Iteratees

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

FLOPS 2012
Kobe, Japan May 24, 2012

Iteratee IO is a style of incremental input processing with precise resource control. The style encourages building input processors from a user-extensible set of primitives by chaining, layering, pairing and other modes of compositions. The programmer is still able, where needed, to precisely control look-ahead, the allocation of buffers, file descriptors and other resources. The style is especially suitable for processing of communication streams, large amount of data, and data undergone several levels of encoding such as pickling, compression, chunking, framing. It has been used for programming high-performance (HTTP) servers and web frameworks, in computational linguistics and financial trading.

We exposit programming with iteratees, contrasting them with Lazy IO and the Handle-based, stdio-like IO. We relate them to online parser combinators. We introduce a simple implementation as free monads, which lets us formally reason with iteratees. As an example, we validate several equational laws and use them to optimize iteratee programs. The simple implementation helps understand existing implementations of iteratees and derive new ones.

What are iteratees and how many of them

Google search on 'iteratee'

Just what are Iteratees and how many? Let's ask Google. Google also shows the related search terms: 'haskell iteratee tutorial', 'iteratee scalaz', 'iteratee play', 'iteratee clojure'. Iteratee is a new word, so all these references do bear on the subject of the talk. Iteratee seem to have something to do with Web frameworks (and what isn't nowadays), and they are supposed to be difficult to understand. There are also lots of flavors and variations. I can't comment on all or even several of them for the lack of time, or familiarity. What I can try to do is to give an overview and a few general principles, and show how to reason with Iteratees. You can then write your own implementation or explanation, adding to the Google count.

Outline

1. What is Iteratee IO
   contrast with Lazy and Handle IO
2. Can you reason with it
   online parser combinators and equational laws
3. Concluding remarks

We first describe Iteratee IO intuitively, and then formally.

A veneer and the real thing

```
import  IterateeM

type R e m          = ...
newtype L e m1 m2 = ...
```

We use an 'industrial strength' IterateeM library, one of many iteratee libraries. I put a simple veneer, simpler than the one in the paper. The types R and L, which we will see in abundance below, and functions of that type belong to the veneer, which is a wrapper around the IterateeM library. The veneer can be adjusted for other libraries.

Count whitespace: Lazy

$$
\begin{aligned}
&\mathsf{cws}_\ell \;::\; \mathsf{String} \;\to \mathsf{Int} \\
&\mathsf{cws}_\ell \;\texttt{""} \;=\; 0 \\
&\mathsf{cws}_\ell \;(\mathsf{c}\colon \mathsf{str}\,) \;\mid\; \mathsf{isSpace}\;\; \mathsf{c} \;=\; 1 + \mathsf{cws}_\ell\;\; \mathsf{str} \\
&\mathsf{cws}_\ell \;(\_\colon \mathsf{str}\,) \;=\; \mathsf{cws}_\ell\;\; \mathsf{str}
\end{aligned}
$$

$$
\mathsf{run}\;\; \mathsf{fname} \;=\; \mathsf{readFile}\;\; \mathsf{fname} \gg\!= \mathsf{print} \circ \mathsf{cws}_\ell
$$

We start with a trivial example: counting white space characters in a file. We first show the Lazy IO example in idiomatic Haskell, with the standard pattern-matching on a string. (Actually, we could do better, and we will.) If the first character of the string is the whitespace, the total number of whitespace is one more than the number of whitespace characters in the tail of the string. The code *clearly* expresses the pure, mathematical function. There is no IO.

Count whitespace: Lazy

$$
\begin{aligned}
&\mathsf{cws}_\ell \;::\; \mathsf{String} \;\to \mathsf{Int} \\
&\mathsf{cws}_\ell \;\text{""} = 0 \\
&\mathsf{cws}_\ell \;(\mathsf{c}\colon \mathsf{str}) \mid \mathsf{isSpace}\;\mathsf{c} = 1 + \mathsf{cws}_\ell\;\mathsf{str} \\
&\mathsf{cws}_\ell \;(\_\colon \mathsf{str}) = \mathsf{cws}_\ell\;\mathsf{str}
\end{aligned}
$$

$$
\mathsf{run}\;\mathsf{fname} = \mathsf{readFile}\;\mathsf{fname} \gg\!= \mathsf{print} \circ \mathsf{cws}_\ell
$$

Where does the IO happen?

