An Intuition for List Folds

LambdaConf, Boulder CO, 2018

Tony Morris

Brisbane

Queensland

Brisbane east coast

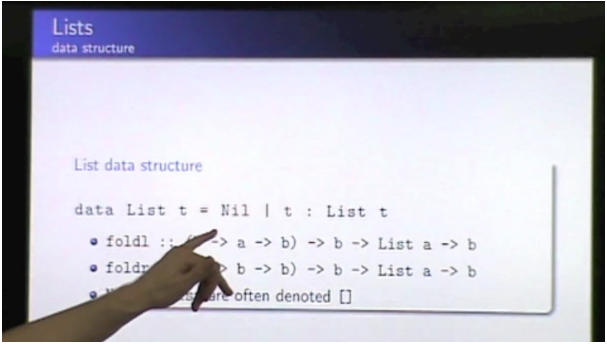
QFPL


https://qfpl.io
133%
Search
DATA 61
CSIRO
Home
Location
People
Projects
ADVANCE QUEENSLAND
Queensland Government
DATA 61
CSIRO
Queensland Functional Programming Lab

Intro

Explain List Folds to Yourself, April 2013

In April 2013, BFPG had a chat about list folds



Introduction

Today

- we are doing a similar thing, with differences
- aiming to beginners, who have only had cursory experiences with lists
- being very explicit about the utility of the developed intuition, and developing it further

Introduction

Today

- we are doing a similar thing, with differences
- aiming to beginners, who have only had cursory experiences with lists
- being very explicit about the utility of the developed intuition, and developing it further

Introduction

Today

- we are doing a similar thing, with differences
- aiming to beginners, who have only had cursory experiences with lists
- being very explicit about the utility of the developed intuition, and developing it further

Lists

What is a list?

What, exactly is a list?

Lists

a list is either

- a **Nil** construction, with no associated data
- a **Cons** construction, associated with one arbitrary value, and another list

*And **never, ever** anything else*

Lists

the shape of a list's construction is

- `Nil :: List a`
- `Cons :: a -> List a -> List a`

Lists

a list using C#

```
interface List<A>{}
class Nil<A> : List<A> {}
class Cons<A> : List<A> { A head; List<A> tail; }
```

*And some tricks (omitted) to enforce **never ever anything else***

Lists

a list using Haskell

```
data List a = Nil | Cons a (List a)
```

never ever anything else *is enforced in haskell*

Some examples of Lists

C#

```
new Cons<int>(12, new Nil<int>())
```

Haskell

```
Cons 12 Nil
```

printed

```
[12]
```

Some examples of Lists

C#

```
new Cons<char>('a', new Cons<char>('b', new Cons<char>('c', new Nil<char>())))
```

Haskell

```
Cons 'a' (Cons 'b' (Cons 'c' Nil))
```

printed

```
['a', 'b', 'c']
```

Some nomenclature

Naming Schmaming

- Sometimes you will see `Nil` denoted `[]`
- and/or `Cons` denoted `:` in an infix position
- like this `1:(2:(3:[]))`
- but this is the same data structure

Some nomenclature

Naming Schmaming

- Sometimes you will see `Nil` denoted `[]`
- and/or `Cons` denoted `:` in an infix position
- like this `1:(2:(3:[]))`
- but this is the same data structure

Some nomenclature

Naming Schmaming

- Sometimes you will see `Nil` denoted `[]`
- and/or `Cons` denoted `:` in an infix position
- like this `1:(2:(3:[]))`
- but this is the same data structure

Some nomenclature

Naming Schmaming

- Sometimes you will see `Nil` denoted `[]`
- and/or `Cons` denoted `:` in an infix position
- like this `1:(2:(3:[]))`
- but this is the same data structure

Lists

Ensure we all know what a list is

Folds

Left, Right, FileNotFound

- You may have heard of *right folds* and *left folds*
- Haskell: `foldr`, `foldl`
- Scala: `foldRight`, `foldLeft`
- C# (BCL): *no right fold*, `Aggregate` *(kind of)*
- C# (xsharpx): `FoldRight`, `FoldLeft`

Folds

Left, Right, FileNotFound

- You may have heard of *right folds* and *left folds*
- Haskell: `foldr`, `foldl`
- Scala: `foldRight`, `foldLeft`
- C# (BCL): *no right fold*, `Aggregate` *(kind of)*
- C# (xsharpx): `FoldRight`, `FoldLeft`

Folds

Developing intuition for folds

- When do I know to use a fold?
- When do I know which fold to use?
- What do the fold functions *actually do*?

