Iteratees

Oleg Kiselyov

oleg@okmij.org

Abstract. 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.

> "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.

1 Introduction

Iteratee IO is a style of compositional incremental input processing with precise resource control. As such it is conducive to handling large amounts of data and programming of long-running servers. Iteratee IO has been proven in practice: it is employed in several commercially deployed web frameworks (e.g., [2]) has been used in financial trading applications [9] and natural language processing. Good performance of iteratee IO is seen from several benchmarks, web-related (included SNAP) and others [6, 10]. Performance, compositionality and high level of abstraction attracted attention. As of May 2011, there are three main implementations of Iteratee IO on Hackage:[1] `iteratee-0.8.3`, `enumerator-0.4.10` and the extensive `iterIO`, as well as several variations. Iteratee IO lends itself to efficient, online parser combinator libraries similar to [1, 13]. First introduced to Haskell [5], Iteratee IO has since been ported to F#,[2] Scala and other languages.

[1] http://hackage.haskell.org
[2] https://github.com/fsharp/fsharpx

The goal of Iteratee IO is to overcome drawbacks of Lazy and Handle-based IO and combine their strong features. Lazy IO, an instance of memory-mapped IO, is an elegant abstraction that effectively eliminates IO, giving programmers an impression that the entire file is available in memory and may be accessed as an ordinary string. There is no longer need to explicitly read, let alone worry about buffer allocations and underflows. The abstraction of a file as an in-memory string comes without guilt: behind the scene, the operating or run-time systems access the file efficiently, reading it on demand and sharing the read data. Lazy IO facilitates compositional input processors like parser combinators.

Lazy IO is so irresistible that it was added to Haskell despite the reservations of its inventors and the failure to develop good techniques for reasoning about its correctness [7, Sec 10.5]. However benign, reading is an observable side-effect, whose occurrence may have to be correlated with other side effects. Such correlations are crucial when performing IO over communication pipes, which is typical of web servers.[3] As Launchbury and Peyton Jones feared, Lazy IO indeed "gives rise to a very subtle class of programming errors". We have seen deadlocks; mishandling of IO errors; running out of file descriptors and similar scarce resources; unpredictable, volatile and sometimes unbearably excessive use of memory. We illustrate the splendors and miseries of Lazy IO in §2.

Handle-based IO is the `stdio`-style IO familiar from C. It is 'strict': IO operations must be explicitly requested. Therefore, it affords precise control of resources and the detection of all IO errors. However, it is very low-level: every read operation is painfully explicit. Handle-based IO hides the buffering, providing the abstraction for a stream of characters. The abstraction does not extend to a stream of other data types, and does not support stream embeddings. The programmer must be constantly aware of the current file position, which makes it tortuous to process layered streams or combine parsers to process the same stream in parallel. §2 illustrates these problems as well.

One wishes for a set of abstractions that free programmers from thinking about IO, and yet provide facilities to control buffering, look ahead, locking, etc. at those moments where it matters. One wishes to derive these abstractions and optimize them by algebraic transformations based on equational laws.

Iteratee IO is an approach to this ideal, amalgamating ideas going back to the IO of Haskell 1.3; Kernel-Prolog's iterator objects [15] uniformly representing files, in-memory collections and processes; the resumption monad, surveyed in [3], and generators of Alphard [12] (which now live as Java streams and generators in modern languages.) Like Handle IO, Iteratee IO gives error handling, the precise control over important operations and resource allocations, incremental processing and high performance. Like Lazy IO, it gives high-level abstractions, encapsulating input processing layers that can be nested and composed sequentially or in parallel. Iteratee IO turns out to offer reasoning principles letting us derive implementations and optimize them.

Although all implementations of Iteratee IO follow the same principles, there are many variations based on historical accidents, handling of buffering, levels

[3] POSIX memory-mapped IO, `mmap`, does not work with communication pipes either.

of generality. There are even more tutorials, varying in generality and points of view [6, 8, 16–18]. The desire for a standpoint to grasp the idea of iteratees, to assess and derive their variations and to reason with them is the motivation for the present paper.

The contributions and the structure of the paper We argue that the essence of Iteratee IO is captured by two inter-related views: stream processing network and incremental online parser combinators. Based on the final co-algebra model of stream processors, for the first time we formulate algebraic laws, which let us derive and simplify iteratee parser combinators.

§2 introduces Iteratee IO, by using an Iteratee library to write a progression of examples abstracted from web server programming and computational linguistics. We use the examples to contrast Iteratee IO with Lazy and Handle-based IO and to give them informal semantics.

§3 defines the semantics formally, as an interpretation of the data type denoting iteratees. The semantics lets us view iteratees as parsers. The rich algebraic structure of the iteratee data type – final co-algebra and free monad – gives rise to algebraic laws, which let us build and reason about iteratee programs compositionally. Appendix B of the full paper[4] details optimizing iteratee parser combinators using the equational laws.

Appendix A proves the equational laws in a more general setting of effectful iteratees – in which input processing is accompanied by effects in some monad. Buffering and look-ahead are two particular examples of such an effect. This insight clarifies the implementation of buffering in iteratee libraries – which so far has been the most confusing feature.

More material about iteratees, including demonstrations, tutorials and references to iteratee libraries are available online at `http://okmij.org/ftp/Streams.html`. The annotated source code for all the examples in the paper can be found in `http://okmij.org/ftp/Haskell/Iteratee/`, which is the base URL for all code files referenced in this paper.

2 Programming with Iteratees

This section introduces programming with iteratees on a series of progressively more complex examples. We stress compositionality – assembling input processors from previously written or library components. We appeal to the intuitions of more familiar Lazy IO and Handle IO when explaining iteratees. Therefore, the examples are also written in Lazy IO, and, when feasible, Handle IO. The contrast lets us see the advantages of Lazy and Handle IO that Iteratee IO inherits, and the drawbacks it is designed to overcome. (In particular, we shall see Lazy IO's unexpected, huge memory consumption and wasting sparse resources like file descriptors.)

