

CS421 Lecture 4: Higher Order Functions¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

June 5, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

- 1 First Class Values
- 2 Higher Order Functions
- 3 Reverse, Folds, and Maps
- 4 Miscellaneous HOFs
- 5 Sample Problems

Objectives

Higher Order Functions (HOFs) are fundamental to the design and use of functional languages. Later, we will study the Lambda Calculus, which formalizes the idea of computing with functions. The purpose of this lecture is to give you an introduction to the creation and use of higher order functions.

As a result of this lecture (plus MP2), you should. . .

- know how to create HOFs
- learn how to understand complex combinations of HOFs
- understand relationships between different types of HOFs

First Class Types

A type is first class if it can be

- Passed as an argument
- Assigned as a value
- Returned as a result

Examples:

- C: scalars, pointers, structures
- C++: same as C, plus objects
- Scheme, LISP: scalars, lists (s-expressions), functions
- ML: same as Scheme, plus user defined data types

First Class Types – Key Point

- The kind of data that can be manipulated well in a language largely determines for which applications the language is well suited
- The ability to treat functions as data is one of the strengths of applicative programming languages

Higher Order Functions

A function is higher-order if it takes a function as an argument or returns one as a result

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

The type $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'a) \rightarrow 'c \rightarrow 'b$ is a higher order type because of

- $('a \rightarrow 'b)$ – the first argument is a function
- and $('c \rightarrow 'a)$ – the second argument is a function
- and $'c \rightarrow 'b$ – the function returns a function

Higher Order Functions

What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

Higher Order Functions

What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

Higher Order Functions

What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

```
1 - : int -> int = <fun>
```

Higher Order Functions

What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

```
1 - : int -> int = <fun>
```

```
1 # compose plus_two;;
```

Higher Order Functions

What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

```
1 - : int -> int = <fun>
```

```
1 # compose plus_two;;
```

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

Higher Order Functions

What do the following functions do?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

Higher Order Functions

What do the following functions do?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;  
2 - : int -> int = <fun>
```

Higher Order Functions

What do the following functions do?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two;;  
2 - : ('_a -> int) -> '_a -> int = <fun>
```

Thrice

Recall our definition:

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

Thrice

Recall our definition:

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

How would you write thrice with compose?

Thrice

Recall our definition:

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

How would you write thrice with compose?

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

Curried vs Uncurried

```
1 # let add_three x y z = x + y + z;;  
2 val add_three : int -> int -> int -> int = <fun>
```

How does this differ from:

```
1 # let add_triple (u,v,w) = u + v + w;;  
2 val add_triple : int * int * int -> int = <fun>
```

Curried vs Uncurried

```
1 # let add_three x y z = x + y + z;;  
2 val add_three : int -> int -> int -> int = <fun>
```

How does this differ from:

```
1 # let add_triple (u,v,w) = u + v + w;;  
2 val add_triple : int * int * int -> int = <fun>
```

`add_three` is *curried*, while `add_triple` is *uncurried*

Partial Application

```
1 # (+);;
2 - : int -> int -> int = <fun>
3 # (+) 2 3;;
4 - : int = 5
5 # let plus_two = (+) 2;;
6 val plus_two : int -> int = <fun>
7 # plus_two 7;;
8 - : int = 9
```

Partial application with operators is also called *sectioning*.

Lambda Lifting

Lambda lifting is a transformation to eliminate *free variables* from a function by adding them as *function parameters*

```
1 # let add_two = (+) (print_string "test"; 2);;
2 test
3 val add_two : int -> int = <fun>
4
5 # let add2 =      (* lambda lifted *)
6     fun x -> (+) (print_string "test"; 2) x;;
7 val add2 : int -> int = <fun>
```

You must remember the rules for evaluation when you use partial application. Can you see the difference between these two?

