
CS 421 – Spring 2007

Lecture Notes Set 8:

Higher Order Functions

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 04-05-hof (slides 21-29)

Made available: February 2, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Recursive Functions with Higher Order Operators

It will usually be to our advantage to lambda lift these recursive functions so we end up with the best polymorphic results. So here we are going to apply what we learned from the last class so we don't have to worry about the `_a` variable to get truly polymorphic results.

2.1 A couple examples from last time... (slide 21)

Recall the functions `sumlist` and `prodlis`.

```
# let rec sumlist list = match list with
  [] → 0 | x::xs → x + sumlist xs;;
val sumlist : int list → int = <fun>
# let rec prodlis list = match list with
  [] → 1 | x::xs → x * prodlis xs;;
val prodlis : int list → int = <fun>
```

These are both forward recursive functions. We will recurse down the list until you get to the end, then you will start to accumulate your value on the way back. Basically the only difference between these two functions are the operations used and that in turn means the base cases are different.

There are a few parts that you need to know, first what are we going to do with the recursive value (in `sumlist` we are going to add the head to it, then in `prodlis`, we are going to multiply it to the head). The next thing we need to know is what do you do when you get to the tail of the list (in `sumlist` we are going to return 0, then in `prodlis`, we are going to multiply return 1). If you know those components, then you know what the function is going to do. With those components, you then can implement the same functions with the fold functions.

In particular, `fold_right` is a specific kind of forward recursion called primitive recursion and you know when you see primitive recursion when you see the basic structure of the functions `sumlist` and `prodlis`, that is the recursive call is on the recursive components of your structural data type. In the code above, that would be `sumlist xs` and `prodlis xs`. We then combine the recursive part with the part that is not recursive, so in the code above, that would be the head of the list. Most forward recursive functions can be coerced into primitive recursion form.

Another example would be if we had a tree structure. The recursive call would be on the branches and if there was some information in the root of each subtree, you would be passing up the information that way.

¹© 2007, Share and Enjoy

Then, you would also need to know what happens in the terminal case. In the code above we are using a list, so there is only one terminal case of the empty list. In a tree, the terminal case would be leaves.

2.2 The Fold Functions (slide 22)

Now lets look at `fold_right` and `fold_left`.

```
# let rec fold_right f list b = match list
  with [] → b | (x :: xs) → f x (fold_right f xs b);;
val fold_right: (a → b → b) → a list → b → b = <fun>
```

`Fold_right` takes a function that tells you what work to do on the list, or rather, it tells you what function or operation to use to combine your non-recursive component of your data structure with your recursive component. Then it also takes the list that you want to recurse over. Lastly, you give it your base case, but the value wont hold only the base case, it will also start accumulating values as `fold_right` starts doing its work.

So if you wanted to use `fold_right` you would have to come up with the function that does the combining and then base value. Then you can see that the function puts things together basically like we did on the previous code.

So remember, `fold_right` keeps pushing function calls onto the stack. There will be as many function calls as there are elements in the list and only when we get to the tail do we start doing any work.

```
#let rec fold_left f a list = match list
  with [] → a | (x :: xs) → fold_left f (f a x) xs;;
val fold_left: (a → b → a) → a → b list → a = <fun>
```

`Fold_left` captures a different type of recursion. That type is iteration, that is, we do some work on the head of the list, and that accumulated value gets passed forward onto the next recursive call. When we get to the end of the list, we want to return that value. But to start this accumulation, we need a base value to start with. This is the value we used in our base case of the recursive functions earlier.

So when using `fold_left` on associative and commutative operations, we can usually see how we want to structure our function. However, if doing something like subtraction from a set value, you would want to iterate over the list and have that value be your starting value and then subtract from that.

Looking at the code, you can see that `fold_left` starts at by performing the argument function `f` on the head of the list and the accumulated value thus far and that new value becomes the accumulated value sent into the next iteration. So we keep sending the value forward and when we get to the tail, then we see that all we have to do is return that value.

Remember, here we are doing the work as we go and when we get to the end, that is when we can simply pop the stack of our final function call and return our value.