The examples revolve around reading potentially very large text and counting specific words and whitespace. The final example, abstracted from interactive

[4] `http://okmij.org/ftp/Haskell/Iteratee/describe.pdf`

systems, tests orchestration: reading a communication pipe up to a terminator but not a byte further. The complete code for all examples – with tests and the extra details left out of the paper – is available online (`IterDemo1.hs`). The code uses the `IterateeM` library available from the same site. Figure 1 lists the interface of the library fragment used in this section, pointing out related functions from the Haskell standard library. A different series of illustrative examples, counting lines and words and searching for the first or all occurrences of a word – implementing `wc` and `grep` – are given in `IterDemo.hs`. The latter set of examples illustrates error handling and the encapsulation of state.

```
type Iteratee el m a -- a processor of the stream of els
                     -- in a monad m yielding the result of type a
instance Monad m ⇒ Monad (Iteratee el m)
instance MonadTrans (Iteratee el)

getchar  :: Monad m ⇒ Iteratee el m (Maybe el) -- cf. IO.getChar, List.head
count_i  :: Monad m ⇒ Iteratee el m Int        -- cf. List.length

run      :: Monad m ⇒ Iteratee el m a → m a   -- extract Iteratee's result

-- A producer of the stream of els in a monad m
type Enumerator el m a = Iteratee el m a → m (Iteratee el m a)
enum_file :: FilePath → Enumerator Char IO a  -- Enumerator of a file

-- A transformer of the stream of elo to the stream of eli
-- (a producer of the stream eli and a consumer of the stream elo)
type Enumeratee elo eli m a =
    Iteratee eli m a → Iteratee elo m (Iteratee eli m a)

en_filter :: Monad m ⇒ (el → Bool) → Enumeratee el el m a
take :: Monad m ⇒ Int → Enumeratee el el m a          -- cf. List.take
enum_words :: Monad m ⇒ Enumeratee Char String m a -- cf. List.words

-- Kleisli (monadic function) composition: composing enumerators
(>>>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)

-- Connecting producers with transformers (cf. (=<<))
infixr 1 .|                    -- right-associative
(.|) :: Monad m ⇒
  (Iteratee el m a → w) → Iteratee el m (Iteratee el' m a) → w

-- Parallel composition of iteratees (cf. List.zip)
en_pair :: Monad m ⇒
  Iteratee el m a → Iteratee el m b → Iteratee el m (a,b)
```

Fig. 1. The interface of the IterateeM library fragment

We start lightly, with counting whitespace characters. The Lazy IO code pattern-matches on the ordinary string (cf. more elegant code later):

```
countWS_lazy :: String → Int
countWS_lazy "" = 0
countWS_lazy (c:str ) | isSpace c = 1 + countWS_lazy str
countWS_lazy (_:str ) = countWS_lazy str
```

There are no appearances of any IO operations, which is the main appeal of Lazy IO. The IO is banished to the highest-level code, which opens a file and "reads it all" into a string, to hand out to countWS_lazy.

```
run_countWSL fname = readFile fname ≫= print ∘ countWS_lazy
```

The run-time system reads the file on demand, so that the counting runs in constant, and small memory.

The Handle IO code is in the style of stdio, familiar to C programmers. It, too, "pattern-matches" on the input stream.

```
countWS_handle :: Handle → IO Int
countWS_handle 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_countWSH fname =
 bracket (openFile fname ReadMode) hClose $ \h →
 countWS_handle h ≫= print
```

It, too, runs in small and constant memory. Error handling stands out. We now differentiate EOF (end-of-file) from other IO errors, which is impossible with Lazy IO. Also unlike Lazy IO, the code is explicit about closing the file, ensuring that the file be closed (IO errors or not) before run_countWSH returns.

The Iteratee IO code below should look quite similar to the earlier examples. The intuition of pattern-matching on the stream still applies; the stream is implicit however. Since there is no explicit 'handle', errors like reading from an already closed handle become impossible.

```
countWS_iter :: Monad m ⇒ Iteratee Char m Int
countWS_iter = loop 0       -- tail-recursive
 where loop n = getchar ≫= check n
       check n Nothing = return n
       check n (Just c) = loop (if isSpace c then n+ 1 else n)
```

We have written an *iteratee*: it reads the stream of characters and produces an Int. Polymorphism over the monad m tells that the iteratee (like countWS_lazy) is pure. The library iteratee getchar[5] (see Figure 1) is quite like try (hGetChar h)

[5] Although some Iteratee libraries indeed provide something like getchar, we implement it ourselves, in `IterDemo1.hs`. §3 explains the idea of the implementation and gives a simpler version, called oneL in Figure 2.

in the Handle IO code: getchar reads a character from the stream and returns it; the result **Nothing** signifies EOF. The Iteratee library takes care of detecting and propagating other IO errors. Iteratee el m is a monad, letting us write iteratee code in the standard monadic style (again compare with the Handle IO code above).

This is how we count whitespace in a file:

```
run_countWSI fname = print =≪ run =≪ enum_file fname countWS_iter
```

Here, enum_file is an *enumerator*: it enumerates a file, passing file data to an iteratee. The precise enumerator/iteratee interaction is described in §3. A user of an Iteratee library should find sufficient the stdin intuition: the iteratee getchar is quite like Prelude.getChar, which reads a character from the buffer containing the current chunk of the standard input. If the buffer is empty, OS is requested to fill it in. Our enum_file opens the file on the 'standard stream' and plays the OS for the iteratee, reading a chunk when the iteratee asks to fill its buffer. When the file is exhausted, or when the iteratee stops asking for more data, the iteratee, encapsulating the resulting state, is returned. The resulting iteratee cannot hold any references to the file: in fact, an iteratee cannot know if its 'stdin' data come from a file or other source. Therefore, enum_file's closing the file upon return is safe.[6] The function run tells the iteratee that the stream is finished and extracts its result, the integer counter in our case. (App. C gives another, plumbing intuition for Iteratee IO.) Like Lazy IO, the file is read incrementally and on demand. Like Handle IO, the file be closed when enum_file returns, IO errors or not. Explicit bracketing is not needed. Like Handle IO (and unlike Lazy IO), iteratees support precise error handling and accounting of sparse resources like file descriptors. Unlike Handle IO, the boring details are hidden away.

