High Level Iteratees

or: An Iteratee Tutorial Tutorial

1. Introduction

Every iteratee implementation and tutorial I've seen so far is at least moderately low-level. I find the usual presentation does quite a bit to obscure the topic. It took quite a few implementations and introductions before I even had a good idea of what an iteratee is and why I'd use one. Since then, every serious use I've made of iteratee libraries has involved writing at least one function that did something conceptually quite simple in a mind-bending, convoluted way. Even if that is entirely due to my own ineptitude, the fact that I have read tutorials and failed to learn from them a better way to do such simple tasks is puzzling.

I have yet to find even a single tutorial in which every example iteratee, or even the majority of them, is written solely in terms of high level primitives. Instead, the style they encourage involves directly implementing complex continuation-passing logic. Until the day a comprehensive iteratee tutorial can be written without a single explicit "feed me" continuation, iteratees are just not ready for the average programmer.

The vast majority of this complexity is incidental to the iteratee concept and can be eliminated by further abstraction. Toward that end, I'd like to present yet another iteratee implementation, based on a radically different approach to the problem: defining iteratees as a stack of already well-understood monad transformers. By doing so, I hope to develop a clear operational semantics for at least one interpretation of the iteratee concept.

This approach will be presented in three parts. First, section 2.1 works through a simple top-down design process starting with a laundry list of features that iteratees should have and gradually building up a monad transformer stack that implements them. Second, section 2.2 demonstrates a bottom up analysis looking at an existing iteratee implementation and reverse-engineering it into an equivalent monad transformer stack. Finally, section 3 explores one possible high-level formulation of enumerators that complements the iteratees developed in section 2.

This is not intended to be the one true specification or even a serious implementation. Rather, consider this an example of an approach for formally specifying the semantics, an alternative to the usual approach where the implementation is the only specification.

The primary reason, I believe, that a treatment like this one is not already commonplace is the challenge of finding a monad transformer that effectively models interruptible computations without opening the *ContT* Pandora's Box. The rest of the things we expect of iteratees are easy to provide with well-known monad transformers. The *Program* monad and its associated transformer *ProgramT* (from the "operational" package) do exactly that. Iteratees can be viewed as *Program* monads with a very simple instruction set consisting of just one operation: Get more input. I believe this is the only previously-missing piece of the "iteratee semantics" puzzle.

2. Iteratees

2.1. Top-down Iteratee Design. This section will be introducing several different iteratee implementations that incrementally add features, so I'll start by defining those features in terms of a few type classes that express the primitive operations associated with them.

2.1.1. *Fundamentals: What an iteratee* is. First, the bare minimum: As I see it, to be an iteratee is to be a process that consumes input and eventually returns a response. Thus, *Iteratee* is a subclass of *Monad* with a single operation, *getInput*, that asks for more input. The other side of that operation, the means by which an enumerator feeds that input to the iteratee, will be discussed in another section. A few useful operations on streams will be defined in Appendix A and used throughout the rest of the code.

```
data Stream sym = EOF | Chunks [sym] deriving (Eq, Show)
type IterStream it = Stream (Symbol it)
class Monad it ⇒ Iteratee it where
  type Symbol it
  getInput :: it (IterStream it)
```

2.1.2. *A practical consideration: look-ahead.* Second, for practical iteratees we'll also want lookahead of a limited sort. We want to be able to get a prefix of the available input without consuming all of it. We also want to be able to see what input is available without consuming any of it or causing the enumerator to do any additional work. Traditionally, an *unget* operation is also provided, but I prefer not to include it even though all of the implementations here could support one. It really doesn't seem like it ought to be possible to "put back" arbitrary data that may or may not ever have been read from the stream in the first place. In a real-world implementation I might expect to see an *unget* operation in a separate `.Internal` module or something, with its use discouraged and a tacit expectation that speed freaks will probably make use of it anyway.

```
class Iteratee it ⇒ Lookahead it where
  getSymbols :: Int → it (IterStream it)
  lookahead  ::        it (IterStream it)
```

