Introduction to Iteratees

Motivation and implementation basics

Alexander Lehmann
<afwlehmann@googlemail.com>

Technische Universität München

December 17th, 2013

Brief history

2008	Oleg Kiselyov, “Incremental multi-level input processing with left-fold enumerator”, DEFUN 2008
2010-05-12	John W. Lato, “Teaching an Old Fool New Tricks’, Monad Reader #16
2010-05-26	Available in scalaz 5.0 Rúnar Óli
2012-03-13	Initial release of the play framework 2.0

We've often seen something along the lines of

```
import io.Source.fromFile

val it: Iterator[String] = fromFile("foo.txt").getLines

var result = 0
while (it.hasNext) {
  val line = it.next
  result += line.toInt
}

// Do something with the result
```

Issues

- ✗ Repetitive pattern (DRY principle)
- ✗ Manual pulling
- ✗ Mutability, imperative style
- ✗ No error handling (we sometimes just forget, right?)
- ✗ No (one|two)-way communication (Exceptions don't count)
- ✗ Resource handling (how long do we need a resource and who is responsible for opening/closing/recovering it?)
- ✗ Missing or rather difficult composability
- ✗ What if the input stream were infinite?

Try to be more “functional” and invert control...

```
val it: Iterator[String] = fromFile("foo.txt").getLines
var result = 0
it foreach { line =>
  result += line.toInt
}
```

define ourselves a re-usable utility function...

```
def enum(it: Iterator[String]): Int = {
  var result = 0
  it foreach { line => result += line.toInt }
  result
}

val foo = enum(fromFile("foo.txt").getLines)
```

being more versatile for the greater good...

```
def enum(it: Iterator[String])(init: Int)
        (f: (Int, String) => Int): Int = {
  var result = init
  it foreach { line => result = f(result, line) }
  result
}

val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Int = enum(it)(0)(_ + _.toInt)
```

possibly even generic...

```
def enum[In,Out](it: Iterator[In])(init: Out)
                (f: (Out, In) => Out): Out = {
  var result = init
  it foreach { x => result = f(result, x) }
  result
}

val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Int = enum(it)(0)(_ + _.toInt)
```

provide sophisticated error handling...

```
def enum[In,Out](it: Iterator[In])(init: Out)
                (f: (Out, In) => Out): Out = {
  var result = init
  try {
    it foreach { x => result = f(result, x) }
  } catch {
    case t: Throwable => /* TODO (fingers crossed) */
  }
  result
}

val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Int = enum(it)(0)(_ + _.toInt)
```

or rather use {your-favorite-"monad"-here} for error handling, asynchronism and composability...

```
def enum[In,Out](it: Iterator[In])(init: Out)
                (f: (Out, In) => Out): Option[Out] = {
  try {
    var result = init
    it foreach { x => result = f(result, x) }
    Some(result)
  } catch {
    case t: Throwable => None
  }
}

val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Option[Int] = enum(it)(0)(_ + _.toInt)
```

only to realize we've already had it all!?

```
val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Try[Int] = it.foldLeft(Try(0)) {
   // f: (Out, In) => Out
   case (acc, line) =>
    acc map { x => x + line.toInt }
}
```

only to realize we've already had it all!?

```
val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Try[Int] = it.foldLeft(Try(0)) {
   // f: (Out, In) => Out
   case (acc, line) =>
    acc map { x => x + line.toInt }
}
```

Really? What about

- producer and consumer talking back and forth
- cannot cope with infinite streams
- resource handling (stick this into another function?)
- asynchronism (could have used `Futures`)
- data which are not available all at once

only to realize we've already had it all!?

```
val it: Iterator[String] = fromFile("foo.txt").getLines
val foo: Try[Int] = it.foldLeft(Try(0)) {
    // f: (Out, In) => Out
    case (acc, line) =>
     acc map { x => x + line.toInt }
}
```

Think of

- `foldLeft` as *Enumerator*
- `acc` together with `f` as *Iteratee*

Enumerators . . .

- are stream producers
- push subsequent chunks of data to an Iteratee, hence folding the Iteratee over the input stream

```
sealed trait Enumerator[F] {
  def apply[T](iter: Iteratee[F,T]): Iteratee[F,T]
  def run[T](iter: Iteratee[F,T]): T
}
```

- act synchronously or asynchronously (think Futures)
- come with batteries included

```
object Enumerator {
  def apply[T](xs: T*): Enumerator[T]
  def fromTextFile(fileName: String): Enumerator[String]
  def fromStream(s: InputStream, chunkSize: Int = 8192)
                : Enumerator[Array[Byte]]
}
```

Enumerators communicate state to the Iteratee:

```
sealed trait Input[+T]

case class  Element[T](x: T) extends Input[T]
case object Empty            extends Input[Nothing]
case object EOF              extends Input[Nothing]
```

Enumerators communicate state to the Iteratee:

```
sealed trait Input[+T]

case class  Element[T](x: T) extends Input[T]
case object Empty            extends Input[Nothing]
case object EOF              extends Input[Nothing]
```

For now, assume enumerators like this:

```
def enum[F,T](xs: List[F])(it: Iteratee[F,T]): Iteratee[F,T]
```

Iteratees . . .

