Iteratee IO
safe, practical, declarative input processing

`http://okmij.org/ftp/Streams.html`

Utrecht, NL   December 17, 2009
Updated for the November 2010 version

Outline

▶ **Introduction**

Non-solutions: Handle-based IO and Lazy IO

Pure Iteratees

General Iteratees

Lazy IO revisited

Introduction

A practical alternative to Handle and Lazy IO for input processing

Good performance

Incremental processing, interleaving, low-latency,
block-based i/o from a single buffer
Encouraging performance as compared to C (`libsnd`)

Correctness

No unsafe operations
predictable resource usage, timely deallocation, preventing access to disposed resources; *Haskell98*

Elegance

Arbitrary nesting; vertical, horizontal and parallel combinations; no code bloat

`http://okmij.org/ftp/Streams.html`

This talk

A practical alternative to Handle and Lazy IO for input processing

- Practical talk for (server) developers
- Generalizing from practical experience
  (Web application server, Takusen, WAVE reader)
- Lots of code
- Use Haskell for concreteness
- Code is in *Haskell98*

`http://okmij.org/ftp/Haskell/Iteratee/README.dr`

Running example

```
PUT /file HTTP/1.1crlf
Host: example.comcr
User-agent: Xlf
content-type: text/plaincrlf
crlf
```

Running example

```
PUT /file HTTP/1.1crlf
Host: example.comcr
User-agent: Xlf
content-type: text/plaincrlf
crlf
```

Running example

```
PUT /file HTTP/1.1crlf
Host: example.comcr
User-agent: Xlf
content-type: text/plaincrlf
crlf

1Ccrlf
body line 1lf body line 2crlfcrlf

7crlf
body licrlf

37crlf
ne 3cr body line 4lf body line 5lfcrlf

0crlfcrlf
```

Running example

```
PUT /file HTTP/1.1crlf
Host: example.comcr
User-agent: Xlf
content-type: text/plaincrlf
crlf

1Ccrlf
body line 1lf body line 2crlfcrlf

7crlf
body licrlf

37crlf
ne 3cr body line 4lf body line 5lfcrlf

0crlfcrlf
```

Running example

```
PUT /file HTTP/1.1crlfHost:
```

```
 example.comcrUser-agent: Xlf content-type: text/plaincr
```

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

Outline

Introduction

- **Non-solutions: Handle-based IO and Lazy IO**

Pure Iteratees

General Iteratees

Lazy IO revisited

Non-solutions: Handle-based IO and Lazy IO

```
type Headers = [String]
type ErrMsg  = String

-- The result of reading headers
data HResult = HR Headers                      -- successful
             | HRFail ErrMsg Headers    -- headers so far
```

Code file: `GHCBufferIO.hs`

Using hGetLine, not quite correctly

```
line_read h = doread []
 where
 doread acc = do
  eof <- hIsEOF h
  if eof then return (HRFail "EOF" (reverse acc))
     else do
          l <- hGetLine h >>= return . strip_cr
          if null l then return (HR (reverse acc))
             else doread (l:acc)

strip_cr [] = []
strip_cr s = if last s == '\r' then init s else s
```

Using hGetLine, not quite correctly

```
line_read h = doread []
 where
 doread acc = do
  eof <- hIsEOF h
  if eof then return (HRFail "EOF" (reverse acc))
     else do
          l <- hGetLine h >>= return . strip_cr
          if null l then return (HR (reverse acc))
             else doread (l:acc)

strip_cr [] = []
strip_cr s = if last s == ’\r’ then init s else s
```

Using hGetChar

```
line_read_cr h = doread [] []
 where
 doread acc curr_line = do
  eof <- hIsEOF h
  if eof then return (HRFail "EOF" (reverse acc))
     else hGetChar h >>= check_term acc curr_line
 check_term acc curr_line ’\n’ = finish acc curr_line
 check_term acc curr_line ’\r’ = do
  eof <- hIsEOF h
  if eof then finish acc curr_line
     else do
          c <- hLookAhead h
          when (c == ’\n’) (hGetChar h >> return ())
          finish acc curr_line
 check_term acc curr_line c = doread acc (c:curr_line)
 finish acc "" = return (HR (reverse acc))
 finish acc line = doread (reverse line:acc) ""
```

Using Lazy IO

```
line_lazy h = hGetContents h >>= return . doparse []
 where
 doparse acc str =                  -- pure function
     case break (\c -> c == ’\r’ || c == ’\n’) str of
       (_,"")             -> HRFail "EOF" (reverse acc)
       (l,’\r’:’\n’:rest) -> finish acc l rest
       (l,_:rest)         -> finish acc l rest

 finish acc "" rest = HR (reverse acc)
 finish acc l rest  = doparse (l:acc) rest
```

When are all resources of the Handle `h` freed?

