Beautiful Folds

Matthias Heinzel

September 20, 2017

Outline

Folds

Folds combine the elements in a structure

Folds

```haskell
foldl' :: (b -> a -> b) -> b -> [a] -> b
```

actually:

```haskell
foldl' :: Foldable f => (b -> a -> b) -> b -> f a -> b
```

Using Folds

```
foldl' :: (b -> a -> b) -> b -> [a] -> b

length :: [a] -> Int
length = foldl' (\ acc e -> acc + 1) 0

sum :: [Int] -> Int
sum = foldl' (+) 0
```

Summing a hundred million elements

```haskell
main = print $ foldl' (+) (0 :: Int) [1 .. 100000000]
```

```
53,168 bytes allocated in the heap
 3,480 bytes copied during GC
44,384 bytes maximum residency (1 sample(s))
17,056 bytes maximum slop
     1 MB total memory in use (0 MB lost due to fragmentation)

GC      time    0.000s  (  0.000s elapsed)
Total   time    0.144s  (  0.082s elapsed)
```

Using multiple folds

```haskell
divide :: Int -> Int -> Double  -- for convenience
divide x y = intToDouble x / intToDouble y
  where
    intToDouble = fromInteger . fromIntegral
```

Using multiple folds

```haskell
divide :: Int -> Int -> Double  -- for convenience
divide x y = intToDouble x / intToDouble y
  where
    intToDouble = fromInteger . fromIntegral

average :: [Int] -> Double
average list = divide (sum list) (length list)

main = print $ average [1 .. 100000000]
```

The problem

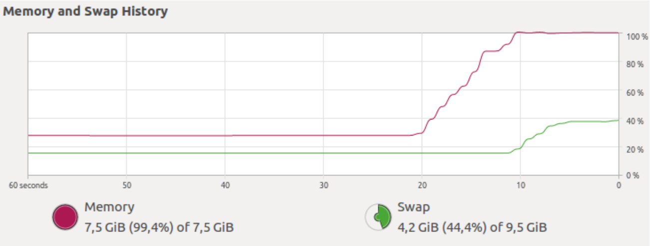

Figure 1: A space leak in its natural habitat

```
 8,000,107,232 bytes allocated in the heap
12,237,661,504 bytes copied during GC
 3,419,124,968 bytes maximum residency (17 sample(s))
   684,952,344 bytes maximum slop
          7760 MB total memory in use (0 MB lost due to fragmentation)

GC      time    9.004s  ( 18.373s elapsed)
Total   time   10.688s  ( 23.502s elapsed)
```

A solution?

```
average' :: [Int] -> Double
average' xs = divide s l
  where
    Pair s l = foldl' f (Pair 0 0) xs
    f (Pair s l) n = Pair (s + n) (l + 1)

data Pair a b = Pair !a !b
```

A solution?

```
8,000,105,904 bytes allocated in the heap
      754,936 bytes copied during GC
       44,384 bytes maximum residency (2 sample(s))
       53,936 bytes maximum slop
            1 MB total memory in use (0 MB lost due to fragmentation)

GC      time    0.028s  (  0.030s elapsed)
Total   time    1.108s  (  1.074s elapsed)
```

[x] Performance

[ ] Code reuse :(

Composition?

```
type Fold i o = [i] -> o
```

too late for composition. . .

Composition?

```haskell
type Fold i o = [i] -> o
```

too late for composition. . .

```haskell
data Fold i o = Fold (o -> i -> o) o
```

could work

Composition?

```haskell
type Fold i o = [i] -> o
```

too late for composition. . .

```haskell
data Fold i o = Fold (o -> i -> o) o
```

could work

```haskell
data Fold i o = Fold (i -> m) (m -> o)  -- what is m?
```

Composition?

```haskell
type Fold i o = [i] -> o
```

too late for composition. . .

```haskell
data Fold i o = Fold (o -> i -> o) o
```

could work

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)
```

Composing Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

compose :: Fold i o -> Fold i o' -> Fold i (o, o')
compose = _
```

