Free Monads in Practice

Haroon Khan
HealthExpense, Inc
February 8, 2016

Introduction

- My journey towards functional enlightenment
- Towards “Compositional Architectures”
  - Readers
  - State
  - Writer
  - Reader Writer State
  - Applicatives
  - Monad Transformers
  - IO monads
  - Scalaz, Scala,

How to tell if a type is Monadic?

- See how it behaves from going to Functor to a Monad
  - Functors transform values
  - Monad let the structure of embedded computations depend on the values of previous computations
- Monadic effects?
  - State, Maybe, Reader, Writer,
  - Continuations: non-local control flow

What is a Free Monad?

- A generalization of the IO Monad

```
trait Free[F[_],A]

case class Return[F[_],A](a: A) extends
Free[F,A]

case class Suspend[F[_],A](f: F[Free[F,
A]]) extends Free[F,A]
```

- For any kind of F
  - Has type parameter with a constructor

Getting Started on using Free Monads

- Scalaz 7.2.0
- Specs2 3.7
- Scala 7.11.6
- “Reasonably Priced Monads”
- “Functional Programming in Scala”

Goals of Project

- Design two “languages” using Free Monads
- Show code for various features
  - Abstraction
  - Composition
  - Testability
  - …
- Tie things togehter

Language A: Simple Stack Language

- Two Operations: Push and Pop

```
sealed trait ForthOp[A]

object ForthOp {

  final case class Push[A](n: Int, next:
Throwable \/ Unit => A) extends ForthOp[A]

  final case class Pop[A](next: Throwable
\/ Int => A) extends ForthOp[A]

}
```

Smart Constructors

```
def push[F[_]](n: Int)(implicit I:
ForthOp -~> F) = inject[F, ForthOp,
Throwable \/ Unit](Push(n,
Free.point(_)))


def pop[F[_]](implicit I: ForthOp -~>
F) = inject[F, ForthOp, Throwable \/
Int](Pop(Free.point(_)))
```

Build Rich based on fundamental blocks

```
def div[F[_]](implicit I: ForthOp -~> F) : Free[F, Throwable \/ Unit ] = {
  def divRec[F[_]](implicit I: ForthOp -~> F): Free[F, Throwable \/ Unit] = for {
    quotient <- pop[F]
    divisor <- pop[F] map { dv =>
      dv flatMap (x => if (x==0) -\/(new IllegalArgumentException("Divide by zero")) else \/-(x))
    }
    remainder <- pop[F]
    qdr <- Free.point[F, Throwable \/ (Int, Int, Int)]( (quotient |@| divisor |@| remainder) { (_, _, _) } )
    res <- qdr match {
      case \/-((q,d,r)) =>
        if (r < d) {
          for {
            _ <- push[F](r);
            res <- push[F](q)
          }
          yield res
        } else for {
          _ <- push[F](r - d);
          _ <- push[F](d);
          _ <- push[F](q+1);
          res <- divRec[F]
        } yield res
      case -\/(th) => Free.point[F, Throwable \/ Unit](th.left[Unit])
    }
  } yield res

  for {
      res <- push[F](0)
      res <- divRec[F]
  } yield res
}
```

Language B: Key Value Store Language (CRUD)

```
sealed trait KVOp[A]



object KVOp {

  case class Create[A](key:String, value:Int, next: (Throwable \/ Unit
=> A)) extends KVOp[A]

  case class Read[A](key:String, next:(Throwable \/ Int => A) ) extends
KVOp[A]

  case class Update[A](key:String, value:Int, next : (Throwable \/ Unit
=> A)) extends KVOp[A]

  case class Delete[A](key:String, next: (Throwable \/ Unit => A))
extends KVOp[A]

}
```

Smart Constructors for CRUD

```
def create[F[_]](key:String, value:Int)(implicit I:KVOp -~> F) =
inject[F, KVOp, Throwable \/ Unit](Create(key, value, Free.point(_)))
```

```
  def read[F[_]](key:String)(implicit I:KVOp -~> F) = inject[F, KVOp,
Throwable \/ Int](Read(key, Free.point(_)))
```

```
  def update[F[_]](key:String, value:Int)(implicit I:KVOp -~> F) =
inject[F, KVOp, Throwable \/ Unit](Update(key, value, Free.point(_)))
```

```
  def delete[F[_]](key:String)(implicit I:KVOp -~> F) = inject[F, KVOp,
Throwable \/ Unit](Delete(key, Free.point(_)))
```

Build a CRUD program

```
for {
        _ <- create("key1", 1)
        _ <- create("key2", 2)
        x <- read("key1")
} yield x
```

