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

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`

We introduce input processing with left-fold enumerator – Iteratee IO – as a safe, declarative and practical alternative to Handle and Lazy IO for input processing. The approach is general and applies to processing of data taken from in-memory structures, databases, files, sockets, etc. Binary and random IO is supported. Our approach is incremental, permitting i/o interleaving. We shall see an example of i/o multiplexing without using threads and in no danger of race conditions. Unlike lazy IO, our approach is *correct*. There is not even hint of UnsafePerformIO. Accessing a disposed resource like a closed handle is impossible in our approach. Our approach permits composing streams and stream processors: the same processor can handle several streams one after another. Or two processors can be applied in succession to the same source. Processors can be combined 'vertically,' so to deal with streams that are chunk-encoded, escaped, UTF8- or otherwise encoded or nested into other streams. Processors and streams can be combined in parallel: a stream can feed several processors in parallel, or the same processor can take data from several streams.

Enumerators and iteratees, which generalize fold, have useful algebraic properties. But we won't talk about them here.

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`

This talk is aimed at practitioners, in particular, server programmers. That is, programmers who write long-running distributed applications and are painfully aware of the issues of reading from sockets, latency, buffering, many layers of decoding, proper resource disposal and sustaining high load. In short, anyone who programmed a network server, a database back-end, or a high-volume data format transcoder. The technique has been validated in a production web-application server written in Haskell, database access library Takusen, TIFF and WAVE file readers. The performance of the Iteratee library on reading WAVE files can exceed the performance of the C-based libsnd library.

All the code is available on-line. You are welcome to download it, even now, and follow along.

Running example

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

Our running example is processing of an HTTP client request: a POST/PUT request. Typically, the processing is some kind of parsing. For illustration we use a simple but realistic line parser. In part 1, we read a sequence of lines from the input until the empty line is encountered. We return the list of the read lines.

For simplicity all the discussed code uses file IO rather than network sockets. All the code is in Haskell98 and can be run in any Haskell compiler. Since only file-based IO will be demonstrated, the operating system does not matter either.

Running example

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

Lines could be terminated by a CRLF combination and also by a single CR or LF. For robustness, we must handle all three line terminators. HTTP generally permits lines within the entity terminated with LF, CR or 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
```

Part 2 of the running example is reading the headers and reading all the lines from the HTTP-chunk-encoded content that follows the headers. Part 2 thus verifies layering of streams, and processing of one stream embedded (chunk encoded) into another stream.

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 5lf crlf

0crlfcrlf
```

Chunks are not aligned at all with the boundaries of the lines of the embedded content. A line may start in one chunk and be terminated in another.

Running example

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

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

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

For efficiency, we read by blocks. Blocks, too, cut arbitrarily across line boundaries. Blocks may cut across CR LF, across chunks and chunk length data.

All this could be embedded into another stream, such as an SSL stream or a message/http.

Outline

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`

We start with the conventional implementation, for the sake of comparison. We only tackle part 1 of the problem: reading headers. `HResult` is the type of the result of the function we are about to write. The code is in the 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
```

We must not forget that GHC does not count CR as the line terminator! So the read string may contain CR at the end, which we must strip off. This is an expensive process producing much garbage. The code may appear simple, but it is not quite correct: it can’t handle lines that are terminated by a single CR. Also, `hGetLine` can’t tell the last incomplete line from the last terminated line.

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) ""
```

This code does solve the whole problem, handling all three CR, LF and CRLF as terminators. There are no obvious errors in the code – there is nothing obvious about the code at all. The code is quite imperative and ugly though. It barely fits on the slide.

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?

Pattern-matching makes for a convenient parsing. We could have used `Prelude.lines`; but the latter can't handle CRLF, and can't tell if the last line was terminated or not. We can't do any IO on the handle afterwards: it is closed, or semi-closed. In the above case, we really don't know when it is going to be closed: whenever the garbage collector collects the non-yet-read portion of string and finalizes it. That event may never occur. We can't use such reckless resource management in any serious programming.

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

As we saw, the proper solution that accounts for all three terminators is not that simple. It is imperative, hard to reason about and hard to see its correctness.
Handle corresponds to a resource: an open file descriptor, an IO buffer. A handle may be open in one function and passed around, to be closed in other functions. It's hard to see the dynamic extent of the resource use. It is quite possible to attempt to read from a closed handle, which is akin to reading from the null pointer. We should strive to avoid such errors in the production code.

