

Regaining Control

with Indexed Monads

Felix Mulder

flatMap(Oslo) 2018

Functional State

The Canonical Example

```
type Seed = Long

def rng(seed: Seed): (Seed, Long)

def rgb(seed: Seed): (Seed, Boolean) = {
  val (newSeed, rand) = rng(seed)
  (newSeed, rand > 0L)
}
```

Adding Three Random Numbers

```
val s0 = 0L  
  
val (s1, r0) = rng(s0)  
val (s2, r1) = rng(s1)  
val (_, r2) = rng(s2)  
  
r0 + r1 + r2  
// res0: Long = 3318706044697439873
```

The Canonical Example

Avoid passing the state?

The Canonical Example

Avoid passing the state?

Get rid of boilerplate?

The Canonical Example

$$S \Rightarrow (S, A)$$

The State Monad

```
case class State[S, A](run: S ⇒ (S, A)) extends AnyVal  
  
val nextLong: State[Seed, Long] = State(seed ⇒ rng(seed))  
  
def nextBool: State[Seed, Boolean] = ???
```


Map

We'd like to implement map in such a way that we do not affect S

Ergo, we want:

$\text{State}[S, A] \Rightarrow \text{State}[S, B]$

The State Monad

```
case class State[S, A](run: S ⇒ (S, A)) extends AnyVal {  
  
  def map[B](f: A ⇒ B): State[S, B] = State {  
    s0 ⇒ {  
      val (s1, a) = run(s0)  
      (s1, f(a))  
    }  
  }  
}
```

The State Monad

```
val nextBool: State[Seed, Boolean] = nextLong.map(_ > 0L)
```

The State Monad

How do we get rid of the explicit state passing?

The State Monad

We want to reason about the `A` value in `State[S, A]`

(without having to worry about `S`!)

The State Monad

We sort of want to pull the value out, to *bind* it...

The State Monad

```
case class State[S, A](run: S ⇒ (S, A)) extends AnyVal {  
  
  // ...  
  
  def flatMap[B](f: A ⇒ State[S, B]): State[S, B] = State {  
    s0 ⇒ {  
      val (s1, a) = run(s0)  
      f(a).run(s1)  
    }  
  }  
}
```

Adding Three Random Numbers

```
val addition: State[Seed, Long] = for {  
  r0 ← nextLong  
  r1 ← nextLong  
  r2 ← nextLong  
} yield r0 + r1 + r2  
  
addition.run(0L)  
// res1: (Seed, Long) = (-7280499659394350823,3318706044697439873)
```


Cooler stuff

```
case class Customer(id: Long, debt: Long, name: String)
```

```
val randomCustomer: State[Seed, Customer] =  
  for {  
    id      ← nextLong  
    debt    ← nextLong  
    isHuman ← nextBool  
    name    = if (isHuman) "Kim" else "Mark Zuckerberg"  
  } yield Customer(id, debt, name)
```

```
randomCustomer.run(1L)._2
```

```
// res2: Customer = Customer(1,7806831264735756412,Mark Zuckerberg)
```

Are we there yet?

Stack safety?

What about effects?

What about effects?

```
import cats.effect.IO

def getNonce(seed: Seed): IO[(Seed, Long)] =
  IO(rng(seed))

val nextNonce: State[Seed, Long] = State(getNonce)
// <console>:17: error: type mismatch;
//   found   : cats.effect.IO[(Seed, Long)]
//   (which expands to) cats.effect.IO[(Long, Long)]
//   required: (Seed, Long)
//   (which expands to) (Long, Long)
//       val nextNonce: State[Seed, Long] = State(getNonce)
//                                     ^
```

StateT

```
case class StateT[F[_], S, A](val run: S ⇒ F[(S, A)])  
  
val nextNonce: StateT[IO, Seed, Long] = StateT(getNonce)
```

Stack Safety

Now depends on $F[_]$

State in Cats

```
import cats.Eval
```

```
type State[S, A] = StateT[Eval, S, A]
```


Where is my indexed Monad?

Also, what are indexed Monads?

Indexed State Monad

```
case class IxState[I, O, A](run: I  $\Rightarrow$  (O, A))
```

Yet Another Naive Implementation

```
case class IxState[I, O, A](run: I  $\Rightarrow$  (O, A)) {  
  
  def map[B](f: A  $\Rightarrow$  B): IxState[I, O, B] = IxState {  
    i  $\Rightarrow$  {  
      val (o, a) = run(i)  
      (o, f(a))  
    }  
  }  
}
```

Yet Another Naive Implementation

```
case class IxState[I, O, A](run: I  $\Rightarrow$  (O, A)) {
```

```
// ...
```

```
def flatMap[OO, B](f: A  $\Rightarrow$  IxState[O, OO, B]): IxState[I, OO, B] =  
  IxState {  
    i  $\Rightarrow$  {  
      val (o, a) = run(i)  
      f(a).run(o)  
    }  
  }  
}
```

Now we can model state transitions!

```
sealed trait OrderStatus
case class Initiated() extends OrderStatus
case class Received()  extends OrderStatus
case class Packed()    extends OrderStatus
case class Shipped()   extends OrderStatus
case class Delivered() extends OrderStatus
```

Helper Functions

```
def received: IxState[Initiated, Received, Unit] =  
  IxState(_  $\Rightarrow$  (Received(), ()))
```

```
def packed: IxState[Received, Packed, Unit] =  
  IxState(_  $\Rightarrow$  (Packed(), ()))
```

```
def shipped: IxState[Packed, Shipped, Unit] =  
  IxState(_  $\Rightarrow$  (Shipped(), ()))
```

```
def delivered: IxState[Shipped, Delivered, Unit] =  
  IxState(_  $\Rightarrow$  (Delivered(), ()))
```

Usage

```
val order = for {  
  _ ← received  
  _ ← packed  
  _ ← shipped  
  _ ← delivered  
} yield ()
```

```
order.run(Initiated())  
// res3: (Delivered, Unit) = (Delivered(),())
```


Static errors!

```
for {  
  _ ← delivered  
  _ ← packed  
} yield ()  
// <console>:19: error: type mismatch;  
//   found   : IxState[Received,Packed,Unit]  
//   required: IxState[Delivered,?,?]  
//           _ ← packed  
//           ^
```

Cats

```
class IndexedStateT[F[_], SA, SB, A](val runF: F[SA  $\Rightarrow$  F[(SB, A)]])
```

Wait a minute, this looks familiar...

StateT in Cats

```
import cats.data.IndexedStateT
```

```
type StateT[F[_], S, A] = IndexedStateT[F, S, S, A]
```

The Epifani

The Epifani

Passing state explicitly



The Epifani

$$S \Rightarrow (S, A)$$



The Epifani

State[S, A]



The Epifani

StateT[F[_], S, A]



The Epifani

**State[S, A] =
StateT[Eval, S, A]**



The Epifani

IndexedStateT[F[_], SA, SB, A]



The Epifani

**StateT[F[_], S, A] =
IndexedStateT[F, S, S, A]**