2.1.3. *Exception handling.* (Traditional) exception handling: The *MonadError* class is actually sufficient for this purpose, but let's a new class that is explicitly about iteratees anyway just for emphasis.

```
class Iteratee it ⇒ IterateeError it where
  type Exc it
  throw  :: Exc it → it a
  handle :: (Exc it → it a) → it a → it a
```

2.1.4. *Implementations.* Now for some implementations. Here's a minimalist iteratee, without support for any of the fancy stuff like lookahead, exceptions, etc. Since we're working primarily with monad transformers, it's trivial for us to include an underlying monad in our simplest example, so we'll go ahead and do so.

```
data Fetch sym a where
  Fetch :: Fetch sym (Stream sym)

newtype Iter1 sym m a = Iter1 (ProgramT (Fetch sym) m a)
  deriving (Functor, Monad, MonadTrans)

runIter1 (Iter1 p) = viewT p >>= step
  where
    step :: Monad m => ProgramViewT (Fetch sym) m a -> m a
    step (Return x)       = return x
    step (Fetch :>>= k) = viewT (k EOF) >>= step

instance Monad m => Iteratee (Iter1 sym m) where
  type Symbol (Iter1 sym m) = sym
  getInput = Iter1 (singleton Fetch)
```

We can easily add lookahead by throwing a state monad onto the stack (note that *Iter2*'s *getInput* doesn't call *Iter1*'s *getInput* unless the *Stream* state is *EOF* or *Chunks* [ ]):

```
newtype Iter2 sym m a = Iter2 (Iter1 sym (StateT (Stream sym) m) a)
  deriving (Functor, Monad)
instance MonadTrans (Iter2 sym) where
  lift = Iter2 . lift . lift

runIter2 (Iter2 i) = runStateT (runIter1 i) (Chunks [])

instance Monad m => Iteratee (Iter2 sym m) where
  type Symbol (Iter2 sym m) = sym
  getInput = do
    stashed <- lookahead
    Iter2 $ if isEmpty stashed
      then do
        input <- getInput
        lift (put (takeStream 0 input))
        return input
      else do
        lift (put (Chunks []))
        return stashed

instance Monad m => Lookahead (Iter2 sym m) where
  lookahead = Iter2 (lift get)
  getSymbols n = do
    input <- getInput
    if isEOF input
      then return EOF
      else do
```

```
      let (result, rest) = splitStreamAt n input
          nResults       = streamLength result
      Iter2 (lift (put rest))
      if isEmpty rest ∧ nResults < n
        then do
          more ← getSymbols (n − nResults)
          return (appendStream result more)
        else return result
```

To that we can add exception handling with *ErrorT*:

```
newtype Iter3 e sym m a = Iter3 (ErrorT e (Iter2 sym m) a)
  deriving (Functor, Monad, MonadError e)
instance Error e ⇒ MonadTrans (Iter3 e sym) where
  lift = Iter3 ∘ lift ∘ lift
runIter3 (Iter3 i) = runIter2 (runErrorT i)
instance (Error e, Monad m) ⇒ Iteratee (Iter3 e sym m) where
  type Symbol (Iter3 e sym m) = sym
  getInput = Iter3 (lift getInput)
instance (Error e, Monad m) ⇒ Lookahead (Iter3 e sym m) where
  lookahead  = Iter3 ( lift  lookahead)
  getSymbols = Iter3 ∘ lift ∘ getSymbols
instance (Error e, Monad m) ⇒ IterateeError (Iter3 e sym m) where
  type Exc (Iter3 e sym m) = e
  throw  = throwError
  handle = flip catchError
```

There are many other interesting monad transformers that we could put into the stack. There are also other interesting constructors we could add to our *ProgramT*'s "instruction" GADT. For example, either approach could be used to implement resumable exceptions (either by adding another *ProgramT* layer or by adding constructors to the *Fetch* GADT to reperesent exceptions). This is why they appear to be such a natural fit in Oleg's implementation: As we'll see later, his exception system is equivalent to the latter.

