NEVER CHANGE STATE

AND STILL GET THINGS DONE

Remko de Jong [@aggenebbisj]
Martijn Blankestijn [@martijnblankest]

CODE.STAR

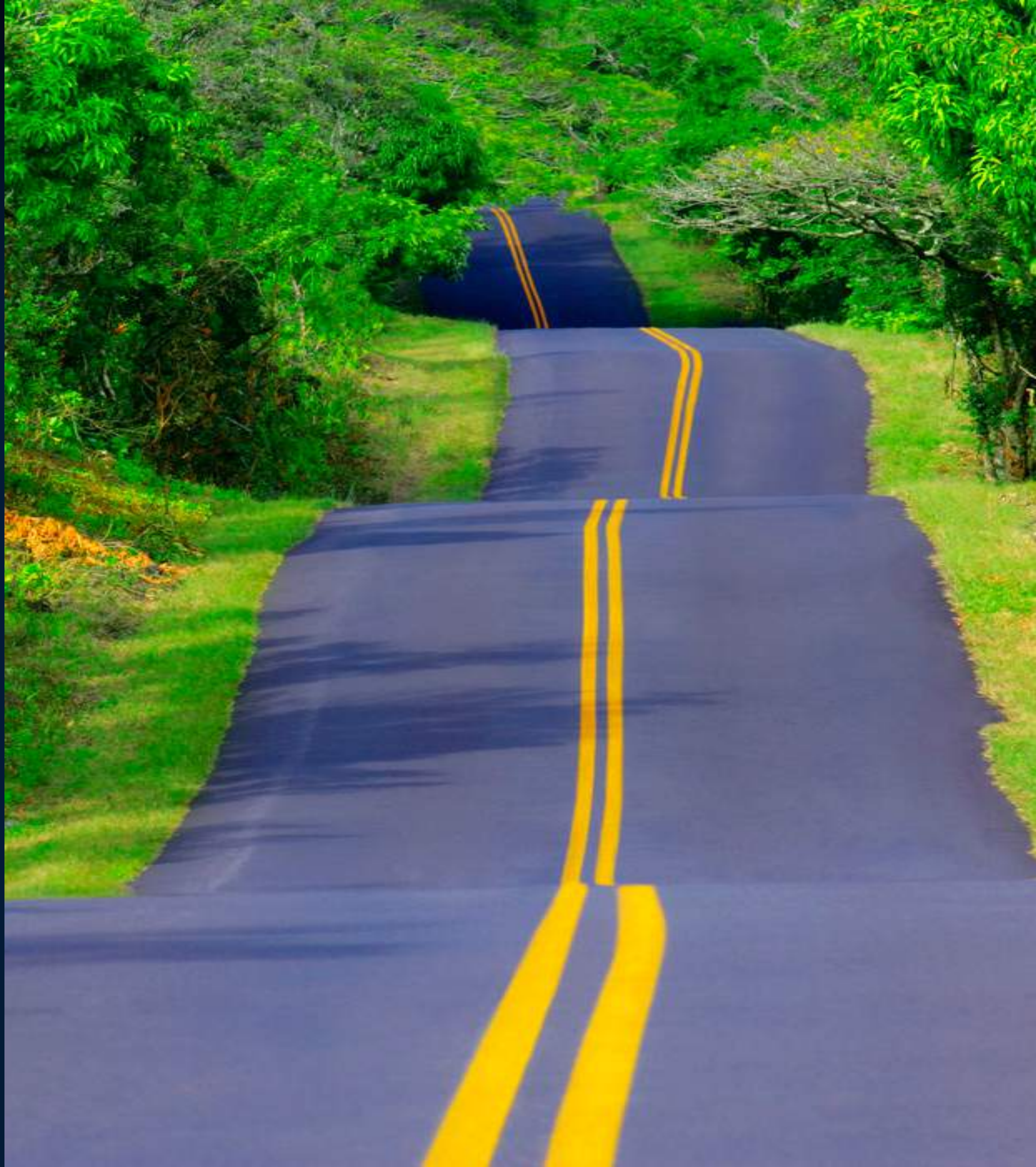
Roadmap

Why?

