Iteratees

Functional Sequence Processors

About Me

- Bryan Murphy
- Software Developer 20+ years
- Scala enthusiast since 2010
- Scala Developer at Playup (1 week)
  - living the dream
- Functionally challenged
- Limited public speaking

Questions & Discussion Welcome

- flexible format
- dialog is good
- questions/corrections/assistance from all experience levels welcome
- use the collective wisdom
- minor diversions tolerated
- whiteboard is your friend

Outline

- Why Iteratees
- Brief Description
- Functional Data Structures
- Problems suited to Iteratees
- Detailed Description (and code)
- Futures
- Reactive Intro

Outline (2)

- Combining Futures and Iteratees
  - problems with naive solution
  - Play Framework solution
  - pattern matching fail
  - fold() and Step
  - long running computations
  - flatten()

Outline (3)

- scalaz
  - taking the red pill …
- Misc
  - Enumeratees
  - Interleaving Enumerators

Why Iteratees

- Investigating functional alternatives to run loop for handling data from socket
- Read blogs and looked at implementations
- Play Iteratees seemed best supported but aspects of interface were a bit baffling

Why Iteratees (2)

```
val f_it: Future[Iteratee[Byte, Int]] = ???
val it: Iteratee[Byte, Int] = Iteratee.flatten(f_it)

sealed trait Step[E, A]
case class Cont[E, A](k: Input[E] => Iteratee[E,A]) extends Step[E, A]

trait Iteratee[E, A] {
      fold[B](folder: Step[E, A] => Future[B]): Future[B]
}
```

Iteratee: Brief Description

- A functional and immutable data type that is suited to processing sequences or streams of data

- The Play Framework Iteratee implementation provides an Iteratee that allows for non blocking or “reactive” modes of operation which is well suited to networking communications and other processing that would involve delays

Functional Data Structures

- immutable - any “change” involves creation of a new object with the changes leaving the original the same
  - eg String, Option, List or Future
- composable
  - enables functions and other instances of the same data type to be combined to produce a new instance of the data type
- map() and flatMap() are often used which allows the data type to be used in for() comprehensions

Functional Data Structures (2)

- sometimes the creation of the composed data type doesn’t do anything at that time but encapsulates the composed functionality for later use eg Future, Stream, Reader and State “monads”
- deterministic ?
  - not always
    - eg Future.firstCompletedOf()

Problems Suited to Iteratees

- Seq has
  - map() - new seq same size as orig
  - fold() - single result after processing all elements
- Awkward to:
  - terminate before end
  - handle subsets of sequence
  - handle continuous data stream with time delays
  - save position - Iterator is mutable
  - eg: byte encoded messages on a socket

Detailed Description

- encapsulates the current state of processing a stream of data
- usually created by an Iteratee that processed the previous item
- usually include some accumulation of the data processed so far
  - and a function that defines what to do next with the next item
  - which is usually to produce a new Iteratee with new accumulated state
  - similar to function in foldLeft
  - foldLeft[B](acc: A)(f: (A, E) => A): A
  - called “Cont”
- Iteratee produced may be in a “Done” state and produce the final result
- Iteratee produced may be in a “Error” state and produce error message
- Input data can be EOF to signal no more data
- Done or Error is expected in response to EOF

Detailed Description (2)

- Iteratee is apply'd to an Enumerator until a final Iteratee is produced
  - Iteratee is Done
  - or Enumeration is finished
  - EOF signal is optional to enable concatenation of enumerators
- Iteratees can be composed such that when one is Done the next Iteratee will get the next item in the sequence

Show me the code already !

Futures

- Four (at least) ways to create Futures
  - immediately with result (no thread)
    - eg successful { 5 }
  - one thread per future
    - eg future { longCalculation() }
  - one thread multiple future
    - val f1 = future { longCalculation() }
    - val f2 = f1.map { _ * anotherLongCalculation() }
    - val f3 = f2.map { _.toString }
  - no thread result later
    - val p = new Promise[Int]
    - val f = p.future

Futures continued

- Promise can be used to create a Future that will be instantiated later
  - val p = Promise[Int]
  - val f = p.future
  - p.success(5)
- Useful for:
  - Fixed thread pool
  - Request queue
  - Creating “reactive” service interface
- for() comprehension leverages flatMap() to create a “sequential” interface

```
val f3 = for ( v1 <- f1
               v2 <- f2
             ) yield v1 + v2
```

