Higher-Order Functions (Pre Lecture)

Dr. Neil T. Dantam

CSCI-400, Colorado School of Mines

Spring 2023

Introduction

Higher-Order Functions

- **Higher-order function**: takes or returns another function
- The common ones:

map	transforms elements
filter	eliminates elements
fold	combines elements

- Useful abstractions in functional and non-functional settings

Outcomes

- Understand the operation of `map`, `filter`, and `fold`
- Apply `map`, `filter`, and `fold` to operate on collections

Outline

Higher-order functions

Definition: Higher-order function

A function that takes another function as an argument or returns another function as its result.

Example (Passing)

Function f(g,a)

1 **return** $g(42, a)$

Example (Returning)

Function f(a)

1 **function** $g(b)$ **is**
2 **return** $a + b$
3 **return** g

Counterexample

Function f(a,b)

1 **return** $a + b$

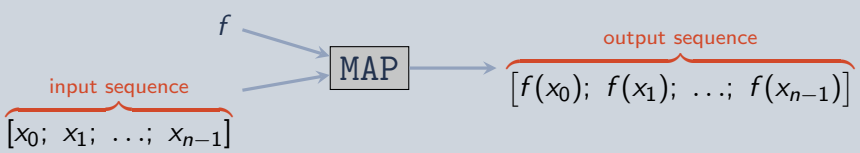
Map function

Definition (map)

Apply a function to every member of an input sequence, and collect the results into the output sequence.

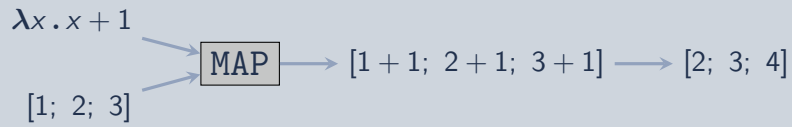
$$\texttt{map} \quad : \quad \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{X}^n}_{\text{input sequence}} \mapsto \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

Illustration



Example: Map

Increment every element by one



Not (¬) every element

Algorithm: Map function on lists

Functional Map

Function map(f, ℓ)

```
1 match ℓ with
2     case NIL → NIL
3     case cons(h, t) →
4         cons(f(h), map(f, t))
```

Imperative Map

Procedure map(f, ℓ)

```
1 Y ← make-sequence(|ℓ|)
2 i ← 0
3 while i < n do
4     Y[i] ← f(ℓ[i])
5     i ← i + 1
6 return Y
```

Exercise 1: Map

Function double-list(ℓ)

Filter

Definition (filter)

Apply a predicate (Boolean-valued function) to every member of an input sequence, and collect the only the input elements where the predicate is true.

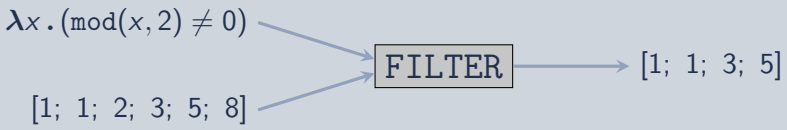
$$\texttt{filter} \;:\; \underbrace{(\mathbb{X} \mapsto \mathbb{B})}_{\text{function}} \times \underbrace{\mathbb{X}^m}_{\text{input sequence}} \mapsto \underbrace{\mathbb{X}^n}_{\text{output sequence}}$$

Illustration

Example: Filter

Evens

Odds



Algorithm: Filter

Filter Algorithm

Function filter(p, s)

```
1 match s with
2     case NIL → NIL
3     case cons(f, r) →
4         if p(f) then
5             cons(f, filter(p, r))
6         else
7             filter(p, r)
```

Exercise 2: Filter

Function remove-empty(ℓ)

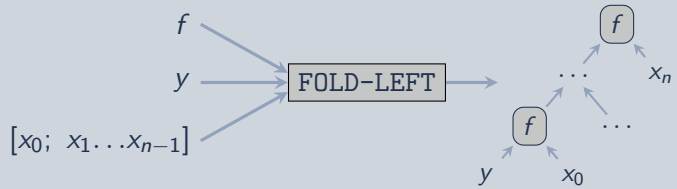
Fold-left

Definition (fold-left)

Apply a binary function to each member of a sequence and the prior result, starting from the left.