2.3 Putting them together (slide 23)

So we take the information we learned from looking at the recursive functions and what we learned about how `fold_left` and `fold_right` work and we can rewrite `sumlist` and `prodlist`.

Recall that we wrote `sumlist` and `prodlist` in a primitive recursive function manner. That means we can use `fold_right` to rewrite these functions.

```
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list → int = <fun>
```

With `sumlist`, we discussed that the function that pulls the non-recursive and recursive parts of the list together was the addition that was being performed. That is then our function. We know we are recursing over our list, so we send that in and then we know what our base case it too, from the base case in our original function. So this is saying that some list is a recursive list that we want to add the elements of starting with 0 on the right and moving back towards to the head.

```
# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list → int = <fun>
```

`Prodlist` is very similar, we discussed that the function that pulls the non-recursive and recursive parts of the list together was the multiplication that was being performed. That is then our function. We know we are recursing over our list, so we send that in and then we know what our base case is too, from the base case in our original function. So

this is saying that some list is a recursive list that we want to multiply the elements of starting with 1 on the right and moving back towards the head.

Using the fold functions you can decompose the process of recursion into the steps of simply coming up with a function that causes the accumulation and the starting case for the accumulation.

Note that since addition and multiplication are both associative and commutative then if trying to write the function with `fold_left` we actually will use the same arguments, just in a different order. (`Fold_left` takes in the base case before the list.)

Also note that `fold_left` is less expensive than `fold_right` since `fold_right` builds a stack and `fold_left` just returns the value at the end. And because you have to pop the entire stack you built for the `fold_right` version, you essentially have twice as many operations as `fold_left`.

2.4 Summary on converting to folding functions (slides 24-25)

Remember that you can write any primitive recursive function with `fold_right`. If you have structural primitive recursion you can write it with `fold_right` and as long as your structure is finite, you are guaranteed that the recursion will terminate. Similarly, you can write any tail-recursive function with `fold_left`.

So here is an example to sum everything up. First look at the code and see if you can tell what the base case, the operation, and the recursive call are.

```
# let rec append list1 list2 = match list1 with
  [] → list2 | x::xs → x :: append xs list2;;
val append : 'a list → 'a list → 'a list = <fun>
```

Well, the answers are \rightarrow base case = "list2", operation = "::" (the cons operation), and recursive call = "append xs list2" which tells that this is a forward recursive function (it does all the function calls to append, and then starts cons-ing the elements together. So, when using a fold function we get:

```
# let append list1 list2 =
  fold_right (fun x y → x :: y) list1 list2;;
val append : 'a list → 'a list → 'a list = <fun>
```

The forward recursion means we use `fold_right`. Next is the first argument of `fold_right`, which is the cons operation. The next argument is the input list, which we know is `list1`. And then the base case is the last argument for `fold_right`. So, we were able to easily rewrite `append` with a fold function by just knowing a few things.

3 Other Higher Order Functions

3.1 Complist (slide 26)

Let's say that you want to take a list of functions and apply each function, sequentially to some value. To do this, we have the function `complist`.

`Complist` is a function that takes a list of functions, `flist` and basically pulls out a function, one by one, from the list and then allows the function to be applied, one by one.

```
# let rec complist flist = match flist with
  [] → (fun x → x)
  | f::fs → compose f (complist fs);;
val complist : ('a → 'a) list → 'a → 'a = <fun>
```

So, we start out with the base case, where if you have the empty list, you do nothing to your argument. So the empty list matches to the identity function. What ever you wanted to apply on `x`, if `flist` is empty, you simply get `x` back. Otherwise, the function, being forward recursive, gets to the last element in the list and basically applies that last function on your input value. That value is returned and then you use the new value with the second to last function. This goes on and on until you get to the head of the list, you apply that function on the value and that value is returned. The type of `complist` is a list of functions goes from `'a` to `'a` and is applied to a value of type `'a` and returns a value of type `'a`.

Let's look closely at `complist`. We discussed the base case leading to the identity function. We know that the function is forward recursive, so the recursive part of the function is the '`complist fs`', that is, we are calling the function on the tail list. The non-recursive part is the head function of the list. Now, you may look at it and ask, well, we know that we have to do some operation on the non-recursive and recursive parts to put them together...but what is that operation? Well, it is the `compose` function that does that. It makes sure that the return value of one function becomes the input to the next one.

