Overview

- Monoids (definition, examples)
- Reducers
- Generators
- Benefits of Monoidal Parsing
  - Incremental Parsing (FingerTrees)
  - Parallel Parsing (Associativity)
  - Composing Parsers (Products, Layering)
  - Compressive Parsing (LZ78, Bentley-McIlroy)
- Going Deeper (Seminearrings)
What is a Monoid?

- A Monoid is *any* associative binary operation with a unit.

- **Associative:** \((a + b) + c = a + (b + c)\)
- **Unit:** \((a + 0) = a = (0 + a)\)

- **Examples:**
  - \((*,1)\), \((+,0)\), \((\text{max}, \text{minBound})\), \((.,\text{id})\), ...
Monoids as a Typeclass

class Monoid m where
    mempty :: m
    mappend :: m -> m -> m

mconcat :: [m] -> m
mconcat = foldr mappend mempty
Built-in Monoids

newtype Sum a = Sum a
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum a `mappend` Sum b = Sum (a + b)

newtype Endo a = Endo (a -> a)
instance Monoid (Endo a) where
  mempty = Endo id
  Endo f `mappend` Endo g = Endo (f . g)
So how can we use them?

- Data.Foldable provides fold and foldMap

```haskell
class Functor t => Foldable t where

  ...

  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m

fold = foldMap id
```
instance (Monoid m, Monoid n) => Monoid (m,n) where
  mempty = (mempty, mempty)
  (a,b) `mappend` (c,d) = (a `mappend` c, b `mappend` d)
Associativity is Flexibility

We can:

- `foldr`: `a+(b+(c+...))`
- `foldl`: `((a+b)+c)+ ...`
- or even consume chunks in parallel:
  `(.+.+.+.+.+.)+(.+.+.+.+.+.)+(.+.+.+.+.+.)+...`
- or in a tree like fashion:
  `((.+)+(.+))+((.+)+(.+o))`
- ...

But we always pay full price

- Containers are Monoid-oblivious
- Monoids are Container-oblivious

Can we fix that and admit optimized folds?

( :) is faster than ( \x xs -> return x ++ xs )

And what about monotypic containers?

Strict and Lazy ByteStrings, IntSets, etc...
Monoid-specific efficient folds

class Monoid m => Reducer c m where
  unit :: c -> m
  snoc :: m -> c -> m
  cons :: c -> m -> m

  c `cons` m = unit c `mappend` m
  m `snoc` c = m `mappend` unit c
Simple Reducers

instance Reducer a [a] where
  unit a = [a]
  cons = (:)

instance Num a => Reducer a (Sum a) where
  unit = Sum

instance Reducer (a -> a) (Endo a) where
  unit = Endo
Reducers enable faster folds

reduceList :: (c `Reducer` m) => [c] -> m
reduceList = foldr cons mempty

reduceText :: (Char `Reducer` m) => Text -> m
reduceText = Text.foldl1’ snoc mempty
class Generator c where
    type Elem c :: *
    mapReduce :: (e `Reducer` m) => (Elem c -> e) -> c -> m
    ...

reduce :: (Generator c, Elem c `Reducer` m) => c -> m
reduce = mapReduce id

instance Generator [a] where
    type Elem [a] = a
    mapReduce f = foldr (cons . f) mempty
Container-Specific Folds

instance Generator Strict.ByteString where
type Elem Strict.ByteString = Word8
mapReduce f = Strict.foldl’ (\a b -> snoc a (f b)) mempty

instance Generator IntSet where
type Elem IntSet = Int
mapReduce f = mapReduce f . IntSet.toList

instance Generator (Set a) where
type Elem (Set a) = a
mapReduce f = mapReduce f . Set.toList
instance Generator Lazy.ByteString where
  mapReduce f =
    Data.Foldable.fold .
    parMap rwhnf (mapReduce f) .
    Lazy.toChunks
Non-Trivial Monoids/Reducers