Handle IO provides buffering, but only over real file descriptors. We can't create a handle over a 'synthetic' file descriptors, such as embedded streams: see SSL or message/http embedding above. Handle IO cannot be 'nested'.

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

I can talk a lot how disturbingly, distressingly wrong lazy IO is theoretically, how it breaks all equational reasoning. Lazy IO entails either incorrect results or poor optimizations. But I won't talk about theory. I stay on practical issues like resource management. We don't know when a handle will be closed and the corresponding file descriptor, locks and other resources are disposed. We don't know exactly when and in which part of the code the lazy stream is fully read: one can't easily predict the evaluation order in a non-strict language. If the stream is not fully read, we have to rely on unreliable finalizers to close the handle. Running out of file handles or database connections is the routine problem with Lazy IO. Lazy IO makes error reporting impossible: any IO error counts as mere EOF.
It becomes worse when we read from sockets or pipes. We have to be careful orchestrating reading and writing blocks to maintain handshaking and avoid deadlocks. We have to be careful to drain the pipe even if the processing finished before all input is consumed. Such precision of IO actions is impossible with lazy IO. It is not possible to mix Lazy IO with IO control, necessary in processing several HTTP requests on the same incoming connection, with `select` in-between.
I have personally encountered all these problems. Leaking resources is an especially egregious and persistent problem. All the above problems frequently come up on Haskell mailing lists.

Outline

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

We observe that resource problems of the Handle-based IO occur because the state of the file traversal is exposed as a Handle. We have to explicitly close the handle. Do it too late, we leak resources. Do it too early, we get the null pointer – closed handle – errors. The sheer number of internet security advisories concerning memory allocation problems indicates that manual management of resources is greatly error-prone. We also have to check the state of the handle – valid, not EOF – at each and every operation on the handle. But there is another way. The state of the traversal can be encapsulated rather than exposed.

Fold

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

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

prod = fold (*) 1 coll
    $\equiv$ (a_n * ...(a_2 * (a_1 * 1)))

As an example, let’s look at a Haskell collection, say, an IntMap. It provides fold, which takes a seed and another function. We shall call it iteratee, because it is being iterated upon each element of the collection. The fold passes the iteratee the initial seed and the first element of the collection. The result is the new seed, to be passed again to the iteratee along with the second element of the collection, etc. After the iteratee has been applied to all elements of the collection, the final seed is returned. Here is an example, computing the product of all elements of the collection of numbers. The seed is the product so far, originally 1. The iteratee multiplies the current seed with the current element, giving the new current product.

Fold

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

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

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

   $\equiv$ (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)
```

Fold encapsulates the traversal and its resources

We can do more interesting things: for example, we may want to skip the first n elements of the collection and compute the product of the rest. Our seed is the pair: the current product, initially 1, and the number of elements yet to skip. The iteratee accumulates the product only after the skipping is done. At the end, we extract the desired product, the second component of the final seed.
We may see that fold is a powerful pattern; lots of papers have been written about it, and I greatly encourage you to read all of them. We don't have time, alas, to talk about its wonderful properties.
We have to talk about practical things. For example, we have no 'traversal' handle. We never had to check if the traversal of the IntMap finished or not. Fold traverses the collection in some way and merely gives the iteratee each encountered element along with the seed. The seed is opaque to to fold. Fold simply passes the seed from one invocation of iteratee to another, and, finally, returns as the result. We see the separation of concerns: fold cares about traversal, allocating resources at the beginning and freeing them at the end. Iteratee cares about processing elements; it need not be concerned about deallocating resources at the end.

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

But the separation of concerns isn't perfect. Although fold is indeed being treated as an abstract enumerator of a collection, iteratee is not being treated as a black-box. The state of the iteratee, its seed, is completely exposed. We see its structure and the components such as n, which are only used internally. Also, the definition of iteratee and the definition of the initial seed are separated. The `iteratee` here can be defined in a separate file. Imagine all the changes we have to make if we change our definition so that the the order of the components in the seed is switched. We'd like to treat iteratee *along* with its seed, and avoid exposing its internal data.

There is the second problem with the above interface. If the current element of the collection is 0, the iteratee can terminate product accumulation. There is no point of further traversal of the collection. Alas, there is no way for iteratee to tell fold that the iteratee is finished and is not interested in further traversal. We see that the traversal interface can be better.

Fold

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

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

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

   $\equiv$ (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
```

Let us design a better interface. For simplicity, in this part of the talk we assume that the collection to traverse is made of characters – such as a string or a file.