This construction of iteratees requires a much broader knowledge base to digest than the existing bottom-up presentations, and the code involves a fair amount of syntactic noise with all the lifting and newtype wrapping and unwrapping. Ultimately, though, I find it considerably simpler to understand because it involves combining a small number of already-well-understood concepts. And to be honest I think that most people that are able to really understand any of the existing expositions of iteratees probably can digest this one as well.

Much more importantly, having a monad transformer stack as a reference implementation allows implementors to make their iteratees vastly simpler to use; a complete set of primitive operations can be derived as a combination of the primitive operations of each of the monad-transformer layers, minus anything the implementor prefers to keep abstract. This benefit comes merely from the existence of this kind of model - the implementation need not be the same. It can be quite aggressively refactored or optimized as long as it provides the set of primitives chosen from the reference model.

2.2. **Bottom-up Iteratee Analysis.** That last point raises an interesting question. If we already have an implementation of iteratees, can we easily "retrofit" a monad-transformer-stack semantics from which to derive an appropriate set of primitives? Here is a somewhat informal procedure I have found useful for this purpose.

We'll start by looking at an existing implementation and rewriting it in a simple type-structural notation. Let's use the implementation from Oleg Kiselyov's original IterateeM.hs as a worked example:

```
type ErrMsg = SomeException
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))
```

Which gives us the basic equations:

$$\begin{aligned} ErrMsg &= SomeException \\ Stream(el) &= 1 + ErrMsg + List(el) \\ Iteratee(el, m, a) &= a + (1 + ErrMsg) * (m(Iteratee(el, m, a) * Stream(el))^{Stream(el)}) \end{aligned}$$

(we won't need to look inside streams, let alone lists, but for completeness recall that $List(x) = 1 + x * List(x)$)

Now, considering this all as a symbolic algebra problem, combine them with some functions corresponding to known monad transformers and manipulate the equations till they reach a suitably simple form. Here are some equations corresponding to a few monad transformers I've found useful:

$$\begin{aligned} ReaderT(r, m, a) &= m(a)^r \\ WriterT(w, m, a) &= m(a * w) \\ StateT(s, m, a) &= ReaderT(s, m, WriterT(s, m, a)) \\ &= m(a * s)^s \\ ErrorT(e, m, a) &= m(e + a) \\ ProgramT(instr, m, a) &= m(ProgramViewT(instr, m, a)) \\ ProgramViewT(instr, m, a) &= a + \sum_t (instr(t) * ProgramT(instr, m, a)^t) \end{aligned}$$

The ProgramViewT equation probably requires a bit of explanation. The $\sum_t$ component corresponds to existential quantification over a new type variable t. Interpreting the *instr* type as a function that maps each set of type parameters to the equation for all constructors that can yield that assignment of type parameters, the notation means exactly what the summation operation suggests. For example, the following GADT:

data *Foo a b* **where**
  Bar :: *Int* → *String* → *Foo X Y*
  Baz :: *Foo Z Z*

would map to the following function (written in a pseudo-Haskell style with pattern matching on the type arguments):

$$\begin{aligned} Foo(X,Y) &= Int * String \\ Foo(Z,Z) &= 1 \\ Foo(\_,\_) &= 0 \end{aligned}$$

so $\sum_{a,b}(Foo(a,b) * Bar(a))$ would expand to $Foo(X,Y) * Bar(X) + Foo(Z,Z) * Bar(Z)$, which (by evaluating Foo) simplifies to $Int * String * Bar(X) + Bar(Z)$. So there is at least one isomorphism between $\exists a, b.\ (Foo\ a\ b, Bar\ a)$ and $Either\ (Int, String, Bar\ X)\ (Bar\ Z)$.