O.O. solution

More Functional

State data structure



Why?


SUPER BIG SALE!
sitma

Warning: Scala ahead

```java
int foo;

final int foo;

public int foo(int x);

public void bar();
```

```scala
var foo: Int

val foo: Int

def foo(x: Int): Int

def bar: Unit
```

```java
Stream.of(1,2,3)
  .map(x -> x + 1)
  .collect(
    Collectors.toList()
  );
```

```scala
List(1,2,3)
  .map(x => x + 1)
```

```java
public int foo(int x);

public void bar();

interface List<E>;

<T> int size(List<T> l);
```

```scala
def foo(x: Int): Int

def bar: Unit

trait List[E]

def size[T](l: List[T]): Int
```

```java
class Foo {
  final int x;

  public Foo(int x) {
    this.x = x;
  }

  public int getX() { … }
  // additional methods
  // like equals, copy
}
```

```scala
case class Foo(x: Int)
```

The Domain

```scala
case class Candy(color: Color)

case class Coin()
```

Object-Oriented Solution

```scala
class Machine(
  private val candies: mutable.Buffer[Candy],
  private var coins: Int) {

  def turn(): Candy = candies.remove(0)

  def insert(coin: Coin): Unit = coins = coins + 1

  def getCoins = coins
}
```

```
> val candies = ArrayBuffer(Candy(BLUE),
                            Candy(RED),
                            Candy(GREEN))

> val machine = new Machine(candies, coins = 0)

> machine.insert(Coin())
> val candy: Candy = machine.turn()

> machine.getCoins shouldBe 1
```

So what is the problem?

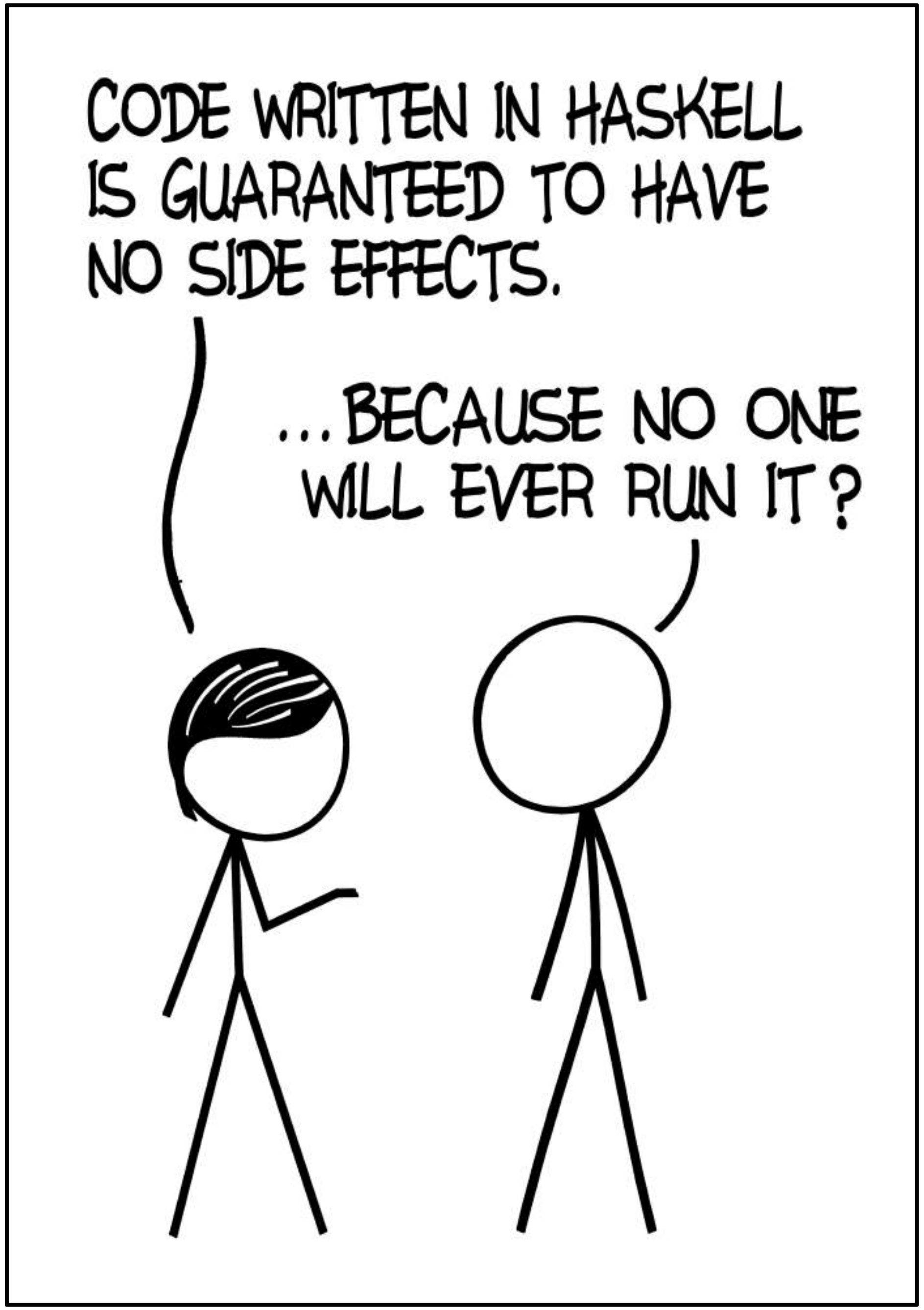
```
def f(x: Int): Int

f(2) == 3

f(2) == 3

f(f(2)) == 4

f(f(f(2))) == 5
```

```
def f(x: Int): Int =
  x + 1
```

```
def f(x: Int): Int

f(2) == 3

f(2) == 5 // Hmm...

f(f(2)) == 14 // OK...

f(f(f(2))) == 64 // WTF?
```

```
var y = 1

def f(x: Int): Int = {
  y = x + y
  y
}
```