- stream consumers
- consume chunks of input (as a whole)
- are immutable, re-usable computations
- state is encoded through type

```
sealed trait Iteratee[F,T] { def run: T }

case class Done[F,T] (result: T, remainingInput: Input[F])
case class Cont[F,T] (k: Input[F] => Iteratee[F,T])
case class Error[F,T](t: Throwable)

// All of Done, Cont and Error extend Iteratee[F,T]
```

So far:

- Enumerators fold iteratees over a stream
- Enumerators communicate state through `Input[T]`
- Iteratees encapsulate result, error or computation

So far:

- Enumerators fold iteratees over a stream
- Enumerators communicate state through `Input[T]`
- Iteratees encapsulate result, error or computation

Let's revisit our "enumerator":

```
def enum[F,T](xs: List[F])(it: Iteratee[F,T]): Iteratee[F,T] =
  (xs, it) match {
    case (h::t, Cont(k)) => enum(t)( k(Element(h)) )
    case _               => it
  }
  // k: Input[In] => Iteratee[F,T]
```

This is all we need to have some working examples.

```
def head[T]: Iteratee[T,T] = {
  def step: Input[T] => Iteratee[T,T] = {
    case Element(x) => Done(x, Empty)
    case Empty      => Cont(step)
    case EOF        => Error(new NoSuchElementException)
  }
  Cont(step)
}
```

Example:

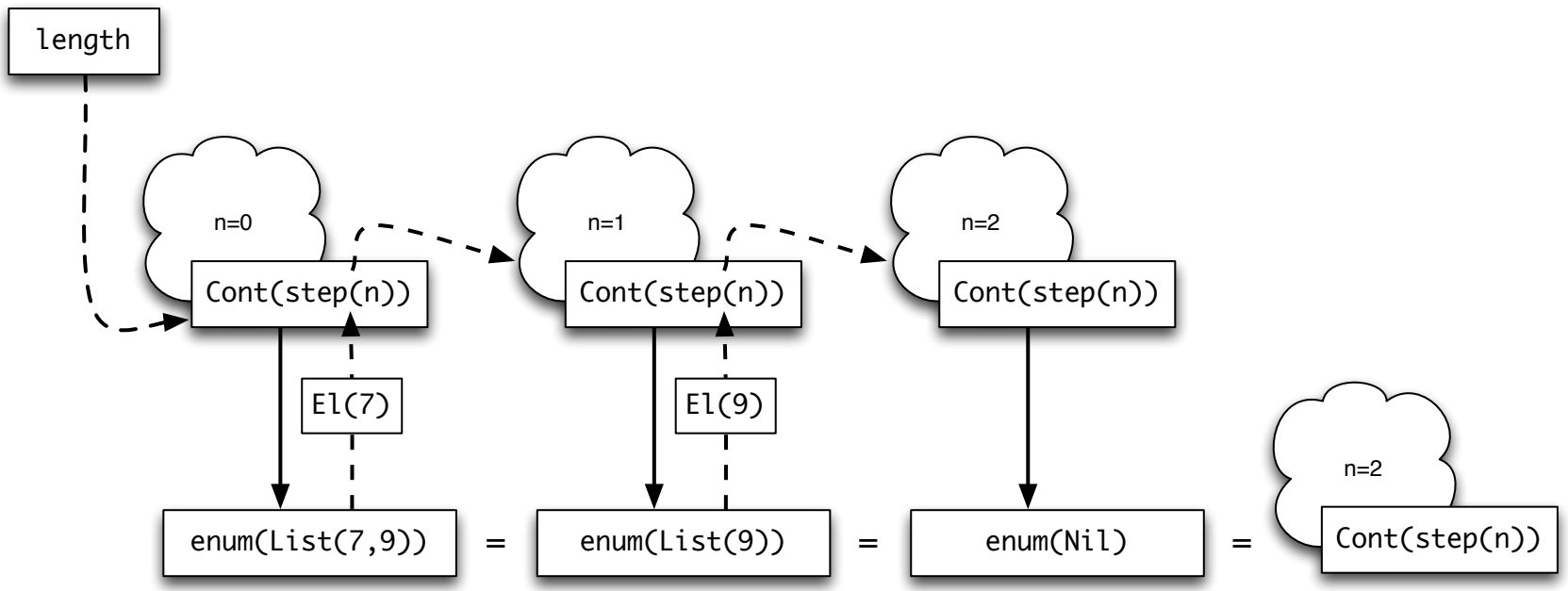
```
scala> enum("Hello world!".toList)(head)
res1: Iteratee[Char,Char] = Done(H,Empty)

scala> res1.run
res2: Char = H
```

```
def length[T]: Iteratee[T, Int] = {
  def step(n: Int): Input[T] => Iteratee[T, Int] = {
    case EOF         => Done(n, EOF)
    case Empty       => Cont(step(n))
    case Element(_) => Cont(step(n+1))
  }
  Cont(step(0))
}
```