Before, an iteratee received the current element of the collection. We'd like our iteratee to handle more than one element, if so immediately available. That greatly improves the efficiency: think of block-based IO rather than character IO. The traversal may encounter an error. Since we wish iteratee encapsulated its internal state, we need to explicitly tell the iteratee that the traversal is finished and it should produce the final answer. So, our iteratee receives not a single element but this value, a Stream. The first variant indicates the termination of the traversal. `Chunk str` gives the immediately available characters. The traversal is not terminated yet.

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.

Here’s our iteratee. In the ’done’ state, it contains the computed result. In the ’cont’ state, the iteratee has not finished the computation and needs more data. When the iteratee gets more data, a chunk, it consumes (some of) them, moving to another state and returning the unconsumed part of the chunk, if any. There is no mentioning of seed here: it is fully encapsulated.
The ’cont’ state looks pretty much like the state monad, doesn’t it?
The ’cont’ state is also used to send an error or other message to the stream producer (e.g., to rewind the stream). The error is restartable: if the producer fixed the error, it replies with a chunk and so resumes the processing.

We assume that all iteratees are ‘good’ – given bounded input, they do the bounded amount of computation and take the bounded amount of resources. We also assume that given a terminated stream, an iteratee moves to the done state, so the results computed so far could be returned.

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)
```

Let’s write some iteratees. The simplest one simply peeks at the current element, without removing it from the stream. After peeking at the element, or determining that it will never be available because the stream is terminated, the Iteratee moves to the done state. The *state* of stream is not affected: the received stream is returned as it is. A `Chunk` may contain the empty string: it means that no elements are currently available, but the stream is not yet exhausted. In that case, we remain in the existing state, waiting for something to become available.

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)
```

The head iteratee is similar, only it does remove the current element from the stream, acting as a stream deconstructor. As another difference from peek, head reports an error if the stream is terminated and so has no current element. The error is restartable however. It propagates to the producer of data. If the producer finds a new source of data, it would send a new chunk in response to the error, effectively resuming processing and recovering from the error. If the producer cannot or would not handle the EOF error, the error would ‘automatically’ propagate up, as we shall see soon.

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

Not all iteratees are so trivial. Here is a more complex one. Whereas `head` was akin to `List.head`, this one is the analogue to the `List.break` function from the Prelude. It takes the break predicate and returns a string of characters, which is the (possibly empty) prefix of the stream. None of the characters in the string prefix satisfy the character predicate. If the stream is not terminated, the first character of the remaining stream satisfies the predicate.
This iteratee has a non-trivial state: the list of characters read so far, none of which satisfy the break predicate. This iteratee also takes advantage of the chunked input.

The helper function `ie_contM` represents the common pattern of an iteratee consuming the whole chunk and wanting more. It simplifies writing Iteratees.

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" $\rightsquigarrow$ "d"... $\ggg$ done 2

Let me mention another parsing combinator, I mean, iteratee, which turns out awfully convenient in practice.
Given a sequence of characters, we attempt to match them against the characters on the stream, returning the count of how many characters have matched. The matched characters are removed from the stream.
For example, if the stream contains ”abd”, then (heads ”abc”) will remove the characters ”ab” and return 2.

The notation $s \ggg i \rightsquigarrow s' \ggg i'$ means that upon ingesting the prefix of the stream s the iteratee i moved to the state i' with s' part of the stream remaining. Often, i' is `done` v.

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
```

Our running example was to read lines. We would like to somehow *combine* the above iteratees to read lines. Perhaps you won't be surprised that iteratees combine, well, like a monad. You don't need to know anything about monad. This scary word simply means that if we have one iteratee that produces a value and the rest of the stream, and another iteratee to handle the rest of the stream, we can combine them to make a bigger iteratee. This infix operator `(>>=)` makes the composition.

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
```

The last line of `case f a of` describes the error propagation. Error also propagates in the `IE_cont` case. So the Iteratee is not only a monad but a Failure monad.

