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 larger 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 resources allocated to serve the request as soon as the response has been sent

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
  - etc.
- 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 errors
- I/O exceptions can occur in pure code

Idea: Use inversion of control

```
data Request = Request
  { ...
  , input :: EnumeratorM IO
  }
```

- The resource (request body) 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 Iteratee a m = a -> ByteString -> m (Either a a)
```

A file enumerator without error handling

```
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 function
- 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 ())

sendRespone :: EnumeratorM IO -> Socket -> IO ()
sendRespone 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