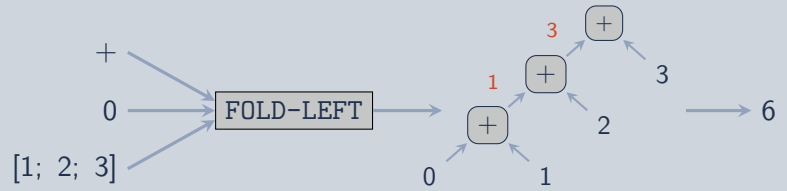
$$\text{fold-left} : \underbrace{(\mathbb{Y} \times \mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

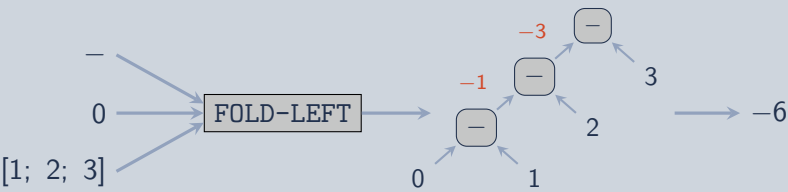


Example: Fold-left

Example (Addition)



Example (Subtraction)



Algorithm: Fold-left

Functional

Function fold-left(f, y, X)

```
1 match X with
2     case NIL → y
3     case cons(a, X′) →
4         fold-left(f, f(y, a), X′)
```

Imperative

Procedure fold-left(f, y, X)

```
1 i ← 0
2 while i < |X| do
3     y ← f(y, X)
4     i ← i + 1
5 return y
```

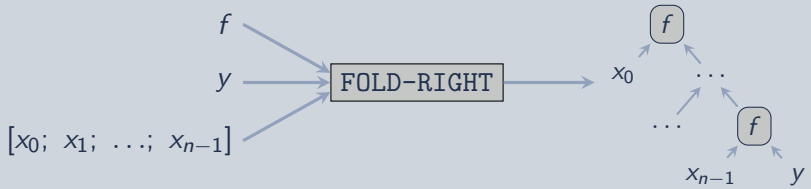
Fold-right

Definition (fold-right)

Apply a binary function to each member of a sequence and the prior result, starting from the right.

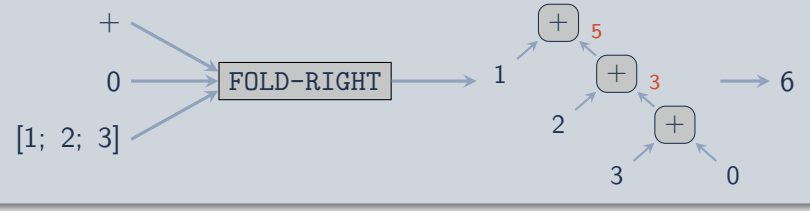
$$\text{fold-right} : \underbrace{(\mathbb{X} \times \mathbb{Y} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

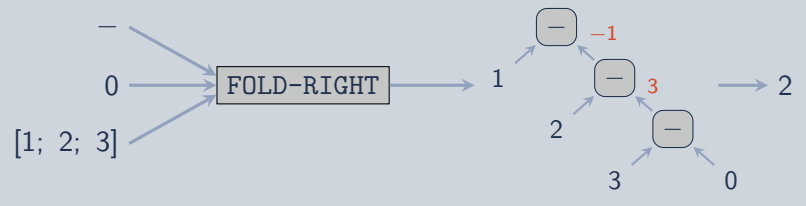


Example: Fold-right

Example (Addition)



Example (Subtraction)



