

---

# CS 421 – Spring 2007

## Lecture Notes Set 7: Higher Order Functions

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

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 04-05-hof (slides 13-)

Made available: January 31, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Some notes on the previous lecture...

### 2.1 The type `'_a`

`'_a` is a outside the scope of this course, but the main idea is there is a clash of two types when it comes to the type system. So you are writing code and there is a requirement that every value that you create must either gain its polymorphism from a `<fun>` bound variable or must be a polymorphic constant, like `unit`. It cannot gain its polymorphism from the application of a polymorphic function to a polymorphic value. So, if you take a function like `map` and you to apply polymorphic function then you have something left over that is polymorphic, for example a list, but now that list's type is not being determined by some `fun x` it's being determined by the application you just used. So we are left with this variable, or like in the example, a list, where we don't know what it is, but it's not allowed to be entirely polymorphic, that is, it can't take on any possible type, but there still may be some freedom. This is called monomorphic type. Then the first time you use it, it gets assigned a type and has to stay that way.

For example, you have a function that applies an operation to some values. And the first time around you use integers, then every time after, you have to use integers. You can't apply the function on strings or floats because its `'_a` has been bound to an actual type. If you lambda lifted the function, then you have your complete polymorphic function and you won't have to worry about this.

What is key here is that when you see that a function is using something of type `'_a`, the first time you apply that function, your function will be bound to that type forever and ever.

Here is a more clear example. Let's say we write a function:

```
# let f x = x;;  
val f : '_a → 'a
```

So we wrote the identity function, it will give back what ever we sent in and therefore will have the same type. And now we write another function.

```
# let g = map f;;  
val g : '_a list → '_a list
```

Then we apply `g` on a list of one element of type integer. So we are applying `g` to the list `[1]`. We end up with a different type for `g` now.

```
val g : int list → int list
```

If you tried to then use `g` on a list of strings OCaml will get mad because there is a conflict in types. However, if you rewrote `g` to look like:

---

<sup>1</sup>© 2007, Share and Enjoy

```
# let g l = map f l;
```

Then you would get the type:  
`val g : 'a list → 'a list`

And now `g` is fully polymorphic and there is no binding of the function `g` to the type `int`.

### 3 Lambda Lifting

When evaluating a term, you evaluate until you get to a value. If that value is a closure, you stop there. You don't evaluate the body of the closure. Doing this give you the ability to control the flow of operations because you can prevent something from occurring for a while by hiding it underneath a "fun x →...", also known as a lambda, or an abstraction.

#### 3.1 An introductory example (slides 13-14)

So, we write a function `add_two`.

```
# let add_two = (+) (print_string "test\n"; 2);;
```

Note that the `(+)` was described in the last lecture. It is simply an abstract way of referring to the addition operation.

```
test
val add_two : int → int = <fun>
```

So here, what is happening is you are trying to evaluate the function `add_two`. Along that path is the primitive function of `print_string`. So you see that `test` is printed out. Then the rest of the evaluation continues. We see the `2` becomes part of a partial application of the `+` operation, and that becomes a function. It is one that is waiting for an integer argument to finish the evaluation of `2 + something`. So we have an input of an integer that returns an integer.

But, what if...what if you wanted the printing of the word "test" to not happen until you get the second argument for the addition operation. We can do this, we can stop the evaluation of the body of the function by hiding it. To hide it, we add the lambda.

```
# let add2 = (* lambda lifted *)
  fun x → (+) (print_string "test\n"; 2) x;;
val add2 : int → int = <fun>
```

Now, when we compile, we see that `add2` is being bound to a function that takes an argument `x` and we think that we cannot evaluate any of it without that argument `x`. So even though we know it could be possible to evaluate the `print_string`, we don't because it immediately goes to a closure that is waiting for `x`.

Note that for `add_two`, we will never again see the word "test" print out, because it is not part of the closure of `add_two`. The second function, `add2`, on the other hand, every time you give it an argument for `x`, you will see "test" printed out.

To see this, look at the following examples.

```
# thrice add_two 5;;
- : int = 11
```

We apply `thrice` to `add_two` and supply `add_two` with the value `5`, and we get our final value of `11`. That's it, it was as simple as that.

```
# thrice add2 5;;
test
test
test
- : int = 11
```

Now notice the difference after applying `thrice` to `add2` and supplying the same value of `5`. For every time the function `add2` is evaluated, we see the word "test" print to the screen. Then after the iterations and we have our result, we output that result.

There is the issue of which one is better...well that depends on what you are doing. If you will have numerous iterations of a function and want to watch as the iterations happen, then lambda lifting is the way to go. If you just

want your final result, most likely, you should choose the original format. So, now you can see that you can control how often things are done by how you write your code.