With all this in mind, here are two rewrites of the *Iteratee* equations above. We first hypothesize, based on superficial similarities of the corresponding equations, that $Iteratee(el, m, a)$ is isomorphic to $ProgramViewT(f, n, b)$ for some f, n and b:

$$Iteratee(el, m, a) = ProgramViewT(f, n, b)$$

$$\begin{aligned} & a + (1 + ErrMsg) * (m(Iteratee(el, m, a) * Stream(el))^{Stream(el)}) \\ = \quad & b + \sum_t (f(t) * n(ProgramViewT(f, n, b))^t) \end{aligned}$$

From here, $a = b$ is an easy assumption, which leaves:

$$\begin{aligned} & (1 + ErrMsg) * (m(Iteratee(el, m, a) * Stream(el))^{Stream(el)}) \\ = \quad & \sum_t (f(t) * n(ProgramViewT(f, n, b))^t) \end{aligned}$$

There are at least two possible ways to unify these two expressions, arising from different possible choices of the function f. One way is to declare that f has only one value in its range: $Stream(t)$. We can let f be either:

$$\begin{aligned} F(el, 1 + ErrMsg) &= Stream(el) \\ F(el, \_) &= 0 \end{aligned}$$

or

$$\begin{aligned} F(el, 1) &= Stream(el) \\ F(el, ErrMsg) &= Stream(el) \\ F(el, \_) &= 0 \end{aligned}$$

These assignments correspond to GADTs:

data *F el t* **where**
  F :: *Maybe ErrMsg* → *F el* (*Stream el*)

or

data *F el t* **where**
  F1 :: *F el* (*Stream el*)
  F2 :: *ErrMsg* → *F el* (*Stream el*)

respectively. So (informally currying F):

$$\begin{aligned}
& (1+ErrMsg)*(m(Iteratee(el,m,a)*Stream(el))^{Stream(el)}) \\
= \; & (1+ErrMsg)*n(ProgramViewT(F(el),n,b)^{Stream(el)}) \\
= \; & (1+ErrMsg)*n(Iteratee(el,m,a)^{Stream(el)})
\end{aligned}$$

$$n(Iteratee(el,m,a)) = m(Iteratee(el,m,a)*Stream(el))$$

$$\begin{aligned}
n &= WriterT(Stream(el),m) \\
Iteratee(el,m,a) &= ProgramViewT(F(el),WriterT(Stream(el),m),a)
\end{aligned}$$

It is important to note that this is only an isomorphism of types, and in particular does *NOT* say that the *Monad* operations that would be provided by library implementations of these monad transformers are the same as the original implementation's. The fact that we have an isomorphism of types, though, means that we can push the implementation's existing operations through to the new type. This is also a valuable exercise because it lets us restate exactly what the implementation was doing in a language of our choosing or, as in this case, see that the type we have come up with is really not a very good fit after all.

Oleg's implementation had state-passing machinery in $\gg=$ which, when pushed through our isomorphism, makes enumerators responsible for knowing when the iteratee has returned unconsumed input and feeding it back to them before generating any more. So although we can shoehorn the iteratee concept into a writer monad, doing so puts an unnecessary burden on our enumerators and really does not capture the spirit of what's going on.

Let's go back now to the choice of f (which we'll start calling g to emphasize that we are now making a different choice of function). We'll try another sensible function in an effort to find a type more like a state monad. Let the unit type be the only element of g's range:

$$\begin{aligned}
G(1+ErrMsg) &= 1 \\
G(_) &= 0
\end{aligned}$$

or, as a GADT:

data G **where**
  $G :: Maybe\ ErrMsg \rightarrow G\ ()$

We can see from its type that G is an operation with *only* side effects. This reinforces Oleg's identification of the iteratee's continuation as a "resumable exception". It requests outside intervention to make things better - more input, handle some exception, etc. Our equations are now:

$$\begin{aligned}
& (1+ErrMsg)*(m(Iteratee(el,m,a)*Stream(el))^{Stream(el)}) \\
= \; & \sum_t (G(t)*n(ProgramViewT(G,n,a))^t) \\
= \; & (1+ErrMsg)*n(ProgramViewT(G,n,a))^1 \\
= \; & (1+ErrMsg)*n(ProgramViewT(G,n,a)
\end{aligned}$$

$$\begin{aligned}
& n(ProgramViewT(G,n,a) \\
= \; & (m(Iteratee(el,m,a)*Stream(el))^{Stream(el)}) \\
= \; & StateT(Stream(el),m,Iteratee(el,m,a))
\end{aligned}$$