Here are some examples:

```
# complist [( - ) 1; ( * ) 3; plus_two];;
- : int → int = <fun>
```

Here, we have created a function, that will first (since it is forward recursive, we start from the tail element) add two, then will multiply the returned value by three, and that value will be subtracted from one. So we essentially have a function that does this: $(1 - (3 * (2 + x)))$. So we get a function that takes an integer to an integer.

```
# complist [( - ) 1; ( * ) 3; plus_two] 5;;
- : int = -20
```

Now we actually applied the `complist` to a value, 5. So, we have $(1 - (3 * (2 + x)))$, where $x = 5$. That turns into $(1 - (3 * (2 + 5))) = (1 - (3 * (7))) = (1 - (21)) = (-20)$. So -20 is returned. Notice that this would be different if done in a tail recursive manner. If we started from the beginning of the list, we would get $((((1 - x) * 3) + 2)$, where $x = 5$. That would give us, $((((1 - 5) * 3) + 2) = (((-4) * 3) + 2) = ((-12) + 2) = (-10)$. So, it is important to understand whether a function is forward or tail recursive.

Note that since lists require each element to have the same type, your functions have to take a type 'a' and return a type 'a'. You also can't have all the functions go from 'a' → 'b' either because if your functions all have the same type, then they all take something of type 'a'. However, since all your functions return 'b', then the next function can't be evaluated since 'b' is the wrong input type.

Also note that you can rewrite `complist` with `fold_right`, since `complist` is forward recursive. You know what the base case is, you know what list you are recursing over, and you know what operation you are using to combine the recursive and non recursive parts of the list. So, `complist` would then look like: `fold_right compose flist (fun x → x)`

3.2 Repeat (slide 27)

Now, what do we do if we want to do something repeatedly, like a for loop? Well, we do two different things. First, we have the function, `repeat`.

```
# let rec repeat n f x =
  match n with 0 → x | _ → f (repeat (n - 1) f x);;
val repeat : int → ('a → 'a) → 'a → 'a = <fun>
```

So, let's take the function apart. First, the arguments of the function are as follows. We first have the number of times you want to repeat the loop. Then you have the function that you want to apply n times. Finally, you have the value that you want to act like your accumulator.

Now, the function starts out by saying, if you want to repeat the loop 0 times, just return your input accumulator, since nothing will be accumulated, you get the same value back. In this case, anything else will have you repeat the function calls until you get to $n = 0$ where you can start performing the input function on the input value/accumulator. That returns to the last function call and the input function is then applied to that number. However, this is not exactly a forward recursive function. This is as close as you can get to forward recursion with the natural numbers. That is, you have to do work, the subtracting 1 from n , to get to your next function call. So `repeat` is for natural numbers what `fold_right` is for lists.

The next example is `iter`.

```
# let rec iter n f x =
  match n with 0 → x | _ → iter (n - 1) f (f x);;
val iter : int → ('a → 'a) → 'a → 'a = <fun>
```

Here, we have an extremely similar situation. The arguments are the same and the base case is the same. The difference comes in how we perform the recursive step. Before, in `repeat`, we recursed first and performed the function of the loop as we returned. In `iter`, we are going to send forward the return value of the function after we have already

applied it to our input value. So, when we get to our base case, we have already done all the work we are going to do and we can return from there. This sounds a lot like tail recursion and it is.

So, `iter` is to natural numbers, what `fold_left` is for lists.

3.3 Mapping (slides 28-29)

Take a look at this example.

```
# let rec inclist list = match list with [] → []
| x :: xs → (1 + x) :: inclist xs;;
val inclist : int list → int list = <fun>
```

The function takes in a list, and then for every element in the list, adds 1 to it.

Now, recall the map function:

```
# let rec map f list =
  match list
  with [] → []
| (h::t) → (f h) :: (map f t);;
val map : ('a → 'b) → 'a list → 'b list = <fun>
```

In the next lecture, the uses of map will be further discussed.