Lambda Lifting

Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

```
1 # thrice add_two 5;;  
2 - : int = 11  
3 # thrice add2 5;;  
4 test  
5 test  
6 test  
7 - : int = 11
```

Curry and Uncurry

```
1 # (+);;  
2 - : int -> int -> int = <fun>  
3  
4 # let curry f x y = f (x,y);;  
5 val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>  
6  
7 # let uncurry f (x,y) = f x y;;  
8 val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Remember these!

Curry and Uncurry

```
1 # let plus = uncurry (+);;  
2 val plus : int * int -> int = <fun>  
3  
4 # plus (3,4);;  
5 - : int = 7  
6  
7 # curry plus 3 4;;  
8 - : int = 7
```

Reversing Arguments

```
1 # let flip f a b = f b a;;
2 val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
3 # map ((-) 1) [5;6;7];;
4 - : int list = [-4; -5; -6]
5 # map (flip (-) 1) [5;6;7];;
6 - : int list = [4; 5; 6]
7 # let (-) = flip (-);;
8 val (-) : int -> int -> int = <fun>
9 # 2 - 5;;
10 - : int = 3
```

Folding Functions over Lists

How are the following functions similar?

```
1 # let rec sumlist list = match list with
2   [ ] -> 0 | x::xs -> x + sumlist xs;;
3 val sumlist : int list -> int = <fun>
4 # sumlist [2;3;4];;
5 - : int = 9
```

```
1 # let rec prodlist list = match list with
2   [ ] -> 1 | x::xs -> x * prodlist xs;;
3 val prodlist : int list -> int = <fun>
4 # prodlist [2;3;4];;
5 - : int = 24
```

Folding

```

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

$$\text{fold_left } f \ a \ [x_1; x_2; \dots; x_n] = f((f (f \ a \ x_1) \ x_2))x_n$$

```

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

$$\text{fold_right } f \ [x_1; x_2; \dots; x_n] \ b = f \ x_1(f \ x_2((f \ x_n b)))$$

Folding

```
1 # let sumlist list = fold_right (+) list 0;;  
2 val sumlist : int list -> int = <fun>  
3 # sumlist [2;3;4];;  
4 - : int = 9  
5 # let prodlist list = fold_right ( * ) list 1;;  
6 val prodlist : int list -> int = <fun>  
7 # prodlist [2;3;4];;  
8 - : int = 24
```

Folding

- Can replace recursion by `fold_right` in any *primitive recursive* definition, meaning any definition that only recurses on the immediate subcomponents of a recursive data structure
- We've seen one recursive data structure – list – defined as either the empty list or a head element and a list (this is the recursive part)
- Recursion can be replaced by `fold_left` in any *tail recursive* definition

Recall: Fold Right

```
1 # let rec fold_right f l i =  
2   match l with  
3     [] -> i  
4     | (x :: xs) -> f x (fold_right f xs i);;  
5 val fold_right :  
6   ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Encoding Recursion with Fold

```

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

Operation Recursive Call Base Case

```

1 # let append list1 list2 =
2   fold_right (fun x y -> x :: y) list1 list2;;
3 val append : 'a list -> 'a list -> 'a list = <fun>
4 # append [1;2;3] [4;5;6];;
5 : int list = [1; 2; 3; 4; 5; 6]
  
```

Combining Lists of Functions

```

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

Why isn't the type more general, like `compose`?

```

1 # complist [( - ) 1; ( * ) 3; plus_two] ;;
2 - : int -> int = <fun>
3 # complist [( - ) 1; ( * ) 3; plus_two] 5;;
4 - : int = -20
  
```

Can you write this with `fold_right`?

Repeating n times

```

1 # let rec repeat n f x =
2   match n with 0 -> x | _ -> f (repeat (n - 1) f x);;
3 val repeat : int -> ('a -> 'a) -> 'a -> 'a = <fun>
4 # repeat 8 (fun x -> x * 2) 1;;
5 - : int = 256
6 # let rec iter n f x =
7   match n with 0 -> x | _ -> iter (n - 1) f (f x);;
8 val iter : int -> ('a -> 'a) -> 'a -> 'a = <fun>
9 # iter 8 (fun x -> x * 2) 1;;
10 - : int = 256
  
```

Which is more efficient?

