Working With Mutable Objects

Chris Addie

Datacom TSS

June 21, 2017

Table of Contents

Table of Contents

Abstract and Audience

Building on my earlier talks on monadic programming in Scala I'll be demonstrating how to work with *mutable objects* using *pure functions*

Audience:

- This presentation intended for imperative and functional programmers interested in safely dealing with mutation
- Code samples will be in Scala

Table of Contents

Stateful Functional Programs

DATACOM TSS

- Functional programs are those composed of *pure functions*
- By definition such programs are stateless
- Programs may produces different results for the same input by mutating the computer running them
- This mutation via can be reasoned about in a *purely functional* manner through *monadic* programming
- `System.IO` and `scalaz.IO` offer such capabilities in Haskell and Scala respectively

Stateful Applications without IO

- Sometimes you may want to develop a long running application that is stateful, but not persistent
- It may also be desirable for application state to lead persistent state for performance reasons
- In such cases you may need a mutable memory reference that is overwritten when the application moves to a new state
- Haskell and Scala offer `Control.Monad.ST` and `scalaz.effect.ST` for computation with mutable context

Stateful Objects

- Scala allows mutable references to exist outside of the ST Monad
- In Java all references are mutable by default
- This means a method could invisibly mutate application state as a side effect:

```
class EvilObject {
  private var s: Int = 0
  def setS(i: Int): Unit = { s = i }
  def getS: Int = s
  override def toString = s"EvilObject($s)"
}
```

Stateful Objects Continued

- Working with such objects is extremely tiresome for a number of reasons:
  - They can't be shared between threads
  - Incremental state is lost because backreferences are mutated
  - They can't be used in the *point free* style because mutation returns nothing
- Unfortunately such objects are extremely common in Java libraries... :(

Solution 1: Copy on Write

Let's introduce a typeclass for things that can be copied and a pure update function

```
trait Replicant[A] {
  def clone(a: A): A
}

implicit class UpdateOps[A: Replicant](a: A) {
  def update(f: A => Unit) = {
    val b = implicitly[Replicant[A]].clone(a)
    f(b); b
  }
}
```

Copy on Write Continued

Now if we implement the Replicant typeclass we can immutably update EvilObject

```
implicit object EvilReplicator extends Replicant[EvilObject] {
  def clone(a: EvilObject) = {
    val b = new EvilObject
    b.setS(a.getS); b
  }
}

scala> val a = new EvilObject
a: EvilObject = EvilObject(0)

scala> a.update(_.setS(3))
res2: EvilObject = EvilObject(3)

scala> a
res3: EvilObject = EvilObject(0)
```

Copy on Write Conclusions

- Any `type T` with a copy constructor can trivially implement `Replicant[T]` :)
- Incremental state is preserved :)
- *point free* style is back! :)
- Objects can sort of be shared if you explicitly share a clone :|
- Syntax is a bit messy :|
- Abstraction introduces large performance impact :(
- Not all types can be Replicants :(

Solution 2: Deferring Mutation

Rather than performing operations with side effects let's describe the effects we intend to have.

```
type Effect[T] = T => Unit
```

We want our effects to compose, and to be able to run them when we're done composing

```
type EffectMonoid[T] = Vector[Effect[T]]
implicit class UpdateOps[T](em: EffectMonoid[T]) {
  def update(e: Effect[T]): EffectMonoid[T] = em :+ e
}

//Not a pure function, actually mutates the world
def unsafeRunEffects[T](t: T)(em: EffectMonoid[T]): Unit =
  em.foreach{ _(t) }
```

Deferring Mutation Continued

Now we can code against the EvilObject API without any side effects

```
type EmEvilObject = EffectMonid[EvilObject]
object EmEvilObject {
  def empty = Vector.empty[Effect[EvilObject]]
}

val emEvilObject = EmEvilObject.empty
  .update{ _.setS(3) }
  .update{ _.setS(5) }
```