- readFile ?
- cws_ℓ ?

Lazy IO is like `mmap`

Where does the IO happen in this code? It may seem IO happens in readFile, which reads the whole file into memory string first. We then count the whitespace and print the result. Only that's not how it works: reading happens on demand, in cws_ℓ . The counting hence happens in constant memory – which is the great advantage of Lazy IO, akin to memory-mapped IO. However, executing IO within a supposedly 'mathematical' function does seem odd. And it is, as we shall see soon.

Count whitespace: Handle

```
cws_h :: Handle → IO Int
cws_h h = loop 0
 where
 loop n = try (hGetChar h) >>= check n
 check n (Right c) = loop (if isSpace c then n+1 else n)
 check n (Left e) | Just ioe ← fromException e,
                    isEOFError ioe = return n
 check _ (Left e) = throw e

run fname =
 print =<<
  bracket (openFile fname ReadMode) hClose cws_h
```

Here is the idiomatic C code – still written in Haskell, with a handle-based IO. We now differentiate EOF from other IO errors: we could not do that with the Lazy IO. We could even recover from some errors (like file locking errors). The code is much more explicit – with error handling and ensuring that the handle is always closed – but is still quite simple. Only one line in that code deals with the counting of the whitespace characters; the others are the 'boilerplate'.

Count whitespace: Iteratee

```
cws_i  :: Monad m ⇒ R Char m Int
cws_i  = loop 0
 where
 loop n = getchar ≫= check n
 check n (Just c) = loop (if isSpace c then n+ 1 else n)
 check n Nothing = return n
```

```
type R e m = Iteratee e m
getchar :: Monad m ⇒ R e m (Maybe e)
instance Monad m ⇒ Monad (R e m)
```

Here is the Iteratee IO code. It has the overall look of the Handle IO: getting the current character and checking it. The code is higher-level: for example, EOF and other IO errors are handled as before, but out of the way.

Count whitespace: Iteratee

```
cws_i  :: Monad m ⇒ R Char m Int
cws_i  = loop 0
 where
 loop n = getchar ≫= check n
 check n (Just c) = loop (if isSpace c then n+ 1 else n)
 check n Nothing = return n
```

```
type R e m = Iteratee e m
getchar :: Monad m ⇒ R e m (Maybe e)
instance Monad m ⇒ Monad (R e m)
```

getchar vs. System.IO.getChar vs. List.head

We use R as a synonym for Iteratee. First of all it's shorter that Iteratee, and space is at a premium on slides. But there is a different reason, as we shall see. Our getchar is the simplest iteratee, which does what it name suggests. It is like getChar of the standard IO library. The type R e m a says that getchar is an iteratee that consumes a stream of elements e and produces, *without any effects*, either e or `Nothing`. So, `getchar` looks like taking the head of the list. Iteratees form a monad, letting us combine them into a bigger code, for the loop.

Count whitespace: Iteratee

```
cws_i  :: Monad m ⇒ R Char m Int
cws_i  = loop 0
 where
 loop n = getchar ≫= check n
 check n (Just c) = loop (if isSpace c then n+ 1 else n)
 check n Nothing = return n

run fname = print =≪ fileL fname ⊛ cws_i

type R e m = Iteratee e m
getchar :: Monad m ⇒ R e m (Maybe e)
instance Monad m ⇒ Monad (R e m)
fileL   :: FilePath → L Char IO IO
```

To run the iteratee code, we hook the iteratee, the consumer, with a producer of the stream, e.g., fileL. Our fileL (see its type) turns a file into a stream of bytes, reading it incrementally. IO happens in fileL, not during the counting (`getchar`'s type shows it does no IO.) Counting requires no more effects than reading, hence IO is repeated twice in fileL's type. Our fileL takes care of closing the file. No bracketing is needed any more.

Hook-ups

Lazy IO

```
readFile fname >>= (\ str -> print . cws_ℓ $ str)
```

Handle IO

```
do
  h <- openFile fname ReadMode
  r <- cws_h h
  hClose h
  return r
```

Iteratee IO

```
fileL fname ⊛ cws_i
```

Let's compare how we hook-up data producers with consumers in all three methods. Handle IO uses a handle to represent an open file. Lazy IO does the same: `str` is pretty much like a handle. A handle represents a resource that has to be accounted for and promptly disposed of. We have to make sure to close the handle, or to beseech GC to do it for us, not too late.