Problems with Handle IO

- It is not that simple
- Handle IO puts the file descriptor in the non-blocking mode:
  not always good for sockets
- Cannot do our own input multiplexing with select/epoll
- Resource leaks, closed handle errors
- Cannot do Handle IO over nested/embedded streams

Problems with Lazy IO

- It is *delusionally* simple
- Theoretical abomination:
  a “pure” computation with observable side-effects
- Permits no IO control
- Practically unacceptable resource management
- Practically unacceptable error reporting
- Danger of deadlocks when reading from pipes

Lazy IO in serious, server-side programming is unprofessional

Outline

Introduction

Non-solutions: Handle-based IO and Lazy IO

▶ **Pure Iteratees**

General Iteratees

Lazy IO revisited

Problems of the exposed traversal state

Handle exposes the (file) traversal state:

- need to pass the Handle around, and explicitly close
- danger of resource leaks or closed-Handle errors
- must check the Handle state on *each* access

Fold

```
fold :: (a -> b -> b) -> b -> IntMap a -> b

fold f z coll ≡ (f $a_n$ ...(f $a_2$ (f $a_1$ z)))

prod = fold (*) 1 coll
    ≡ ($a_n$ * ...($a_2$ * ($a_1$ * 1)))
```

Fold

```
fold :: (a -> b -> b) -> b -> IntMap a -> b
```

$$\texttt{fold f z coll} \equiv \texttt{(f } a_n \texttt{ ...(f } a_2 \texttt{ (f } a_1 \texttt{ z)))}$$

```
prod = fold (*) 1 coll
```

$$\equiv \texttt{(}a_n \texttt{ * ...(}a_2 \texttt{ * (}a_1 \texttt{ * 1)))}$$

```
prodbut n = snd (fold iteratee (n,1) coll)
 where iteratee a (n,s) =
           if n <= 0 then (n,a*s) else (n-1,s)
```

Fold encapsulates the traversal and its resources

Fold

```
fold :: (a -> b -> b) -> b -> IntMap a -> b

fold f z coll ≡ (f a_n ...(f a_2 (f a_1 z)))

prod = fold (*) 1 coll
   ≡ (a_n * ...(a_2 * (a_1 * 1)))

prodbut n = snd (fold iteratee (n,1) coll)
 where iteratee a (n,s) =
          if n <= 0 then (n,a*s) else (n-1,s)
```

Seed exposes the iteratee state
No interface for early termination

Fold

```
fold :: (a -> b -> b) -> b -> IntMap a -> b
fold f z coll ≡ (f a_n ...(f a_2 (f a_1 z)))

prod = fold (*) 1 coll
   ≡ (a_n * ...(a_2 * (a_1 * 1)))

prodbut n = snd (fold iteratee (n,1) coll)
 where iteratee a (n,s) =
          if n <= 0 then (n,a*s) else (n-1,s)
```

Seed exposes the iteratee state
No interface for early termination

Iteratee

```
data Stream = EOF (Maybe ErrMsg) | Chunk String
```

Iteratee

```
data Stream = EOF (Maybe ErrMsg) | Chunk String
```

```
data Iteratee a =
    IE_done a
  | IE_cont (Maybe ErrMsg) (Stream -> (Iteratee a,Stream))
```

Code file: Iteratee.hs

The internal 'state' of the iteratee – the seed – is fully encapsulated.