Lazy IO permits a far more elegant solution: a one-liner, using the standard Prelude functions on lists:

```
countWS'_lazy :: String → Int
countWS'_lazy = length ∘ filter isSpace
```

We filter out the characters other than whitespace, and count the remainder. As expected for pure Haskell, filtering and counting is done incrementally and lazily. The intermediate list is never fully constructed. Iteratee IO matches the algorithm and the elegance:

```
countWS'_iter :: Monad m ⇒ Iteratee Char m Int
countWS'_iter = id .| (en_filter isSpace) count_i
```

We use three new library primitives, Figure 1: the iteratee count_i, like length, returns the length of the stream. The *enumeratee* en_filter is a stream transformer. The type of Enumeratee elo eli m a almost fits the pattern of Enumerator eli m a: indeed, the enumeratee is an enumerator for the inner stream (of eli-type elements), taking data from the outer stream. That is, Enumeratee elo eli m a converts a stream of elos into a stream of elis. The conversion is not necessarily in lock-step, as is the case for en_filter: although the outer and the

[6] In that respect, enum_file is similar to Scheme's with-input-from-file.

inner streams have elements of the same type, the inner streams has generally fewer elements. Although conceptually 'stream' is an (infinitely) long sequence of elements, at any given time only a single, small chunk of the stream is present in memory. Our en_filter requests a chunk from the outer stream and creates a filtered chunk, to pass to an iteratee. All stream conversion is *strictly* incremental. There is not even a chance of producing a large intermediate data structure, and no need to trust laziness or GHC fusion rules. The combinator (.|), akin to run, 'runs' the enumeratee, that is, terminates the inner stream and extracts the result of the inner iteratee, passing it to the consumer, the left argument of (.|). (There are cases, not described in the paper, where the inner stream should be left unterminated so it can be passed to another enumeratee: e.g., processing of HTTP chunk-encoded streams.) To count whitespace in a file, we write enum_file fname countWS'_iter. The equational law f (g .| h) ≡ (f ∘ g) .| h gives enum_file fname .| en_filter isSpace count_i, which resembles the Unix pipeline.

We modify the example to count the occurrences of the word "the" (assuming the input is text with words of bounded size). Lazy IO code is most straightforward, relying on Prelude.words to parse the input string into a list of words. We filter out words other than "the" and count the remainder. Thanks to Haskell laziness, the whole operation runs in constant space.

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

Handle IO code for this example is complex. Not only should we search for "the", we also have to make sure the character before and after (if exist) is whitespace. On the top of it, we have to deal with errors and EOF. The simplest solution is to explicitly write the Finite State Machine recognizer, see `IterDemo1.hs` for details. The result is too big to put in the paper, reminding us that Handle IO is really low level. An abstraction is direly needed, for example, in a form of a lexer generator – or Iteratee IO.

Here is the Iteratee IO code (Recall, (.|) is right-associative)

```
countTHE_iter :: Monad m ⇒ Iteratee Char m Int
countTHE_iter = id .| enum_words .| en_filter (== "the") count_i
```

It is quite like Lazy IO, converting a character stream to a word stream to the filtered stream, which is then counted.

Let us extend the example so to count the word "the" within a sequence of files, as if they are concatenated. We shall count "the" even if it is split between two files. The Lazy IO code re-uses the previously written counting function countTHE_lazy, which now receives a string that is the concatenation of all files' contents.

```
run_manyTHEL fnames =
  mapM readFile fnames >>= print ∘ countTHE_lazy ∘ concat
```

The code is elegant; the processing is incremental, reading only one file at a time. Alas, we have to open all files first! The action readFile, which opens a file

and prepares it for lazy reading, is performed in sequence on all files – prior to counting. Since counting is pure, we cannot execute IO actions like readFile in the counting code. Therefore, we need as many file descriptors as there are files in the fnames list. If the list is long, we may run out of file descriptors. Ideally, however, we only need one file descriptor, opening and closing it as we go. We get the first intimation of Lazy IO resource mis-management – with no interface to correct.

Handle IO code to count in multiple files is even more complex than the single-file counter.We do not show the code for the lack of space.

The Iteratee IO code again looks quite like Lazy IO code, re-using the previously written iteratee countTHE_iter. Kleisli (monad functional composition) (>>>) builds an enumerator from two others, effectively sending to an iteratee first the chunks of the first stream and then the chunks of the second. In short, composing enumerators concatenates their sources. We elaborate on that property and state it formally, in §3.

```
run_manyTHEI fnames = print =<< run =<<
  foldr1 (>>>) (map enum_file fnames) countTHE_iter
```

(one can use the regular foldr keeping in mind that the unit of (>>>) is return). Unlike Lazy IO, only a single file descriptor is used during the whole counting; only one file is open at any given time.

As a test of compositionality, we combine the two counting operations and count "the" and whitespace, together. Lazy IO code is elegantly straightforward.

```
run_countPairL fname = do
  str ← readFile fname
  print (countWS_lazy str, countTHE_lazy str)
```

Here we run into one of Lazy IO pitfalls: the counting is no longer incremental. The whole file is loaded in memory. For applications processing large files or long streams, Lazy IO is too unreliable for use in production.

The Iteratee code also re-uses previously written counters. It pairs them, relying on the parallel composition of iteratees en_pair.

```
run_countPairI fname =
  print =<< run =<< enum_file fname (countWS_iter `en_pair` countTHE_iter)
```

One may think of en_pair as 'splitting' (or duplicating) the stream. In reality en_pair does no copying or buffering: it receives a chunk and passes it to its two argument iteratees. If both iteratees want more data, a new chunk is requested. Unlike Lazy IO, the processing remains incremental and in constant memory: As we read a block from file, we send the block to two iteratees.

Our final example demonstrates early, prior to EOF, termination. We modify the previous "the" and the whitespace counter to count only within the prefix of the stream of the size at most N. This 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 a web client expects the reply first, before it sends the next request. If we attempt to read ahead after

N bytes, the deadlock ensues. The lazy IO code uses the Prelude.take to lazily obtain the prefix of the file content, which is processed as before.

```
run_ntermL n fname = do
  str0 ← readFile  fname
  let  str  = Prelude.take n str0
  print  (countWS_lazy str,  countTHE_lazy str)
```

As in the previous example, this counting does not run in constant memory. There are bigger problems. First, since we generally stop reading before EOF, the run-time system will not close the file descriptor. It will be closed when the corresponding finalizer is run, which may happen very late. Leaking of file descriptors puts us in danger of running out of them, which indeed happens in practice when using Lazy IO with programs that process lots of files. Most serious is the real danger of a deadlock. 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. Deadlock does routinely happen in practice, when using lazy IO for interactive services.[7]

Lazy IO was designed to give the impression that IO is not even happening. When dealing with communication pipes and request-response servers, even reading is an observable effect. The precise control of reading actions is crucial. Lazy IO becomes a wrong abstraction.