More Functional



Referential Transparency

*“If an **expression** can be **replaced with** its corresponding **value** **without changing** the program's **behavior**”*


Let's make stuff immutable!
FOR
DUMMIES

Easier to

Reason about

Test

Compose

Parallellize

Recipe for immutability

- Pass state explicitly
- Make a copy
- Enjoy

Remember?

```
class Machine(
  private val candies: mutable.Buffer[Candy],
  private var coins: Int) {

  def turn(): Candy = candies.remove(0)

  def insert(coin: Coin): Unit = coins = coins + 1

  def getCoins = coins
}
```

```scala
case class Machine(
  candies: immutable.List[Candy],
  coins: Int
)

object Machine {

  def turn(m: Machine): (Machine, Candy) =
    ( m.copy(candies = m.candies.tail), m.candies.head )

  def insert(coin: Coin, m: Machine): Machine =
    m.copy(coins = m.coins + 1)
}
```

```
> val candies = List(Candy(BLUE), Candy(RED), Candy(GREEN))

> val m0 = Machine(candies, 0)

> val m1: Machine  = Machine.insert(Coin(), m0)
> val (m2, candy0) = Machine.turn(m1)

> val m3           = Machine.insert(Coin(), m2)
> val (m4, candy1) = Machine.turn(m1)

> m4.coins shouldBe 2
> m4.candies.size shouldBe 1
```

+ Simple
+ Immutable

- Extra argument
- Extra return value
- Error prone


Can we do better ?

State
Data
Structure

```
def turn(m: Machine): (Machine, Candy)
```

```
Machine => (Machine, Candy)
```

```
S => (S, A)
```

```
f: S => (S, A)
```

```
case class State[S, A](f: S => (S, A)) {



}
```

```scala
case class State[S, A](f: S => (S, A)) {
  def run(initial: S): (S, A) =
}
```

```scala
case class State[S, A](f: S => (S, A)) {
  def run(initial: S): (S, A) = f(initial)
}
```

```
> State(f).run(s) == f(s)
```

Let’s refactor...

```
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def turn(m: Machine): (Machine, Candy) =
    ( m.copy(candies = m.candies.tail), m.candies.head )


}
```

