

---

# CS 421 – Spring 2007

## Lecture Notes Set 6:

### Recursion and Higher Order Functions

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 02-ocaml (slides 30-end) 04-05-hof (slides 1-12)

Made available: January 29, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Types of Recursion (slide 30)

Previously, we discussed a few types of recursion. Structural recursion, remember is when we take apart a data type a do the recursion on its elements, like a list. Then forward recursion, which can be used as a more specific type of structural recursion, is where you would do all the recursive calls first, and then work your way back performing the work that needs to be done.

### 2.1 Examples of Forward Recursion (slide 31)

We went over a couple examples of forward recursion, `double_up` and `poor_rev`. Remember, `double_up` takes a list and recurses to the very end of the list. Once it gets there, it matches the empty list to the empty list and returns to the last call. That next call takes the tail element and cons's it to itself, so the tail element appears twice. Then that call returns, and we work on the element next to the tail element. We do this until we get back to the head. So, the key here is that we did all the recursive calls first, and then performed the actions.

Another example is `poor_rev`. The very first thing we do here is recurse to the tail. We, again, match the empty list to the empty list and then return that call. The next call after that takes the first element of the list it sees and appends it to the end. Which in this case, since we recursed to the end of the list, is the tail element. So we append, to the end of the list, the tail element, then return, and then append what used to be the second to the tail of the list. This keeps going until the element that once was the head is now the tail. Note that `append` is a very expensive function. `Append` traverses the entire list to add an element to the end of the list. There is no tail pointer. So if you have a very long list, it may take some time to add a new element to the end.

### 2.2 Mapping Recursion (slides 32-33)

Another type of structural recursion we have seen is mapping recursion. Here, we take a structure and we want to apply some function to each element to alter their values. So, in the example we take the a list and we mapping each element to itself times two.

```
# let rec doubleList list = match list
  with [] → []
   | x::xs → 2 * x :: doubleList xs;;
val doubleList : int list → int list = <fun>
```

So if we performed `doubleList` on the list `[2;3;4]` we would get:

---

<sup>1</sup>© 2007, Share and Enjoy

```
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Note that this is also an example of forward recursion. The first thing it does is recurse to the end of the list. Then as it starts to return calls from the stack, it cons's on the doubled value of the element.

Another thing to note is that OCaml actually has a map function built in. It is a higher order function because it takes a function as one of its arguments. So to write doubleList with the built in function:

```
# let doubleList list =
  List.map (fun x → 2 * x) list;;
val doubleList : int list → int list = <fun>
```

What happens is you have this List.map function that takes in two arguments. The first is the function that you want mapped to your list. The second argument is the list. The function you supply in this case takes an element and returns the doubled value. But with the List.map function, you are able to traverse the list, apply the function to each element without using the more primitive recursion as was used in the previous doubleList example.

## 2.3 Folding Recursion (slides 34-35)

Then there is folding recursion. This type of recursion takes in a structure, but doesn't necessarily return something of the same type. So here in the example, we have the function multList, which appears to be multiplying each element of the list together to get one single value in the end.

```
# let rec multList list = match list
  with [] → 1
       | x::xs → x * multList xs;;
val multList : int list → int = <fun>
```

So, we again with forward recursion, we traverse to the end of the list, and when we find the empty list, we return 1. Then in the next call on the stack, we multiply that 1 with the element that is considered the head at that point and return that value. Then in the next call on the stack we do the multiplication again, and we do this until we get back to the actual head of the list, and finally return the actual result. So on the list [2;4;6] we get 48 because the multList function goes in and multiplies 1 \* 6, which equals 6, and then the 6 we just got times 4, which equals 24, and then that value by 2 which equals 48. Or, looking at order of operations-wise, it computes (2 \* (4 \* (6 \* 1))). Obviously, with multiplication, any order you multiply these numbers by will give the same result, but just makes sure you understand why it was written in that order.

