



Edward Kmett


Iteratees, Parsec, and Monoids



A PARSING TRIFECTA




Overview

- The Problem
 - Deriving Iteratees á la Oleg
 - Buffered Iteratees
 - Layering Parsec Over Iteratees
 - Local Context-Sensitivity
 - Monoids From Invariants
 - Layering Monoids (Layout, Parsing)
 - Conclusion
- 




The Problem

- Monadic parsing is *convenient*
 - Monoids are *parallel* and *incremental*

 - How to include both feature sets?
- 



Parser Combinators

- Parser combinators make it possible to express parsers directly in Haskell using an embedded domain specific language encoding of your grammar.
 - Parsec is the most well-known parser combinator library in use in Haskell.
- 

Parser Combinator Libraries

- Parsec 3
 - Monadic, depth-first, backtracking
- Iteratees
 - Monadic, resumable, *non*-backtracking
- UU-ParsingLib
 - Monadic, breadth-first, backtracking
- Parsimony (coming soon)
 - Monoidal parsing of Alternative grammars

The Problem In More Detail

- Monadic parsing is *convenient*
- Monoids are *parallel* and *incremental*
- How to include both feature sets?
 - People know how to use Parsec effectively.
 - Parsec needs the full input and can't be resumed.
 - An efficient parsing monoid can't be monadic, because of context-sensitivity



Monoidal Parsing Issues

- As a Monoid we need to parse input chunks with no contextual information about the chunks.
 - We don't know how far along in the file we are
 - We don't know if we're in the middle of parsing a string
 - We don't know what lexemes are valid in our current state.

What are our options?

- Walk an Applicative grammar top-down or bottom up in Earley or CYK style and merge sub-parses. (Parsimony)
- Extract a small representation of the state transitions represented by the values spanned by our monoid for smaller languages. (i.e. Bicyclic, Regex...)
- Hunt for frequently occurring invariants in the lexer where context sensitivity vanishes and start new monadic parsers at those locations, folding together their results.

Monoid-Wrapped Parsec Issues

- We need Parsec to be able to run up to a point and block waiting for Input in order for it to work Monoidally.
- But, Parsec offers `getInput`, which can be used to obtain the unparsed portion of the input Stream.
- We need to be able to resume the same Parsec parser with multiple future parses to deal with incremental parsing, so this at first seems to be paradoxical.



Inverting Parsec: Iteratees

- An Iteratee is a monad that consumes input and performs computation until it needs more, then asks for more input.
- They are perhaps best understood by thinking about resumable computations as a monad.

Partiality as a Monad

```
data Partial a
  = Return a
  | Cont (Partial a)
```

```
instance Monad Partial where
  return = Return
  Return a >>= f = f a - the first monad law!
  Cont k >>= f = Cont (k >>= f)
```

```
runPartial :: Partial a -> Int -> Either (Partial a) a
runPartial (Done a) _ = Right a
runPartial c@(Cont _) 0 = Left c
runPartial (Cont k) !n = runPartial k (n - 1)
```

Iteratees á la Oleg

```
data Input = Chunk ByteString | EOF
```

```
data Iteratee a  
  = Return a Input  
  | Cont (Input -> Iteratee a)
```

```
instance Monad Iteratee where  
  return = Return a (Chunk empty)  
  Return a (Chunk e) >>= f | null e = k  
  Return a i >>= f = case f a of  
    Return b _ -> Return b i  
    Cont k -> k i  
  Cont k >>= f = Cont (k >=> f)  
  -- (>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
```

Reading input in an Iteratee

```
next :: Iteratee (Maybe Char)
```

```
next = Cont next'
```

```
next' :: Input -> Iteratee (Maybe Char)
```

```
next' EOF = Return Nothing EOF
```

```
next' (Chunk i)
```

```
  | null i = next
```

```
  | otherwise = Return (Just (head i)) (Chunk (tail i))
```

But once it consumes part of the input, it is gone!

Backtracking Iteratees

- The problem with the default Iteratee design is that it conflates current position info with the contents of the buffer.
- We need to retain the full input buffer history for backtracking purposes, but this requires an efficient means of concatenating ByteString and indexing into the buffer. ByteString alone, lazy or not, won't cut it asymptotically!

Monoids to the rescue

```
newtype Cursor = Cursor { getCursor :: Int }  
    deriving (Eq, Ord, Num, Show, Read, Enum)
```

```
instance Monoid Cursor where  
    mempty = 0  
    mappend = (+)
```

Efficient Buffers

```
newtype Chunk = Chunk { getChunk :: S.ByteString } deriving
  (Eq, Ord, Show, Read)
```

```
instance Measured Cursor Chunk where
  measure (Chunk xs) = Cursor (S.length xs)
```

```
type Buffer = FingerTree Cursor Chunk
```

```
index :: Cursor -> Buffer -> Word8
  index !i !t = S.index a $ getCursor (i - measure l) where
    (l,r) = F.split (> i) t
    Chunk a :< _ = F.viewl r
```

```
-- O(log (min (n,m-n))) for m chunks and an offset into the nth chunk
```