Hook-ups

Lazy IO

```
readFile fname >>= (\ str -> print . cws_ℓ $ str)
```

Handle IO

```
do
  h <- openFile fname ReadMode
  r <- cws_h h
  hClose h
  return r
```

Iteratee IO

```
fileL fname ⊛ cws_i
```

What you do not have, you cannot abuse

There are no such worries with iteratees however: the "handle" is implicit, and so there is nothing to leak. What you do not have, you cannot abuse.

Better count whitespace: Lazy

$$\begin{aligned}
&\mathsf{cws}_\ell \;::\; \mathsf{String} \;\to\; \mathsf{Int} \\
&\mathsf{cws}_\ell \;=\; \mathsf{length} \circ \mathsf{filter}\quad \mathsf{isSpace}
\end{aligned}$$

$$\mathsf{run}\;\;\mathsf{fname} = \mathsf{readFile}\;\mathsf{fname} \gg\!= \mathsf{print} \circ \mathsf{cws}_\ell$$

Lazy IO permits a far more elegant solution. It is a one-liner, using the standard Prelude functions on lists. We filter only whitespace characters, and count them.

Better count whitespace: Iteratee

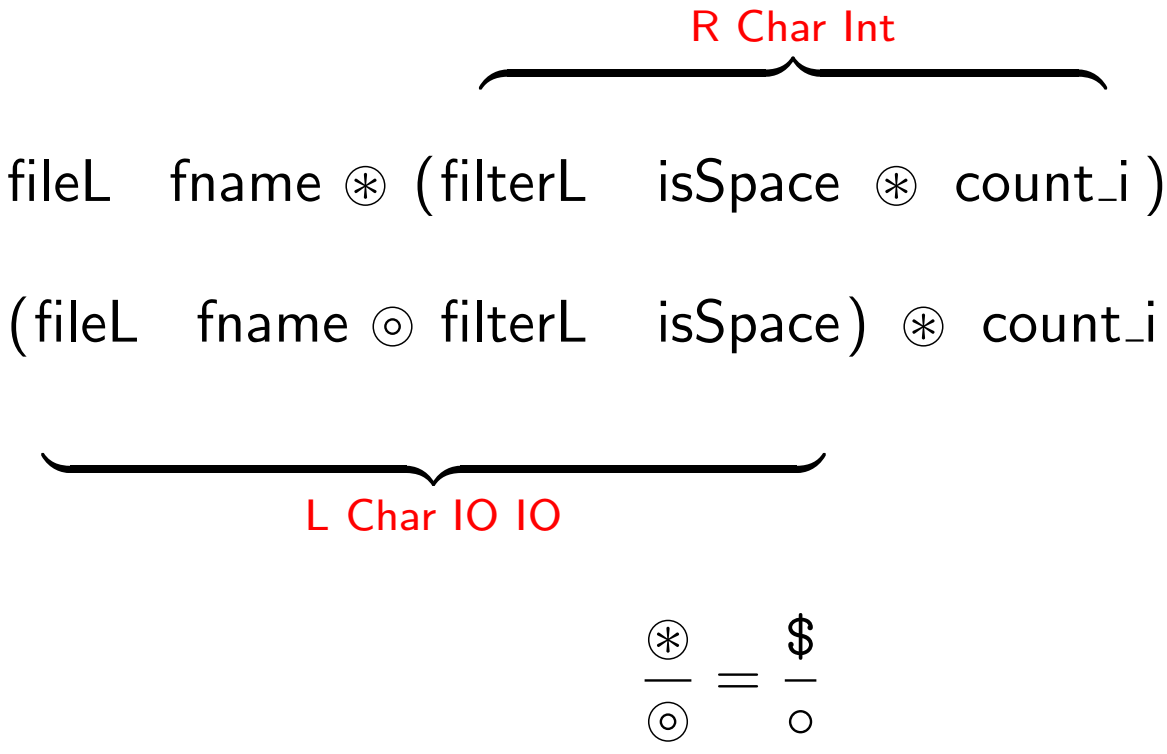
```
cws_i  :: Monad m ⇒ R Char m Int
cws_i  = filterL  isSpace ⊛ count_i

run fname = print =≪ fileL  fname ⊛ cws_i

count_i  :: Monad m ⇒ R e m Int
filterL  :: Monad m ⇒ (e → Bool) → L e m (R e m)
```

Iteratee IO affords the same elegance. count_i is like List.length, counting items in a stream. filterL is what it sounds like, the analogue of List.filter. Its type says it is a stream producer on one end and a consumer on the other end.