Simplest Iteratees

```
peek :: Iteratee (Maybe Char)
peek = IE_cont Nothing step
 where
 step s@(Chunk [])     = (peek, s)
 step s@(Chunk (c:_)) = (IE_done (Just c), s)
 step s                = (IE_done Nothing, s)
```

```
head :: Iteratee Char
head = IE_cont Nothing step
 where
 step (Chunk [])     = (head, Chunk [])
 step (Chunk (c:t)) = (IE_done c, (Chunk t))
 step s              = (IE_cont (Just "EOF") step, s)
```

Simplest Iteratees

```
peek :: Iteratee (Maybe Char)
peek = IE_cont Nothing step
 where
 step s@(Chunk [])     = (peek, s)
 step s@(Chunk (c:_)) = (IE_done (Just c), s)
 step s                = (IE_done Nothing, s)
```

```
head :: Iteratee Char
head = IE_cont Nothing step
 where
 step (Chunk [])     = (head, Chunk [])
 step (Chunk (c:t)) = (IE_done c, (Chunk t))
 step s              = (IE_cont (Just "EOF") step, s)
```

Complex Iteratee

```
ie_contM k = (IE_cont Nothing k, Chunk [])

break :: (Char -> Bool) -> Iteratee String

break cpred = IE_cont Nothing (step [])
 where
 step before (Chunk [])  = ie_contM (step before)
 step before (Chunk str) =
     case Prelude.break cpred str of
       (_,[])     -> ie_contM (step (before ++ str))
       (str,tail) -> (IE_done (before ++ str), (Chunk tail)
 step before stream = (IE_done before, stream)
```

Non-trivial state; benefiting from chunked input

Another Complex Iteratee

```
heads :: String -> Iteratee Int

heads str = loop 0 str
 where
 loop cnt ""          = return cnt
 loop cnt str         = IE_cont Nothing (step cnt str)
 step cnt str s@(Chunk "")            = (loop cnt str,s)
 step cnt (c:t) s@(Chunk (c':t')) =
     if c == c' then step (succ cnt) t (Chunk t')
        else (IE_done cnt, s)
 step cnt _ stream                    = (IE_done cnt, stream)
```

Semantics

"abd"... $\ggg$ heads "abc" $\leadsto$ "d"... $\ggg$ done 2

Combining Iteratees

```
instance Monad Iteratee where
    return = IE_done

    IE_done a   >>= f = f a
    IE_cont e k >>= f = IE_cont e (docase . k)
     where
     docase (IE_done a, stream)   = case f a of
         IE_cont Nothing k -> k stream
         i                 -> (i,stream)
     docase (i, s)  = (i >>= f, s)
```

Horizontal Iteratee composition

```
(>>=) :: Iteratee a -> (a -> Iteratee b)
         -> Iteratee b
```

Combining Iteratees

```
instance Monad Iteratee where
    return = IE_done

    IE_done a   >>= f = f a
    IE_cont e k >>= f = IE_cont e (docase . k)
     where
     docase (IE_done a, stream)   = case f a of
         IE_cont Nothing k -> k stream
         i                 -> (i,stream)
     docase (i, s)  = (i >>= f, s)
```

Horizontal Iteratee composition

```
(>>=) :: Iteratee a -> (a -> Iteratee b)
         -> Iteratee b
```

Reading lines

```
type Line = String   -- The line of text, no terminators

read_lines :: Iteratee (Either [Line] [Line])
read_lines = lines’ []
 where
 lines’ acc = break (\c -> c == ’\r’ || c == ’\n’) >>=
      \l -> terminators >>= check acc l
 check acc _  0  = return . Left  . reverse $ acc
 check acc "" _  = return . Right . reverse $ acc
 check acc l  _  = lines’ (l:acc)
 terminators =  heads "\r\n" >>=
   \n -> if n == 0 then heads "\n" else return n
```

Reading lines

```
lines’ acc = break (\c -> c == ’\r’ || c == ’\n’) >>=
     \l -> terminators >>= check acc l
check acc _  0  = return . Left  . reverse $ acc
check acc "" _  = return . Right . reverse $ acc
check acc l  _  = lines’ (l:acc)
terminators =  heads "\r\n" >>=
  \n -> if n == 0 then heads "\n" else return n
```

```
doparse acc str =             -- for comparison
    case break (\c -> c == ’\r’ || c == ’\n’) str of
      (_,"") -> HRFail "EOF" (reverse acc)
      (l,’\r’:’\n’:rest) -> finish acc l rest
      (l,_:rest) -> finish acc l rest
finish acc "" rest = HR (reverse acc)
finish acc l rest  = doparse (l:acc) rest
```

