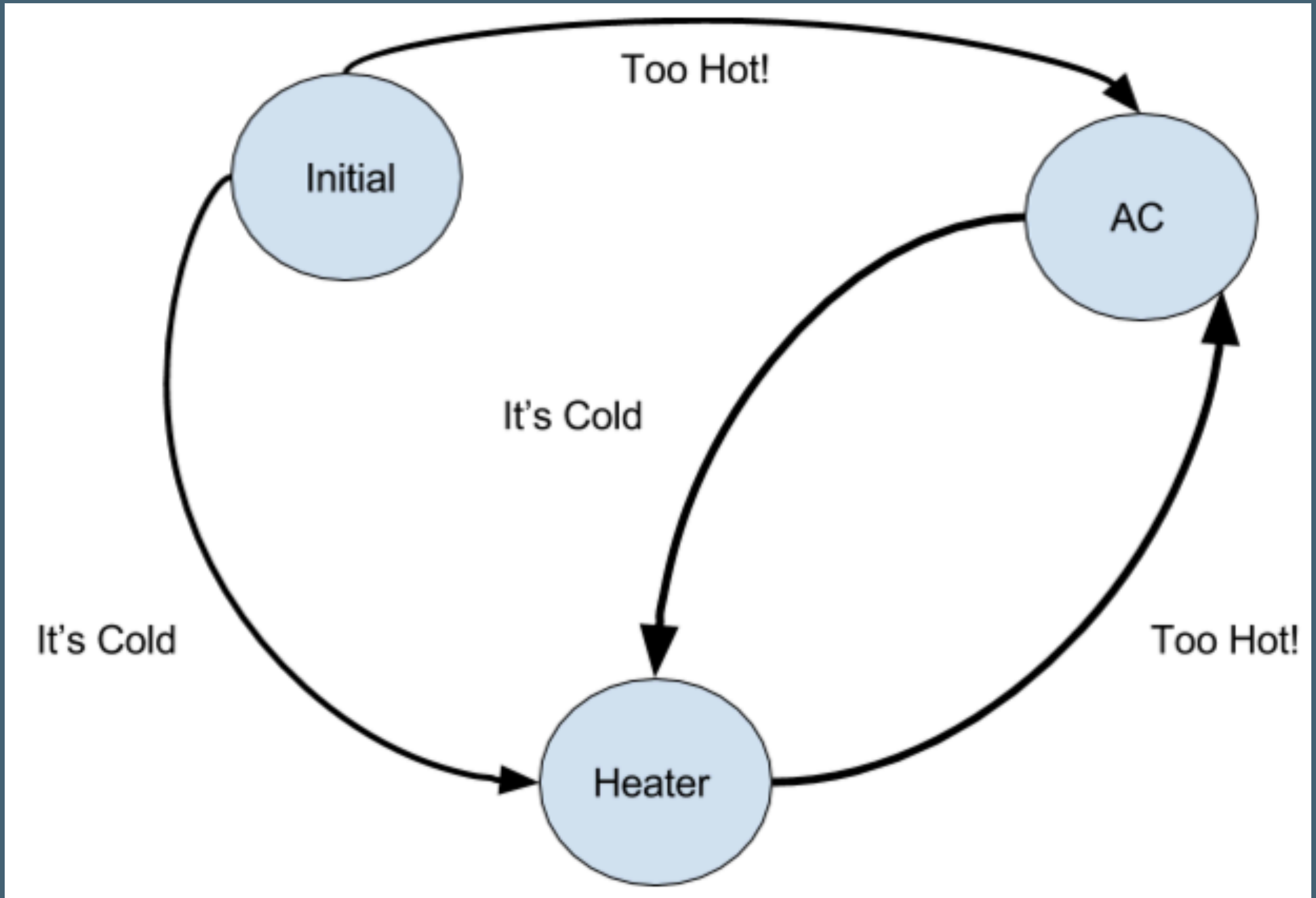
An Immutable State Machine

Now, Wait just a damn minute …

Isn't the State Machine all about **mutable** state?