Applications and compositions



Let's look more closely at the two-sided nature of filterL. Before we counted the whitespace as in the first expression, connecting the count_j *consumer* with filterL to build a bigger consumer, of the unfiltered stream of characters. We connect that latter consumer with the stream producer fileL. We also can do the other way around. We combine a *producer* of the raw stream with filterL to get a bigger *producer*, of the filtered stream. We hook up the result with the counting consumer. The iteratee composition operators indeed look and feel like functional application and composition.

With Lazy and Handle IO, the producers are provided by the run-time library and can't be extended. With Lazy and Handle IO, we only build consumers. Iteratee IO offers a symmetric approach, letting us build either consumers or producers, depending on what suits us.

THE count: Lazy

```
ct_ℓ    :: String  → Int
ct_ℓ    = length ∘ filter  (== "the") ∘ words

run fname = readFile fname >>= print ∘ ct_ℓ

type Word = String
words :: String  → [Word]
```

A more elaborate problem: counting the occurrences of the word "the" (assuming the input is text with words of bounded size). We must count "the" as the word by itself, not being a part of another word. That is, in order for "the" to be counted, the character before and after (if exist) must be whitespace.

The Lazy IO code again clearly expresses the algorithm: we split the stream into words, filter "the", and count.

THE count: Handle

```
ct_h :: Handle → IO Int
ct_h h = getchar >>= s1 0
 where
 s1 n (Just c) | isSpace c = getchar >>= s1 n
 s1 n (Just 't') = getchar >>= st n
 s1 n (Just _) = getchar >>= sskip n
 s1 n Nothing = return n

 st n (Just 'h') = getchar >>= sth n
 st n x = sskip n x

 sth n (Just 'e') = getchar >>= sthe n
 sth n x = sskip n x

 sthe n (Just c) | isSpace c = getchar >>= s1 (n+ 1)
 sthe n Nothing = return (n+ 1)
 sthe n _ = getchar >>= sskip n

 sskip n Nothing = return n
 sskip n (Just c) | isSpace c = getchar >>= s1 n
 sskip n _ = getchar >>= sskip n

 getchar :: IO (Maybe Char)
 getchar = try (hGetChar h) >>= \c → case c of
              Right x → return $ Just x
              Left e | Just ioe ← fromException e,
                          isEOFError ioe → return Nothing
              Left e → throw e

run fname =
 bracket (openFile fname ReadMode) hClose $ \h →
 ct_h h >>= print
```

Here is complete Handle IO code. You can't see the code because the font is too small. Making the font bigger won't make the code more understandable, I'm afraid. It is a mess: an encoding of a state machine.

THE count: Iteratee

$$\begin{aligned}
&\mathsf{ct}_i \quad :: \textbf{Monad}\ \mathsf{m} \Rightarrow \mathsf{R\ Char\ m\ Int}\\
&\mathsf{ct}_i \quad = \mathsf{wordsL} \circledast \mathsf{filterL}\quad (== \text{"the"}) \circledast \mathsf{count_i}
\end{aligned}$$

$$\mathsf{run\ fname} = \mathsf{print} =\!\ll \mathsf{fileL}\ \ \mathsf{fname} \circledast \mathsf{ct}_i$$

$$\mathsf{wordsL} :: \textbf{Monad}\ \mathsf{m} \Rightarrow \mathsf{L\ Word\ m\ (R\ Char\ m)}$$

The Iteratee IO does the same stream processing as Lazy IO, converting one stream to another (characters to words, words to filtered words).

THEs count: Lazy

$$\mathsf{run}\ \mathsf{fnames} = \mathsf{mapM}\ \mathsf{readFile}\ \mathsf{fnames} \gg\!= \mathsf{print} \circ \mathsf{ct}_\ell \circ \mathsf{concat}$$

Let us count the occurrences of the word “the” in a sequence of files assuming the files are concatenated together. Therefore, we will count the word ‘the’ if the letter ‘t’ is in one file but ‘he’ is in the next one. With Lazy IO we re-use the previously written counting function ct_ℓ . We will pass it a string that is the concatenation of files’ contents. The code is elegant.

