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:
  - \(((\times),1), ((+),0), (\max, \min\text{Bound}), ((.),\text{id}), \ldots\)
Monoids as a Typeclass

- (from Data.Monoid)

- class Monoid m where

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

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 = 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
```
Monoids allow succinct definitions

instance Monoid [a] where
  mempty = []
  mappend = (++)

concat :: [[a]] -> [a]
concat = fold
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap = foldMap
Monoids are Compositional

instance (Monoid m, Monoid n) => Monoid (m,n) where
  mempty = (mempty,mempty)
  (a,b) `mappend` (c,d) = (a `mappend` c, b `mappend` d)
Associativity allows Flexibility

We can:

- `foldr`: a+(b+(c+...))
- `foldl`: ((a+b)+c)+ ... 
- or even consume chunks in parallel: 
  (.+.+.+.+.+.)+(.+.+.+.+.)+(.+.+.+.+.)+...
- or in a tree like fashion: 
  (((.+)+(.+))(((.+)+(.+0))))
- ...
But we always pay full price

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

Can we fix that and admit optimized folds? (Reducers)
  - (:) is faster than (++) . return

And what about non-Functorial containers? (Generators)
  - Strict and Lazy ByteString, IntSet, etc...

Foldable doesn’t help us here.
Monoid-specific efficient folds

(from Data.Monoid.Reducer)

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
Reducers enable faster folds

- \( \text{reduceList} :: (c \ `\text{Reducer}` \ m) \Rightarrow [c] \rightarrow m \)
- \( \text{reduceList} = \text{foldr} \ \text{cons} \ \text{mempty} \)

- \( \text{reduceText} :: (\text{Char} \ `\text{Reducer}` \ m) \Rightarrow \text{Text} \rightarrow m \)
- \( \text{reduceText} = \text{Text.foldl'} \ \text{snoc} \ \text{mempty} \)

- (We’ll come back and generalize the containers later)
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
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 TAG Grammars
Example: File Position Info

-- we track the delta of column #s

data SourcePosition = Cols Int | ...

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

instance Reducer SourcePosition where
    unit _ = Cols 1

-- but what about newlines?
Handling Newlines

data SourcePosition = Cols Int | Lines Int Int
instance Monoid SourcePosition 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 SourcePosition where
    unit ‘\n’ = Lines 1 1
    unit _ = Cols 1

    -- but what about tabs?
Handling Tabs

data SourcePosition = ... | Tabs Int Int

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

instance Monoid SourcePosition 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 SourcePosition where
  unit 't' = Tab 0 0
  unit 'n' = Line 1 1
  unit _ = Cols 1
#line pragmas and start of file

data SourcePosition file =
  = Pos file !Int !Int
  | Line !Int !Int
  | Col !Int
  | Tab !Int !Int
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 = ...

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 Char.
We only track up to 3 bytes on either side.
Non-Functorial Containers

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
Now we can use 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 =
  fold .
  parMap rwhnf (mapReduce f) .
Lazy.toChunks
An aside: Dodging mempty

-- Fleshing out Generator

class Generator c where

    type Elem c :: *

    mapReduce :: (e `Reducer` m) => (Elem c -> e) -> c -> m
    mapTo :: (e `Reducer` m) => (Elem c -> e) -> m -> c -> m
    mapFrom :: (e `Reducer` m) => (Elem c -> e) -> c -> m -> m

    mapReduce f = mapTo f mempty
    mapTo f m = mappend m . mapReduce f

    mapFrom f = mappend . mapReduce f

  -- minimal definition mapReduce or mapTo
Dodging mempty

instance Generator [c] where
    type Elem [c] = c
    mapFrom f = foldr (cons . f)
    mapReduce f = foldr (cons . f) mempty

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

This avoids some spurious ‘mappend mempty’ cases when reducing generators of generators.
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 ()
asm :: (Generator c, Alternative f, f a ~ Elem c) => c -> f a
and :: (Generator c, Elem c ~ Boolean) => c -> Boolean
or :: (Generator c, Elem c ~ Boolean) => c -> Boolean
any :: Generator c => (Elem c -> Boolean) -> c -> Boolean
all :: Generator c => (Elem c -> Boolean) -> c -> Boolean
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]
elm :: (Generator c, Eq (Elem c)) => Elem c -> c -> Boolean
filter :: (Generator c, Reducer (Elem c) m) => (Elem c -> Boolean) -> c -> m
filterWith :: (Generator c, Reducer (Elem c) m) => (m -> n) -> (Elem c -> Boolean) -> c -> n
find :: Generator c => (Elem c -> Boolean) -> c -> Maybe (Elem c)
sum :: (Generator c, Num (Elem c)) => c -> Elem c
product :: (Generator c, Num (Elem c)) => c -> Elem c
notElem :: (Generator c, Eq (Elem c)) => Elem c -> c -> Boolean
Generator Combinators

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

reduceWith \( f = f \cdot \text{reduce} \)

mapReduceWith \( f \  g = f \cdot \text{mapReduce} \  g \)

\[
\begin{align*}
\text{sum} &= \text{reduceWith } \text{getSum} \\
\text{and} &= \text{reduceWith } \text{getAll} \\
\text{any} &= \text{mapReduceWith } \text{getAny} \\
\text{toList} &= \text{reduce} \\
\text{mapM}_\_ &= \text{mapReduceWith } \text{getAction} \\
\end{align*}
\]

...
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 SourcePosition.
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.
I Want More Structure!

A Monoid is to an Applicative as a Right Seminearring is to an Alternative.

If you throw away the argument of an Applicative, you get a Monoid, if you throw away the argument of an Alternative you get a RightSemiNearRing.

In fact any Applicative wrapped around any Monoid forms a Monoid, and any Alternative wrapped around a Monoid forms a RightSemiNearring.