Example:

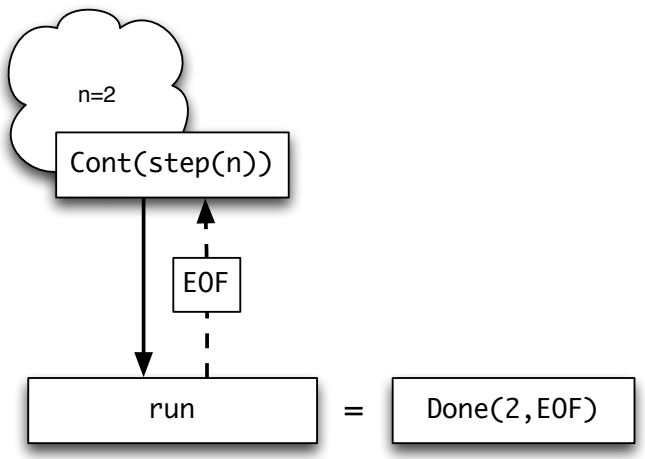
```
scala> enum("Hello world!".toList)(length)
res3: Iteratee[Char,Int] = Cont(<function1>)

scala> res3.run
res4: Int = 12
```

How this works



How this works

Further use that iteratee or apply `run` to it...



There's a lot of pattern matching going on. Frameworks to the rescue:

```
object Iteratee {
  def apply[F,T](el:    Element[F] => Iteratee[F,T],
                 empty: => Iteratee[F,T],
                 eof:   => Iteratee[F,T]): Iteratee[F,T]

  def fold[F,T](z: T)(f: (T,F) => T): Iteratee[F,T]
}
```

Because that's what we need most of the time.

Wouldn't it be awesome if Iteratees were composable?

```
def drop1Keep1[T] = for {
  _ <- drop[T](1)
  h <- head
} yield h
```

```
def pair[T] = for {
  h1 <- head[T]
  h2 <- head
} yield (h1, h2)
```

They are indeed! Iteratees compose sequentially (and so do Enumerators).

```
scala> enum("Hello World!".toList)(drop1Keep1).run
res1: Char = e

scala> enum("Hello World!".toList)(pair).run
res2: (Char, Char) = (H,e)
```

As we know, `for` translates to `map` and `flatMap`:

```
trait Iteratee[F,T] {
  def map[U](f: T => U): Iteratee[F,U]
  def flatMap[U](f: T => Iteratee[F,U]): Iteratee[F,U]
}
```

As we know, `for` translates to `map` and `flatMap`:

```
trait Iteratee[F,T] {
  def map[U](f: T => U): Iteratee[F,U]
  def flatMap[U](f: T => Iteratee[F,U]): Iteratee[F,U]
}
```

Adding this to `Cont` is straight-forward.

```
def map[U](f: T => U): Iteratee[F,U] =
  Cont(elt => k(elt) map f)

def flatMap[U](f: T => Iteratee[F,U]): Iteratee[F,U] =
  Cont(elt => k(elt) flatMap f)

// k: Input[F] => Iteratee[F,T]
```

So is adding to Done.

```
def map[U](f: T => U): Iteratee[F,U] =
  Done(f(x), remainingInput)

def flatMap[U](f: T => Iteratee[F,U]): Iteratee[F,U] =
  f(x) match {
    case Done(xPrime, _) => Done(xPrime, remainingInput)
    case Cont(k)         => k(remainingInput)
    case err @ Error(_)  => err
  }
```

Now why is that so great?

- Easily compose simple (or complex) iteratees

```
def drop1Keep1[T] = for {
  _ <- drop[T](1)
  h <- head
} yield h
```

```
def pair[T] = for {
  h1 <- head[T]
  h2 <- head
} yield (h1, h2)
```

- No repetitive pattern-matching hell
- Benefit from utility functions, e.g. `sequence` or `repeat`

Example:

```
scala> def fivePairs[T] = sequence(List.fill(5)(pair[T]))
...
scala> enum((1 to 10).toList)(fivePairs).run
res1: List[(Int, Int)] = List((1,2), (3,4), (5,6), (7,8), (9,10))
```

Now why is that so great?