Reading lines

```
lines’ acc = break (\c -> c == ’\r’ || c == ’\n’) >>=
     \l -> terminators >>= check acc l
check acc _  0  = return . Left  . reverse $ acc
check acc "" _  = return . Right . reverse $ acc
check acc l  _  = lines’ (l:acc)
terminators =  heads "\r\n" >>=
  \n -> if n == 0 then heads "\n" else return n
```

```
doparse acc str =                  -- for comparison
    case break (\c -> c == ’\r’ || c == ’\n’) str of
      (_,"") -> HRFail "EOF" (reverse acc)
      (l,’\r’:’\n’:rest) -> finish acc l rest
      (l,_:rest) -> finish acc l rest
finish acc "" rest = HR (reverse acc)
finish acc l rest  = doparse (l:acc) rest
```

Enumerators

```
type Enumerator a    = Iteratee a -> Iteratee a
type EnumeratorM m a = Iteratee a -> m (Iteratee a)
```

Enumerators

```
type Enumerator a    = Iteratee a -> Iteratee a
type EnumeratorM m a = Iteratee a -> m (Iteratee a)
```

```
(>>>):: Enumerator a -> Enumerator a -> Enumerator a
(>>>) = flip (.)

(>>.):: Monad m =>
  EnumeratorM m a -> EnumeratorM m a -> EnumeratorM m a

e1 >>. e2 = \i -> e1 i >>= e2
```

Trivial Enumerators

```
enum_eof :: Enumerator a
enum_eof (IE_cont Nothing k) =
             check . fst $ k (EOF Nothing)
 where
 check i@IE_done            = i
 check i@(IE_cont (Just _) _) = i
 check _ = throwErr "Divergent Iteratee"
enum_eof i = i
```

Trivial Enumerators

```
enum_pure_1chunk :: String -> Enumerator a
enum_pure_1chunk str (IE_cont Nothing k) =
                             fst (k (Chunk str))
enum_pure_1chunk _   iter = iter
```

```
enum_pure_nchunk :: String -> Int -> Enumerator a
enum_pure_nchunk str@(_:_) n (IE_cont Nothing k) =
    enum_pure_nchunk s2 n . fst $ (k (Chunk s1))
 where (s1,s2) = splitAt n str
enum_pure_nchunk _ _ iter = iter
```

File Enumerator

```
enum_fd :: Fd -> EnumeratorM IO a
enum_fd fd iter =
 allocaBytes (fromIntegral buffer_size) (loop iter)
 where
  buffer_size = 5 -- for tests
  loop (IE_cont Nothing k) = do_read k
  loop iter = \p -> return iter
  do_read k p = do
   n <- myfdRead fd p buffer_size
   case n of
    Left errno -> return . fst $ k (EOF (Just "IO error"))
    Right 0    -> return $ IE_cont Nothing k
    Right n    -> do
        str <- peekCAStringLen (p,fromIntegral n)
        loop (fst $ k (Chunk str)) p
```

Block IO; No resource leaks

Reading headers

```
test_driver filepath = do
  fd <- openFd filepath ReadOnly Nothing defaultFileFlags
  result <- fmap run $
            enum_fd fd read_lines_and_one_more_line
  closeFd fd
  print result
 where
  read_lines_and_one_more_line = do
     lines  <- read_lines
     after  <- break (\c -> c == ’\r’ || c == ’\n’)
     status <- is_finished
     return (lines,after,status)
```

Running example

```
PUT /file HTTP/1.1crlfHost:
```

```
 example.comcrUser-agent: Xlf content-type: text/plaincr
```

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

Stream adapters: Enumeratees

```
type Enumeratee a = Iteratee a -> Iteratee (Iteratee a)
```

Stream nesting

- buffering,
- framing,
- character encoding,
- compression, encryption, SSL, etc.

Stream adapters: Enumeratees

```
type Enumeratee a = Iteratee a -> Iteratee (Iteratee a)
```

Stream nesting

- buffering,
- framing,
- character encoding,
- compression, encryption, SSL, etc.

