Explaining List Folds

An easy explanation of the fold-left and fold-right functions

Tony Morris

Brisbane Functional Programming Group, 23 April 2013

List data structure

```
data List t = Nil | t : List t
```

- `foldl :: (b -> a -> b) -> b -> List a -> b`
- `foldr :: (a -> b -> b) -> b -> List a -> b`
- `Nil` and `List` are often denoted `[]`

Examples of List values

- `1:2:3:Nil`
- `1:(2:(3:Nil))`
- `'x':'y':'z':Nil`
- `A:B:C:[]`

Folds
folding left and right

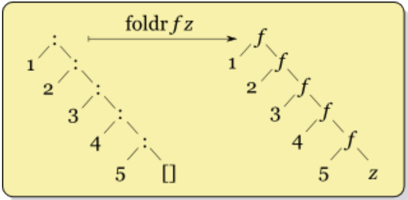
- We are going to be discussing the `foldl` and `foldr` functions on **cons lists**.
- In the Scala programming language, these are called `foldLeft` and `foldRight`.
- The C# programming language provides an approximation for `foldl` called `Aggregate`[1].
- Our discussion is language-independent and so applies equally to Haskell, Scala and more.

[1] there is no `foldr` equivalent as the structure is not a proper cons list

Folds
explanations

There are all types of explanations of list fold functions out there.

Fold Diagrams



Short, concise descriptions

- `foldl` applies a function to a list, associating to the left.
  - `\f z -> (f (f (f a z) b) c)`
- `foldr` applies a function to a list, associating to the right.
  - `\f z -> (f a (f b (f c z)))`

But then I am hit with more questions

- How does folding right start from the right but work on infinite lists?
- How do I recognise when it is appropriate to use a fold function?
- When do I choose to use one over the other?

Goals

Goals for today

- Develop a robust and accurate description for the list fold functions.
- Infer answers to practical questions from this description.
- Propose a tacit argument that you should use this description when discussing with others.

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 is a machine that requires three values:

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

It will give you back a value of the type `b`.

```
foldl :: (b -> a -> b) -> b -> List a -> b
```

foldl

What does this machine do?

- OK, so `foldl` takes three arguments.
- But what does this machine do to those three arguments to compute the return value?

foldl

A standard loop, exactly in a way in which we are familiar

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

Really, is that all?

To compute the product of the list, let:

1. `f = *`
2. `z = 1`

Yes, that is all

```
product list =
  var r = 1
  foreach(e in list)
    r = *(r, e)
  return r
```

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

Another example

To reverse a list, let:

1. `f = \xs x -> x : xs`
2. `z = Nil`

Reversing a cons list

```
reverse list =
  var r = Nil
  foreach(e in list)
    r = :(e, r)
  return r
```

```
reverse list =
  foldl (\xs x -> x : xs) [] list
```

Observations about `foldl`

- We might compute the `length` of a list with `foldl`.
- We might compute the `sum` of a list with `foldl`.
- Importantly, `foldl` **will never work on an infinite list**.

There is nothing more or less to `foldl` than what has just been described.

foldr

The `foldr` function is a machine that requires three values[2]:

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

It will give you back a value of the type `b`.

```
foldr :: (a -> b -> b) -> b -> List a -> b
```

[2] similar to `foldl`, although the function's arguments are swapped in order

foldr

What does the `foldr` machine do?

- Like `foldl`, `foldr` takes three arguments.
- But what this machine do to those three arguments?
- A loop like `foldl`? Something else?

foldr

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

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

1. Every occurrence of the cons constructor `(:)` with `f`.
2. Any occurrence of the nil constructor `[]` with `z`[3].

[3] The nil constructor may be absent —an infinite list

foldr

Constructor Replacement?

- Let `list = A : (B : (C : (D : [])))`
- The expression `foldr f z list`
- `list = A ‘f‘ (B ‘f‘ (C ‘f‘ (D ‘f‘ z)))`

foldr

example —append

- Suppose we wish to append two lists
  - `list1 = U : (V : (W : []))`
  - `list2 = X : (Y : (Z : []))`
  - `result = U : (V : (W : (X : (Y : (Z : []))))))`
- How might the `foldr` machine help us?
- Is this a candidate problem for constructor replacement?

foldr

example —append

```
U : (V : (W : [] ) )
                X : (Y : (Z : [] ) )

U : (V : (W : (X : (Y : (Z : [] ) ) ) ) )
```

foldr

example —append

```
U : (V : (W : [] ) )
              X : (Y : (Z : [] ) )

U : (V : (W : (X : (Y : (Z : [] ) ) ) ) )
```

In `list1`:

- replace `(:)` with `(:)`
- replace `[]` with `list2`

foldr

example —append

- How do we perform constructor replacement?
- `foldr ? ? ?`

foldr

example —append

- How do we perform constructor replacement?
- `foldr ? ? ?`
- On what are we performing constructor replacement?
- `foldr ? ? list1`

foldr

example —append

- How do we perform constructor replacement?
- `foldr ? ? ?`
- On what are we performing constructor replacement?
- `foldr ? ? list1`
- What are we replacing the [] constructor with?
- `foldr ? list2 list1`

foldr

example —append

- How do we perform constructor replacement?
- `foldr ? ? ?`
- On what are we performing constructor replacement?
- `foldr ? ? list1`
- What are we replacing the [] constructor with?
- `foldr ? list2 list1`
- What are we replacing the `(:)` constructor with?
- `foldr (:) list2 list1`

foldr

example —append

```
append list1 list2 =
  foldr (:) list2 list1
```

foldr

More examples

You can repeat this exercise for

- `map :: (a -> b) -> List a -> List b`
- `filter :: (a -> Bool) -> List a -> List a`
- `concat :: List (List a) -> List a`
- `concatMap :: (a -> List b) -> List a -> List b`
- and **many more**

Try it!

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 `[]` constructor *if it ever comes to exist*.
- The expression `foldr (:) []` leaves the list *unchanged*.
  - In other words, passing the list constructors to `foldr` produces an *identity* function.
  - A function that produces an identity, given constructors for a data type, is called its *catamorphism*.
  - `foldr` is the list catamorphism.

Summary

- `foldl` performs an *imperative loop*, just like we are familiar with3.
- `foldl` will **never** work on an infinite list.
- `foldr` performs *constructor replacement*.
- `foldr` **may** work on an infinite list.
- `foldr` is the list catamorphism.
- Everything discussed applies equally to all programming languages.