… and return the State structure

```scala
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def turn(): State[Machine, Candy] =
   State(m =>
    ( m.copy(candies = m.candies.tail), m.candies.head )
   )

}
```

And inserting a coin?

```
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def insert(coin: Coin, m: Machine): Machine =
    m.copy(coins = m.coins + 1)

}
```

“currying”

```scala
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def insert(coin: Coin)(m: Machine): Machine =
    m.copy(coins = m.coins + 1)

}
```

Make the return value explicit

```scala
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def insert(coin: Coin)(m: Machine): (Machine, Unit) =
    ( m.copy(coins = m.coins + 1), () )

}
```

And then use the State structure

```
case class Machine(candies: List[Candy], coins: Int)

object Machine {

  def insert(coin: Coin): State[Machine, Unit] =
   State(m =>
    ( m.copy(coins = m.coins + 1), () )
   )
}
```

```
// declaration
> val program: State[Machine,Unit] = Machine.insert(Coin())

// execution
> val m0 = Machine(candies, coins = 0)
> val (m1, _) = program.run(m0)

> m1.coins shouldBe 1
```

Functional Composition

“Building the Library”

mapping a function

```
val f: Double => Int

val g: Int => String
```

```
            +----+             +----+
            |    |             |    |
Double ->   | f  |  -> Int ->  | g  |  -> String
            |    |             |    |
            +----+             +----+
```

h

```
val h: Double => String = f.map(g)
```

```scala
case class State[S, +A](f: S => (S, A)) {
  def run(initial: S): (S, A) = f(initial)
  def map[B](transform: A => B): State[S, B]


}
```

```
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](



    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 =>


    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 =>
                  run(s0)


    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 =>
      val (s1, a) = run(s0)

    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 =>
      val (s1, a) = run(s0)
      (s1,                )
    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 =>
      val (s1, a) = run(s0)
      (s1, transform(a))
    )
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def map[B](transform: A => B): State[S, B] =
    State[S, B](s0 => {
      val (s1, a) = run(s0)
      (s1, transform(a))
    })
}
```

State[S, A]

```
          +----+
          |    |  -> s1
s0 ---->| f  |
          |    |  -> a
          +----+
```

```
          +----+
          |    | -> s1
s0  -->|  f  |
          |    | -> a   -> | transform | ---> b
          +----+
```

```
          +----+
          |    | -> s1 ---------------------> s1
s0 -->|  f |           +-------------+
          |    | -> a  -> | transform | ---> b
          +----+           +-------------+
```

State[S, B]

```
    +-------------------------------+
    |                               |
    |  +---+                        |
    |  |   |  -> s1 --------------------> s1
s0 -->| f |          +-----------+  |
    |  |   |  -> a  -> | transform | ---> b
    |  +---+          +-----------+  |
    |                               |
    +-------------------------------+
```