```
# multList [2;4;6];;
- : int = 48
```

Like, map, OCaml has a built in function for fold\_right. To use it, you use the command List.fold\_right. So using this new function we can rewrite multList without using primitive recursion.

```
# let multList list =
  List.fold_right
    (fun x → fun p → x * p)
    list 1;;
```

Here, we call the built in function, List.fold\_right and we pass in the arguments. The first argument is a function. The next is the function itself and finally, we have the element corresponding to our base case. Now, the function looks a little strange, but here is how it works. Where it says fun x, the x is referring to the current element we are working on. In fun p, the p is referring to the result of the recursive case that we just finished. So, with the example list of [2;4;6]. The List.fold\_right will traverse the list until it gets to the base case, where it will then return 1 to the value p. Now, p holds the value of 1 and we have finally have a value for x, which is 6. We do the multiplication of 6 \* 1 and return that result to the previous call for the value p. So our new p is 6, or (6 \* 1), and our new x is 4. We do the math again and get 24, return that to p and have a new x of 2, do the math again and are now able to return the final result of 48.

Note that the way fold\_right works is the recursion is done on the tail. We will discuss the function fold\_left that is slightly more efficient because it is tail-recursive, but can't be used every where.

### 3 Running Times (slides 36-37)

Running time is probably familiar to you but it is something that is very important when programming, recursion especially. So, common running times are constant,  $O(1)$ , where it doesn't matter how much input you give a function, the function will take the same amount of time to finish. (We like that kind of function.) There is linear running time,  $O(n)$ , where the function takes as long as the input, so if you double the input, you double the running time. With quadratic running time,  $O(n^2)$ , if you double the input, then you end up quadrupling the running time. Then you get to really bad running time of exponential,  $O(2^n)$ . This is the case where if you were to simply increment the input, you end up doubling the running time. (We don't like this kind of running time.)

#### 3.1 Linear (slide 38)

So for lists, we can expect the functions to have linear running times. That is, for every element, we will usually, be doing some constant work. But with an input of  $n$ , we will be doing the constant work  $n$  times, because we are making only one recursive call per step, and that gives us our linear running time.

#### 3.2 Quadratic (slide 39)

However, there are functions on lists that take more than linear time. Remember the function `por_rev`. We used `append` in that function during every recursive call. So even though we were only making one recursive call per step, the work done during each step was not constant. The `append` function had to traverse the list during each call. Passing one element might be constant time, but here we can potentially be traversing  $n$  elements, so that is  $n$  constant operations. Then we will be doing the  $n$  operations  $n$  times (since we are making  $n$  recursive calls) and in the end, we get an  $n*n$ , or a quadratic running time. (Note that even though the running time is more specifically  $\frac{1}{2} * n^2$  we simply round to  $O(n^2)$ ).

#### 3.3 Exponential (slides 40-41)

Now, even worse than quadratic is exponential running time. Again, we don't like this type of running time. Here, even if a step of the recursion only takes constant time, but each step also makes two or more recursive calls then every one of those calls will make two or more recursive calls, and those will make two or more recursive calls. Note that it is very easy to write something that could have been linear, but can end up being exponential.

The naive fibonacci function has an exponential running time.