The Iteratee IO code, like the earlier Lazy IO code, differs from the previous run_countPairI in one change, take – from IterateeM rather than Prelude.

```
run_ntermI  n fname =
  print  =≪ run =≪ enum_file  fname . |
   IterateeM. take  n  (countWS_iter ‘en_pair‘  countTHE_iter)
```

Like en_filter, the enumeratee take substreams its outer stream, namely, takes the prefix of the size at most n. As soon as take n gets its n elements, it stops asking for more data, prompting its enumerator, enum_file, to close the file. The Iteratee code is just as concise as the Lazy IO code; both are quite alike. Since IO is now done strictly, the iteratee code gives full control over file opening, closing, and reading. (`IterDemo1.hs` has another early termination example, reading the file up to the first occurrence of a given string.)

We have seen that both Lazy IO and Iteratee IO allow assembling of the whole program from independent building blocks. Both IO styles permit the incremental processing, reading file data on demand. Because Iteratee IO is not lazy, the Iteratee library can ensure timely deallocation of resources, precise IO error handling, precise control of reading actions. Iteratee IO, unlike Lazy IO, guarantees the incremental processing.

3 Enumerators and the semantics of iteratees

This section outlines the conceptual design of iteratees, viewing iteratees and enumerators as communicating sequential processes. Iteratee processes are mod-

[7] `http://www.haskell.org/pipermail/haskell-cafe/2008-August/046532.html`

eled as a data type; enumerators become interpreters, thus defining the semantics of iteratees as parsers of an enumerator's source. We show the compositional construction of iteratee-parsers and elucidate the algebraic laws that help design iteratee parser combinators and simplify iteratee programs. The accompanying code with derivations and several examples is available at `IterDeriv.hs`.

Our running example is reading lines from the standard input until the empty line, returning them in a list. The example is part of the common task of reading HTTP or e-mail headers. A line is a maximal sequence of non-newline characters. First, we write the example in the familiar C-style, with getChar – or its non-exceptional version getchar0, which, like the one in the C standard library, returns the current character or EOF.

```
type LChar = Maybe Char -- lifted character
getchar0 :: IO LChar
```

The function to read one line is later used to read all lines up to the empty line:

```
getline0  :: IO String
getline0  = loop ""
 where loop acc = getchar0 >>= check acc
       check acc (Just c) | c ≠ '\n' = loop (c:acc)
       check acc _                   = return (reverse acc)

getlines0  :: IO [String]
getlines0  = loop []
 where loop acc = getline0 >>= check acc
       check acc "" = return (reverse acc)
       check acc l  = loop (l:acc)
```

We may view getline0 and getlines0 as processes receiving lifted characters on a dedicated channel stdin and terminating with a value (a line or a list of lines). The simplest model represents such processes as a data type with a variant for each process operation – finished or inputting a character (see [4] for a good explanation of such modeling). We will call these processes *iteratees*[8].

```
data I a = Done a
         | GetC (LChar → I a)
```

Here is the data type model of the line reader

```
getline  :: I String
getline  = loop ""
 where loop acc = GetC (check acc)
       check acc (Just c) | c ≠ '\n' = loop (c:acc)
       check acc _                   = Done (reverse acc)
```

which looks almost identical to getline0. However, Done and GetC merely represent process operations. Terms like getlines hence do not "do" anything; they

[8] For the origin of the name, see `http://okmij.org/ftp/Scheme/enumerators-callcc.html`.

have to be interpreted so to run the corresponding process. Our interpreter uses a given finite string as the source of characters to send to the process.

```
eval :: String → I a → a
eval ""    (GetC k) = eval "" $ k Nothing
eval (c:t) (GetC k) = eval t  $ k (Just c)
eval str   (Done x) = x
```

When the string is exhausted, the process is sent EOF (that is, **Nothing**). Hopefully the process then finishes and we can return the produced result. One may view eval str i as a Unix pipeline `cat str | i`.

The data type I a has a rich structure. I a is a final co-algebra of the functor $T(X) = A + X^{LChar}$ – which helps us prove algebraic laws of iteratees below. The data type represents finitely branching trees with finite and infinite branches.[9] The interpreter eval s traces the path s in the tree. Last but not least, I a is a free monad (for good explanation and references, see [14]):

```
instance Monad I where
    return = Done

    Done x >>= f = f x
    GetC k >>= f = GetC (k >>> f)
```

(see Figure 1 for Kleisli composition (>>>)). Therefore, we may build iteratee processes by chaining simpler ones with the monadic (>>=) operation. For example, we chain getlines to build the process model of the reader of lines; the result looks identically to getlines0:

```
getlines :: I [String]
getlines = loop []
 where loop acc = getline >>= check acc
       check acc "" = return (reverse acc)
       check acc l  = loop (l : acc)
```

The next step is simple but momentous: we factor out eval into two interpreters, separating out the sending of data from the sending of EOF. The first factor feeds the characters, until there are no more data or the iteratee process is finished. The resulting iteratee is returned. The second interpreter tells the iteratee that there are no more data, and extracts its result.

```
en_str :: String → I a → I a
en_str ""    i = i
en_str (c:t) (GetC k) = en_str t $ k (Just c)
en_str _     (Done x) = Done x

run :: I a → a
run (GetC k) = run $ k Nothing
```

[9] Recall that Haskell data types are co-inductive, letting us construct infinite terms, such as getline.

```
run (Done x) = x
```

Clearly, eval str $\equiv$ run $\circ$ en_str str. We call en_str *enumerator*, and its argument str a source. The function run is analogous to eof of the UUparsing library [13]. Enumerators and run of the IterateeM library, Figure 1, look more general than en_str and run above since IterateeM lets iteratees and enumerators perform side effects in a monad m, when obtaining and processing input data. See App. A for such a generalization.

The point of factoring out eval is obtaining interpreters that can be functionally composed. Furthermore,

Equational Law 1 (Composition)

$$\mathit{en_str}\ (s1 \mathbin{+\!\!+} s2) \equiv \mathit{en_str}\ s2 \circ \mathit{en_str}\ s1$$

In words: the composition of enumerators corresponds to the concatenation of their sources. The law holds for more general effectful enumerators and iteratees, see App. A for the formulation and proof. If we overlook the last clause of en_str's definition, en_str is an instance of foldr (which inspired the names 'iteratee' and 'enumerator'). The law of composition therefore is hardly surprising.

Another law of en_str illustrates the compositionality of the iteratee semantics and lets us view iteratees as parsers and build parser combinator libraries. An iteratee is a value of the data type I a, which per se has "no semantics". The interpreter en_str gives a semantics to iteratees, as a function from finite strings to either Done v or GetC k. When the result of en_str s i is Done v, we say that the iteratee i has recognized the string s, parsing it to the value v. It follows from the law of composition that if i has recognized the string s, i recognizes s ++ s2 for any s2. We say that i *properly recognizes* the string s if i recognizes s but not any proper prefix of s.