Folds

Developing intuition for folds

- When do I know to use a fold?
- When do I know which fold to use?
- What do the fold functions *actually do*?

Folds

Developing intuition for folds

- When do I know to use a fold?
- When do I know which fold to use?
- What do the fold functions *actually do*?

Developing intuition for folds

There is much effort toward answering these questions

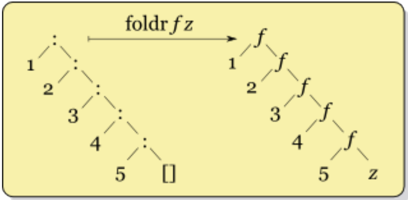


Figure: right fold diagram

Developing intuition for folds

There is much effort toward answering these questions

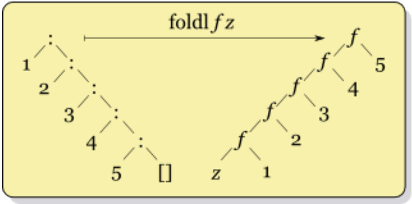


Figure: left fold diagram

Developing intuition for folds

and terse explanations

- the right fold does folding from the right and left fold, folding from the left
- choose the right fold when you need to work with an infinite list

Developing intuition for folds

and terse explanations

- the right fold does folding from the right and left fold, folding from the left
- choose the right fold when you need to work with an infinite list

Developing intuition for folds

Unfortunately

many of these explanations are misrepresentative, incomplete, or wrong

Goals

Goals for today

- Develop a robust and accurate description and intuition for each list fold function
- Ask and answer practical questions, given this intuition

Goals

First things first

In practice, the `foldl` and `foldr` functions are **very different**.

So let us think about and discuss each separately.

foldl

The `foldl` function accepts three values:

1. `f :: b -> a -> b`
2. `z :: b`
3. `list :: List a`

to get back a value of the type `b`.

```
foldl :: (b -> a -> b) -> b -> List a -> b
B FoldLeft<A, B>(Func<B, A, B>, B, List<A>)
```

foldl

?

How does `foldl` take three values to that return value?

foldl

all left folds are loops

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

foldl

all left folds are loops

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

foldl

refactor some loops

let's look at a real code example

switch to code example

foldl

all left folds are loops

Let's sum the integers of a list

foldl

sum the integers of a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

?

foldl

sum the integers of a list

```
\list ->
  var r = 0
  foreach(a in list)
    r = +(r, a)
  return r
```

Replace the values in the loop

foldl

sum the integers of a list

```
sum list = foldl (\r a -> (+) r a) 0 list
sum = foldl (+) 0
```

foldl

multiply the integers of a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

?

foldl

multiply the integers of a list

```
\list ->
  var r = 1
  foreach(a in list)
    r = *(r, a)
  return r
```

Replace the values in the loop

foldl

multiply the integers of a list

```
product list = foldl (\r a -> (*) r a) 1 list
product = foldl (*) 1
```

foldl

all left folds are loops

Let's reverse a list

foldl

reverse a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

?

foldl

reverse a list

```
\list ->
  var r = Nil
  foreach(a in list)
    r = flipCons(r, a)
  return r

flipCons = \r a -> Cons a r
```

Replace the values in the loop

foldl

reverse a list

```
reverse list = foldl (\r a -> Cons a r) Nil list
reverse = foldl (flip Cons) Nil
```

foldl

all left folds are loops

Let's compute the length of a list

foldl

length of a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

?

foldl

length of a list

```
\list ->
  var r = 0
  foreach(a in list)
    r = plus1(r, a)
  return r

plus1 = \r a -> r + 1
```