Composing Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

compose :: Fold i o -> Fold i o' -> Fold i (o, o')
compose = _

instance (Monoid a, Monoid b) => Monoid (Pair a b) where
  mempty = Pair mempty mempty
  mappend (Pair a b) (Pair a' b') = Pair (mappend a a') (mappend b b')
```

Composing Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

compose :: Fold i o -> Fold i o' -> Fold i (o, o')
compose (Fold pre1 post1) (Fold pre2 post2) = Fold pre post
  where
    pre i = Pair (pre1 i) (pre2 i)
    post (Pair m1 m2) = (post1 m1, post2 m2)
```

Composing Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Functor (Fold i) where

  -- (a -> b) -> Fold i a -> Fold i b
  fmap f (Fold pre post) = _
```

Composing Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Functor (Fold i) where

  -- (a -> b) -> Fold i a -> Fold i b
  fmap f (Fold pre post) = Fold pre (f . post)
```

Folds are Applicatives

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Applicative (Fold i) where

  -- a -> Fold i a
  pure x = _

  -- Fold i (a -> b) -> Fold i a -> Fold i b
  Fold preF postF <*> Fold preX postX = _
```

Folds are Applicatives

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Applicative (Fold i) where

  -- a -> Fold i a
  pure x = Fold (const ()) (const x)

  -- Fold i (a -> b) -> Fold i a -> Fold i b
  Fold preF postF <*> Fold preX postX = _
```

Folds are Applicatives

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Applicative (Fold i) where

  -- a -> Fold i a
  pure x = Fold (const ()) (const x)

  -- Fold i (a -> b) -> Fold i a -> Fold i b
  Fold preF postF <*> Fold preX postX = Fold pre post
    where
      pre i = Pair (preF i) (preX i)
      post (Pair mF mX) = postF mF (postX mX)
```

The average Fold, again

```
length :: Fold i Int
length = Fold (Sum . const 1) getSum

sum :: Fold Int Int
sum = Fold Sum getSum

average :: Fold Int Double
average = divide <$> sum <*> length
```

Running Folds

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

run :: Fold i o -> [i] -> o
run (Fold pre post) = post . foldl' mappend mempty . map pre
```

Running Folds

```
average :: Fold Int Double
average = divide <$> sum <*> length

main = print $ run average [1 .. 100000000]
```

```
19,200,106,912 bytes allocated in the heap
     3,389,824 bytes copied during GC
        44,384 bytes maximum residency (2 sample(s))
        30,600 bytes maximum slop
             1 MB total memory in use (0 MB lost due to fragmentation)

GC      time    0.092s  (  0.074s elapsed)
Total   time    4.184s  (  4.197s elapsed)
```

Running Folds in Parallel

```haskell
runInChunksOf :: Int -> Fold i o -> [i] -> o
runInChunksOf n (Fold pre post) =
  post . reduce . parMap rseq inner . chunksOf n
  where
    reduce = foldl' mappend mempty
    inner = reduce . fmap pre
```

Running Folds at Scale

```haskell
data Fold i o =
  forall m. (Monoid m, Binary m) => Fold (i -> m) (m -> o)
```

across multiple devices

similar to Map-Reduce

The Essence

```haskell
data Fold i o = forall m. Monoid m => Fold (i -> m) (m -> o)

instance Functor (Fold i) where
  fmap f (Fold pre post) = Fold pre (f . post)

instance Applicative (Fold i) where
  pure x = Fold (const ()) (const x)
  Fold preF postF <*> Fold preX postX = Fold pre post
    where
      pre i = Pair (preF i) (preX i)
      post (Pair mF mX) = postF mF (postX mX)

run :: Fold i o -> [i] -> o
run (Fold pre post) = post . foldl' mappend mempty . map pre
```

What we have

A representation of Folds that

- composes nicely
- traverses structures only once
- is reasonably fast
- can be run in parallel

Where to learn more

- Gabriel Gonzalez' MuniHac talk (youtube.com/watch?v=6a5Ti0r8Q2s)
- Algebird library in Scala (github.com/twitter/algebird)

Questions?