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
```

This is the Iteratee IO solution to the problem of reading headers. We combine the iteratees to read a sequence of lines up to the empty line. A line can be terminated by CR, LF or CRLF. We return the read lines, in order, not including the terminating empty line. Upon EOF or a stream error, we return the complete, terminated lines accumulated so far, in the `Left` alternative.

The code is the combination of other iteratees; there is no longer any mentioning of streams.

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
```

For comparison, here is a similar function from the Lazy IO code. The parsing is very similar: find the break character, check what it is, and if it is CR, look ahead to the next character and check if it is LF. The iteratee version has no `rest`. Iteratee does not deal with the future, only with the present.

Count the number of lines of code! It is 7 in both cases. An alternative to Lazy IO can be just as compact!

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
```

Unlike Lazy IO, the iteratee now distinguishes the stream EOF from the stream error. The error is a part of `IE_cont`, and it is propagated transparently.

Enumerators

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

This was the story about iteratees, but what about fold – the one that takes our iteratee and, well, iterates it upon the collection? We shall call such a procedure enumerator. Enumerator takes the iteratee, applies it to each element of the collection until the collection is exhausted or the iteratee said it had enough. And then enumerator returns the result. Which is, well, the final value of the iteratee. So, enumerator is the *iteratee transformer*.

For the time being, our Iteratee were designed to have no effects. Enumerators may have effects, for example, to read from a file. Hence we also need `EnumeratorM`. We soon get rid of that asymmetry.

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
```

Obviously as Iteratee transformers, enumerators can be composed – just like functions. The composition means: iterate the iteratee upon the first collection. And then iterate over the second collection. Thus the ordinary functional composition of enumerators corresponds to concatenation, so to speak, of their collections. We can use the same iteratee to process data from a string followed by data read from a file followed by data received from a socket. And iteratee could not tell from which collection the character came from – not does the iteratee care.

We use `(>>>)` for left-to-right composition; such an operator, in the more general case of categories, is defined in Control.Category.

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
```

Here is the most primitive enumerator: it applies the iteratee to the terminated stream. It could be written simpler, but I want to report an error if an iteratee is bad and didn’t move to the done state upon receiving the EOF.

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
```

The pure 1-chunk enumerator passes a given string to the iteratee in one chunk. We see the commonly occurring pattern in writing enumerators: if the iteratee wants more data, we give them to it. If the iteratee does not want more (it is done or reporting an error), we return the iteratee as it was. The pure 1-chunk enumerator does no IO and is useful for testing of base parsing.

The pure n-chunk enumerator passes the given string to the iteratee in chunks of size n. It is useful for testing of handling of chunk boundaries.

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

Finally an interesting enumerator, which reads a file by blocks. It uses a single IO buffer, which it allocates at the beginning and frees at the very end. All the allocation and deallocation is contained within the enumerator code. We know exactly when the clean-up occurs. There can't be any 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)
```

We come back to the running example, part 1: We read lines, terminated by the empty line, and one extra line. I should remind how the input looks like. Here are block boundaries, cutting across headers and line terminators. And we did not have to care about any of that!

Running example

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

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

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

For efficiency, we read by blocks. Blocks, too, cut arbitrarily across line boundaries. Blocks may cut across CR LF, across chunks and chunk length data.

All this could be embedded into another stream, such as an SSL stream or a message/http.

Stream adapters: Enumeratees

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

Stream nesting

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

Stream adapters, or Enumeratees, handle nested – encapsulated – streams. Stream nesting is rather common: buffering, character encoding, compression, encryption, SSL are all examples of stream nesting. On one hand, an Enumeratee is an Enumerator of a nested stream: it takes an iteratee for a nested stream, feeds its some data, returning the resulting iteratee when the nested stream is finished or when the iteratee is done. On the other hand, an Enumeratee is an Iteratee for the outer stream, taking data from the parent enumerator. One can view an Enumeratee as a AC/DC or voltage converter, or as a ‘vertical’ composition of iteratees (compared to monadic bind, which plumbs two iteratees ‘horizontally’).

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

In the trivial case (e.g., Word8 to Char conversion), one element of the output stream is mapped to one element of the nested stream. Generally, we may need to read several elements from the outer stream to produce one element for the nested stream. Sometimes we can produce several nested stream elements from a single outer stream element.

Stream adapters: Enumeratees

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

That many-to-many correspondence between the outer and nested streams justifies the type of the enumeratee. Suppose that the enumeratee has received EOF on its, that is, the outer stream. The enumeratee, as the outer iteratee, must move to the Done state. Yet the nested iteratee is not finished. The enumeratee then has to return the nested iteratee as its result.

Stream adapters: Enumeratees

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

Enumeratee is an EnumeratorM in an Iteratee monad

If we look at the type of the Enumeratee carefully we see that it is EnumeratorM, where monad m is chosen to be Iteratee. That explains that Enumeratee acts as an enumerator to the inner iteratee, but obtains data from an outer stream.

Simplest nesting: framing

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

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

One of the simplest Enumeratees is `take`. The nested stream is a prefix of the outer stream of exactly n elements long. Such nesting arises when several independent streams are concatenated.

We read `n` elements from a stream and apply the given (inner) iteratee to the stream of the read elements. Unless the stream is terminated early, we read exactly `n` elements (even if the inner iteratee has accepted fewer).

Simplest nesting: framing

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

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

The definition of take implies the take non-law. It should not surprise us given the non-law of atomic transactions in the STM monad, or the non-law of rounding. All three non-laws express the significance of transaction boundaries.

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
```