- Easily compose simple (or complex) iteratees

```
def drop1Keep1[T] = for {
  _ <- drop[T](1)
  h <- head
} yield h
```

```
def pair[T] = for {
  h1 <- head[T]
  h2 <- head
} yield (h1, h2)
```

- No repetitive pattern-matching hell
- Benefit from utility functions, e.g. `sequence` or `repeat`

Example:

```
scala> def alternates[T] = repeat(drop1Keep1[T])
...
scala> enum((1 to 10).toList)(alternates).run
res2: Stream[Int] = Stream(2, ?)
// res2.toList == List(2, 4, 6, 8, 10)
```

Enumeratees...

- are both consumer and producer
- link in between Enumerator and Iteratee
- feed transformed input from Enumerator to "inner" Iteratee, for instance
  - `filter`
  - `take`
  - `...`
- are (of course?) composable
- allow vertical (parallel) stacking of Iteratees (one to many)

See eamelink's play-related examples @

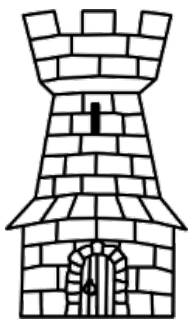
`https://gist.github.com/eamelink/5639642`

Brief discussion:

- ✓ Enumerators, Iteratees, Enumeratees
- ✓ Easy to use
- ✓ Composable in sequence and in parallel
- ✓ Tail recursion ⇒ no stack overflow
- ✓ Thin layer ⇒ high performance
- ✓ Good match for asynchronuous environments, e.g. the play framework
- ✗ Pure form lacks functional treatment of side effects

Where to go from here

- Read blogs & tutorials, start out with [1] to [6]
- Use or rather implement iteratees
- Check out the play framework
  - Asynchronous iteratees
  - Numerous factory methods, e.g. easily bridge between Rx's Observables and Enumerators [3]
  - All the other goodies from play
- Check out scalaz 7
  - Implementation in terms of monad transformers $\Rightarrow$ your choice of `IO`, `Future`, `Option`, ...
  - All what's missing in the standard library
  - See examples and explanations at [1] and [2]



Thank you for your attention

Anyone trying to understand monads will inevitably run into the [...] monad, and the results are almost always the same: bewilderment, confusion, anger, and ultimately Perl.

Daniel Spiewak, Dec. 2010

References:

[1] "Enumeration-based I/O With Iteratees" @ http://blog.higher-order.com/blog/2010/10/14/scalaz-tutorial-enumeration-based-io-with-iteratees/

[2] "learning Scalaz: Enumeration-Based I/O with Iteratees" @ http://eed3si9n.com/learning-scalaz/Iteratees.html

[3] "RxPlay: Diving into iteratees and observables and making them place nice" @ http://bryangilbert.com/code/2013/10/22/rxPlay-making-iteratees-and-observables-play-nice/

[4] "Understanding Play2 Iteratees for Normal Humans" @ http://mandubian.com/2012/08/27/understanding-play2-iteratees-for-normal-humans/

[5] "Incremental multi-level input processing and collection enumeration" @ http://okmij.org/ftp/Streams.html

[6] "Teaching an Old Fool New Tricks" @ http://themonadreader.wordpress.com/2010/05/12/issue-16/

Backup slides

`run` makes great effort to yield the final result:

```
def run: T = this match {
  case Done(rslt, _) => rslt

  case Error(t)      => throw t

  // k: Input[F] => Iteratee[F,T]
  case Cont(k)       => k(EOF).run   // may loop forever
}
```

`run` makes great effort to yield the final result:

```
def run: T = this match {
  case Done(rslt, _)    => rslt

  case Error(t)         => throw t

  // k: Input[F] => Iteratee[F,T]
  case Cont(k)          => k(EOF) match {
    case Done(rslt, _) => rslt
    case Cont(_)        => sys.error("Diverging iteratee!")
    case Error(t)       => throw t
  }
}
```

```
def drop[T](n: Int): Iteratee[T, Unit] = {
  def step: Input[T] => Iteratee[T, Unit] = {
    case EOF         => Done((), EOF)
    case Empty       => Cont(step)
    case Element(_) => drop(n-1)
  }
  if (n <= 0) Done((), Empty) else Cont(step)
}
```

Example:

```
scala> enum("Hello World!".toList)(drop(3))
res5: Iteratee[Char,Unit] = Done((),Empty)

scala> res5.run
// blank
```

How this works