THEs count: Lazy

$$\mathsf{run}\ \mathsf{fnames} = \mathsf{mapM}\ \mathsf{readFile}\ \mathsf{fnames} \gg\!= \mathsf{print} \circ \mathsf{ct}_\ell \circ \mathsf{concat}$$

- reading is incremental
- N open files

The files are read incrementally; the second file will be read only after the first one finished. Alas, we have to open all of the files first! The readFile action, which opens the file and prepares it for lazy reading, is performed first. Therefore, we need as many descriptors as there are files in the fnames list. If the list is obtained from scanning a directory tree, we may run out of file descriptors! That is particularly disturbing since we really need only one file descriptor, opening and closing it as we go. We get the first intimation how Lazy IO mis-manages resources – sometimes taking much more than needed, and giving the programmer no facilities to control the resources.

THEs count: Handle

The first approximation:

```
run fnames = mapM counter fnames >>= print ∘ sum
 where counter fname =
          bracket (openFile fname ReadMode) hClose ct_h
```

- reading is incremental
- 1 open file at a time
- does not quite work

We only need one file descriptor for the whole operation: the next file is opened only after the previous file is closed. Alas, this solution is deficient since EOF is counted as a word terminator. We can't handle the case of the word 'the' split across the files. We have to re-write our state machine to perform an action upon the detection of EOF to re-associate the handle with another file. This re-writing is left as an exercises to the reader, to drive down the point of how really low-level Handle IO is.

THEs count: Iteratee

$$\mathsf{run}\ \mathsf{fnames} = \mathsf{print} =\!\!\ll \mathsf{foldr1}\ \ \mathsf{mappend}\ (\mathbf{map}\ \mathsf{fileL}\ \mathsf{fnames}) \circledast \mathsf{ct}_i$$

- reading is incremental
- 1 open file

Mappending producers concatenates their sources

The iteratee code again looks pretty much like Lazy IO code. However, only one file descriptor is used for all processing. fileL prints the debugging message when the file is opened and closed. We then clearly see that, unlike Lazy IO, only one file is open at any given time.

THE whitespace count: Lazy

```
run fname = do
  str ← readFile fname
  print (cws_ℓ str , ct_ℓ str )
```

- Looks great! Code reuse!

To count both “the” and the whitespace, we just combine the previously written counters. Looks great!

THE whitespace count: Lazy

```
run fname = do
  str ← readFile fname
  print (cws_ℓ str , ct_ℓ str )
```

- Looks great! Code reuse!
- Not incremental

Alas, it doesn't work great! Now, the whole file is loaded in memory. The processing is no longer incremental. We have come across one of many surprises of Lazy IO. It is not modular; it is not compositional, if we care about complexity or resource consumption..

THE whitespace count: Iteratee

```
run fname =
  print =≪ fileL fname ⊛ (cws_i `en_pair` ct_i )

en_pair :: Monad m ⇒ R e m a → R e m b → R e m (a,b)
```

Like Lazy IO, we re-use the previously written counters, pairing them. Unlike Lazy IO, the processing remains incremental. As we read a block from file, we send the block to two iteratees for handling.

THE whitespace count prefix: Lazy

Early termination

```
run n fname = do
  str0 ← readFile fname
  let str = Prelude.take n str0
  print (cws_ℓ str , ct_ℓ str )
```

We now count the occurrences of the word “the” and the white space within the prefix of the stream of the size at most N. The example is abstracted from reading HTTP request content with the explicitly specified Content-Length. We should not attempt to read even a single byte after N since the web client expects the reply first, before it will send the next request. If we attempt to read ahead after N bytes, the deadlock ensues.

