

CS421 Lecture 4 Supplement: Reverse, Folds, and Maps¹

Sameer Sundresh
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

May 31, 2006

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

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>
```

```
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?