- Tracking Accumulated File Position Info
- FingerTree Concatenation
- Delimiting Words
- Parsing UTF8 Bytes into Chars
- Parsing Regular Expressions
- Recognizing Haskell Layout
- Parsing attributed PEG, CFG, and TAGs!
Generator Combinators

mapM_ :: (Generator c, Monad m) => (Elem c -> m b) -> c -> m ()
forM_ :: (Generator c, Monad m) => c -> (Elem c -> m b) -> m ()
msum :: (Generator c, MonadPlus m, m a ~ Elem c) => c -> m a
traverse_ :: (Generator c, Applicative f) => (Elem c -> f b) -> c -> f ()
for_ :: (Generator c, Applicative f) => c -> (Elem c -> f b) -> f ()
asum :: (Generator c, Alternative f, f a ~ Elem c) => c -> f a
and :: (Generator c, Elem c ~ Bool) => c -> Bool
or :: (Generator c, Elem c ~ Bool) => c -> Bool
any :: Generator c => (Elem c -> Bool) -> c -> Bool
all :: Generator c => (Elem c -> Bool) -> c -> Bool
foldMap :: (Monoid m, Generator c) => (Elem c -> m) -> c -> m
fold :: (Monoid m, Generator c, Elem c ~ m) => c -> m
toList :: Generator c => c -> [Elem c]
concatMap :: Generator c => (Elem c -> [b]) -> c -> [b]
elem :: (Generator c, Eq (Elem c)) => Elem c -> c -> Bool
filter :: (Generator c, Reducer (Elem c) m) => (Elem c -> Bool) -> c -> m
filterWith :: (Generator c, Reducer (Elem c) m) => (m -> n) -> (Elem c -> Bool) -> c -> n
find :: Generator c => (Elem c -> Bool) -> c -> Maybe (Elem c)
sum :: (Generator c, Num (Elem c)) => c -> Maybe (Elem c)
product :: (Generator c, Num (Elem c)) => c -> [Elem c]
notElem :: (Generator c, Eq (Elem c)) => Elem c -> c -> Bool
Generator Combinators

- Most generator combinators just use mapReduce or reduce on an appropriate monoid.

```
reduceWith f = f . reduce
mapReduceWith f g = f . mapReduce g

sum = reduceWith getSum
and = reduceWith getAll
any = mapReduceWith getAny
toList = reduce
mapM_ = mapReduceWith getAction
...```
Example: File Position Delta

- We track the delta of column #s

```haskell
data Delta = Cols Int | ...

instance Monoid Delta where
  mempty = Cols 0
  Cols x `mappend` Cols y = Cols (x + y)

instance Reducer Delta Char where
  unit _ = Cols 1

-- but what about newlines?
Handling Newlines

- After newline, preceding columns are useless, and we know an absolute column #

```haskell
data Delta = Cols Int | Lines Int Int | ...

instance Monoid Delta where
    Lines l _ `mappend` Lines l' c' = Lines (l + l') c'
    Cols _ `mappend` Lines l' c' = Lines l c'
    Lines l c `mappend` Cols c' = Lines l (c + c')

instance Reducer Delta where
    unit '\n' = Lines 1 1
    unit _ = Cols 1
```

- but what about tabs?
Handling Tabs

data Delta = Cols Int | Lines Int Int | Tabs Int Int | ...

nextTab :: Int -> Int
nextTab !x = x + (8 – (x – 1) `mod` 8)

instance Monoid Delta where

... Lines l c `mappend` Tab x y = Lines l (nextTab (c + x) + y)
Tab{} `mappend` l@Lines{} = l
Cols x `mappend` Tab x’ y = Tab (x + x’) y
Tab x y `mappend` Cols y’ = Tab x (y + y’)
Tab x y `mappend` Tab x’ y’ = Tab x (nextTab (y + x’) + y’)

instance Reducer Char Delta where

unit `\t` = Tab 0 0
unit `\n` = Line 1 1
unit _ = Cols 1
#line Directives

data Delta =
    = Pos !ByteString !Int !Int
    | Line !Int !Int
    | Col !Int
    | Tab !Int !Int
instance Monoid Delta where
  mempty = Cols 0
  Cols c `mappend` Cols d = Cols (c + d)
  Cols c `mappend` Tab x y = Tab (c + x) y
  Lines l c `mappend` Cols d = Lines l (c + d)
  Lines l _ `mappend` Lines m d = Lines (l + m) d
  Lines l c `mappend` Tab x y = Lines l (nextTab (c + x) + y)
  Tab x y `mappend` Cols d = Tab x (y + d)
  Tab x y `mappend` Tab x' y' = Tab x (nextTab (y + x') + y')
  Pos f l _ `mappend` Lines m d = Pos f (l + m) d
  Pos f l c `mappend` Cols d = Pos f l (c + d)
  Pos f l c `mappend` Tab x y = Pos f l (nextTab (c + x) + y)
  _ `mappend` other = other