THE whitespace count prefix: Lazy

Early termination

```
run n fname = do
  str0 ← readFile fname
  let str = Prelude.take n str0
  print (cws_ℓ str , ct_ℓ str )
```

- Not in constant memory
- Leaking file descriptor (scarce resource)
- Can deadlock – and does, in practice

Reading is an observable effect

As before this code does not run in constant memory. There is another problem: since we may read only a part of the file, the file descriptor will not be closed (until a finalizer is run, which may happen very late). There is a real danger of running out of file descriptors (which regularly happens in practice with Lazy IO). There is the third problem: the run-time system may speculatively read-ahead, at any time and for any reason. The programmer has no way whatsoever to control this read-ahead or even be informed about it. Therefore, deadlock may happen (and does routinely happen in practice, when using lazy IO for interactive services). Lazy IO was designed to give the impression that IO is not even happening. Alas, when dealing with request-response servers and multiple IO operations, even reading is an observable effect.

THE whitespace count prefix: Iteratee

```
run n fname =
  print =≪ fileL fname ⊛
        takeL n ⊛ (cws_i `en_pair` ct_i )

takeL :: Monad m ⇒ Int → L e m (R e m)
```

- in constant memory, incremental
- reading only n bytes and not a byte longer
- the file is closed immediately afterwards

The primitive takeL, like its List's namesake, ensures that no more than n characters are read; afterwards, it will not speculatively ask for any further file data.

The overall code looks quite like Lazy IO. Now it actually works as intended.

Taste of the equational laws

Composition

$$\mathsf{strL}\ (\mathsf{s1} +\!\!+ \mathsf{s2}) \equiv \mathsf{strL}\ \ \mathsf{s1}\ \ \text{`}\mathsf{mappend}\text{`}\ \mathsf{strL}\ \mathsf{s2}$$

Chaining

$$\mathsf{strL}\ \ (\mathsf{s1} +\!\!+ \mathsf{s2}) \circledast (\mathsf{i}\ \gg\!= \mathsf{f}) \equiv$$
$$(\mathsf{strL}\ \ \mathsf{s1} \circledast \mathsf{i}\,) \gg\!= \backslash \mathsf{x} \to \mathsf{strL}\ \ \mathsf{s2} \circledast \mathsf{f}\ \ \mathsf{x}$$

(provided i properly recognizes `s1`)

Zero

$$\mathsf{failure}\ \ \gg\!= \mathsf{f} \equiv\ \mathsf{failure}$$

Right distributivity

$$\mathsf{i}\ \gg\!= \backslash \mathsf{x} \to (\mathsf{k1}\ \mathsf{x} \lhd\ \mathsf{k2}\ \mathsf{x}) \equiv\ (\mathsf{i}\ \gg\!= \mathsf{k1}) \lhd\ (\mathsf{i}\ \gg\!= \mathsf{k2})$$

Please see the paper for details and applications

Let me give you the taste of equational laws of Iteratees. We consider iteratees as parsers. The laws justify the modular construction of iteratee parsers, treating them as parser combinators.

We exposit programming with iteratees, contrasting them with Lazy IO and the Handle-based, stdio-like IO. We relate them to online parser combinators. We introduce a simple implementation as free monads, which lets us formally reason with iteratees. As an example, we validate several equational laws and use them to optimize iteratee programs. The simple implementation helps understand existing implementations of iteratees and derive new ones.

Frequently Questioned Answers

Can Iteratee IO really:

- do binary IO
- arbitrarily change position in stream: `fseek`
- write to a file, with exception safety: `iterFile`
- split the stream (process it in parallel)
- process several files together, out of lockstep: `merge`

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

Here are a few answers that people frequently find hard to believe. Yes, really, you can do all that. Please see the above URL for details, which are not in the paper.

Conclusions

Iteratee IO is

- incremental IO with precise resource control
- compositional:
  assembling consumers *or producers*
- especially good for communication streams and BigData
- a library of incremental parsing combinators and
  can be reasoned as such

Conclusions

Iteratee IO is

- incremental IO with precise resource control
- compositional:
  assembling consumers *or producers*
- especially good for communication streams and BigData
- a library of incremental parsing combinators and
  can be reasoned as such

Iteratee IO is a strict lazy IO

A reviewer of the paper wrote in his summary that the Iteratee IO can somewhat perversely be described as ‘strict Lazy IO’. Iteratee IO is what Lazy IO was meant to be.

Parting thought

"We should have some ways of coupling programs like garden hose – screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also."

M. D. McIlroy. October 11, 1964.
`http://doc.cat-v.org/unix/pipes/`

Almost fifty years later, we are coming to such a way of coupling programs and doing IO.