Replace the values in the loop

foldl

length of a list

```
length list = foldl (\r a -> r + 1) 0 list
length = foldl (const . (+1)) 0
```

foldl

refactoring, intuition

- a left fold is what you would write if I insisted you remove all duplication from your loops
- all left folds are exactly this loop
- **exactly**

foldl

refactoring, intuition

- a left fold is what you would write if I insisted you remove all duplication from your loops
- all left folds are exactly this loop
- exactly

foldl

refactoring, intuition

- a left fold is what you would write if I insisted you remove all duplication from your loops
- all left folds are exactly this loop
- **exactly**

foldl

some observations

- a left fold will **never** work on an infinite list
- a correct intuition for left folds is easy to build on existing programming knowledge (loop)

foldl

some observations

- a left fold will **never** work on an infinite list
- a correct intuition for left folds is easy to build on existing programming knowledge (loop)

Lists

Ensure we have developed intuition for left fold

foldl

The `foldr` function accepts three values:

1. `f :: a -> b -> b`
2. `z :: b`
3. `list :: List a`

to get back a value of the type `b`.

```
foldr :: (a -> b -> b) -> b -> List a -> b
B FoldRight<A, B>(Func<A, B, B>, B, List<A>)
```

foldl

?

How does `foldr` take three values to that return value?

foldr

constructor replacement

The `foldr` function performs **constructor replacement**.

The expression `foldr f z list` replaces in `list`:

- Every occurrence of `Cons (:)` with `f`.
- Any occurrence of `Nil []` with `z`[1].

[1] The `Nil` constructor may be absent —i.e. the list is an infinite list of `Cons`

foldr

constructor replacement?

- suppose `list = Cons A (Cons B (Cons C (Cons D Nil)))`
- the expression `foldr f z list`
- produces `f A (f B (f C (f D z)))`

foldr

right folds replace constructors

Let's multiply the integers of a list

foldr

multiply the integers of a list

Supposing

```
list = Cons 4 (Cons 5 (Cons 6 (Cons 7 Nil)))
```

foldr

multiply the integers of a list

Supposing

```
list = Cons 4 (Cons 5 (Cons 6 (Cons 7 Nil)))
```

?

foldr

multiply the integers of a list

- let `Cons` = `(*)`
- let `Nil` = `1`

foldr

multiply the integers of a list

Supposing

```
list = (*) 4 ((*) 5 ((*) 6 ((*) 7 1)))
```

```
product list = foldr (*) 1 list
product = foldr (*) 1
```

foldr

right folds replace constructors

Let's and `(&&)` the booleans of a list

foldr

and (&&) the booleans of a list

Supposing

```
list = Cons True (Cons True (Cons False (Cons True Nil)))
```

foldr

and (&&) the booleans of a list

Supposing

```
list = Cons True (Cons True (Cons False (Cons True Nil)))
```

?

foldr

and (&&) the booleans of a list

- let `Cons` = `(&&)`
- let `Nil` = `True`

foldr

and (&&) the booleans of a list

Supposing

```
list = (&&) True ((&&) True ((&&) False ((&&) True True)))
```

```
conjunct list = foldr (&&) True list
conjunct = foldr (&&) True
```

foldr

right folds replace constructors

Let's append two lists

foldr

append two lists

Supposing

```
list1 = Cons A (Cons B (Cons C (Cons D Nil)))
list2 = Cons E (Cons F (Cons G (Cons H Nil)))
```

foldr

append two lists

Supposing

```
list1 = Cons A (Cons B (Cons C (Cons D Nil)))
list2 = Cons E (Cons F (Cons G (Cons H Nil)))
```

?

foldr

append two lists

- let `Cons` = `Cons`
- let `Nil` = `list2`

foldr

append two lists

Supposing

```
list1 = Cons A (Cons B (Cons C (Cons D list2)))
list2 = Cons E (Cons F (Cons G (Cons H Nil)))
```

```
append list1 list2 = foldr Cons list2 list1
append = flip (foldr Cons)
```

foldr

right folds replace constructors

Let's map a function on a list

foldr

map a function (`f`) on a list

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

foldr

map a function (f) on a list

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

?