Equational Law 2 (Chaining) *If iteratee i properly recognizes s1, then*

$$\mathit{en_str}\ (s1 \mathbin{+\!\!+} s2)\ (i \ggg\!= f) \equiv \mathit{en_str}\ s1\ i \ggg\!= \mathit{en_str}\ s2 \circ f$$

The proof of the general version of this law is given in App. A.

The law of chaining tells us how to build a recognizer for a string from the recognizers of the string's prefix and suffix, thus defining the meaning of the sequential iteratee composition ($\ggg\!=$). To represent choice we need a parallel composition: the left-biased alternation combinator.

```
(◁) :: I a → I a → I a
Done x ◁ _ = Done x
_ ◁ Done x = Done x
GetC k1 ◁ GetC k2 = GetC (\c → k1 c ◁ k2 c)
```

The parser i1 ◁ i2 recognizes whatever the first finishing parser recognizes; in the event of a tie, the result of i1 is preferred. Whereas the left and right unit of ($\ggg\!=$) is Done, the left and right unit of ◁ is failure, Figure 2, which keeps requesting input even after receiving EOF. It is a "diverging iteratee": run failure

diverges. (In IterateeM library, run reports an error if an iteratee asks for data after receiving EOF.)

Equational Law 3 (Zero) *The failure is the left zero of bind*

$$failure \gg= f \equiv failure$$

Equational Law 4 (Right distributivity)

$$i \gg= \backslash x \to (k1\ x \lhd k2\ x) \equiv (i \gg= k1) \lhd (i \gg= k2)$$

The law is similar to the law L10 in the parallel parser combinator library [1]. Since $\lhd$ commits to whatever a parser recognizes first, the left distributivity does not hold:

$$(i1 \lhd i2) \gg= k \not\equiv i1 \gg= k \lhd i2 \gg= k$$

Primitive parsers

```
failure  :: I a      -- The parser of nothing
failure  = GetC (const failure )

empty :: a → I a  -- The parser of the empty string
empty v = Done v

oneL :: I LChar    -- The parser of one lifted  character
oneL = GetC Done
```

Parser combinators: chaining and alteration

```
(≫= ) :: I a → (a → I b) → I b
(◁ )  :: I a → I a → I a
```

Derived parsers

```
-- The parser of a one-character string
one :: I Char
one = oneL ≫= maybe failure return

-- The parser of a character satisfying  the given predicate
pSat :: (LChar → Bool) → I LChar
pSat pred = oneL ≫= \c → if pred c then return c else failure
```

Fig. 2. Parser combinator library for simple iteratees

We thus arrive at the simple parser combinator library, Figure 2, which lets us *derive* the parsers getline and getlines that we previously built by intuition. First, we use the library to write the line reader in the 'obviously correct' way, expressing our definition of a line:

```
pGetline :: I String
pGetline = nl ◁ liftM2 (:) one pGetline
 where nl = do
```

```
        pSat (\c → c == Just '\n' || c == Nothing)
        return ""
```

It is quite inefficient: if the current character is not newline, nl turns into failure, which does nothing but keeps receiving characters and discarding them. Such wasteful operations should be eliminated. Noting that pSat and one start with oneL that can be factored out by the right distributivity, and applying the laws of failure gives us

```
pGetline' :: I String
pGetline' = oneL >>= check
 where check (Just '\n') = return ""
       check Nothing     = return ""
       check (Just c)    = liftM (c:) pGetline'
```

(See App. B for the complete equational derivation.) This iteratee is a non-tail-recursive version of getline that we wrote ad hoc earlier with explicit process constructors GetC and Done. (The tail-recursive conversion through accumulator is standard.) The correctness of getlines follows from the law of chaining.

The parser combinators in Figure 2 are efficient: the input stream is consumed character-by-character and is never backtracked. These parser combinators are somewhat similar to camlp4 parsers [11] in structuring a parser as a team of concurrent simple recursive-descent 'stream parsers'. Figure 2 library races the stream parsers in parallel until one succeeds. Camlp4 orders stream parsers by precedence and lets a higher-precedence parser run for a as long as it could. Camlp4 relies on look-ahead, which iteratee parser combinators in this section do not have (although it could be emulated in the continuation-passing style).

Look-ahead, or a fixed put-back, is an 'effect', to be expressed with effectful iteratees, see App. A. Effectful iteratees also permit a better error reporting. App. A thus lays out the way towards implementing the interface in Figure 1 and running our illustrative examples, §2.

4 Conclusions

We have introduced Iteratee IO, a compositional incremental input processing style with precise resource control. Like Lazy IO, it provides high abstraction, composability, combinator libraries, and on-demand IO. Because Iteratee IO is not lazy, the Iteratee library can ensure timely deallocation of resources, precise IO error handling, and strict control of reading actions. Incremental processing can now be guaranteed. Iteratee IO is therefore particularly suitable for programming long-running servers and processing large amounts of data. Compared to Handle IO, Iteratee IO is much higher level.

We have presented a view of iteratees as processes, represented by final co-algebras and free monads. The view shows how to reason with iteratees and implement them, motivating the basic design of iteratees and explaining their compositions. The theory of effectful iteratees clarifies the vexing issues of buffering and look-ahead. The iteratee libraries have many other features such as error

reporting, restartable exceptions, random IO, and merging of several streams. The iteratee-as-process view helps in understanding these advanced parts, too.

The capabilities and applications of Iteratee IO are still being discovered. For example, it was recently shown[10] that monadic regions and iteratees easily combine; therefore it is possible after all to write an exception-safe iterFile (an iteratee that writes the stream data to a file), ensuring the output file always closed.

The theory of effectful iteratees hints at the possibility of reasoning about computations with arbitrary IO effects (involving communication pipes, locking, shared memory, etc.), being very specific, at times, about the allocation of resources and the precise sequencing of operating system calls. We could derive observational equivalences of IO programs by extending equivalences of simple sample programs asserted by the programmer.

Even though Lazy IO compromises equational reasoning, it was introduced because Haskell was perceived – by its creators – as not expressive enough for incremental high-level IO: "We fear that there may be no absolutely secure system – that is, which one guarantees the Church-Rosser property – which is also expressive enough to describe the programs which systems programmers (at least) want to write..." [7, Sec 10.5]. The pessimism turns out unwarranted. We can write high-level programs with incremental IO and precise resource control – in safe Haskell.