Outer-stream elements to inner-stream elements:
many-to-many

Stream adapters: Enumeratees

```
type Enumeratee a = Iteratee a -> Iteratee (Iteratee a)
```

Stream adapters: Enumeratees

```
type Enumeratee a = Iteratee a -> Iteratee (Iteratee a)
```

Enumeratee is an EnumeratorM in an Iteratee monad

Simplest nesting: framing

```
take :: Int -> Enumeratee a
```

$$b_1 \cdots b_n \ldots \ggg \texttt{take n}\ i \quad \leadsto \quad \ldots \ggg \texttt{done}\ i'$$

where $b_1 \cdots b_n \ggg i \leadsto \_ \ggg i'$

Simplest nesting: framing

```
take :: Int -> Enumeratee a
```

$b_1 \cdots b_n \ldots \ggg \texttt{take}\ \texttt{n}\ i \quad \rightsquigarrow \quad \ldots \ggg \texttt{done}\ i'$
where $b_1 \cdots b_n \ggg i \rightsquigarrow \_ \ggg i'$

Non-law of `take`

```
take n i1 >> take m i2 /= take (n+m) (i1 >> i2)
```

compare:

```
atomically (m1 >> m2) /= atomically m1 >> atomically m2
round (x1 + x2)       /= round x1 + round x2
```

Simplest nesting: framing

```
take :: Int -> Enumeratee a

take 0 iter@IE_cont         = return iter
take n (IE_cont Nothing k)  = IE_cont Nothing (step n k)
 where
 step n k (Chunk []) = ie_contM (step n k)
 step n k chunk@(Chunk str) | length str < n =
   (take (n - length str) . fst $ (k chunk), Chunk [])
 step n k (Chunk str) =
   (IE_done (fst $ k (Chunk s1)), (Chunk s2))
  where (s1,s2) = splitAt n str
 step n k stream = (IE_done (fst $ k stream), stream)
take n iter        = drop n >> return iter
```

Chunk decoding

- $\texttt{"0"}\ \texttt{CRLF}\ \texttt{CRLF}\ \ldots \ggg \texttt{enum\_cd}\ i \quad \leadsto \quad \texttt{done}\, i$
- $n_{hex}\ \texttt{CRLF}\ b_1 \cdots b_n\ \texttt{CRLF}\ \ldots \ggg \texttt{enum\_cd}\ i \quad \leadsto$
  $\ldots \ggg \texttt{enum\_cd}\ i'$
  where $b_1 \cdots b_n \ggg i \quad \leadsto \quad \_ \ggg i'$

Chunk decoding

```
enum_chunk_decoded :: Enumeratee a
enum_chunk_decoded iter = read_size
 where
 read_size = break (== ’\r’) >>=
             checkCRLF iter . check_size
 checkCRLF iter m = do
   n <- heads "\r\n"
   if n == 2 then m else frame_err "..." iter
 check_size "0" = checkCRLF iter (return iter)
 check_size str@(_:_) =
     maybe (frame_err "Chunk size" iter) read_chunk $
     read_hex 0 str
 check_size _ = frame_err "Error reading chink size" iter

 read_chunk size = take size iter >>= \r ->
   checkCRLF r $ enum_chunk_decoded r
```

Complete test

```
test_driver filepath = do
  fd <- openFd filepath ReadOnly Nothing defaultFileFlags
  result <- fmap run (enum_fd fd read_headers_body)
  closeFd fd
  print result
 where
  read_headers_body = do
     headers <- read_lines
     body    <- return . run =<<
                  enum_chunk_decoded read_lines
     status  <- is_finished
     return (headers,body,status)
```

Running example

```
PUT /file HTTP/1.1crlfHost:
```

```
 example.comcrUser-agent: Xlf content-type: text/plaincr
```

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

Outline

Introduction

Non-solutions: Handle-based IO and Lazy IO

Pure Iteratees

▶ **General Iteratees**

Lazy IO revisited

General Streams and Iteratees

```
data Stream el = EOF (Maybe ErrMsg) | Chunk [el]

data Iteratee el m a =
     IE_done a
   | IE_cont (Maybe ErrMsg)
             (Stream el -> m (Iteratee el m a, Stream el))
```

```
instance Monad m => Monad (Iteratee el m)
instance MonadTrans (Iteratee el)
```

Code file: `IterateeM.hs`

Sample General Iteratees

```
head  :: Monad m => Iteratee el m el
break :: Monad m => (el -> Bool) -> Iteratee el m [el]


dropWhile :: Monad m =>
  (el -> Bool) -> Iteratee el m ()

drop  :: Monad m => Int -> Iteratee el m ()
line  :: Monad m => Iteratee Char m (Either Line Line)


stream2list :: Monad m => Iteratee el m [el]
print_lines :: Iteratee Line IO ()
```

General Enumerators

```
type Enumerator el m a =
      Iteratee el m a -> m (Iteratee el m a)
```

General Enumerators

```
type Enumerator el m a =
      Iteratee el m a -> m (Iteratee el m a)
```

Why not the following type?