foldr

map a function (f) on a list

- let Cons = \x -> Cons (f x)
- let Nil = Nil

foldr

map a function (f) on a list

Supposing

```
consf x = Cons (f x)

list = consf A (consf B (consf C (consf D Nil)))
```

```
map f list = foldr (\x -> Cons (f x)) Nil list
map f = foldr (Cons . f) Nil
```

foldr

right folds replace constructors

Let's flatten a list of lists

foldr

flatten a list of lists

Supposing

```
list = Cons lista (Cons listb (Cons listc (Cons listd Nil)))
```

foldr

flatten a list of lists

Supposing

```
list = Cons lista (Cons listb (Cons listc (Cons listd Nil)))
```

?

foldr

flatten a list of lists

- let `Cons` = `append`
- let `Nil` = `Nil`

foldr

flatten a list of lists

Supposing

```
list = append lista (append listb (append listc (append listd Nil)))
```

```
flatten list = foldr append Nil list
flatten = foldr append Nil
```

foldr

right folds replace constructors

Let's filter a list on predicate

foldr

filter a list on predicate (p)

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

foldr

filter a list on predicate (p)

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

?

foldr

filter a list on predicate (p)

- let `Cons` `= \x -> if p x then Cons x else id`
- let `Nil` `= Nil`

foldr

filter a list on predicate (p)

Supposing

```
applyp x = if p x then Cons x else id

list = applyp A (applyp B (applyp C (applyp D Nil)))
```

```
filter p list = foldr (\x -> if p x then Cons x else id) Nil list
filter p = foldr (\x -> if p x then Cons x else id) Nil
filter p = foldr (\x -> bool id (Cons x) (p x)) Nil
filter p = foldr (bool id . Cons <*> p) Nil
```

foldr

right folds replace constructors

Let's get the head of a list, or default for no head

```
:: a -> List a -> a
```

foldr

the head of a list, or default for no head

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

foldr

the head of a list, or default for no head

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

?

foldr

the head of a list, or default for no head

- let `Cons = \x _ -> x`
- let `Nil  = thedefault`

foldr

the head of a list, or default for no head

Supposing

```
constant x _ = x

list = constant A (constant B (constant C (constant D thedefault)))
```

```
heador thedefault list = foldr constant thedefault list
heador thedefault = foldr constant thedefault
heador = foldr constant
```

foldr

right folds replace constructors

Let's sequence a list of effects `(f a)` and produce an effect (f) of list

`:: Monad f => List (f a) -> f (List a)`

foldr

list of effects (`f a`) to effect (`f`) of list

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

foldr

list of effects (`f a`) to effect (`f`) of list

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

?

foldr

list of effects `(f a)` to effect `(f)` of list

- let `Cons`
  `= \a b -> do { x <- a; y <- b; return (Cons x y) }`
- let `Nil` `= return Nil`

foldr

list of effects (f a) to effect (f) of list

Supposing

```
lift2cons a b = do { x <- a; y <- b; return (Cons a b)}

list = lift2cons A (lift2cons B (lift2cons C (lift2cons D return Nil)))
```

```
sequence list = foldr (lift2cons) (return Nil) list
sequence = foldr (lift2cons) (return Nil)
```

foldr

Observations

- `foldr` may work on an infinite list.
  - There is no *order* specified, however, there is associativity.
  - Depends on the strictness of the given function.
  - Replaces the `Nil` constructor *if it ever comes to exist*.
- The expression `foldr Cons Nil` leaves the list *unchanged*.
  - In other words, passing the list constructors to `foldr` produces an *identity* function.

foldr

Observations

- `foldr` may work on an infinite list.
  - There is no *order* specified, however, there is associativity.
  - Depends on the strictness of the given function.
  - Replaces the `Nil` constructor *if it ever comes to exist*.
- The expression `foldr Cons Nil` leaves the list *unchanged*.
  - In other words, passing the list constructors to `foldr` produces an *identity* function.

Summary

the key intuition

- left fold performs a *loop*, just like we are familiar with
- right fold performs *constructor replacement*

Summary

from this we derive some observations

- left fold will *never* work on an infinite list
- right fold *may* work on an infinite list
- These observations are independent of specific programming languages

Summary

from this we also solve problems

- `product = ...`
- `append = ...`
- `map = ...`
- `length = ...`
- ...

Summary

- intuitively, this is precisely what list folds do
- this intuition is **precise** and requires no footnotes

Summary

THE END