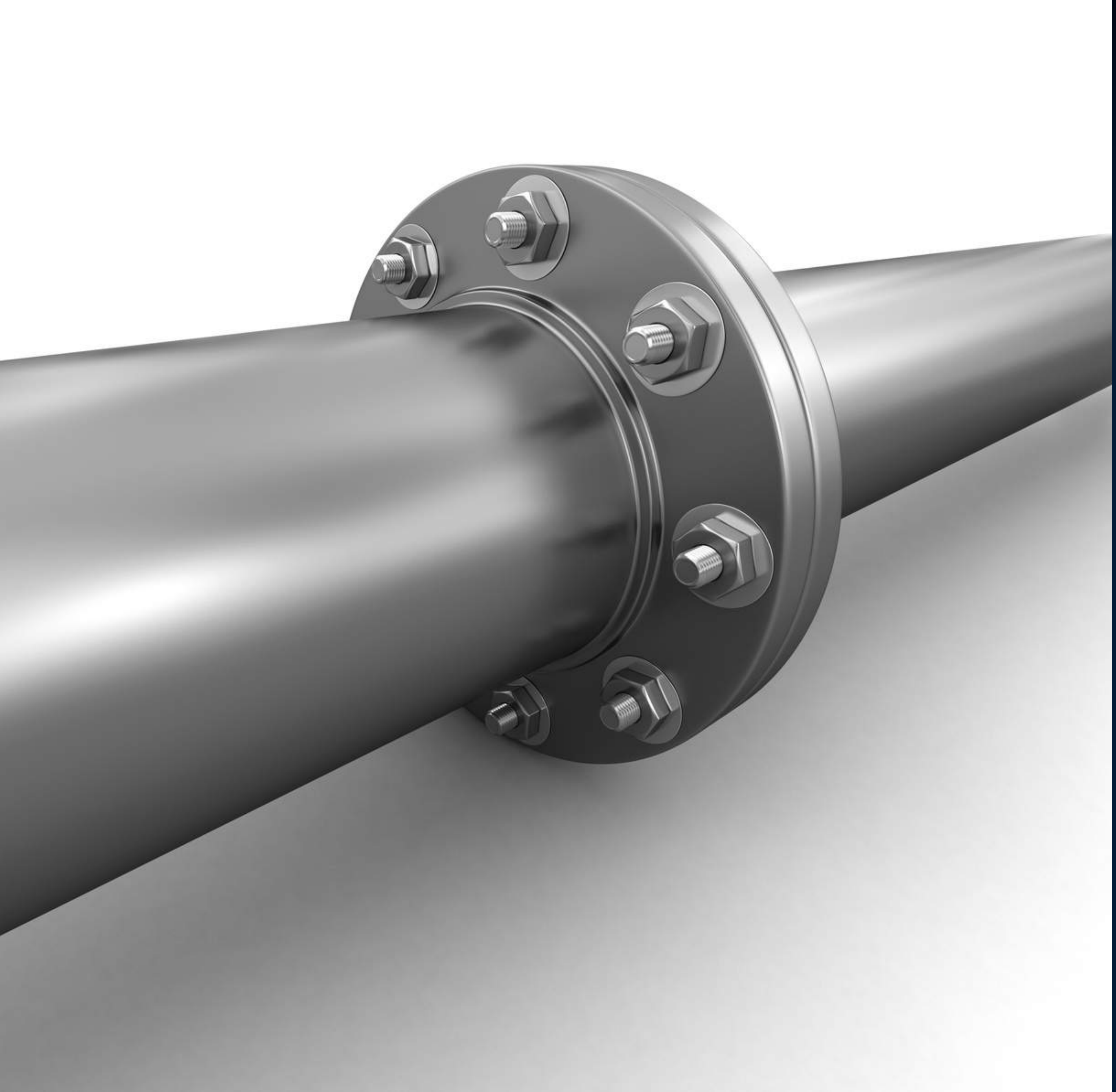
State[S, B]

```
     +-------------------------------+
     |                               |
     |                               |
     |                               |
     |                            ---> s1
s0 -->                               |
     |                            ---> b
     |                               |
     |                               |
     +-------------------------------+
```

```
// declaration
> val turn    : State[Machine, Candy]  = Machine.turn()
> val program: State[Machine, String] =
                          turn.map(candy => candy.color)

// execution
> val m0 = Machine(candies, coins = 0)
> val (m1, color) = program.run(m0)

> color shouldBe BLUE
```

```
// declaration
> val program = Machine.turn().map(candy => candy.color)



// execution
> val m0 = Machine(candies, coins = 0)
> val (m1, color) = program.run(m0)

> color shouldBe BLUE
```

```
// declaration
> val program = turn().map(_.color)



// execution
> val m0 = Machine(candies, coins = 0)
> val (m1, color) = program.run(m0)

> color shouldBe BLUE
```



Sequencing state functions

```scala
case class State[S, +A](f: S => (S, A)) {
  def run(initial: S): (S, A) = f(initial)
  def flatMap[B](g: A => State[S, B]): State[S, B]


}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def flatMap[B](g: A => State[S, B]): State[S, B] =
    State(s0 => {
      val (s1, a) = run(s0)

    })
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def flatMap[B](g: A => State[S, B]): State[S, B] =
    State(s0 => {
      val (s1, a) = run(s0)
      g(a)
    })
}
```

```scala
case class State[S, +A](f: S => (S, A)) {

  def run(initial: S): (S, A) = f(initial)

  def flatMap[B](g: A => State[S, B]): State[S, B] =
    State(s0 => {
      val (s1, a) = run(s0)
      g(a).run(s1)
    })
}
```