Reactive

- There is now a manifesto
- Avoid blocking
- Setup callbacks
- Minimise threads
- Facilitated by:
  - Future and Promise
    - functional interface
  - Actor
    - message passing interface
    - can return Future as well

Futures and Iteratees

- To be non blocking we need to add Future
  - Cont(k: Input[E] => Future[Iteratee[E, A])
  - map(f: A => Future[B]): Future[Iteratee[E, B]
  - flatMap(f: A => Future[Iteratee[E, B]): Future[Iteratee[E, B]
- But this interface is not “monadic” (flatMap interface is wrong)
  - no use of for() syntax :-(
- Composition is clumsy:

```
def getStringIt: Iteratee[Byte, String] = ???
def getIntIt: Iteratee[Byte, Int] = ???
def getUser: Future[Iteratee[Byte, User]] =
    getStringIt.flatMap(s => getIntIt.map(i => successful(User(s, i))))
```

Futures and Iteratees (2)

- Alternative is to combine the Future into the Iteratee which creates two levels:
  - Outer level Iteratee which has a Future “aspect” and can be convert to/from Future[Iteratee]
    - supports map() and flatMap()
    - not sealed just define abstract fold() method
  - Inner level Iteratee state which has no future aspect
    - usually called Step
    - sealed trait

Play Framework Iteratee

- The Play Framework Iteratee uses the second approach

fold() and Step

- Pattern matching callback

```
sealed trait Step[E, A]
case class Done[E, A](a: A, rem: Input[E]) extends Step[E, A]
case class Cont[E, A](k: Input[E] => Iteratee[E, A]) extends Step[E, A]

trait Iteratee[E, A] {
    def fold(folder: Step[E, A] => Future[B]): Future[B]
}
```

Pattern matching fail

- Iteratee may not be complete so can’t pattern match on it
- Step can be pattern matched but is hidden inside Iteratee
- Need an operation like Future.map to act on the Step when it is available
- Register a callback

```
// Instead of:
val inp: Input[E]
it match {
    case Cont(k) => k(inp)
    case _ => it
}
// Do this
it fold {
    case Cont(k) => future { k(inp) }
    case _ => successful { it }
} // returns Future[Iteratee[E, A]]
```

Long running computations howto

```
// Cont takes Input[E] => Iteratee[E, A]
def myIt: Iteratee[Int, String] = Cont {
    case El(e) =>
        val fs: Future[String] = getFromDb(e)
        val fi: Future[Iteratee[E, A]] = fs.map(s => Done(s))
        Iteratee.flatten(fi)
    case EOF => Done(“”)
}
```

flatten()

```
Iteratee.flatten[E, A](f: Future[Iteratee[E, A]) : Iteratee[E, A] =
    new Iteratee[E, A] {
        def fold[B](folder: Step[E, A] => Future[B]): Future[B] =
            f.flatMap(it => it.fold(folder))
    }
```

Creating Iteratees

- Lots of helper methods so don't usually use step()
- Done(), Cont(), Error() to create Iteratees in those "states"
- Iteratee.fold[E, A](state: A)(f: (A, E) => A): Iteratee[E, A]
- Iteratee.foldM[E, A](state: A)(f: (A, E) => Future[A]) : Iteratee[E, A]
- Iteratee.foreach[E](f:E => Unit): Itereate[E, Unit)
- Iteratee.consume[E]: Iteratee[E, E]
  - concatenates so E should be traversable
- Iteratee.getChunks[E]: Iteratee[E, List[E]]
- Iteratee.head[E]: Iteratee[E, Option[E]]
- Iteratee.foldM[E, A](state: A)(f: (A, E) => Future[A]) : Iteratee[E, A]

scalaz

- Haskell for Scala
- need to take the red pill
  - full on functional
- scalaz 7 has Iteratees
- similar to Play but different
- Uses “monad transformer” pattern to combine Iteratee and Future (or any other monad)

Misc

- Play Framework has more:
  - Enumeratees
    - transformers From => To
    - operate on both Enumerators and Iteratees
    - provides filter, take, drop, “group”, …
  - Interleaving Enumerators
    - Non deterministically join streams of data

References

- Play Doco: http://www.playframework.com/documentation/2.3-SNAPSHOT/Iteratees
- James Roper's Blog: http://jazzy.id.au/default/2012/11/06/iteratees_for_imperative_programmers.html
- Runar's Blog: http://apocalisp.wordpress.com/2010/10/17/scalaz-tutorial-enumeration-based-io-with-iteratees/
- Mandubian: http://mandubian.com/2012/08/27/understanding-play2-iteratees-for-normal-humans/