Acknowledgments

I am very thankful to John W. Lato, Paulo Tanimoto, Johan Tibell and Alistair Bayley for extensive insightful discussions. Many helpful comments and suggestions from Gregory Collins, Jason Dagit, Nicolas Frisby, David Mazières, Chung-chieh Shan, Wren Ng Thornton and anonymous reviewers are gratefully acknowledged.

[10] http://www.haskell.org/pipermail/haskell-cafe/2012-January/098704.html

Bibliography

[1] Claessen, Koen. 2004. Parallel parsing processes. *Journal of Functional Programming* 14(6):741–757.

[2] Collins, Gregory David. 2011. snap-server: An iteratee-based http server library. `http://snapframework.com/docs/latest/snap-server/index.html`.

[3] Harrison, William L. 2006. The essence of multitasking. In *11th int. conf. on algebraic methodology and software technology (AMAST 2006)*, 158–172.

[4] Hinze, Ralf. 2001. Deriving backtracking monad transformers. In *ICFP*, 186–197. ACM Press.

[5] Kiselyov, Oleg. 2008. Incremental multi-level input processing with left-fold enumerator: predictable, high-performance, safe, and elegant. ACM SIGPLAN 2008 Developer Tracks on Functional Programming (DEFUN 2008).

[6] Lato, John W. 2010. Iteratee: Teaching an old fold new tricks. In *The monad.reader*, ed. Brent Yorgey, vol. 16, 19–35.

[7] Launchbury, John, and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

[8] Millikin, John. 2010. Understanding iteratees. `http://john-millikin.com/articles/understanding-iteratees/`.

[9] Parker, Conrad. 2011. Iteratees at Tsuru Capital. `http://blog.kfish.org/2011/09/iteratees-at-tsuru.html`.

[10] Quick, Kevin. 2011. Fun with the ST monad. `http://www.haskell.org/pipermail/haskell-cafe/2011-February/089689.html`.

[11] de Rauglaudre, Daniel. 2003. Camlp4 - Reference Manual, version 3.07. `http://caml.inria.fr/pub/docs/manual-camlp4/`.

[12] Shaw, Mary, William A. Wulf, and Ralph L. London. 1977. Abstraction and verification in Alphard: defining and specifying iteration and generators. *Communications of the ACM* 20(8):553–564.

[13] Swierstra, S. Doaitse. 2008. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, vol. 5520 of *LNCS*, 252–300. Springer.

[14] Swierstra, Wouter. 2008. Data types á la carte. *Journal of Functional Programming* 18(4):423–436.

[15] Tarau, Paul. 2000. Fluents: A refactoring of Prolog for uniform reflection and interoperation with external objects. In *Computational Logic: First international conference*, ed. John Lloyd. LNCS 1861.

[16] Thornton, Wren Ng. 2011. Fun with the ST monad. `http://www.haskell.org/pipermail/haskell-cafe/2011-February/089687.html`.

[17] Yamamoto, Kazu. 2011. A tutorial on the enumerator library. `http://www.mew.org/~kazu/proj/enumerator/`.

[18] Yang, Edward Z. 2012. Why iteratees are hard to understand. `http://blog.ezyang.com/2012/01/why-iteratees-are-hard-to-understand/`.

A Effective iteratees

The data received by an iteratee may come from a file or a network. To get a chunk of that data, an enumerator had to perform IO. An iteratee may also need effects, e.g., to write a log, report exceptions, rewind the input stream. The theory of effectful iteratees developed in this section applies to all these cases. The theory extends the final-coalgebra representation of iteratee processes introduced in §3. We generalize the equational laws stated in §3 and prove them. The full details are in the accompanying source code `IterDerivM.hs`.

As in §3, we view iteratees as processes, modeled by a final co-algebra of their operations – terminated or requesting a character. After receiving a character, the iteratee may now incur an effect, in an arbitrary monad m:

```
data I m a = Done a
           | GetC (LChar → m (I m a))
```

The model of the line reader process below looks identically to that of getline in §3. Indeed, this line reader had no effects besides requesting a character.

```
getline :: Monad m ⇒ I m String
getline = loop ""
 where
 loop acc = GetC (check acc)
 check acc (Just c) | c ≠ '\n' = return (loop (c:acc))
 check acc _                   = return (Done (reverse acc))
```

Let us introduce an effect, of emitting a string:

```
class Monad m ⇒ PutS m where
  putS :: String → m ()

instance PutS IO where
  putS = putStrLn
```

so that we may model a line reader that writes the debugging trace of each received character:

```
getlineT :: (PutS m, Monad m) ⇒ I m String
getlineT = loop ""
 where
 loop acc = GetC (trace acc)
 trace acc c = putS ("got␣" ++ show c) ≫ check acc c
 check acc (Just c) | c ≠ '\n' = return (loop (c:acc))
 check acc _                   = return (Done (reverse acc))
```

The interpreters of the iteratee term representation – enumerators and run – are defined as before, §3. The presence of the "call-by-value application" (=≪) reveals that the evaluation order now matters.

```
en_str :: Monad m ⇒ String → I m a → m (I m a)
```

```
en_str ""      i        = return i
en_str (c:t) (GetC k) = en_str t =<< k (Just c)
en_str _ i@Done{} = return i

run :: Monad m => I m a -> m a
run (Done x) = return x
run (GetC k) = run =<< k Nothing
```

As in §3, monadic operations let us compose iteratee processes: I m is a monad – a free monad.

```
instance Monad m => Monad (I m) where
  return = Done

  Done x >>= f = f x
  GetC k >>= f = GetC (k >>> (return . (>>= f)))
```

Somewhat surprisingly (since monads do not generally compose), the composition of m and I m is also a monad, with the following bind operation

```
type IM m a = m (I m a)
bind :: Monad m => IM m a -> (a -> IM m b) -> IM m b
bind m f = m >>= check
 where
 check (Done x) = f x
 check (GetC k) = return (GetC (\c -> bind (k c) f))
```

We hence combine the logging line reader getlineT to read several lines until the empty line:

```
getlinesT :: (PutS m, Monad m) => I m [String]
getlinesT = loop []
 where
 loop acc = getlineT >>= check acc
 check acc "" = return (reverse acc)
 check acc l  = loop (l:acc)
```

Here is the complete example of reading lines from a given string, printing each character as it is being processed.

```
t111 = print =<< run =<< en_str "abd\nxxx\nf" getlinesT
```

The equational laws of iteratees and enumerators, §3, generalize to the effectful case.