Putting everything together[1]:

$$\begin{aligned}
& Iteratee(el,m,a) \\
= \; & a+(1+ErrMsg)*(m(Iteratee(el,m,a)*Stream(el))^{Stream(el)}) \\
= \; & a+(1+ErrMsg)*StateT(Stream(el),m,Iteratee(el,m,a)) \\
= \; & a+\sum_t (G(t)*StateT(Stream(el),m,ProgramViewT(G,n,a))^t) \\
= \; & ProgramViewT(G,StateT(Stream(el),m),a)
\end{aligned}$$

This is a nicer conclusion than the previous one because it formalizes Oleg's informal remarks that an iteratee is a kind of a state monad. From this definition we can see that it really is. Keep in mind, again, that the operations we get for free from the library implementations are not *exactly* the same as the ones we get when we push the original type's capabilities through our isomorphism. It's mostly a standard *ProgramT* monad, but due to the original implementation of $\gg=$, the *G Nothing* operation is effectively a no-op unless the state is empty.

It is an eye-opening (and highly recommended) exercise to perform this sort of derivation for several different implementations of iteratees and compare the resulting transformer stacks. It's particularly interesting to note just how widely varied the semantics are.

3. Enumerators

Going back to the code in section 2.1, I'd like to add what I find to be an elegant definition of enumerators. Essentially, an enumerator is a state monad over an arbitrary iteratee. I would just use *StateT* but I also want to require my definition of *Enumerator* to be independent of the iteratee and its return type. To do so with *StateT* would require impredicative types, which are deprecated these days.

I haven't yet written much of any explanation about this code, so take it as a brain-dump. There is probably a lot of room for improvement and clarification.

---

[1]Incidentally, the fact that this *Iteratee* is equivalent to *ProgramViewT* and not to *ProgramT* exposes a subtle problem with the implementation (although it will have been obvious to some already): This *Iteratee* type is *NOT* a monad transformer. It violates the law that requires $lift \circ return = return$. $lift\ (return\ x)$ will ask the enumerator for input before passing on x to the rest of the program. I'm not entirely sure whether it obeys the monad laws either, though I suspect it does. I have checked the identity laws but not associativity.

Note that $\mathit{feed}\ \mathit{enum}\ \mathit{iter1} \gg \mathit{iter2} \not\equiv \mathit{feed}\ \mathit{enum}\ (\mathit{iter1} \gg \mathit{iter2})$ - This is unavoidable, and really should be expected: if *iter2* asks for input, it's just too late in the first case for *enum* to respond, while in the second *enum* has no way to distinguish which input requests come from which iteratee. Additionally, it is very much an open question whether in the former case the remaining input (if any) from *iter1* should be available in *iter2* or should be silently discarded. Ultimately, I am of the opinion that the former pattern of calls really just ought to be discouraged.

$\mathbf{class}\ \mathit{Monad}\ \mathit{it} \Rightarrow \mathit{Iterable}\ \mathit{it}\ \mathit{sym}\ \mathbf{where}$
$\quad \mathit{step} :: \mathit{it}\ a \to \mathit{it}\ (\mathit{Either}\ (\mathit{Stream}\ \mathit{sym} \to \mathit{it}\ a)\ a)$

$\mathbf{newtype}\ \mathit{Enum1}\ \mathit{sym}\ m\ a = \mathit{Enum1}\ (\forall \mathit{it}\ t. \mathit{Iterable}\ \mathit{it}\ \mathit{sym} \Rightarrow \mathit{it}\ t \to \mathit{it}\ (\mathit{it}\ t, a))$