State[S, A]

```
             +----+
             |    |  --> s1
s0 ---->     | f  |
             |    |  -->  a
             +----+
```

```
          +----+                    +----+
          |    |  --> s1            |    |
s0 --->   | f  |                    | g  |
          |    |  -->  a  -->       |    |
          +----+                    +----+
```

```
         +----+                   +----+
         |    |  --> s1 -->       |    |  ---> s2
s0 --->  | f  |                   | g  |
         |    |  -->  a -->       |    |  ---> b
         +----+                   +----+
```

State[S, B]

```
    +-----------------------------+
    |    +----+            +----+ |
    |    |    | --> s1 --> |    | ---> s2
s0 ---> | f  |            | g  | |
    |    |    | -->  a --> |    | ---> b
    |    +----+            +----+ |
    +-----------------------------+
```

State[S, B]

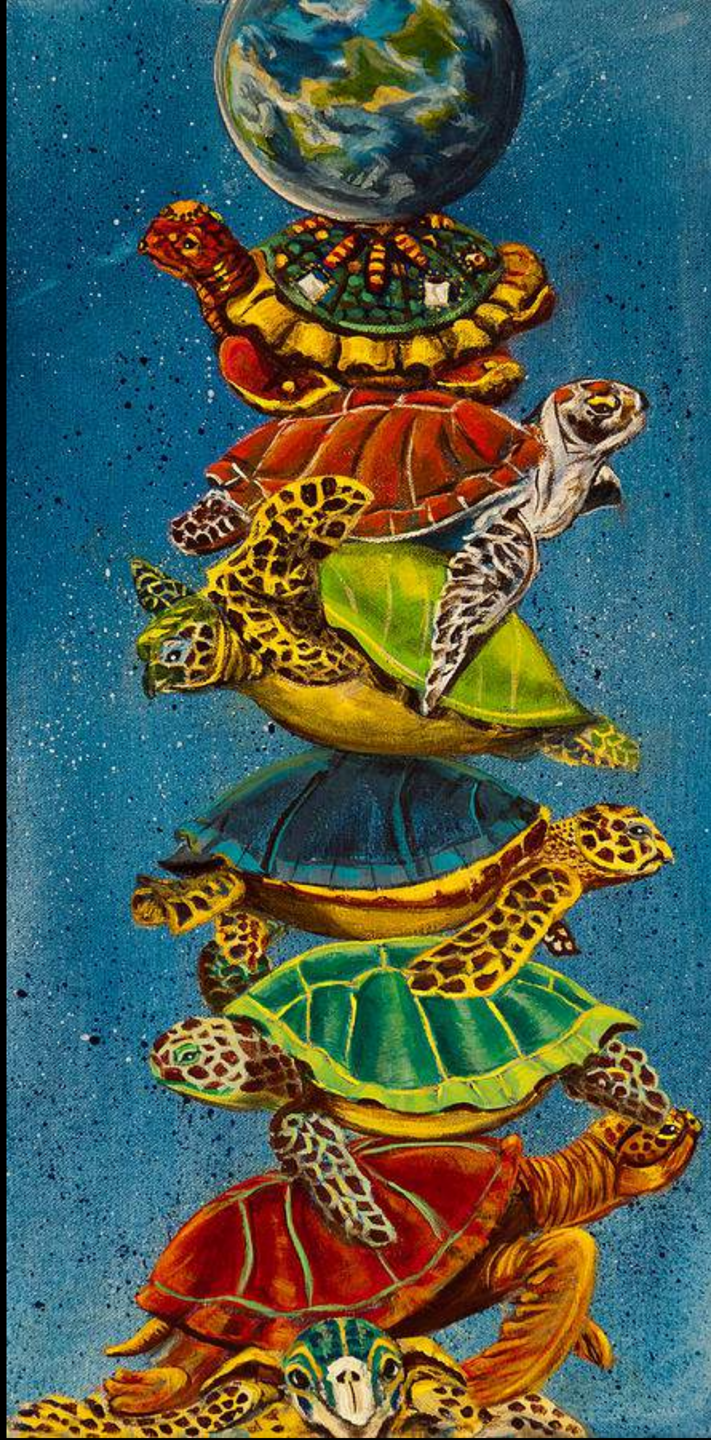
```
       +-------------------------+
       |                         |
       |                      ---> s2
s0 ---->                         |
       |                      ---> b
       |                         |
       +-------------------------+
```