```
# let rec naiveFib n = match n
  with 0 → 0
      | 1 → 1
      | _ → naiveFib (n-1) + naiveFib (n-2);;
```

Notice that during each recursive step, two calls are made.

But to improve this, you can write a function that takes a pair instead of a single value. That way, the function gets the last value and the second to last value in the same function call and now there is only one recursive call to be made per step.

## 4 Optimization with Recursion

### 4.1 General Idea (slides 42-43)

Now, when making all these calls, we are building up a stack. But what if we know that one of those functions sole job is to return a value. The function isn't going to do any work on the value, just return it. For example, function `f` calls function `g`. Then function `g` calls function `h`. Now, when `h` is done with it's work, it has to return it's value to `g`. Function `g`, however isn't going to do any work. It is simply going to return the value to `f`. During all these function calls, the address of the function is saved on the function stack, so we know where to return the results. The question is, if `g` is only going to return the value from `h` to `f`, do we need to save the address of `g`?

The answer is no. The function `h` will return the value directly to `f`. This may sound like it isn't that special, so we skip one return, big deal, right?

Wrong, if we have a function that is tail recursive, that is does the work as before the recursion, then by the time we end to the base case, we have our final answer. So, instead of having a stack full of returns, we can make only one return and be done. We don't have to spend time popping function addresses off the stack.

## 4.2 Tail-Recursion (slides 44-47)

So tail-recursion makes use of this optimization. All the work is done as you recurse. Generally, doing this will require the use of an accumulator, a place holder for your partial value that you use to hold the result so far.

Here is an example of tail-recursion.

```
# let rec rev_aux list revlist =  
  match list with [ ] → revlist  
  | x :: xs → rev_aux xs (x::revlist);;
```

As arguments, you pass in the list and a place holder, `revlist`, for the list that you're going to create, the reversed list. So, our matching function, if it sees an empty list we return the place holder list, that is, the list that we created. You might be thinking, wait, isn't that empty? No, because we are adding elements to the place holder list as we recurse, by the time we see the empty list, we have created the reversed list we were looking for. We return that to our original call (because we just learned that the OCaml optimization doesn't require the intermediate returns) and then we are done. To make this reversed list, we look at the recursive step. There, we cons the head onto our place holder list, and then call the function on the tail list. So, as we recurse we grow `revlist`.

The running time for this is linear. The reason is because we don't have to traverse an entire list every time we cons an element. So we are doing a constant operation,  $n$  times which gives us  $O(n)$  running time, as opposed to the quadratic running time of `poor_rev` that we saw earlier.

Here is how `poor_rev` and `rev_aux` work in comparison with each other.

```
poor_rev [1,2,3] =  
(poor_rev [2,3]) @ [1] =  
((poor_rev [3]) @ [2]) @ [1] =  
(((poor_rev [ ]) @ [3]) @ [2]) @ [1] =  
((( [ ] @ [3]) @ [2]) @ [1]) =  
([3] @ [2]) @ [1] =  
(3:: ([ ] @ [2])) @ [1] =  
[3,2] @ [1] =  
3 :: ([2] @ [1]) =  
3 :: (2:: ([ ] @ [1])) = [3, 2, 1]
```

With `poor_rev`, we take each element, one by one and append each element. But we don't actually perform the appending yet, we prepare to do it and instead, first do the recursion. So we prepare to append each element. Then, when we get to the empty list we append the original tail element there. Then we return and find the next element we are supposed to append, we traverse to the end of the current list (we look for the empty list), do the append and then return. Then we do that again. Finally, we end up with our reversed list.

Now, let's look at `rev_aux`.

```
rev [1,2,3] =  
rev_aux [1,2,3] [ ] =  
rev_aux [2,3] [1] =  
rev_aux [3] [2,1] =  
rev_aux [ ] [3,2,1] = [3,2,1]
```

Here, since we are using a place holder list, all we have to do is at each step, cons the head onto the place holder. We do this until we reach the empty list of the original list and then we find that our place holder has the list we are looking for and we can return that directly to the original function call because there is no more work left to do. Notice the first line calls `rev` while each preceding line calls `rev_aux`. The reason for this is because the user doesn't need to know that we need a place holder list. So, with `rev`, you just send in the list you want to reverse and `rev` calls `rev_aux` which sends in the empty place holder list needed.

## 5 Higher Order Functions

### 5.1 First-Class Types (new slide set: slides 2-3)

A type is considered a first class type if it can be passed as an argument, assigned a value, and returned as a value. An example of this is in C, there are scalars, pointers and structures. In C++, there are the same types as in C and C++ also includes classes.

One can tell what types of applications a language is suited for by looking at how data can be used. Languages that can treat functions as a first class data type is a great strength for applicative programming. Note that this does not affect what kind of programs or functions that you can write, but rather, what kind of functions are easier to write.

### 5.2 The higher order functions (slide 4)

Functions that take a function as an argument and/or returns a function as a result are considered higher order functions. Let's look at an example.

