Stateful Computions on Heterogeneous Lists

Chris Addie

Datacom TSS

October 14, 2016

Table of Contents

Table of Contents

Abstract and Audiance

Building on my last presentation this talk will be exploring another use for the State Monad I intend to cover:

- State Monad refresher
- Partial functions and collect
- Combining State, Option, and collect into the 'pluck' combinator

Audiance:

- This presentation intended for programmers with some understanding of functional programming principles
- Code samples will be in Scala

Table of Contents

Stateful Computations

From the last talk: *A stateful computation is a function that takes some state and returns a value along with some new state*

```
s -> (s, a)
```

The State Monad lifts stateful computations so that they can be composed without explicitly passing around the state

The State Monad

DATACOM TSS

```scala
trait State[S, +A] {
  def run(initial: S): (S, A)
  //bind
  def flatMap[B](f: A => State[S, B]): State[S, B]
}

object State {
  //return
  def apply[S, A](f: S => (S, A)): State[S, A]
}
```

State Combinators

State combinator are functions you can compose to create more complex stateful computions from simple building blocks

```
//State.state: lift a value into a State
def state[S, A](a: A) = State[S, A]{ s => (s, a) }

//State.get: gets the implicit state value out of a State Monad
//Called init in Scalaz. I don't know why
def get[S] = State[S, S]{ s => (s, s) }

//State.gets: Apply f to the state and return the result
//State.gets(identity) is the same as State.get
def gets[S, A](f: S => A) = State[S, A]{ s => (s, f(s)) }

//State.put: Ignore input state and replace with new S
def put[S](s: S) = State[S, Unit]{ _ => (s, ()) }

//State.modify: Mutate the state without producing a result
def modify[S](f: S => S) = State[S, Unit]{ s => (f(s), ()) }
```

Table of Contents

Partial Functions

Most scala programmers will have seen a *case* statement before

```
//Will return T for scala.Some[T], but not scala.None
case Some(blurg) => blurg
```

Case statements are most often seen inside match statements, but they don't have to be... Case statements are actually defining *partial functions*

Partial Functions Continued

The term *partial function* comes from mathematics to mean a function that is only defined for some inputs.

In the domain of natural numbers:

$$f(x, y) = x - y$$

Is a partial function that is only defined when $x > y$

In Scala partial functions are functions that can be applied to *Any* type but are only defined for some of them

Partial Functions Continued

Partial functions can hence be used to defeat type safety (boo)

```
List(1, "Cake") map { case s: String => s.toUpperCase }
```

Compiles but produces a runtime error:

```
scala.MatchError: (of class java.lang.Integer)
```

Collect

Introducing collect:

```
collect[B](pf: PartialFunction[A, B]): List[B]
```

List.collect applies the partial function pf, only to the values in the list for which that function is defined.

```
List(1, "Cake") collect { case s: String => s.toUpperCase }
res1: List[String] = List(CAKE)
```

Type safety is restored!

Collect Continued

At DTSS heterogeneous lists are the core data type we use to describe security events

```
abstract class Field(val key: String, val value: JsonWritable) ...
case class Event(val fields: Fields) ...
```

Collect makes it possible to write functions that can process only the Field types that a particular function is interested in:

```
private def addGeoFields(fields: List[Field]) = {
  fields ::: (fields collect {
    case ip@IpAddress(k, v) =>
      if(!ip.isPrivate && !v.isLoopbackAddress) {
        getIpGeo(v.getHostAddress) map { StringField(s"Ip Geo ($k)", _) }
      } else None
  } collect { case Some(geoField) => geoField })
}
```

Table of Contents

The Problem

- Sometimes I want to collect somes values I know about, and then all the remaining values.
- I need a collect that optionally extracts a value, and removes it from the collection only if it existed.
- I don't want to mutate the collection as a side effect
- We can solve this problem with a new State combinator that only applies to `State[List[_], _]`

The Solution

```scala
//Applies f to the first value of type A that also matched predicate p
//Mutates state to remove the first A matching p
def pluck[A : ClassTag, B](
  f: A => B = (a: A) => a,
  p: A => Boolean = (a: A) => true
): State[List[_], Option[B]] = for {
  a <- State.gets[List[_], Option[A]] {
    //Extract all the elements of type A
    _.collect { case a: A => a }
    //only if they also match predicate p
    .filter { p }
    //Return the Some of the first element or None if the list is empty
    .headOption
  }
  //filter the state, removing the element that was extracted, if one was.
  _ <- State.modify[List[_]]{ _.diff(a.toList) }
  //Yield f(a) lifted into the Option Monad
} yield( a map(f) )
```

Pluck in Action

```
//Optionally produce an Issue from and Event only if mandatory fields are all present
def toIssue(e: Event) = {
  import scalaz.State
  val issueBuilder = for {
    summaryOpt  <- pluck[Summary, String]( (s: Summary) => s.prettyValue |> escape |> truncate )
    severityOpt <- pluck[SeverityField, Severity]( _.v.v )
    clientIdOpt <- pluck[ClientId, String]( _.prettyValue |> escape)
    //pluck GroupId and SourceData just because we don't want them to appear in the issue
    _           <- pluck[GroupId, Unit]
    _           <- pluck[SourceDataField, Unit]
    //Get the remaining fields
    rawFields   <- State.gets[Fields, List[(String, String)]] {
      _ map { (f: Field) => (f.prettyName |> escape, f.prettyValue |> escape) }
    }
    issue       <- State.state[Fields, Option[Issue]]( for {
      summary  <- summaryOpt
      severity <- severityOpt
      clientId <- clientIdOpt
    } yield new Issue(summary, severity, clientId, txt.issueDescirption(rawFields).toString))
  } yield issue

  issueBuilder(bestEffortOrdering(keyOrder)(e).fields)._2
}
```

Questions?