Imagine *Enum1* as:

$\mathbf{newtype}\ \mathit{Enum1}\ \mathit{sym}\ m\ a = \mathit{Enum1}\ (\mathit{StateT}\ (\forall \mathit{it}\ t. \mathit{Iterable}\ \mathit{it}\ \mathit{sym} \Rightarrow \mathit{it}\ t)\ m\ a)$

$\mathbf{instance}\ \mathit{Functor}\ m \Rightarrow \mathit{Functor}\ (\mathit{Enum1}\ \mathit{sym}\ m)\ \mathbf{where}$
$\quad \mathit{fmap}\ f\ (\mathit{Enum1}\ e) = \mathit{Enum1}\ (\mathit{liftM}\ (\mathit{fmap}\ f) \circ e)$
$\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{Enum1}\ \mathit{sym}\ m)\ \mathbf{where}$
$\quad \mathit{return}\ x = \mathit{Enum1}\ (\lambda \mathit{it} \to \mathit{return}\ (\mathit{it}, x))$
$\quad \mathit{Enum1}\ x \ggg\!= f = \mathit{Enum1}\ (\lambda \mathit{it} \to \mathbf{do}$
$\quad\quad x\_\mathit{it} \leftarrow x\ \mathit{it}$
$\quad\quad \mathbf{case}\ x\_\mathit{it}\ \mathbf{of}$
$\quad\quad\quad (\mathit{it}', x') \to (\lambda(\mathit{Enum1}\ e) \to e)\ (f\ x')\ \mathit{it}')$
$\mathbf{instance}\ \mathit{MonadTrans}\ (\mathit{Enum1}\ \mathit{sym})\ \mathbf{where}$
$\quad \mathit{lift}\ x = \mathit{Enum1}\ (\lambda \mathit{it} \to \mathit{lift}\ x \ggg\!= \lambda r \to \mathit{return}\ (\mathit{it}, r))$

$\mathbf{class}\ \mathit{Enumerator}\ \mathit{enum}\ \mathbf{where}$
$\quad \mathit{feed} :: \mathit{Iterable}\ \mathit{it}\ \mathit{sym} \Rightarrow \mathit{enum}\ \mathit{sym}\ m\ a \to \mathit{it}\ t \to \mathit{it}\ (\mathit{it}\ t, a)$
$\quad \mathit{yieldStream} :: m\ (\mathit{Stream}\ \mathit{sym}) \to \mathit{enum}\ \mathit{sym}\ m\ ()$

$\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Enumerator}\ \mathit{Enum1}\ m\ \mathbf{where}$
$\quad \mathit{feed}\ (\mathit{Enum1}\ e)\ \mathit{it} = \mathbf{do}\ (\mathit{it}', x) \leftarrow e\ \mathit{it}; \mathit{return}\ (\mathit{it}', \mathit{Right}\ x)$
$\quad \mathit{yieldStream}\ \mathit{getSyms} = \mathit{Enum1}\ (\lambda \mathit{it} \to \mathbf{do}$
$\quad\quad \mathit{it} \leftarrow \mathit{step}\ \mathit{it}$
$\quad\quad \mathbf{case}\ \mathit{it}\ \mathbf{of}$
$\quad\quad\quad \mathit{Right}\ x \to \mathit{return}\ (\mathit{return}\ x, ())$
$\quad\quad\quad \mathit{Left}\ k \to \mathbf{do}$
$\quad\quad\quad\quad \mathit{syms} \leftarrow \mathit{lift}\ \mathit{getSyms}$
$\quad\quad\quad\quad \mathit{return}\ (k\ \mathit{syms}, ()))$