Algorithm: Fold-right

Recursive

Function fold-right(f, y, X)

```
1 match X with
2     case NIL → y
3     case cons(a, X′) →
4         f(a, fold-right(f, y, X′))
```

Procedural

Procedure fold-right(f, y, X)

```
1 i ← |X| − 1
2 while i ≥ 0 do
3     y ← f(X_i, y)
4     i ← i − 1
5 return y
```

Outline

Generalized Operations

- `map`, `filter`, `fold` are often defined on lists
- Possible to define on other collections or data types:
  - Arrays
  - Trees
  - Sets
  - key-value pairs (aka "maps")

Fold is Fascinating

Loop

Procedure f(ℓ)

```
1 a ← init
2 while ¬empty(ℓ) do
3 |   f ← first(ℓ)
4 |   ℓ ← rest(ℓ)
5 |   a ← frob(a, f)
6 return a
```

Recursion

Function f(ℓ)

```
1 function h(ℓ, a) is
2 |   match ℓ with
3 |   |   case NIL → a
4 |   |   case cons(f, r) →
5 |   |   |   h(r, frob(a, f))
6 h(ℓ, init)
```

Fold

Function f(ℓ)

```
1 fold(frob, init, ℓ)
```

Compact, Abstract, General

Example: Length via Fold

Tail-Recursive

Function length(ℓ)

```
1 function h(ℓ, y) is
2     match ℓ with
3         case NIL → y
4         case cons(x, d) →
5             h(d, 1 + y)
6 h(ℓ, 0)
```

Fold

Function length(ℓ)

```
1 fold-left(λy x . 1 + y, 0, ℓ)
```

Example: Length via Fold

`length([1; 2; 3])`

fold-left

Function fold-left(f, y, ℓ)

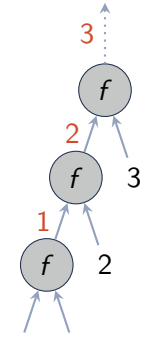
```
1 match ℓ with
2 | case NIL → y
3 | case cons(x, r) →
4 |   fold-left(f, f(y,x), r)
```

length

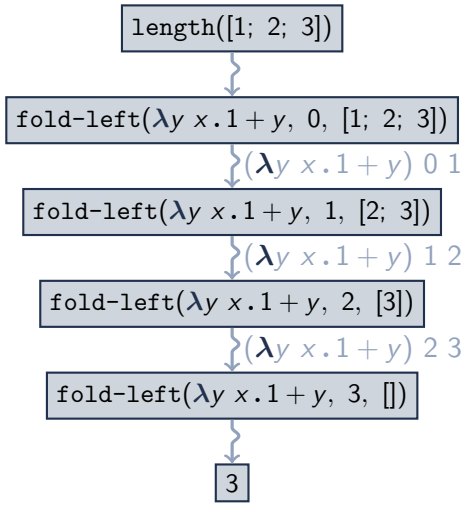
Function length(ℓ)

```
1 fold-left(λy x.1 + y, 0, ℓ)
```

Calls



Trace



Using Fold

Questions

- What is the "result" (y)?
- How do I use the current element (x) to update the the result?
- Do I need to start from the beginning or end of the list?

Fold-Left

result *current element*

function $h($ y , x) **is**

```
    /* Use x to update y        */
    ...
```

`fold-left`$(h,$ initial-value, $\ell)$

Fold-Right

current element *result*

function $h($ x , y) **is**

```
    /* Use x to update y        */
    ...
```

`fold-right`$(h,$ initial-value, $\ell)$

Exercise 3: Count via Fold

Tail-Recursive

Function count(ℓ, p)

```
1 function h(ℓ, a) is
2     match ℓ with
3         case NIL → a
4         case cons(f, r) →
5             if p(f) then
6                 h(r, 1 + a)
7             else
8                 h(r, a)
9 h(ℓ, 0)
```