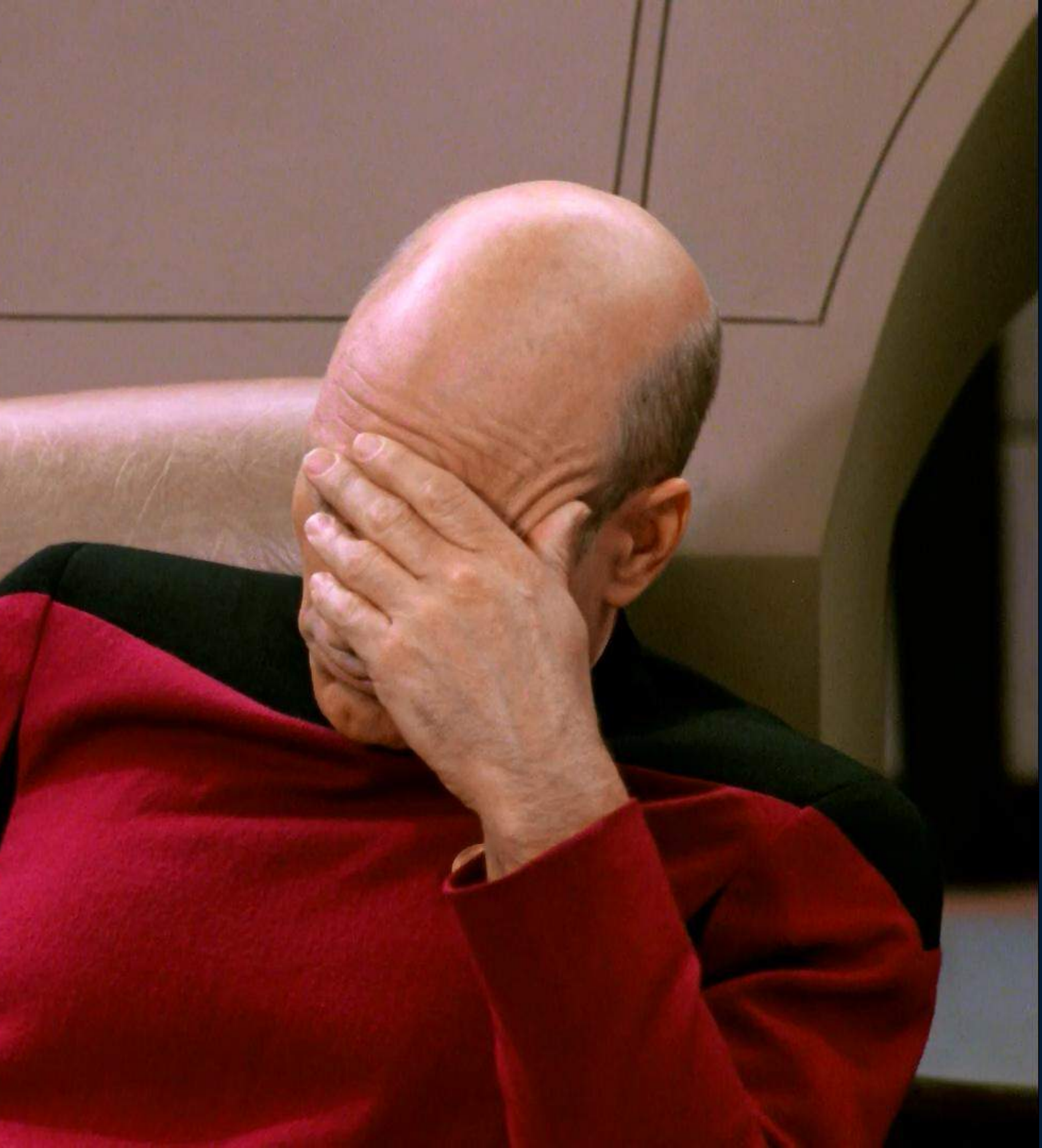
```
// declaration
> val m0 = Machine(candies, coins = 0)

> val program = insert(Coin()).flatMap(_ => turn())

// execution
> val (m1, candy) = program.run(m0)

> candy shouldBe Candy(BLUE)
```

Ok. Looks nice.