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

Troublesome code:

```
  do let iter = enum_file file1 iter_count
     some_action
     run (enum_file file2 iter)
```

General Enumerators

```
type Enumerator el m a =
      Iteratee el m a -> m (Iteratee el m a)
```

Why not the following type?

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

Troublesome code:

```
  do let iter = enum_file file1 iter_count
     some_action
     run (enum_file file2 iter)
```

General Enumerators

```
type Enumerator el m a =
      Iteratee el m a -> m (Iteratee el m a)

(>>>):: Monad m =>
  Enumerator el m a -> Enumerator el m a ->
  Enumerator el m a
-- (>>>) = flip (.)
e1 >>> e2 = \i -> e2 =<< (e1 i)
```

Sample General Enumerators

```
enum_eof :: Monad m => Enumerator el m a

enum_fd :: Fd -> Enumerator Char IO a
```

Sample General Enumeratees

```
type Enumeratee elo eli m a =
   Iteratee eli m a -> Iteratee elo m (Iteratee eli m a)
```

```
take :: Monad m => Int -> Enumeratee el el m a
enum_chunk_decoded :: Monad m => Enumeratee Char Char m a
```

Enumeratee is an `Enumerator eli m a` in an `Iteratee elo m` monad

Sample General Enumeratees

```
type Enumeratee elo eli m a =
   Iteratee eli m a -> Iteratee elo m (Iteratee eli m a)
```

```
take :: Monad m => Int -> Enumeratee el el m a
enum_chunk_decoded :: Monad m => Enumeratee Char Char m a
```

Enumeratee is an `Enumerator eli m a` in an `Iteratee elo m` monad

```
runI :: Monad m => Iteratee eli m a -> Iteratee elo m a
runI = lift . run
```

More interesting Enumeratees

```
map_stream :: Monad m =>
   (elo -> eli) -> Enumeratee elo eli m a

enum_lines :: Monad m => Enumeratee Char Line m a

sequence_stream :: Monad m =>
   Iteratee elo m eli -> Enumeratee elo eli m a
```

True IO interleaving

```
line_printer = enum_lines print_lines
```

```
print_headers_print_body = do
     lift $ putStrLn "Lines of the headers follow"
     line_printer
     lift $ putStrLn "Lines of the body follow"
     runI =<< enum_chunk_decoded line_printer

test_driver_full iter fpath = do
  fd <- openFd fpath ReadOnly Nothing defaultFileFlags
  run =<< enum_fd fd iter
  closeFd fd; putStrLn "Finished reading"

test_driver_mux iter fpath1 fpath2 = do ...
```

Outline

Introduction

Non-solutions: Handle-based IO and Lazy IO

Pure Iteratees

General Iteratees

▶ **Lazy IO revisited**

Lazy IO vs. Iteratee IO

```
driver1 (i:j:rest) =
   print (max_cycle_len i j) >> driver1 rest
driver1 _ = return ()
main1 = getContents >>= driver1 . map read . words
```

Code file: `GetContentsLess.hs`

Lazy IO vs. Iteratee IO

```
driver1 (i:j:rest) =
   print (max_cycle_len i j) >> driver1 rest
driver1 _ = return ()
main1 = getContents >>= driver1 . map read . words

driver2 = do
          i <- head; j <- head
          lift (print (max_cycle_len i j)) >> driver2
main2 = run =<< enum_file "/dev/tty"
          (enum_words . map_stream read $ driver2)
```

Code file: `GetContentsLess.hs`

Binary and random IO

RandomIO.hs

Reading 16- or 32-bit signed and unsigned integers in big- or little-endian formats;
Seeking within a file

Tiff.hs

An extensive example of:

- random and binary IO;
- on-demand incremental processing with iteratees.

Conclusions

Iteratee IO: *safe* and *practical* alternative to Lazy and Handle IO

- Compositionality
  - Iteratees compose horizontally as monads
  - Iteratees compose vertically:
    nesting, embedded stream processors
  - Iteratee compose to process the same stream in parallel, or two streams in parallel
  - Enumerators are iteratee transformers,
    compose as functions
- Good resource management
- Good error handling
- Inherent incremental processing
- Safe IO interleaving
- Based on left fold, for any FP language

Good performance, Correctness, Elegance