And here is the code.

Chunk decoding

- `"0" CRLF CRLF` $\ldots \ggg$ `enum_cd` $i \quad \rightsquigarrow \quad$ `done` i
- n_{hex} `CRLF` $b_1 \cdots b_n$ `CRLF` $\ldots \ggg$ `enum_cd` $i \quad \rightsquigarrow$
  $\ldots \ggg$ `enum_cd` i'
  where $b_1 \cdots b_n \ggg i \quad \rightsquigarrow \quad \_ \ggg i'$

Here is the HTTP chunk-decoding specification, in our notation.

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
```

And here is the corresponding implementation.

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)
```

Here is the complete running example: reading the lines of the headers, and reading the lines of the chunk-encoded body. We use exactly the same iteratee to read the lines: (i) from the original collection, file; (ii) and from the nested and encoded collection, chunk-encoded body. The complete test is at the end of the file `Iteratee.hs`; please try it.

Recall what the input looks like: the IO buffer cuts across headers, chunks and line terminators.

Running example

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

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

```
lfcrlf1Ccrlfbody l
```

```
ine 2crlfcrlf7
```

For efficiency, we read by blocks. Blocks, too, cut arbitrarily across line boundaries. Blocks may cut across CR LF, across chunks and chunk length data.

All this could be embedded into another stream, such as an SSL stream or a message/http.

Outline

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`

We have talked about pure Iteratees, which process and collect their inputs but can't do side effects themselves. Clearly for incremental IO, we'd like to be able to write out the results as soon as we have computed them. We need iteratees that can do at least IO.
We also generalize streams to deliver arbitrary elements rather than just characters.

Iteratee is a generic stream processor, what is being folded over a stream. It now takes this general stream. The new iteratee also can do side-effects, in a monad `m`. The iteratee is also a monad and a monad transformer. Again that is not so surprising given that the last argument of `IE_cont` is `StateT el m (Iteratee el m a)`.

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 ()
```

Here are a few sample Iteratees. First are the ones we have seen. The code is virtually the same as before; only types are more general. The Iteratee `dropWhile` is essentially `break` with the inverse break predicate.

A pure iteratee `stream2list` is quite useful in unit and interactive tests, to 'show' a stream. The iteratee `print_stream` is the first effectful iteratee.

General Enumerators

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

The type of the enumerator is also more general.

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)
```

Why can't we define enumerators with the type `Iteratee m a -> Iteratee m a`? Actually, we can. We indeed can use the regular functional composition to compose Enumerators. The approach, albeit attractive and successful, is problematic.

Consider the following code where `iter_count` is, for example, an iteratee that returns the count of items in the input stream. The code returns the combined count of characters in `file1` and `file2`. It indeed does that. The question is: when exactly `some_action` is performed relative to the opening and closing of `file1`? That is, is `some_action` done before `file1` is opened? A more important question: is `file1` opened before `file2`?