$\mathit{yield} :: \mathit{Enumerator}\ \mathit{enum}\ m \Rightarrow [\mathit{sym}] \to \mathit{enum}\ \mathit{sym}\ m\ ()$
$\mathit{yield}\ \mathit{cs} = \mathit{yieldStream}\ (\mathit{return}\ (\mathit{Chunks}\ \mathit{cs}))$
$\mathit{yieldEOF} :: \mathit{Enumerator}\ \mathit{enum}\ m \Rightarrow \mathit{enum}\ \mathit{sym}\ m\ ()$
$\mathit{yieldEOF} = \mathit{yieldStream}\ (\mathit{return}\ \mathit{EOF})$

The fact that we have to inspect and react to whether the iteratee did anything with our input suggests to me that we might prefer to do something smarter with an iteratee that isn't hungry: preferably, short-circuit it with something like ErrorT or MaybeT so that once an iteratee is satisfied the whole enumerator can terminate immediately. Either way, we also want to provide a way to react to the iteratee being done so that we can cleanup any open handles, etc.

I probably shouldn't try to shoehorn these into having the same type for *yieldStream*. In fact, I really don't think that's the right interface at all. Oh well. The real point is the enumerator types - I think they're a meaningful step in the right direction.

```
data EnumError sym e
  = IterateeFinished (Stream sym)
  | EnumError e
instance Error e ⇒ Error (EnumError sym e) where
  noMsg = EnumError noMsg
  strMsg = EnumError ∘ strMsg

newtype Enum2 e sym m a = Enum2 (ErrorT (EnumError sym e) (Enum1 sym m) a)
  deriving (Functor, Monad)
instance Error e ⇒ MonadTrans (Enum2 e sym) where
  lift = Enum2 ∘ lift ∘ lift
instance (Monad m, Error e) ⇒ MonadError e (Enum2 e sym m) where
  throwError e = Enum2 (throwError (EnumError e))
  catchError (Enum2 x) h = Enum2 (catchError x h′)
    where
      h′ (EnumError e) = (λ(Enum2 y) → y) (h e)
      h′ other = throwError other
```

This *EnumeratorError* class is poorly named, and also its operations arguably should be in the base *Enumerator* class.

```
class Enumerator enum ⇒ EnumeratorError enum where
  catchIterateeFinished :: enum sym m a → (Stream sym → enum sym m a) → enum sym m a
  finally ::
    enum sym m a
    → enum sym m b
    → enum sym m a

instance (MonadError e m, Error e) ⇒ EnumeratorError (Enum2 e) m where
  catchIterateeFinished (Enum2 x) h = Enum2 (catchError x h′)
    where
      h′ (IterateeFinished str) = (λ(Enum2 y) → y) (h str)
      h′ other = throwError other

  Enum2 x ‘finally‘ Enum2 y = Enum2
    ((x ≫= λr → y ≫ return r) ‘catchError‘ h)
    where h err = y ≫ throwError err
```

Using *finally*, we can implement a nice *bracket* function that opens a resource, runs all the code that needs it, and guarantees that it'll be safely closed (at least, insofar as it is possible to do so).

```
bracket open use close = lift open ≫= λrsrc → (use rsrc ‘finally‘ lift (close rsrc))

instance (Error e, MonadError e m) ⇒ Enumerator (Enum2 e) m where
  feed (Enum2 e) iter = do
```

```
    mbE ← feed (runErrorT e) iter
    case mbE of
      (it, Left s)                          → return (it, Left s)
      (it, Right (Left (IterateeFinished s))) → return (it, Left s)
      (it, Right (Left (EnumError e)))      → lift (throwError e)
      (it, Right (Right x))                 → return (it, Right x)
  yieldStream getSyms = Enum2 (ErrorT (Enum1 (λit → do
    it ← step it
    case it of
      Right x → return (return x, Left (IterateeFinished (Chunks [])))
      Left k → do
        syms ← lift getSyms
        return (k syms, Right ())))))
```

```
enumFile path = bracket
  (logIO "open" (IO.openFile path IO.ReadMode >>= λh → IO.hSetBuffering h (IO.BlockBuffering (Just 25
  (enumHandle 16)
  (logIO "close" ∘ IO.hClose)
```

```
enumHandle bufSiz h = do
  isEOF ← lift (IO.hIsEOF h)
  if isEOF then return ()
    else do
      buf ← lift (logIO "hGet" (BS.hGet h bufSiz))
      yield (BS.unpack buf)
      enumHandle bufSiz h
```

```
logIO msg act = putStr msg >> act
```