Equational Law 5 (Effectful Composition)

$$en_str\ (s1 +\!\!+ s2) \equiv en_str\ s1 \ggg en_str\ s2$$

Here s1 must be a finite string; s2 is arbitrary. The law reads just like the original law of composition, this time, in terms of Kleisli composition. Let us prove it.

If we pass the Done x iteratee to the enumerators on both sides of the equality, the results are clearly equal: en_str s (Done x) ≡ return (Done x) regardless of s. The other case is the enumerators applied to a GetC k iteratee, which we prove by induction on s1. In the base case, the empty s1, en_str "" is return, which is the unit of Kleisli composition. The inductive case:

```
en_str ((c: s1') ++ s2) (GetC k)
≡ -- property of list append
en_str (c:( s1' ++ s2)) (GetC k)
≡ -- second clause of en_str
en_str (s1' ++ s2) =<< k (Just c)
≡ -- inductive hypothesis
(en_str s1' >>> en_str s2) =<< k (Just c)
≡ -- definition of (>>>)
k (Just c) >>= (\x → en_str s1' x >>= en_str s2)
≡ -- associativity of bind
(k (Just c) >>= en_str s1') >>= en_str s2
≡ -- second clause of en_str
en_str (c: s1') (GetC k) >>= en_str s2
≡ -- definition of (>>>)
(en_str (c: s1') >>> en_str s2) (GetC k)
```

The law of chaining of §3 becomes:

Equational Law 6 (Effectful Chaining) *1. If the iteratee i properly recognizes s1, then*

en_str (s1++ s2) (i >>= f) ≡ en_str s1 i ‘bind‘ en_str s2 ∘ f

2. If the iteratee i does not recognize s, then

en_str s (i >>= f) ≡ en_str s i >>= (return ∘ (>>= f))

The proof of part 1 is by induction on s1, which must be finite by the definition of proper recognition. In the base case, the iteratee i properly recognizing the empty string s1 is Done x and so en_str s1 i is return (Done x), which is the left unit of bind. In the inductive case, the iteratee i properly recognizes the string c:s1'. Therefore, i must have the form GetC k where k (**Just** c) is an action that must yield an iteratee i' properly recognizing s1'. We calculate:

```
en_str ((c: s1') ++ s2) (GetC k >>= f)
≡ -- property of list append
en_str (c:( s1' ++ s2)) (GetC k >>= f)
≡ -- definition of bind of (I m)
en_str (c:( s1' ++ s2)) (GetC (k >>> (return ∘ (>>= f))))
≡ -- second clause of en_str definition
en_str (s1' ++ s2) =<< (k >>> (return ∘ (>>= f))) (Just c)
≡ -- definition of (>>>)
en_str (s1' ++ s2) =<< (k (Just c) >>= (return ∘ (>>= f)))
```

```
≡ -- rearrangements
(k (Just c) >>= (return ∘ (>>= f))) >>= en_str (s1' ++ s2)
≡ -- associativity of bind
k (Just c) >>= (\i' → return (i' >>= f) >>= en_str (s1' ++ s2))
≡ -- left unit law
k (Just c) >>= (\i' → en_str (s1' ++ s2) (i' >>= f))
≡ -- inductive hypothesis: i' does properly recognize s1'
k (Just c) >>= (\i' → en_str s1' i' `bind` en_str s2 ∘ f)
≡ -- a property of bind: m `bind` f ≡ m >>= (\x → return x `bind` f)
k (Just c) >>= (\i' →
   en_str s1' i' >>= (\x → return x `bind` en_str s2 ∘ f))
≡ -- associativity
(k (Just c) >>= en_str s1') >>=
  (\x → return x `bind` en_str s2 ∘ f)
≡ -- second clause of en_str definition
en_str (c:s1') (GetC k) >>= (\x → return x `bind` en_str s2 ∘ f)
≡ -- the same property of bind
en_str (c:s1') (GetC k) `bind` en_str s2 ∘ f
```

The proof of part 2 is analogous, see `IterDerivM.hs` for the complete derivation. Since the proofs relied only on monad laws, the laws of effectful composition and chaining hold for *any effect* whatsoever.

The divergent failure iteratee now reads

```
failure :: Monad m ⇒ I m a
failure = GetC (const (return failure ))
```

The law Zero of §3 remains the same for effectful iteratees: failure >>= f ≡ failure. The proof is trivial bisimulation.

The left-biased alternation of effectful iteratees has the form

```
(◁) :: Monad m ⇒ I m a → I m a → I m a
i@Done{} ◁ _ = i
_ ◁ i@Done{} = i
GetC k1 ◁ GetC k2 = GetC (\c → liftM2 (◁) (k1 c) (k2 c))
```

To state the right-distributivity law, we need a definition: An iteratee i is idempotent if

```
en_str s i >>= \x → return (x,x) ≡
en_str s i >>= \x → en_str s i >>= \y → return (x,y)
```

for any finite string s. The right distributivity law has the same form as given in §3 – with the side-condition that i must be an idempotent iteratee.

The proof is by bi-similarity. We define the relation R on iteratees as a set of all pairs (iA,iB) where

```
iA = i >>= \x → (k1 x ◁ k2 x)
iB = (i >>= k1) ◁ (i >>= k2)
```

and we consider all observations of related iteratees. Here we only show the case when i is of the form GetC k for some k, in which case iA is GetC kA and iB is GetC kB for some kA and kB. The full proof is in `IterDerivM.hs`. If we feed iA a character c, we observe

```
case GetC k >>= \x → (k1 x ◁ k2 x) of GetC kA → kA c
≡ −− definition on (>>= )
(k >>> (return ∘ (>>= \x → (k1 x ◁ k2 x)))) c
≡
k c >>= \i' →
return (i' >>= \x → (k1 x ◁ k2 x))
```

For iteratee iB, we have

```
case (GetC k >>= k1) ◁ (GetC k >>= k2) of GetC kB → kB c
≡ −− definitions
case GetC (k >>> (return ∘ (>>= k1))) ◁
     GetC (k >>> (return ∘ (>>= k2)))
 of GetC kB → kB c
≡
liftM2 (◁ )
 (k c >>= \ix → return (ix >>= k1))
 (k c >>= \iy → return (iy >>= k2))
≡ −− definition of liftM2
(k c >>= \ix → return (ix >>= k1)) >>= \i1 →
(k c >>= \iy → return (iy >>= k2)) >>= \i2 →
return (i1 ◁ i2)
≡ −− associativity, unit laws
k c >>= \ix → k c >>= \iy →
return ((ix >>= k1) ◁ (iy >>= k2))
≡ −− monad unit law
k c >>= \ix → k c >>= \iy → return (ix, iy) >>=
\ (ix, iy) → return ((ix >>= k1) ◁ (iy >>= k2))
≡ −− idempotence
k c >>= \ix → return (ix, ix) >>=
\ (ix, iy) → return ((ix >>= k1) ◁ (iy >>= k2))
≡ −− monad laws
k c >>= \i' →
return ((i' >>= k1) ◁ (i' >>= k2))
```

Thus feeding c to the related iA and iB incurs the same effect (associated with k c) and produces the iteratees that are also related by R.

