

# Regaining Control

with Indexed Monads

Felix Mulder

flatMap(Oslo) 2018

# Who am I?

- Scala 2.12 Docs Compiler
- Scala 3 Compiler Engineer @ EPFL w/ Martin Odersky
- Software Engineer @ Klarna

# Functional State

# The Canonical Example

```
type Seed = Long

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

def rng(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?

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



# The State Monad

```
case class State[S, A](run: S ⇒ (S, A)) extends AnyVal
```

# The State Monad

```
val nextLong: State[Seed, Long] = State(seed ⇒ rng(seed))
```

# The State Monad

```
val nextBool: State[Seed, Boolean] = ???
```

# Map

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

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

# 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(seed ⇒ getNonce(seed))  
// <console>:20: 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(seed ⇒ getNonce(seed))  
//                                                                    ^
```



# StateT

```
case class StateT[F[_], S, A](val run: S ⇒ F[(S, A)])
```

```
val nextNonce: StateT[IO, Seed, Long] = StateT(seed ⇒ getNonce(seed))
```

# Stack Safety

Now depends on  $F[_]$

# Requirements on `F[_]`

`Functor[F]` and `FlatMap[F]`

for `map` and `flatMap`

# State in Cats

```
import cats.Eval
```

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

**Where is my indexed Monad?**

**Also, what are indexed Monads?**

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

$$\mathbf{I} \Rightarrow (0, A)$$



# Chaining State Transitions

$(S1 \Rightarrow (S2, A)) \Rightarrow$   
 $(S2 \Rightarrow (S3, A)) \Rightarrow$   
 $(S3 \Rightarrow (S4, A)) \dots$

# 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)  
      }  
    }  
}
```

# Chained State Transitions

`IxState[S1, S2, A] ⇒`  
`IxState[S2, S3, B] ⇒`  
`IxState[S3, S4, C] ...`

## 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

```
object IxState {  
  def set[I, O](o: O): IxState[I, O, Unit] =  
    IxState(_  $\Rightarrow$  (o, ()))  
}
```

# Helper Functions

```
def received: IxState[Initiated, Received, Unit] =  
  IxState.set(Received())
```

```
def packed: IxState[Received, Packed, Unit] =  
  IxState.set(Packed())
```

```
def shipped: IxState[Packed, Shipped, Unit] =  
  IxState.set(Shipped())
```

```
def delivered: IxState[Shipped, Delivered, Unit] =  
  IxState.set(Delivered())
```



# Usage

```
val order = for {  
  _ ← received  
  _ ← packed  
  _ ← shipped  
  _ ← delivered  
} yield ()  
  
order.run(Initiated())  
// res3: (Delivered, Unit) = (Delivered(),())
```

# Static errors!

```
for {  
  _ ← delivered  
  _ ← packed  
} yield ()  
// <console>:22: 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]**



# Designing APIs Using IndexedStateT

# Designing APIs Using IndexedStateT

```
def createOrder(init: OrderInit): IndexedStateT[IO, Initiated, Received, Unit] =  
  IndexedStateT.setF(persist(init).as(Received()))
```

```
def packed: IndexedStateT[IO, Received, Packed, Unit] =  
  IndexedStateT.setF(persist(Packed()))
```

```
def shipped: IndexedStateT[IO, Packed, Shipped, Unit] =  
  IndexedStateT.setF(persist(Shipped()))
```

```
def delivered: IndexedStateT[IO, Shipped, Delivered, Unit] =  
  IndexedStateT.setF(persist(Delivered()))
```

## Abstracting over F[\_]

These structures allow you to stay generic. Don't commit too early.

- `F = Id`
- `F = Option`
- `F = OptionT[IO, ?]`
- `F = EitherT[IO, Throwable, ?]`
- `F = MonadError[Throwable, ?]`



# References

- Cats State - Typelevel Cats Documentation
- Control.Monad.State - Hackage
- pandoc-include-code - Oskar Wickström // @owickstrom
- tut - doc/tutorial generator for scala // @tpolecat

**Thank You!**