Mapping

What do these functions have in common?

```

1 # let rec inclist list = match list with
2   | [ ] -> [ ]
3   | x :: xs -> (1 + x) :: inclist xs;;
4   val inclist : int list -> int list = <fun>
5 # inclist [2;3;4];;
6 - : int list = [3; 4; 5]
7 # let rec doublelist list = match list with
8   | [ ] -> [ ]
9   | x :: xs -> (2 * x) :: doublelist xs;;
10  val doublelist : int list -> int list = <fun>
11 # doublelist [2;3;4];;
12 - : int list = [4; 6; 8]
```

Map from Fold

```
1 # let map f list =  
2   fold_right (fun x y -> f x :: y) list [ ];;  
3 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
4 # map ((+)1) [1;2;3];;  
5 - : int list = [2; 3; 4]
```

Can you write `fold_right` (or `fold_left`) with just `map`? How, or why not?

Related Function: Zip

```
1 # let rec zip list1 list2 =  
2   match (list1,list2) with ([ ], _) -> []  
3   | (_, [ ]) -> []  
4   | (x::xs, y::ys) -> (x,y)::zip xs ys;;  
5   val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>  
6   # zip [1;2;3] [4;5;6];;  
7   - : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Zipwith

```
1 # let rec zipwith f list1 list2 =
2   match (list1,list2) with ([ ], _) -> []
3   | (_, [ ]) -> []
4   | (x::xs, y::ys) -> f x y ::zipwith f xs ys;;
5   val zipwith : ('a -> 'b -> 'c) ->
6     'a list -> 'b list -> 'c list = <fun>
7   # zipwith (+) [1;2;3] [4;5;6];;
8   - : int list = [5; 7; 9]
9   # zipwith (fun x y -> (x,y)) [1;2;3] [4;5;6];;
10  : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Zip from Zipwith

How do you write zip from zipwith with no explicit recursion?

Zip from Zipwith

How do you write zip from zipwith with no explicit recursion?

```
1 # let zip = zipwith (fun x y -> (x,y));;  
2 val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>  
3 # zip [1;2;3] [4;5;6];;  
4 - : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Sample problems

- Write a function `flipuc` that flips the arguments to an uncurried function, using just `curry`, `flip` and `uncurry`
- Write a function that has type $('a \rightarrow 'b) \rightarrow 'a * 'c \rightarrow 'b$
- Use `fold_right` to write a function that takes a list and returns it.
- Use `fold_right` to write a function to remove all negative elements from a list

Problem 1

Write a function `flipuc` that flips the arguments to an uncurried function, using just `curry`, `flip` and `uncurry`

```
1 # let flipuc f = uncurry (flip (curry f));;
2 val flipuc : ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>
3 # let cons (x,xs) = x::xs;;
4 val cons : 'a * 'a list -> 'a list = <fun>
5 # let snoc = flipuc cons;;
6 val snoc : '_a list * '_a -> '_a list = <fun>
7 # snoc(snoc ([1],2),3);;
8 - : int list = [3; 2; 1]
```

Problem 2

Write a function that has type $('a \rightarrow 'b) \rightarrow 'a * 'c \rightarrow 'b$

```
1 # let app_fst f (a,b) = f a;;
2 val app_fst : ('a -> 'b) -> 'a * 'c -> 'b = <fun>
3 # app_fst ((+) 1) (3, 7);;
4 - : int = 4
5 # app_fst ((+) 1) (4, "hi");;
6 - : int = 5
```

Problem 3

Use `fold_right` to write a function that takes a list and returns it.

```
1 # let listId list =  
2   fold_right (fun x xs -> x::xs) list [];;  
3 val listId : 'a list -> 'a list = <fun>  
4 # listId [1;2;3];;  
5 - : int list = [1; 2; 3]
```

Problem 4

Use `fold_right` to write a function to remove all negative elements from a list

```
1 # let gezero list =  
2   fold_right  
3   (fun x xs -> if x >= 0 then x::xs else xs)  
4   list [ ];;  
5 val gezero : int list -> int list = <fun>  
6 # gezero [1;0;3;-5;7;-2];;  
7 - : int list = [1; 0; 3; 7]
```