Interpreting “Programs”

```
object forthOpInterpreter extends (ForthOp ~> Id)
{
  val stack = new mutable.Stack[Int]()
  override def apply[A](fa: ForthOp[A]): Id[A] =
fa match {
    case Push(n, next) =>
      stack.push(n)
      next(\/-(()))
    case Pop(next) => try{
      next(\/-(stack.pop()))
    } catch {
      case (e:java.util.NoSuchElementException)
=> next(-\/(e))
    }
  }
}
```

Natural Transforms: ~>

- Transforms any F Type to a Monad e.g. Id
- Can be any Monad
  - State, Reader, Writer, etc.
- This is an important for composition

Impure vs Pure Interpreters

```
package object forthInterpreters {
  type NTState[A] = State[List[Int], A]
}


object forthOpStateInterpreter extends (ForthOp ~> NTState) {
  override def apply[A](fa: ForthOp[A]): NTState[A] = fa match
{
    case Push(n, next) => State {
      state => ((n :: state), next(\/-()))
    }
    case Pop(next) => State {
      case top :: rest => (rest, next(\/-(top)))
      case err @ Nil   => (err, next(-\/(new
NoSuchElementException())))
    }
  }
}
```

Another Pure Interpreter

```
package object kvInterpreters {
  type NTState[A] = State[Map[String, Int], A]
}

object kvStateInterpreter extends (KVOp ~> NTState)  {
  override def apply[A](fa: KVOp[A]): NTState[A] = fa match {
    case Create(key, value, next) => State {
      state => (state + (key -> value), next(\/-(())))
    }

    case Read(key, next) => State {
      state => (state, next(state.get(key).toRightDisjunction(new
Throwable("Key Undefined!"))))
    }
    case Update(key, value, next) => State {
      state => (state + (key -> value), next(().right[Throwable]))
    }

    case Delete(key, next) => State {
      state => (state - key, next(().right[Throwable]))
    }
  }
}
```

“Composing” Free Monads

- Monads don’t compose
- But we can use Monad Transformers!!
- Monad Transformers are Monads
- Natural Transforms can be used to layer Monads

Set up Boiler plate

- Think of how Monadic Layers will be

```
 type KVState[OM[_], A] = StateT[OM,
Map[String, Int], A]


  type ForthOpKVState[A] =
KVState[forthInterpreters.NTState, A]
```

Intermediate Natural Transforms

```
object kvStateToForthKVState extends (kvInterpreters.NTState
~> ForthOpKVState) {

        override def apply[A](innerMonad:
kvInterpreters.NTState[A]): ForthOpKVState[A] = {
    StateT[forthInterpreters.NTState, Map[String, Int], A] {
      (state) =>
innerMonad.run(state).point[forthInterpreters.NTState]
    }
  }
}
object forthStateToForthKVState extends
(forthInterpreters.NTState ~> ForthOpKVState) {

        override def apply[A](outerMonad:
forthInterpreters.NTState[A]): ForthOpKVState[A] =
outerMonad.liftM[KVState]
}
```

“Composing” Algebras

- Each Algebra is an AST that is evaluated
- Needs to be a Coproduct

```
object algebra {
  type C0[A] = Coproduct[ForthOp, KVOp, A]
  type AppAlgebra[A] = C0[A]
  object All extends ForthOpInstances with
KVOpInstances
}
```

“Composing” Interpreters

```
val interpreter = (forthOpStateInterpreter andThen
forthStateToForthKVState) or (kvStateInterpreter
andThen kvStateToForthKVState)
```

```
val prg1 = programs.createPrg[algebra.C0]("update1",
2, 1).foldMap(interpreter).run(Map()).run(List())
```

Sample Program

```
 def incrementPrg[F[_]](key:String, n: Int )(implicit
FO:ForthOp -~> F, KVO: KVOp -~> F) = for {
    valueA <- read[F](key)
    valueB = n.right[Throwable]
    // wrap this in a free monad, to get the types inline
    abv <- Free.point[F, Throwable \/ (Int, Int)]( (valueA |@|
valueB ) { (_, _) } )
    cv <- abv.fold( _.left[Int] |> (Free.point[F, Throwable \/
Int]) , {
      case (a, b) => for {
        _ <- push[F](a)
        _ <- push[F](b)
        _ <- add[F]
        r <- pop[F]
      } yield r
    })
    rr <- cv.fold(_.left[Unit] |> (Free.point[F, Throwable \/
Unit]), update[F](key, _))
  } yield rr
```

Conclusions

- Look closely….
  - And you will see a continuation monad
- Can model all sorts of fancy things
  - Exceptions
  - Options
  - I/O
- Topics for further study
  - Free Applicatives
  - Free Monad Transformers

References

- Chiusano, P and Bjarnason, R. *Functional Programming in Scala.* 2014
- Bjarnsaon, R. *Compositional Application Architecture with Reasonably Priced Monads.* 2014
- Scalaz Library
- Cats Library
  - *Cats has a great explanation for Free Monads written in a tutorial style covering the same topic but leveraging the cats library*