Buffered Iteratee

```
data Iteratee a
  = Done a !Buffer !Bool
  | Cont (Buffer -> Bool -> Iteratee a)
```

```
instance Monad Iteratee where
  return a = Done a F.empty False
```

```
Done a h False >>= f | F.null h = f a
```

```
Done a h eof >>= f = case f a of
```

```
  Done b _ _ -> Done b h eof
```

```
  Cont k -> k h eof
```

```
Cont k >>= f = Cont (\h eof -> k h eof >>= f)
```

Care and Feeding of Iteratees

```
type Enumerator a = Iteratee a -> Iteratee a
```

```
supplyBuffer :: Buffer -> Enumerator a
```

```
supplyBuffer b (Cont k) = k b False
```

```
supplyBuffer _ other = other
```

```
supplyStrictByteString :: S.ByteString -> Enumerator a
```

```
supplyStrictByteString = supplyBuffer . F.singleton . Chunk
```

Generic Supplies

```
class Supply t where
  supply :: t -> Enumerator a
  supplyList :: [t] -> Enumerator a
  supplyList = foldr (andThen . supply) id
```

```
instance Supply a => Supply [a] where
  supply = supplyList
```

```
instance Supply Buffer where
  supply = supplyBuffer
```

```
instance Supply Strict.ByteString where ...
instance Supply Char where ...
```

Supplying EOF

```
data EOF = EOF
```

```
instance Supply EOF where
```

```
    supply _ (Cont k)      = k F.empty True
```

```
    supply _ (Done a h _) = Done a h True
```

A Digression: Failure

```
data Iteratee a
  = Done a !Buffer !Bool
  | Fail String !Buffer !Bool
  | Cont (Buffer -> Bool -> Iteratee a)

instance Monad Iteratee where
  return a = Done a F.empty False

  Done a h False >>= f | F.null h = f a
  Done a h eof >>= f = case f a of
    Done b _ _ -> Done b h eof
    Cont k -> k h eof
    Fail s _ _ -> Fail s h eof
  Cont k >>= f = Cont (\h eof -> k h eof >>= f)
  Fail s h eof >>= _ = Fail s h eof

  fail s = Fail s F.empty False

instance MonadPlus Iteratee where ...
```

Reading from Buffered Iteratees

```
get :: Cursor -> Iteratee Word8
get i = Cont getWord8' where
  get' :: Buffer -> Bool -> Iteratee Word8
  get' h eof
    | n < measure h = Done (index n h) h eof
    | eof = Fail "EOF" h eof
    | otherwise = Cont (\h' eof' -> get' (h >< h') eof')
```

Iteratee-Based Parsec Streams

```
instance Stream Cursor Iteratee Word8 where
  uncons !n = do
    w <- get n
    return $ Just (c, n + 1)
  <|> return Nothing
```

-- I actually use a Char Stream that does UTF8 decoding, but it won't fit on a slide!

A UTF8 Iteratee Parsec Stream

```
instance Stream Cursor Iteratee Char where
  uncons !n = (getWord8 n >= uncons') <|> return Nothing where
    uncons' c
      | c <= 0x7f =
          return $ Just (toEnum (fromEnum c), n + 1)
      | c >= 0xc2 && c <= 0xdf = do
          d <- getWord8 (n + 1)
          return $ Just (b2 c d, n + 2)
      | c >= 0xe0 && c <= 0xef = do
          d <- getWord8 (n + 1)
          e <- getWord8 (n + 2)
          return $ Just (b3 c d e, n + 3)
      | c >= 0xf0 && c <= 0xf4 = do
          d <- getWord8 (n + 1)
          e <- getWord8 (n + 2)
          f <- getWord8 (n + 3)
          return $ Just (b4 c d e f, n + 4)
      | otherwise =
          return $ Just (replacementChar, n + 1)
```

-- I lied. It will.

Putting It Together

```
type P = ParsecT Cursor () Iteratee
```

```
parser = string "hello" <|> string "goodbye"
```

```
example =
```

```
    supply EOF $
```

```
    supply "bye" $
```

```
    supply "good" $
```

```
    runParser parser () "-" (Cursor 0)
```

```
-- Done "goodbye" (Chunk (pack "goodbye"))
```

```
-- note the successful backtrack.
```

```
-- You can obtain the current cursor location with getInput
```

Slicing w/ Sharing

```
slicelt :: Cursor -> Cursor -> Iteratee S.ByteString
slicelt !i !j = Cont slice' where
  slicelt' :: Buffer -> Bool -> Iteratee S.ByteString
  slicelt' h eof
    | j <= measure h || eof = Done (sliceBuffer h) h eof
    | otherwise = Cont $ \h' eof' -> slicelt' (h >< h') eof'
sliceBuffer :: Buffer -> S.ByteString
sliceBuffer !t
  | req <= rmn = Strict.take req first
  | otherwise =
    Strict.concat $ Lazy.toChunks $
    Lazy.take (fromIntegral req) $
    Lazy.fromChunks $ first : map getChunk (toList r')
where
  (l,r) = F.split (> i) t
  Chunk a :< r' = F.view l r
  first = Strict.drop (getCursor (i - measure l)) a
  req = getCursor $ j - i
  rmn = Strict.length first
```

