From Enumerator to Conduit

Erik de Castro Lopo

February 16, 2012

Overview

- What is the problem?
- Iteratee/Enumerator
- The difficult stuff
- Conduit

What is the problem?

Lazy evaluation and I/O is fundamentally mismatched

```
 a <- readFile "input"
process a
```

- How much memory does it use?
- When does the file handle get closed?
- What happens to errors?

Iteratee / Enumerator history

- First paper by Oleg Kiselyov in 2009
- Became a library maintained by John Lato
- Another library named Enumerator by John Millikin in 2010
- IterIO by David Mazieres in 2011
- Conduit by Michael Snoyman in 2011
- Pipes by Gabriel Gonzalez in 2012

Iteratee / Enumerator concept

Main ideas:

- Data is read from and written to I/O in chunks
- Stream of chunks pass a set of processing elements
- First chunk is completely processed before the second chunk is read from the input
- Stream fusion in the GHC compiler makes it fast

Apparently its pretty simple

On StackOverflow, Magnus Therning asks for an introduction or simple examples for iteratee. The response:

"I learned them by rewriting Oleg's code.

So that would certainly be one path:

implement a left-fold based IO layer."

- Don Stewart (Aug 23 2009)

Magnus then blogs his understanding

And receives a comment:

"If this is all there is to it, why is there so much hype? It's just the standard stream fusion hylomorphism stuff with a left fold."

- Wren Thornton (Aug 27 2009)

Iteratee and Enumerator implementations

The good:

- Memory usage for I/O is now bounded

The bad:

- Correct handling of file handles etc still uncertain
- Error handling still ad-hoc
- Client code relies on inversion-of-control
- Composition still problematic

Example #1 : Enumerator version

Proxy a simple HTTP request:

```
serveRequest ::  Manager -> Request IO
          -> Iteratee ByteString IO Response
serveRequest m req
 = liftIO $
    return $ ResponseEnumerator
           $ \f -> DE.run_
           $ http req (mkBuilder f) m

mkBuilder ::
    (Status -> RespHeaders -> Builder IO b)
    -> Status -> ResponseHeaders
    -> Iteratee ByteString IO b
mkBuilder f s rh
 = DE.joinI (EL.map fromByteString $$ f s rh)
```

Example #1 : Conduit version

Proxy a simple HTTP request:

```
serveRequest ::  Manager -> Request IO
             -> ResourceT IO Response
serveRequest mgr req = do
    Response sc rh body <- http req mgr
    return $ ResponseSource sc rh
           $ fmap fromByteString body
```

Example #2 : Enumerator version

Streaming the HTTP POST body from the client to the server required this helper function:

```
type BString = ByteString

enumIteratee ::  MonadIO m => Int64
 -> (Int -> Iteratee ByteString m BString)
 -> Enumerator BString (Iteratee BString m) c
enumIteratee maxlen takeMax = inner 0
  where
    ...
```

Example #2 : Conduit version

Conduit version did not require **any** helper function!

Advantages of Conduits over Iteratee / Enumerator

- Easier composition
- Better resource handling (ResourceT)