Fold

Function count(ℓ, p)

Example: Copy via Fold

Recursive

Function copy(ℓ)

```
1 match ℓ with
2 |  case NIL → NIL
3 |  case cons(a, d) →
4 |  |  cons(a, copy(d))
```

Fold

Function copy(ℓ)

```
1 fold-right(cons, NIL, ℓ)
```

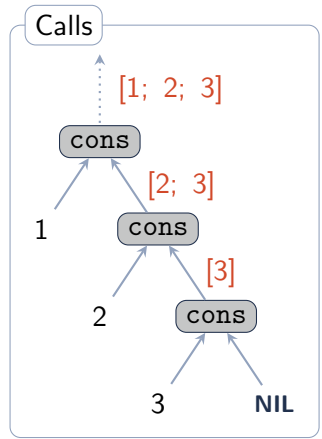
Example: Copy via Fold

`copy([1; 2; 3])`

copy

Function copy(ℓ)

```
1 fold-right(cons, NIL, ℓ)
```



Exercise 4: Append via Fold

Recursive

Function append(ℓ_a, ℓ_b)

```
1 match ℓ_a with
2 |  case NIL → ℓ_b
3 |  case cons(a, d) →
4 |  |  cons(a, append(d, ℓ_b))
```

Recursive

Function append(ℓ_a, ℓ_b)

Exercise 5: Reverse via Fold

Recursive

Function reverse(ℓ)

```
1 function h(ℓ, r) is
2     match ℓ with
3         case NIL → r
4         case cons(a, d) →
5             h(d, cons(a, r))
6 h(ℓ, NIL)
```

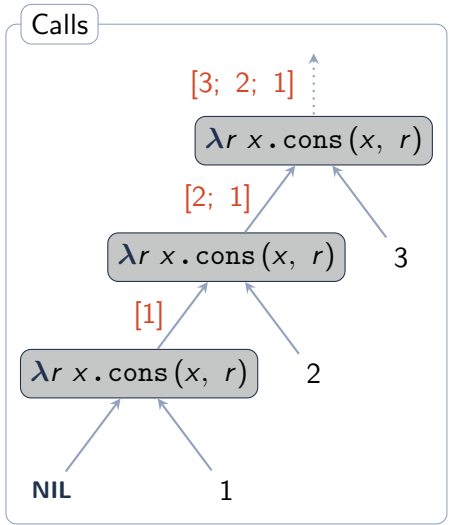
Fold

Function reverse(ℓ)

Exercise 5: Reverse via Fold

`reverse([1; 2; 3])`

reverse



Example: Map via Fold

Recursive Map

Function map(f, ℓ)

```
1 match ℓ with
2     case NIL → NIL
3     case cons(x, r) →
4         cons (f(x), map(f, r))
```

Copy via Fold

Function copy(ℓ)

```
1 fold-right(λx y . cons (x, y), NIL, ℓ)
```

Map via Fold

Function map(f, ℓ)

```
1 fold-right(λx y . cons (f(x), y), NIL, ℓ)
```

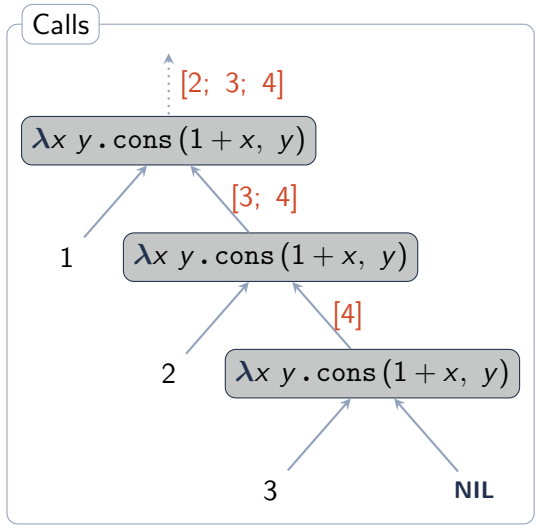
Example: Map via Fold

`map`$(\lambda x\,.\,1+x,\ [1;\ 2;\ 3])$

map

Function map$(f,\ \ell)$

1 **function** $h(x,\ y)$ **is**
2 ⌊ cons$(f(x),\ y)$
3 fold-right$(h,$ **NIL**$,\ \ell)$