Treating look-ahead as an effect, the file `IterDerivM.hs` generalizes the parser combinators of Figure 2 for look-ahead. Buffering, the processing of input by chunks rather than by individual characters, can be handled similarly.

B Deriving the optimal version of pGetline

This section shows the detailed steps in the conversion of the "obviously correct" but grossly inefficient line reader pGetLine, §3, to the efficient version pGetline'. The conversion relies on the equational laws of iteratees.

Our starting point is pGetline written using the iteratee parsing combinators from Figure 2:

```
pGetline :: I String
pGetline = nl ◁ liftM2 (:) one pGetline
 where nl = do
         pSat (\c → c == Just '\n' || c == Nothing)
         return ""
```

First, we inline the definitions of one and pSat and desugar the do-notation:

```
pGetline1 = nl ◁ char
 where
   nl = (oneL >>= \c → if c == Just '\n' || c == Nothing
                        then return c else failure ) >> return ""
   char = (oneL >>= maybe failure return ) >>= \c → liftM (c:) pGetline1
```

We re-associate the bind chains to the right:

```
pGetline2 = nl ◁ char
 where
   nl = oneL >>= (\c → if c == Just '\n' || c == Nothing
                        then return c >> return ""
                        else failure  >> return "")
   char = oneL >>= (\c → maybe failure return c >>= \c →
                          liftM (c:) pGetline2 )
```

distribute bind into case and apply Monad and Zero laws:

```
pGetline3 = nl ◁ char
 where
   nl = oneL >>= (\c → if c == Just '\n' || c == Nothing
                        then return ""
                        else failure )
   char = oneL >>= (\c → case c of
                           Just c → liftM (c:) pGetline3
                           Nothing → failure )
```

and the right-distributivity law:

```
pGetline4 = oneL >>= \c → nl' c ◁ char' c
 where
   nl' c = if c == Just '\n' || c == Nothing
            then return "" else failure
   char' c = case c of
```

```
Just c → liftM (c:) pGetline4
Nothing → failure
```

We pull out the case analysis on the read character, essentially "narrowing"

```
pGetline5 = oneL >>= check
 where
   check (Just '\n') = return "" ◁ liftM ('\n':) pGetline5
   check Nothing     = return "" ◁ failure
   check (Just c)    = failure   ◁ liftM (c:) pGetline5
```

The facts that failure is the left and the right unit of ◁ , and return x is its left zero give us pGetLine'.

C The plumbing intuition for Iteratee IO

The diagrammatic notation for iteratee programs introduced in this section helps visualize the flow of input data, giving an idea of iteratee processing at a glance. The notation is inspired by the "Piping and Instrumentation Diagram Standard Notation"[11] used in Industrial Engineering for a similar purpose. For illustration, we show the diagrams for the examples in §2.

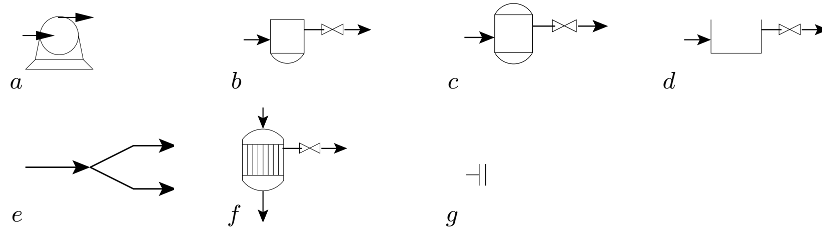
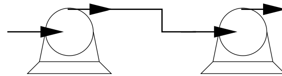


Fig. 3. Notation for primitive components and combinators

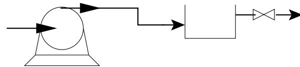
Figure 3 describes the notation for the Iteratee library components, Figure 1. Enumerator (a) is a pump, pumping data so long as it can flow. When the consumer is saturated and will not accept more data, the pressure rises and the pump shuts off. A general iteratee (b) is a reservoir with an overflow pipe. When the reservoir is filled up (i.e., the iteratee is Done), the further input data flow through the overflow pipe to the next iteratee in chain. The overflow pipe is shut by default. When the reservoir fills up (that is, the iteratee gets all data it needs) and there is no further iteratee, the data stream has nowhere to flow, the pressure rises and the pump (enumerator) shuts off. The iteratee getchar (c) is a small reservoir, which can only hold a single byte. In contrast, count_i (d) is an open reservoir, accepting any amount of input data. The pairing combinator en_pair is the Y-connector (e), splitting the stream in two. Enumeratee is

[11] See the example, `https://controls.engin.umich.edu/wiki/index.php/PIDStandardNotation`

a reactor (f), transforming the incoming (top) stream to the stream of 'reactor products'. When the bottom flow stops, that is, the iteratee consuming the reactor product finishes, the incoming flow continues through the right (overflow) end. The combinator (.|) (g) terminates that overflow pipe. There are cases (not shown in the paper) when the overflow continues: for example, when processing multi-part MIME messages. Kleisli composition of enumerators connects pumps in sequence:

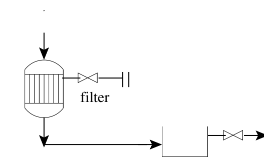


We now show the plumbing diagrams for the examples in §2. We start with the simplest pipeline (too simple to mention in §2) that measures the output of a pump: enum_file fname count_i.



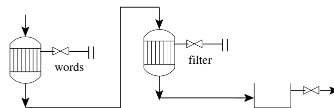
The white-space gauge:

```
countWS'_iter = id .| (en_filter isSpace) count_i
```



The gauge for "the":

```
countTHE_iter = id .| enum_words .| en_filter (== "the") count_i
```



The counter of both "the" and the whitespace in the prefix of the input stream

```
run_nterml n fname =
  print =<< run =<< enum_file fname .|
   IterateeM.take n (countWS_iter `en_pair` countTHE_iter)
```

is drawn as follows.