instance Reducer Char Delta where
  unit 'n' = Lines 1 1
  unit 't' = Tab 0 0
  unit _ = Cols 1

data Delta
  = Pos S.ByteString !Int !Int
  | Lines !Int !Int
  | Tab !Int !Int
  | Cols !Int
  deriving (Eq, Show, Data, Typeable)

nextTab :: Int -> Int
nextTab x = x + (8 - x `mod` 8)
Example: Parsing UTF8

- Valid UTF8 encoded Chars have the form:
  - [0x00...0x7F]
  - [0xC0...0xDF] extra
  - [0xE0...0xEF] extra extra
  - [0xF0...0xF4] extra extra extra

  where extra = [0x80...0xBF] contains 6 bits of info in the LSBs and the only valid representation is the shortest one for each symbol.
UTF8 as a Reducer Transformer

data UTF8 m = Segment !Prefix m !Suffix | Chunk !Suffix

instance (Char `Reducer` m) => Monoid (UTF8 m)
    where ...

instance (Char `Reducer` m) => (Byte `Reducer` UTF8 m)
    where ...

Given 7 bytes we must have seen a full Char. We only need track up to 3 bytes on either side.
Putting the pieces together so far

We can:

- Parse a file as a Lazy ByteString,
- Ignore alignment of the chunks and parse UTF8, automatically cleaning up the ends as needed when we glue the reductions of our chunks together.
- We can feed that into a complicated Char `Reducer` that uses modular components like Delta.
Compressive Parsing

- LZ78 decompression never compares values in the dictionary. Decompress in the monoid, caching the results.
- Unlike later refinements (LZW, LZSS, etc.) LZ78 doesn’t require every value to initialize the dictionary permitting infinite alphabets (i.e. Integers)
- We can compress chunkwise, permitting parallelism
- Decompression fits on a slide.
Compressive Parsing

newtype LZ78 a = LZ78 [Token a]
data Token a = Token a !Int

instance Generator (LZ78 a) where
type Elem (LZ78 a) = a
mapTo f m (LZ78 xs) = mapTo' f m (Seq.singleton mempty) xs

mapTo' :: (e `Reducer` m) => (a -> e) -> m -> Seq m -> [Token a] -> m
mapTo' _ m _ [] = m
mapTo' f m s (Token c w:ws) = m `mappend` mapTo' f v (s |> v) ws
    where v = Seq.index s w `snoc` f c
Other Compressive Parsers

- The dictionary size in the previous example can be bounded, so we can provide reuse of common monoids **up to** a given size or within a given window.
- Other extensions to LZW (i.e. LZAP) can be adapted to LZ78, and work even better over monoids than normal!
- Bentley-McIlroy (the basis of bmdiff and open-vcdiff) can be used to reuse all common submonoids **over** a given size.
Going Deeper

Algebraic Structure Provides Opportunity

<table>
<thead>
<tr>
<th>Structure</th>
<th>Example Opportunity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semigroup</td>
<td>Parallelized Folds</td>
</tr>
<tr>
<td>Monoid</td>
<td>Unit</td>
</tr>
<tr>
<td>Group</td>
<td>Inverses/Undo</td>
</tr>
<tr>
<td>Commutative Monoid</td>
<td>Reordering Computation</td>
</tr>
<tr>
<td>Applicative</td>
<td>Synthesized Attributes</td>
</tr>
<tr>
<td>Abelian Group</td>
<td>Out-Of-Order Undo</td>
</tr>
<tr>
<td>Ringoid</td>
<td>Cancellative Zero</td>
</tr>
<tr>
<td>Right Seminearring</td>
<td>Context-Free Recognizers</td>
</tr>
<tr>
<td>Alternative</td>
<td>Context-Free Attribute Grammars</td>
</tr>
<tr>
<td>Monad</td>
<td>Context-Sensitivity</td>
</tr>
</tbody>
</table>
Conclusion

- Monoids are everywhere
- Reducers allow efficient use of Monoids
- Generators can apply Reducers in parallel
- Monoids/Reducers are composable
- Compression can improve performance
- Algebraic structures provide opportunity