## 4 Lambda Lifting with Unknown Types (slides 15-17)

In the previous lecture we discussed the function `compose`. Here is the function with one of its arguments supplied.

```
# let f1 = compose plus_two;;  
val f1 : ('a → int) → 'a → int = <fun>
```

Notice now that the function has gone from polymorphic to a monomorphic one. This means that the next time you use `f1`, you have bound `'a` to a type.

```
# let f2 = fun g → compose plus_two g;;  
val f2 : ('a → int) → 'a → int = <fun>
```

In function `f2`, we add the lambda lifting and by that we don't even look at the `compose plus_two`. That way, the argument `g` can be anything we want it to be, without restrictions.

Note that there is a way to keep polymorphism and still have the side effects, like printing the word "test", only happen at declaration time. This involves using the `let` keyword and will be discussed later.

Now, when looking at function `f1`, and we give it the second argument of `plus_two`, we see that the function now takes an integer and returns an integer.

```
# f1 plus_two;;  
- : int → int = <fun>
```

If we tried to use `f1` with `List.length` now, we would get an error because `List.length` takes in a list while the function `f1` is expecting something that takes in an integer.

```
# f1 List.length;;
```

However if we did this with `f2`, everything will turn out to be fine. There won't be any errors.

```
# f2 plus_two;;  
- : int → int = <fun>  
# f2 List.length;;  
- : 'a list → int = <fun>
```

## 5 Revisit: Curried vs Uncurried (slide 18)

A few lectures ago, we discussed the difference between a function that takes in a collection of arguments and actually performs the function on one argument that returns a function and performs on the second argument, etc (or in other words, creates partial applications of functions) and a function that takes in one argument, like a tuple, that holds all the needed data. We learned that the former is known as curried and the latter is known as uncurried.

### 5.1 The functions (slide 19)

Here we can write functions that do the conversion from uncurried to curried and vice versa. The first function takes an uncurried function and makes it curried. That can be seen in the type. We have a function that takes in `'a * 'b`, or a pair and returns something of type `'c` and we turn that into something that takes in an argument of `'a` and then an argument of `'b` and returns something of type `'c`.

```
# let curry f x y = f (x,y);;  
val curry : ('a * 'b → 'c) → 'a → 'b → 'c = <fun>
```

The next example does the opposite. It turns a curried function into an uncurried one. Looking at the type, we take in a function that takes something of type `'a` and then something of `'b` and returns something of `'c` and turns that into a function that takes something of `'a * 'b`, or a tuple, and returns something of type `'c`.

```
# let uncurry f (x,y) = f x y;;  
val uncurry : ('a → 'b → 'c) → 'a * 'b → 'c = <fun>
```

So you can easily switch between the two forms by using these functions. If you don't plan on using partial applications of functions, then maybe uncurried is the way to go for that function. But if you ever have a case that you wish the function was curried, you can switch back.

## 5.2 Some examples (slide 20)

The first thing we start off with is checking the way the type works for the plus operation.

```
# (+);;  
- : int → int → int = <fun>
```

Here, we see that the plus operation is curried, in that it takes one argument and then can wait for another. So we can write the uncurried version of the operation.

```
# let plus = uncurry (+);;  
val plus : int * int → int = <fun>
```

Now, we have to supply both arguments for plus to work, whereas, the operator itself allows for partial application.

## 6 Folding

In structural recursion, there are three types of recursion that are special cases of general recursion that are worth noting. One is forward recursion, where you recurse to the base case and then work your way back out. Then there is tail recursion. Here you do the work first and then do the recursive call. Then, we also have the idea of folding.

### 6.1 A couple examples (slide 21)

So here are a couple examples for forward recursion.

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

This first example adds up all the values of a list. It first recurses to the empty list, and then adds the element of that is considered the head of the list with the accumulation thus far and returns it.

```
# sumlist [2;3;4];;  
- : int = 9
```

So the sample list, [2;3;4] gives  $4 + 3 + 2 = 9$

In the next example we do the same as sumlist, except we are multiplying instead of adding.

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

So on the sample list of [2;3;4] we get:

```
# prodlist [2;3;4];;  
- : int = 24
```

When you have a recursive function that works on a finite structure and that the operation is one that is finite, then you know that your function will terminate.

### 6.2 Folding Functions (slides 22-23)

Now we look at both `fold_left` and `fold_right`. `fold_left` is exactly the idea of iterating over a list. Anything you can calculate by iteration from left to right, you can calculate with `fold_left`. Note that `fold_left` itself is tail recursive.

`fold_right` is similar to the mathematical idea of primitive recursion. Without going in depth, let us just leave this at, anything you can do with primitive recursion can be done with `fold_right`. It is relatively powerful and still guaranteed to terminate.

So when using either of these functions, you have to send it the function that tells how to figure out the value. Then there is the place holder, where the accumulation will be stored and lastly, the structure you are folding over.

Next time, we will discuss the conversion of recursive functions to `fold_left` and `fold_right`.