Finally: as you may have guessed by now (based on my choice of primitives or on Oleg's choice of names) an Enumerator really has nothing at all to do with iteratees except that an iteratee consumes one and through a peculiar inversion of control, the enumerator is given primary control of execution, pretty much for the sole purpose of allowing it to detect when the iteratee stops reading from it. Aside from that twist, an enumerator is just like a Pythonic "generator" or a Ruby method with a "block" parameter. So let's make an enumerator type that reflects that notion. I won't bother making instances, just a function to tranlate this enumerator to any of the others.

```
data Yield sym m t where
  Yield :: m (Stream sym) → Yield sym m ()
newtype Enum3 sym m a = Enum3 (PromptT (Yield sym m) m a)
  deriving (Functor, Monad)
```

```
runEnum3 :: (Monad m1, Monad m2) ⇒ (m1 (Stream sym) → m2 ()) → (∀x.m1 x → m2 x) → Enum3 sym
runEnum3 y l (Enum3 e) = runPromptT return (bindP y) (λx k → l x >>= k) e
  where
    bindP :: (Monad m1, Monad m2) ⇒ (m1 (Stream sym) → m2 ()) → Yield sym m1 t → (t → m2 r) →
    bindP y (Yield s) k = y s >> k ()
```

```
enum3ToEnum e = runEnum3 yieldStream lift e
```

$$
\begin{array}{l}
\mathit{feedAndRun}\ \mathit{enum}\ \mathit{iter} = \textbf{do} \\
\quad (\mathit{iter}, \mathit{enumRes}) \leftarrow \mathit{feed}\ \mathit{enum}\ \mathit{iter} \\
\quad \mathit{iterRes} \leftarrow \mathit{iter} \\
\quad \mathit{return}\ (\mathit{enumRes}, \mathit{iterRes})
\end{array}
$$

Appendix A. Simple operations on Streams

$$
\begin{array}{l}
\mathit{streamToList}\ \mathit{EOF} = [\,] \\
\mathit{streamToList}\ (\mathit{Chunks}\ \mathit{cs}) = \mathit{cs}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{streamLength}\ \mathit{str} & = \mathit{length} & (\mathit{streamToList}\ \mathit{str}) \\
\mathit{isEmpty}\quad \mathit{str} & = \mathit{null} & (\mathit{streamToList}\ \mathit{str})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{isEOF}\ \mathit{EOF} & = \mathit{True} \\
\mathit{isEOF}\ \_ & = \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
\mathit{appendStream}\ \mathit{s1}\ \mathit{s2} \\
\quad \mid \mathit{isEOF}\ \mathit{s1} \wedge \mathit{isEOF}\ \mathit{s2} = \mathit{EOF} \\
\quad \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{Chunks}\ (\mathit{streamToList}\ \mathit{s1} +\!\!+\ \mathit{streamToList}\ \mathit{s2})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{splitStreamAt}\ n\ \mathit{EOF} & = (\mathit{EOF}, \quad \mathit{EOF} \quad) \\
\mathit{splitStreamAt}\ n\ (\mathit{Chunks}\ \mathit{cs}) & = (\mathit{Chunks}\ \mathit{xs}, \mathit{Chunks}\ \mathit{ys}) \\
\quad \textbf{where}\ (\mathit{xs}, \mathit{ys}) = \mathit{splitAt}\ n\ \mathit{cs} &
\end{array}
$$

$$
\begin{array}{l}
\mathit{takeStream}\ n = \mathit{fst}\ \circ \mathit{splitStreamAt}\ n \\
\mathit{dropStream}\ n = \mathit{snd} \circ \mathit{splitStreamAt}\ n
\end{array}
$$