Deferred Mutation in Action

If we introduce an implicit to reify our type we can treat the Effect Monoid as that type directly

```
implicit def reifyEmEo(emeo: EmEvilObject): EvilObject = {
  val eo = new EvilObject
  unsafeRunEffects(eo)(emeo); eo
}

def useEvilObject(eo: EvilObject) = println(eo)

scala> useEvilObject(emEvilObject)
EvilObject(5)
```

Defferred Mutaion Conclusions

- Incremental state is preserved :)
- Can use *Point free* style :)
- EffectMonoids can be shared :)
- Works for all types :)
- Syntax is still bit messy :|
- Abstraction introduces small performance impact :|

Table of Contents

Computation With Effects

In our trivial example we've been dealing with functions that produce nothing and only have effects

Real world code tends to look more like this:

```
def realWorldFunction[A,B](a: A): B = {
  //do some computation
  a.evilSideEffectingFunction()
  //somehow produce B
}
```

Computation With Effects

We can refactor such functions to produce effects instead of enacting them

```
def realWorldFunction[A,B,C](a: A): (B, EffectMonid[C]) = {
  //do some computation
  val effects = CMonoid.Empty.update {
    _.evilSideEffectingFunction
  }
  //somehow produce B
}
```

But this ends up being wordy and akward.
What we really want is some sort of computation context which deals with accumulating Effects for us....

Writer to the Rescue!

Let's refactor our helper to use Writer

Also let's work with a more complicated, real-world example: `com.espertech.esper.client.Configuration`

```
def writeEffect[T](effect: Effect[T]): Writer[EffectMonid[T], _] =
  WriterT.tell( Vector(effect) )

type EsConfig = EffectMonid[Configuration]
implicit def toConfig(esc: EsConfig): Configuration = {
  val c = new Configuration
  unsafeRunEffects(c)(esc); c
}

//Pure function for adding type to EsConfig
private def addType[T: ClassTag]: Writer[EsConfig, _] = {
  val typeDef = new ConfigurationEventTypeLegacy()
  typeDef.setAccessorStyle(ConfigurationEventTypeLegacy.AccessorStyle.PUBLIC)
    writeEffect[Configuration] {
    _.addEventType(simpleClassName[T], className[T], typeDef)
  }
}
```

Computation with Writer

We want to compute some value while also Composing the effects of doing so invisibly

```
type EsMapping = Map[String, Object]
object EsMapping {
  def empty: EsMapping = Map.empty
}

private def addArrayMapping[T: ClassTag](em: EsMapping) = for {
  _ <- addType[T]
} yield ( em + (simpleClassName[T] -> classTag[Array[T]].runtimeClass) )

implicit class ConfMapOps(w: Writer[EsConfig, EsMapping]) {
  def withMapping[T: ClassTag] = w.flatMap { addArrayMapping[T] }
}
```

Computation with Writer Continued

Now we can use our Writer combinators in a for comprehension

```
private val addTypesMonad: Writer[EsConfig, EsMapping] = for {
  _ <- writeEffect[Configuration]{ _.addImport("dtss.soc.common.security.*") }
  _ <- addType[CertificateSubject]
  _ <- addType[ZonedDateTime]
  _ <- addType[DomainName]
  //...
  m <- addArrayMapping[CertificateSubjectField](EsMapping.empty)
    .withMapping[DateTimeField]
    .withMapping[DomainField]
    .withMapping[EmailAddressField]
  //...
  _ <- writeEffect[Configuration] { _.addEventType("SecurityEvent", m) }
} yield(m)

protected val config: Configuration = addTypesMonad.run._1
```

`Writer[EffectMonoid[A],B]` Conclusions

DATACOM TSS

- Inside a Writer Monad EsConfig behaves just like Configuration
- You can pass around EsConfig as though it were an Immutable Confiration object
- You get incremental state, and backreferences are preserved
- This is a generic solution that works for any mutable object
- Could still be improved by using type-level programming to pass a list of types, instead of caling addType for each one

Questions?