But can you do more than two?

```
> val program =
  insert(Coin)
    .flatMap(_ => turn()
      .flatMap(_ => insert(Coin)
        .flatMap(_ => turn()
          .flatMap(_ => insert(Coin)
            .flatMap(_ => turn()
              .flatMap(_ => insert(Coin)
                .flatMap(_ => turn()
                  ...
```





Cumbersome

for-comprehension

```scala
val program = for {
  _ <- insert(Coin())
  _ <- turn()
  _ <- insert(Coin())
  candy <- turn()
} yield candy

val m0 = Machine(candies, coins = 0)
val (m1, candy) = program.run(m0)
```

Changing State

```
object State {

  def get[S]: State[S, S]


}
```

```scala
object State {

  def get[S]: State[S, S] =
    State(s =>
      (?, ?)
    )

}
```

```
object State {

  def get[S]: State[S, S] =
    State(s =>
      (?, s)
    )

}
```

The value is the current state

```
object State {

  def get[S]: State[S, S] =
    State(s =>
      (s, s)
    )

}
```

And the state
is not changed!

```scala
object State {

  def set[S](newS: S): State[S, Unit]



}
```

```scala
object State {

  def set[S](newS: S): State[S, Unit] =
    State(oldState =>
      (?, ?)
    )

}
```

```scala
object State {

  def set[S](newS: S): State[S, Unit] =
    State(oldState =>
      (?, ())
    )

}
```

There is no
return value

```scala
object State {

  def set[S](newS: S): State[S, Unit] =
    State(oldState =>
      (newS, ())
    )

}
```

```
object State {

  def set[S](newS: S): State[S, Unit] =
    State(_ =>
      (newS, ())
    )

}
```

Since we're not using this...

Do the refactoring

```
def insert(coin: Coin): State[Machine, Unit] =
  State(m =>
    ( m.copy(coins = m.coins + 1), () )
  )

def insert(coin: Coin): State[Machine, Unit] =
  for {
    m <- State.get[Machine]
    _ <- State.set(m.copy(coins = m.coins + 1))
  } yield ()
```

Get & Set == Modify

```scala
object State {
  def modify[S](f: S => S): State[S, Unit] =
    State( s =>
      (f(s), ())
    )
}

def insert(coin: Coin): State[Machine, Unit] = {
  State.modify(m => m.copy(coins = m.coins + 1))
}
```

Final version of the library

```
case class State[S, A](f: S => (S, A)) {
  def run(initial: S): (S, A)
  def map[B](transform: A => B): State[S, B]
  def flatMap[B](g: A => State[S, B]): State[S, B]
}

object State {
  def get[S]: State[S, S]
  def set[S](newS: S): State[S, Unit]
  def modify[S](f: S => S): State[S, Unit]
}
```

Comparison

```scala
class Machine(
  val candies: Buffer[Candy],
  var coins: Int) {

  def getCoins = coins

  def turn(): Candy =
    candies.remove(0)



  def insert(coin: Coin): Unit =
    coins = coins + 1

}
```

```scala
case class Machine(
  candies: List[Candy],
  coins: Int)

object Machine {

 def turn(): State[Machine, Candy] =
   State(m =>
    (m.copy(candies = m.candies.tail),
     m.candies.head))

 def insert(coin: Coin): State[Machine,Unit] =
    modify(m => m.copy(coins = m.coins + 1))

}
```

In conclusion

+ Simple (scala)
+ Immutable
+ Automatic state wiring
+ For-comprehension

- More complex
- Performance impact