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)
```

The answer to the first question is clear: `some_action` is done before `file1` is opened. The result of `(enum_file file1 iter_count)` is a pure value `Iteratee m a`. That value encapsulates an action but the action is not performed yet. The question about the order of opening of `file1` and `file2` cannot be answered from the above code. The value `iter` encapsulates the action of opening and closing `file1`. If the enumerator `enum_file` executes the action in `iter` as the very first thing, then `file1` will be opened before `file2`. But nothing forces `enum_file` to behave this way; it may open `file2` before checking the result of `iter`. Then `file2` would be opened before `file1`. We have lost the precise control over action sequencing. In pure computations, that is no problem: the results are the same either way. Effects however demand precision on the sequence of actions. When enumerator has the type `Iteratee m a -> m (Iteratee m a)` then there is no longer uncertainty about the order of opening the files. Since enumerator takes `Iteratee` but produces the monadic value `m (Iteratee m a)`, we have to run the action to get `Iteratee m a`, in order to pass to the next enumerator. It is the type system that forces the sequencing on us. That property is well worth preserving.

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)
```

However, our choice for the type of the enumerator imposes a slight burden, of using the operation =<<, the flipped monadic bind, which can be regarded as a sort of ‘call-by-value application’. We will see many occurrences of such an operation. This is the standard monadic operation; one may use bind too.

To compose enumeratees, we have to use the monadic composition rather than the ordinary functional composition. We have to use bind. The types practically force on us the implementation.

Sample General Enumerators

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

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

There is nothing to tell here but to show these general types. The code for these enumerators remains the same.

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

With more general types, the story becomes much more interesting. The code for take and `enum_chunk_decoded` is almost identical to the one shown earlier. But the types are more general.

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
```

Again the types show that an Enumeratee is the Enumerator in the Iteratee monad. The function `runI` is a ‘variant’ of run in the `Iteratee elo m` monad. It is used to terminate (send EOF to) the inner Iteratee and return the result in the outer Iteratee.

We shall see many occurrences of `runI` below. The function `runI` can be defined like lifted run. The real implementation is obtained from the above by inlining.

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
```

More general types of enumeratees let us write more general functions, such as `map_stream`, which transforms the stream of elements `elo` to the stream of elements `eli` and applies the given iteratee to this nested stream. The code is simple, just as you can expect from map.

We generalized line reader to be another stream transformer: from a stream of characters to a stream of lines. We can either accumulate and return all lines as we did before, using the *pure* iteratee `stream2list`, or print the lines just as we receive them.

The transformer `map_stream` maps one element of the outer stream to one element of the nested stream. The transformer `sequence_stream` is more general: it may take several elements of the outer stream to produce one element of the inner stream. The transformation from one stream to the other is specified as `Iteratee elo m eli`. This is a generalization for `Monad.sequence`.

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 ...
```

The simple iteratee `line_printer` used to process a variety of streams: embedded, interleaved, etc. And here how we use it: we read the headers and print each header right after it has been read. We then read the lines of the chunk-encoded body and print each line as it has been read. We use standard Haskell application and bind operations, to feed iteratees to enumerators and to extract the results. Demo: `testm1`, read and print the headers, and then stop after the the empty line. We don't read the whole stream. With `testm2`, we read and print the headers and the body. We show embedded stream: chunk-encoded body.

Running these tests demonstrates true interleaving, of reading from the two file descriptors and of printing the results. All IO is interleaved, and yet it is safe. There are no unsafe operations.

Outline

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`

Finally, let us briefly revisit lazy IO. The great part of its attraction is writing IO processors by composition, like in this sample code. It reads a pair of integers from the standard input, evaluates a pure function on these two arguments, prints the result, and awaits more input.

The same program can be implemented safely with the predictable resource usage. We use the same processing function `max_cycle_len`. We merely replace the driver and the main function. The number of lines of code stays the same!

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`

The function `main2` converts a stream of characters to the stream of integers. Just like `main1`, it does the conversion by applying a sequence of transformers. In `main1`, the stream transformers were composed via `(.)`. The composition operation remains the same, only now we build the transformers inside out rather than outside in. In both cases, the composition takes one line. Incidentally, the code for `enum_words` is quite similar to the code for `Prelude.words`, see `IterateeM.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.

We briefly mention binary and endian IO, with applications to reading TIFF and WAVE files.

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

We have achieved both high performance and encapsulation of input processing layers that can be freely composed. The technique has been validated in a production web-application server written in Haskell and database access library Takusen.
All IO is interleaved, and yet it is safe. No unsafe operations are used.
Separation of concerns: The enumerator knows how to get to the next element; the Iteratee knows what to do with the next element.
The left-fold enumerator as a general concept has been used in other functional languages.