{triple} \), that is passed to another function like \( n\_times \) needs to be defined at the top-level. As usual, it is better style to define such functions locally if they are needed only locally. So we could write:

\[1\]It would be better to make this function tail-recursive using an accumulator.
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 its body (the expression between in and end). To evaluate the body here, the run-time simply looks up triple in the dynamic environment, and the resulting function is the value that becomes 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!), 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 argument f.

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. So function-bindings are not just syntactic sugar for anonymous functions bound to a name.

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

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.

Neophytes are often tempted to use anonymous functions in a way that is particularly bad style. 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 tail of its argument. But there is already a function that does exactly that, namely tl. So just do this:

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

In general, there is no reason to write fn x => f x when we can just write f. (This error is analogous to the beginner’s habit of writing if x then true else false instead of x.)

4 Maps and filters

We now consider a massively useful higher-order function over lists:
fun map (f, lst) =  
    case lst of  
        [] => []  
      | fst::rest => (f fst)::(map(f, rest))  
(* map : ('a -> 'b) * 'a list -> 'b list *)

(The ML standard library provides a very similar function List.map, which we'll discuss in an upcoming lecture.) Our map function takes a function f and a list lst 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]) (* result: [5,9,13,17] *)  
val x5 = map (hd, [[1,2],[3,4],[5,6,7]]) (* result: [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, so long as the argument type of f is the element type of the list (they are both 'a). 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 instantiated with int list and 'b is instantiated with int.

Function map exemplifies an incredibly important idiom: it divides work into two parts: The map function knows how to traverse (i.e., visit all the elements of) a recursive data structure (in this example, a list). Function f passed to map knows how to process each element of the data structure (in this example, increment each number). You could imagine either of these tasks—traversing a complicated piece of data or doing some computation for each of the pieces—being vastly more complicated and best done by different developers, who shouldn’t make assumptions about the other task. This factoring of concerns is exactly what using map with helper function f does.

Here is another massively useful higher-order function for lists. It takes a helper 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 helper function returns true:

fun filter (f, lst) =  
    case lst of  
        [] => []  
      | fst::rest => if f fst  
        then fst::(filter (f, rest))  
        else filter (f, rest)  
(* filter : ('a -> bool) * 'a list -> 'a list *)

Suppose we’re dealing with a list of pairs, and that the second component of each pair has type int. Here’s a function that returns those list elements whose second component is even:

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

Notice how we use a pattern for the argument to our anonymous function.

5 Functions that return other functions

Functions can also return functions. Here is a silly example:

fun double_or_triple f =  
    if f 7  
    then fn x => 2 * x  
    else fn x => 3 * x
The if-expression makes the type of \( f \) clear and as usual the two branches of the if must have the same type, in this case \( \text{int} \rightarrow \text{int} \). So the type of \texttt{double_or_triple} is \( (\text{int} \rightarrow \text{bool}) \rightarrow (\text{int} \rightarrow \text{int}) \). However, the REPL will print the it as \( (\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{int} \). The two mean the same thing: the parentheses are unnecessary because \( \rightarrow \) is \textit{right associative}, meaning that it groups from the right to the left. For example, \( \text{t1} \rightarrow \text{t2} \rightarrow \text{t3} \rightarrow \text{t4} \) is the same as \( \text{t1} \rightarrow (\text{t2} \rightarrow (\text{t3} \rightarrow \text{t4})) \). Right associativity is less common than left associativity, but you might have seen it before. The Java and C assignment operators are right associative: \( \texttt{a = b = c;} \) means \( \texttt{a = (b = c);} \);