Exercise 6: Filter via Fold

Recursive

Function filter(p, ℓ)

```
1 match ℓ with
2     case NIL → NIL
3     case cons(x, r) →
4         if p(x) then
5             cons(x, filter(p, r))
6         else
7             filter(p, r)
```

Fold

Function filter(p, ℓ)

Every

Definition (every)

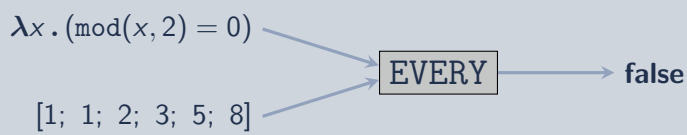
Return true when predicate evaluates to true on every member of the input sequence. Otherwise, return false.

$$\texttt{every} \quad : \quad \underbrace{(\mathbb{X} \mapsto \mathbb{B})}_{\text{function}} \quad \times \quad \underbrace{\mathbb{X}^n}_{\text{input sequence}} \quad \mapsto \mathbb{B}$$

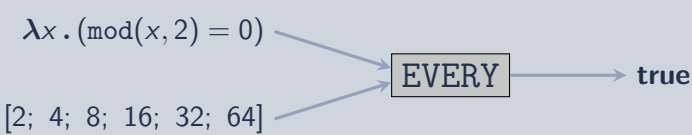
Illustration

Example: Every

Evens



Evens



Exercise 7: Every via Fold

Recursive

Function every(p, ℓ)

```
1 match ℓ with
2     case NIL → true
3     case cons(x, r) →
4         if p(x) then
5             every(p, r)
6         else
7             false
```

Fold

Function every(p, ℓ)

```
1 function h(y, x) is
2     if p(x) then y
3     else false
4 fold-left(h, true, ℓ)
```

What's a potential issue?

Efficiency Notes

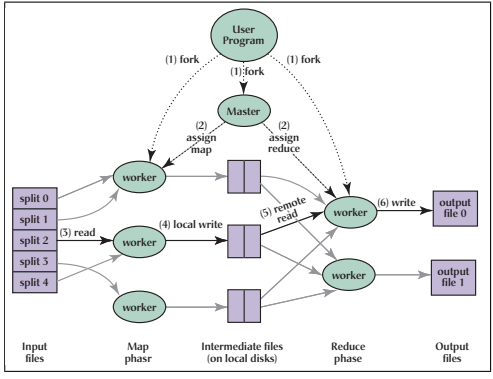
- Fold-left is tail recursive:
  any decent implementation on lists will use constant $O(1)$ space
- Fold-right is *not* tail recursive:
  an implementation on lists will (likely) use linear $O(|\ell|)$ stack space
- Folding over other structures (arrays, trees) may need different amounts of space.
- Fold must visit every element, so time is linear $O(|\ell|)$

Application: MapReduce

Function MapReduce(f,g,X)

1 $Y \leftarrow$ parallel-map(f, X)
2 **return** reduce(g, Y)

- Idea:
  - (parallel) map
  - (serial) reduce/fold
- Provides scalability, fault-tolerance
- Implementations:
  - Google MapReduce
  - Apache Hadoop



Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM. 2008.

Summary

- Common Higher-order Functions:

map	transforms elements
filter	eliminates elements
fold	combines elements

- Compact and Abstract:
  - Factors out control to operate over a collection
  - Per-element operation is independent of collection type

References

- Clarkson. Functional Programming in OCaml.
  - Ch 4 Higher-order programming