

Left-fold enumerators

Johan Tibell
Google

Sep 2008

Hyena - A web application server

Proper resource management is crucial

- Free allocated resources (e.g. file descriptors) in a timely manner
- Constant memory usage

Performance matters

- Low latency
- High throughput

Safety and ease of use

- We use Haskell!

A web application interface

```
-- A web application
type Application = Request -> IO Response

data Method = Get | Post | Put | Delete | ...

-- An HTTP request
data Request = Request
  { requestMethod :: Method
  , requestUri    :: ByteString
  , headers       :: [(ByteString, ByteString)]
  , input         :: ??? -- Request body
  }

type StatusCode = Int

-- An HTTP Response
type Response = (StatusCode, ???) -- Response body
```

Constraints

- In rare cases like a file upload the request body might be large
- Somewhat more frequently the response body might be large e.g. when streaming a video from YouTube
- We want to use a constant amount of memory to serve the request
- We need to free allocated resources as soon as they aren't needed anymore
- The size of the request or response body isn't always known

Streams

```
class Stream s where
  read :: s -> Int -> IO ByteString
  close :: s -> IO ()
```

- Imperative
- Manual resource management
 - Must free resources when the end of the stream has been reached,
 - or in case of an exception
 - Hurts modularity

Lazy I/O

- Create a lazy byte string by reading lazily from the socket

```
import Data.ByteString.Lazy as L
data Request = Request
  { ...
  , input :: L.ByteString
  }
```

What's wrong with lazy I/O?

- To generate a response to send back to the client we might need to:
 - Open files
 - Make HTTP requests to back-end servers
 - Connect to a database
- Lots of side effects for something that claims to be pure in it's type!
- Resources aren't freed in a timely manner in presence of some types of errors
- I/O exceptions can occur in pure code

Idea: Use inversion of control

```
type Response = (StatusCode, EnumeratorM IO)
```

- The resource is iterated over using a left fold.

```
type EnumeratorM m = forall a. IterateeM m -> a -> m a
```

- The caller provides a function to call whenever data is available

```
type IterateeM a m = a -> ByteString -> m (Either a a)
```

A file enumerator

```
fileEnum :: FilePath -> EnumeratorM IO
fileEnum fname iteratee seed = do
  h <- openBinaryFile fname ReadMode
  let loop f z = do
        block <- hGetNonBlocking h 1024
        if null block then return z
        else do
          z' <- f z block
          case z' of
            Left z'' -> return z''
            Right z'' -> loop f z''
  seed' <- loop iteratee seed
  hClose h >> return seed'
```

Pros

- Allocated resources are always be freed at the earliest possible time by the enumerator
- We can still interleave e.g. reading from disc with sending data over the network and use $O(1)$ memory

```
sendChunk :: Socket -> IterateeM () IO
```

```
sendChunk sock _ bs = send sock bs >> return (Right ())
```

```
sendResponse :: EnumeratorM IO -> Socket -> IO ()
```

```
sendResponse enum sock = enum (sendChunk sock) ()
```

- `sendChunk` is an unfold

Pros cont.

- We can use kqueue, epoll, and other OS event systems to drive many enumerators in parallel
 - We store the current iteration state, the seed, together with the file descriptors and whenever an event notification is received we compute a new seed

Using enumerators

- I've implemented a web server, Hyena, using enumerators for all socket and file I/O
 - I use a resumable parser for parsing HTTP headers, the parser state is passed around as a seed by the enumerator
 - Composing enumerators to provide transparent HTTP chunked encoding is not too difficult
 - Have done much optimizations yet, still 2000+ requests / second thanks to block based I/O

Example: HTTP chunked encoding

- Chunked encoding is used when the number of bytes that will be transmitted is unknown
- Each chunk is preceded by a header giving its length e.g. "1a\r\n" says that the following chunk is 26 bytes long
- The last chunk has zero length and is preceded by "0\r\n"

HTTP chunked encoding

```
chunkEnum :: Monad m => EnumeratorM m -> EnumeratorM m
chunkEnum enum f initSeed = fst `liftM` enum go
                                (initSeed, Left S.empty)
```

where

```
go (seed, Left acc) bs =
  case S.elemIndex nl bs of
    Just ix ->
      let (line, rest) = S.splitAt (ix + 1) bs
          hdr           = S.append acc line
          chunkLen     = pHeader hdr
      in case chunkLen of
          Just n -> go (seed, Right n) rest
          Nothing -> error $ "malformed header"
    Nothing -> return $ Right
                (seed, Left (S.append acc bs))
```

HTTP chunked encoding cont.

```
go (seed, Right n) bs =
  let len = S.length bs
  in if len < n
    then do
      seed' <- f seed bs
      case seed' of
        Right seed'' -> return $ Right
          (seed'', Right $! n - len)
        Left seed'' -> return $ Left
          (seed'', Right $! n - len)
    else let (bs', rest) = S.splitAt n bs
          in do
            seed' <- f seed bs'
            case seed' of
              Right seed'' -> go (seed'',
                Left S.empty) rest
              Left seed'' -> return $ Left
                (seed'', Left rest)
```

Conclusions

- Enumerators are a safe and efficient
- Not too onerous programming model
- Question: How well do enumerators compose compared to e.g. functional pipelines