```
# let compose f g = fun x → f (g x);;
val compose : ('a → 'b) → ('c → 'a) → 'c → 'b = <fun>
```

Looking at the type first, it takes a function from some type 'a to 'b and then takes a function from some type 'c back to 'a. It will return a function from some type 'c to 'b.

In terms of the compose function itself. Function g is going to take a value of some type 'c (the 'c which is alone in the type). Then, with function g taking in 'c it is going to return something of type 'a (this is the 'c → 'a part. This is then taken by function f and it will return something of 'b (which is the 'a → 'b part) and then ends up return the result of type 'b (the 'b at the end of the type) from the whole function.

Note that the 'c → 'b does not need to be wrapped in parentheses. The ('a → 'b) and ('c → 'a) do, however, because they are like mini functions of a larger whole.

### 5.3 Examples (slides 5-10)

If the function plus\_two has the following type:

```
# plus_two;;
- : int → int = <fun>
```

What does the type of compose plus\_two plus\_two look like?

```
- : int → int = <fun>
```

This function will take an integer add two to that, and then add two to that result and return. So we are simply taking one integer and turning it into another integer.

What about compose plus\_two?

```
- : ('_a → int) → '_a → int = <fun>
```

Here, we take in something of type '\_a and turn that into an integer. '\_a is basically saying that it is waiting for it's other functional argument, but it's waiting to be bound to something, the function g, what ever that may be. And then, after that, it will take that value and turn it into an integer. So overall, the function is taking something from what ever function g gives it, the '\_a and returns an integer.

Recall the function thrice. It takes in a function and an initial value and then will apply that function three times, starting with the initial value x. It takes something of type 'a and returns something of type 'a.

```
# let thrice f x = f (f (f x));;
val thrice : ('a → 'a) → 'a → 'a = <fun>
```

So, how can you write thrice with compose? Well, you compose f and f together. That will return something of 'a and thing you compose that with another f. And then you can see that this new thrice will give you the same type as the old thrice.

```
# let thrice f = compose f (compose f f);;
val thrice : ('a → 'a) → 'a → 'a = <fun>
```

## 5.4 Even more examples (slides 11-12)

We can do many things with these higher order functions.

Say we write a function `flip` that takes a function and then the function takes two arguments. The `flip` function basically flips the two arguments.

```
# let flip f a b = f b a;;
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

Now, we have seen the `map` function before, but there is new notation here. The `(-)` is a way to abstractly refer to the minus operation. For example, `(-) 3 4` means `3 - 4`.

So the following function is going to map each element of the list to to the `(1 - element)` operation. So you first get if you were looking at the head of the list, you see `(-) 1 5` which translates to `1 - 5` which equals `-4`. Note that the `((-) 1)` is a partial application of the `(-)` operation. The operator has one of its arguments, but is waiting for its other argument. Partial application is also known as sectioning.

```
# map ((-) 1) [5;6;7];;
- : int list = [-4; -5; -6]
```

Now we try applying the `flip` function to the `(-)` operation. So, we will be flipping the arguments of the `(-)` operation. Again, looking at the head of the list, now instead of `(-) 1 5`, we are actually going to calculate `(-) 5 1`, or `5 - 1` which equals `4`.

```
# map (flip (-) 1) [5;6;7];;
- : int list = [4; 5; 6]
```

So, can we change the `(-)` operation. The answer is yes. If we redefine the `(-)` to `flip (-)`.

```
# let (-) = flip (-);;
val (-) : int -> int -> int = <fun>
```

So when we try `2 - 5`, and we expect to see `-3`, since we performed `flip` on `(-)` we actually get `3`.