CS 320: Concepts of Programming Languages

Lecture 9: Folds, Folds, Folds

Ankush Das

Nathan Mull

Oct 1, 2024

Function of the Day

- Today, we will talk about one function and one function only: folds
- This function is very similar to the other higher-order functions
- It abstracts a very common pattern on lists
- Pattern: *combine the elements of a list together*
- Two Versions: combining elements *left to right* and combining elements *right to left*

Let's Observe The Pattern!


```
'a list -> int
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

```
int list -> int
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

```
int list -> int
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

- Recall size, sum, and prod functions for lists. What's the pattern?
- Return a base value when list is empty
- Recursive function on tail
- Combine the result of the recursive call to h using some operator

Abstracting This Pattern!

'a list -> int

```
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

int list -> int

```
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

int list -> int

```
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

Using fold_right

'a list -> int

```
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

int list -> int

```
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

int list -> int

```
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> int

```
let size l =
  fold_right                     l 0
```

int list -> int

```
let sum l =
  fold_right                    l 0
```

int list -> int

```
let prod l =
  fold_right                    l 1
```

Using fold_right

'a list -> int

```
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

int list -> int

```
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

int list -> int

```
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> int

```
let size l =
  fold_right (fun _h res -> 1 + res) l 0
```

int list -> int

```
let sum l =
  fold_right                             l 0
```

int list -> int

```
let prod l =
  fold_right                             l 1
```

Using fold_right

'a list -> int

```
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

int list -> int

```
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

int list -> int

```
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> int

```
let size l =
  fold_right (fun _h res -> 1 + res) l 0
```

int list -> int

```
let sum l =
  fold_right (fun h res -> h + res) l 0
```

int list -> int

```
let prod l =
  fold_right                          l 1
```

Using fold_right

'a list -> int

```
let rec size l =
  match l with
  | [] -> 0
  | _::t -> 1 + size t
```

int list -> int

```
let rec sum l =
  match l with
  | [] -> 0
  | h::t -> h + (sum t)
```

int list -> int

```
let rec prod l =
  match l with
  | [] -> 1
  | h::t -> h * (prod t)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> int

```
let size l =
  fold_right (fun _h res -> 1 + res) l 0
```

int list -> int

```
let sum l =
  fold_right (fun h res -> h + res) l 0
```

int list -> int

```
let prod l =
  fold_right (fun h res -> h * res) l 1
```

Sum using fold_right


```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

```
int list -> int
let sum l =
  fold_right (fun h res -> h + res) l 0
```

```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 +  (2 +  (3 +  (4 +  (5 +  (0 )))))
```

Prod using fold_right

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

int list -> int

```
let prod l =
  fold_right (fun h res -> h * res) l 1
```

```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 *  (2 *  (3 *  (4 *  (5 *  (1 )))))
```

Size using fold_right


```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

```
'a list -> int
let size l =
  fold_right (fun _h res -> 1 + res) l 0
```

```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 +  (1 +  (1 +  (1 +  (1 +  (0 )))))
```

Picture of fold_right


```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

More Examples


```
'a list -> 'a list
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

```
('a -> 'b) -> 'a list -> 'b list
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h) :: (map f t)
```

```
'a list list -> 'a list
let rec flatten l =
  match l with
  | [] -> []
  | h::t -> h @ flatten t
```

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

```
'a list -> 'a list
let reverse l =
  List.fold_right                          l []

('a -> 'b) -> 'a list -> 'b list
let map f l =
  List.fold_right                          l []

'a list list -> 'a list
let flatten l =
  List.fold_right                          l []
```

More Examples

'a list -> 'a list

```
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

('a -> 'b) -> 'a list -> 'b list

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h) :: (map f t)
```

'a list list -> 'a list

```
let rec flatten l =
  match l with
  | [] -> []
  | h::t -> h @ flatten t
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> 'a list

```
let reverse l =
  List.fold_right (fun h res -> res @ [h]) l []
```

('a -> 'b) -> 'a list -> 'b list

```
let map f l =
  List.fold_right                            l []
```

'a list list -> 'a list

```
let flatten l =
  List.fold_right                          l []
```

More Examples

'a list -> 'a list

```
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

('a -> 'b) -> 'a list -> 'b list

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h) :: (map f t)
```

'a list list -> 'a list

```
let rec flatten l =
  match l with
  | [] -> []
  | h::t -> h @ flatten t
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> 'a list

```
let reverse l =
  List.fold_right (fun h res -> res @ [h]) l []
```

('a -> 'b) -> 'a list -> 'b list

```
let map f l =
  List.fold_right (fun h res -> (f h)::res) l []
```

'a list list -> 'a list

```
let flatten l =
  List.fold_right                         l []
```