Slicing in a Parsec Parser

```
slice :: Cursor -> Cursor -> P ByteString
slice mark release =
    lift (sliceIt mark release)
```

```
sliced :: P a -> P ByteString
sliced parse = do
    mark <- getInput
    parse
    release <- getInput
    slice mark release
```

Sliced Recognizers

```
data Token = Ident ByteString | Symbol ByteString
```

```
ident = Ident <$>
```

```
    sliced (lower >> skipMany isIdent)
```

```
symbol = Symbol <$>
```

```
    sliced (skipMany isSymbol)
```



An Iteratee Based Monoid

Assume we have a language where we spot the invariant in the lexer that after any newline, that is not preceded with a backslash, we can know that any lexeme is valid.

We will scan input bytestrings for non-backslashed newlines and start an iteratee after that.



Since, this comes from an actual language, for which whitespace is used for Haskell-style layout, we'll have to track the whitespace on the edges of our result

Local Context Sensitivity

Many PL grammars are only *locally* context-sensitive.

Real compilers usually use “error productions” at the next newline or semicolon to resume parsing after a syntax error

So, scan input for those and start a parser lexer that feeds tokens to a Reducer, then merge those results, monoidally.

Recall: File Position Monoid

```
data Delta
  = Pos !ByteString !Int !Int
  | Lines !Int !Int
  | Cols !Int
  | Tab !Int !Int
```

Introducing Lex

```
data LexResult t ts
  = LexSegment !Delta ts !Delta
  | LexChunk !Delta
  deriving (Eq, Show, Data, Typeable)
```

```
data Lex t ts
  = LexEmpty
  | LexRaw !Bool !Buffer !Bool
  | LexCooked !Buffer (Iteratee (LexResult t ts))
```

```
class Spaceable t where
  space :: Delta -> t
```

```
class (Spaceable t, Reducer t ts) => Lexable t ts | ts -> t where
  lexer :: Iteratee (LexResult t ts)
```


Making Lex into a Monoid

```
(<-<) :: Supply c => Iteratee a -> c -> Iteratee a
```

```
(<-<) = flip supply
```

```
instance Lexable t ts => Monoid (Lex t ts) where
```

```
  mempty = LexEmpty
```

```
  LexEmpty `mappend` b = b
```

```
  a `mappend` LexEmpty = a
```

```
  LexRaw _ xs False `mappend` LexRaw True ys _ =
```

```
    LexCooked xs $ lexer <-< tailBuffer ys
```

```
  LexRaw a xs _ `mappend` LexRaw _ ys d    = LexRaw a (xs `mappend` ys) d
```

```
  LexRaw _ xs _ `mappend` LexCooked ys m    = LexCooked (xs `mappend` ys) m
```

```
  LexCooked xs m `mappend` LexRaw _ ys _    = LexCooked xs (m <-< ys)
```

```
  LexCooked xs m `mappend` LexCooked ys n    = LexCooked xs $ merge (m <-< ys <-< EOF) n
```

```
merge :: Lexable t ts =>
```

```
  Iteratee (LexResult t ts) -> Iteratee (LexResult t ts) -> Iteratee (LexResult t ts)
```

```
merge = ... an exercise for the reader ...
```

Applying Lex

```
newtype LexSource t ts =  
  LexSource { getLexSource :: ByteString }
```


```
instance Lexable t ts =>  
  Measured (Lex t ts) (LexSource t ts) where  
  measure = unit . getLexSource
```

```
type Source = FingerTree (Lex t ts) (LexSource t ts)
```

Just build up a Source, and it can be measured as a reduction of the token stream.



Going Deeper

- We can feed tokens to any Token-Reducer.
 - Use another monoidal layer, using an Iteratee-based Parsec parser and scan the lexemes for a nice resumption point like layout introducing keywords.
 - And run a simple left-to-right monadic calculation that balances parentheses/brackets/braces and layout.
- 

Bitonic Layout

Process layout monoidally by keeping track of a monoid of unfinished closing contexts, reduced results and opening contexts for layout parens, and bracket-like keyword pairs like case .. of, and if .. else.

```
-- ...)| ... | (... (... + ... ) ... ) | ... ( ... =  
-- ...)| ... | (...
```

```
data Bitonic m  
  = Bitonic (Seq (Closing m)) m (Seq (Opening m))
```

Layout typically connotes the detail parser to use inside so we can finish parsing here, declarations after 'where/let', statements after 'do', etc.

Conclusion

- We can parse monoidally using Parsec parsers if we plan ahead this permits incremental and parallel parsing, which is useful for modern compiler demands:
 - Interactive type checking
 - “Intellisense” and tab-completion.
 - Exact syntax highlighting as you type.
- These same patterns are useful at multiple levels of the parser. (Lexing, layout, etc.)
- Buffered Iteratees can invert control for Parsec effectively and, with slicing, improve sharing between tokens and the source tree.
- Slides and Source will be Available Online via <http://comonad.com/>