More Examples

'a list -> 'a list
```
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

('a -> 'b) -> 'a list -> 'b list
```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h) :: (map f t)
```

'a list list -> 'a list
```
let rec flatten l =
  match l with
  | [] -> []
  | h::t -> h @ flatten t
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```
let rec fold_right op l base =
  match l with
  | [] -> base
  | h::t -> op h (fold_right op t base)
```

'a list -> 'a list
```
let reverse l =
  List.fold_right (fun h res -> res @ [h]) l []
```

('a -> 'b) -> 'a list -> 'b list
```
let map f l =
  List.fold_right (fun h res -> (f h)::res) l []
```

'a list list -> 'a list
```
let flatten l =
  List.fold_right (fun h res -> h @ res) l []
```

Fold Left

- `fold_right` folds from right to left, i.e., starts evaluation from the end and returns when it reaches the beginning
- The dual function is called `fold_left`; this starts folding from the left and returns when it reaches the end
- Usually used for accumulating the result (e.g., in a tail recursive function)

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
let rec fold_left op acc l =
  match l with
  | [] -> acc
  | h::t -> fold_left op (op acc h) t
```

fold_right vs fold_left


```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold_right vs fold_left

```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

fold_right vs fold_left


```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

fold left

```
(((((acc op 1) op 2) op 3) op 4) op 5)
```

Examples of fold_left

('a -> bool) -> 'a list -> 'a list

```
let filter p l = fold_left (fun res h -> if p h then res @ [h] else res) [] l
```

'a list -> 'a list -> 'a list

```
let append l1 l2 = fold_left (fun res h -> res @ [h]) l1 l2
```

'a list -> 'a list

```
let reverse l = fold_left (fun res h -> h::res) [] l
```

int list -> int

```
let max l = fold_left (fun m h -> if h > m then h else m) 0 l
```

fold_right vs fold_left

- `fold_right` folds from right to left; used for right-associative operators
- `fold_left` folds from left to right; used for left-associative operators
- Available in the standard List library
  `List.fold_left`
  `List.fold_right`
- What is the difference?

Example: What if op is — ?


```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

Example: What if op is — ?


```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

Example: What if op is — ?


```
1 :: (2 :: (3 :: (4 :: (5 :: ([])))))
```

fold right

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

fold left

```
(((((acc op 1) op 2) op 3) op 4) op 5)
```

Example: fold_right vs fold_left


```
List.fold_right ( - ) [1; 2; 3; 4; 5] 0
= 1 -  (2 -  (3 -  (4 -  (5 -  0))))
```

```
= 3
```

```
List.fold_left ( - ) 0 [1; 2; 3; 4; 5]
= ((((0 - 1) - 2) - 3) - 4) - 5
```

```
= -15
```

- The order of arguments makes it intuitive!
- But please rely on the type checker to remind you of the types!

Associative Operations

- What are associative operations?

```
(a op b) op c = a op (b op c)
```

- For these operations, `List.fold_left` behaves exactly like `List.fold_right`

- Why? See for yourself

Associative Operations

- What are associative operations?

```
(a op b) op c = a op (b op c)
```

- For these operations, `List.fold_left` behaves exactly like `List.fold_right`
- Why? See for yourself

```
1 op (2 op (3 op (4 op (5 op (base)))))
```

```
(((((acc op 1) op 2) op 3) op 4) op 5)
```

Informal Recipe for Folding

- If the operator is associative, both perform the same
- Choose `List.fold_left`. Why? Because it is tail recursive
- Otherwise, see if you need to make the recursive call on the tail: Choose `List.fold_right`
- If you need to accumulate data from left to right: Choose `List.fold_left`
- Otherwise, don't use folding! You are probably better off doing a custom recursive function

Making fold_right Tail Recursive


```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let fold_right op l base =
  List.fold_left (fun x y -> op y x) base (List.rev l)
```

- Folding from right is the same as folding the reverse list from the left
- `List.fold_left` is tail recursive; hence this function is also tail recursive; just need to change the order of `op`
- Now, we have a tail recursive way of folding both from the left and from the right!

Short Circuiting

- Suppose we want to filter out && all elements of a boolean list
- Use fold:

```
bool list -> bool
let and_all l = List.fold_left (&&) true l
```

- We can also define a custom function

```
bool list -> bool
let and_all l =
  match l with
  | [] -> true
  | h::t -> if h then and_all t else false
```

- Which one's more efficient?

Short Circuiting

- Suppose we want to filter out && all elements of a boolean list
- Use fold:

```
bool list -> bool
let and_all l = List.fold_left (&&) true l
```

- We can also define a custom function

```
bool list -> bool
let and_all l =
  match l with
  | [] -> true
  | h::t -> if h then and_all t else false
```

- Which one's more efficient?

Conclusion

- Folds can also work on other data structures
- *Homework: How would you fold over a tree?*
- Read OCaml book 4.4, 4.5, 4.6
- Exercises in OCaml book 4.9
- Next lecture, we will cover our final abstraction: monads!