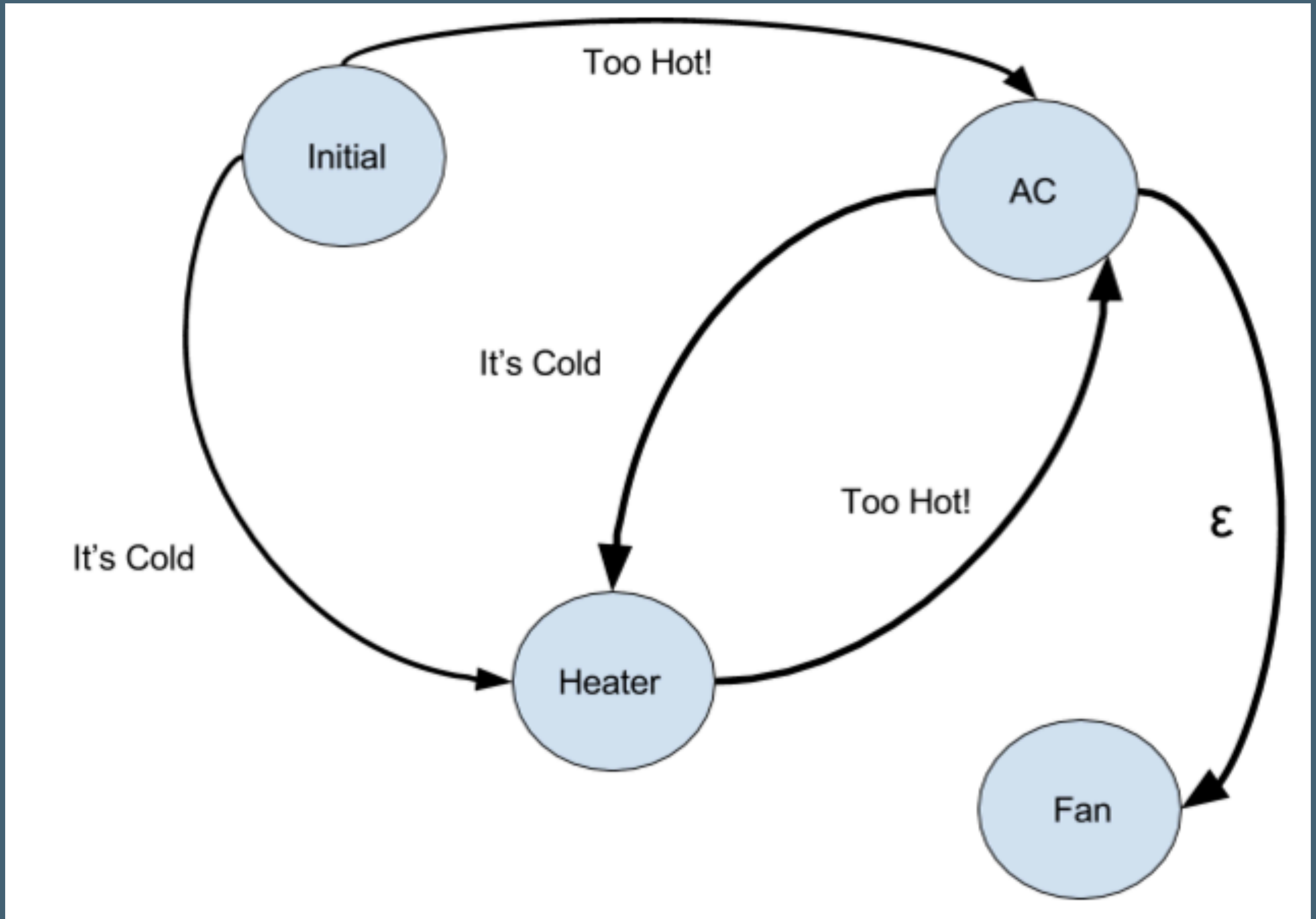
A Simple State Machine



Deterministic vs. Non-Deterministic Finite Automata

- Both recognize *Regular Languages* (i.e. regexs)
- NFAs can be in multiple states "at once"
- There is a variation of NFAs that support ε-transitions
- NFAs can be used to model complex systems w/ fewer states

State Machine with ε–transition added



Implementation in Scala: First Cut

(circa 2012)

```scala
class Transition[S, M <: StateMachine[S, M]]
  (start: S, end :S, guard: Option[M => Boolean]) {
    val startState = start
    val endState = end

    // yuck!
    def willFollow(stateMachine: M, withGuard : Boolean) =
      if (!withGuard && guard == None) true;
      else if(withGuard && guard == None) false;
      else (withGuard && guard.get(stateMachine))
}

class EpsilonTransition[S, M <: StateMachine[S, M]](start: S,end :S)
  extends Transition[S, M](start, end, None)
```

```scala
abstract class StateMachine[S, M <: StateMachine[S, M]]
  (transitions: Set[Transition[S, M]],
   initialStates: Set[S]) { this: M =>
     private var activeStates = initialStates // Downhill from here!
     private val stateDrains = transitions
       .groupBy(_.startState)

     def act() // ... Next Slide

     def transitionsToFollow(states : Set[S], guard : Boolean) =
       states.flatMap(stateDrains.getOrElse(_, Set()))
         .filter(_.willFollow(this, withGuard))
  }
```

```scala
def act() = {
  var entryStates = Set[S]()
  var exitStates = Set[S]()

  transitionsToFollow(activeStates, true).foreach {transition =>
      exitStates += transition.startState
      entryStates += transition.endState
  }

  entryStates = entryStates ++ transitionsToFollow(entryStates,
    false).map(_.endState)

  activeStates = ((activeStates -- (exitStates -- entryStates))
    ++ entryStates)
}
```

```scala
object HvacTransitions {
    val all = Set(
        new EpsilonTransition[HvacState, HVac](aircon, fan),
        new Transition[HvacState, HVac](aircon, fan, Some(_.temperature < 75)),
        new Transition[HvacState,HVac](heater, fan, Some(_.temperature > 50)),
        new Transition[HvacState, HVac](aircon, heater, Some(_.temperature < 50)),
        new Transition[HvacState, HVac](heater, aircon, Some(_.temperature > 75)),
        new Transition[HvacState, HVac](heater, fan, Some(_.temperature > 50)),
      new Transition[HvacState, HVac](fan, heater, Some(_.temperature < 50)),
      new Transition[HvacState, HVac](fan, aircon, Some(_.temperature > 75)))
}

class HVac extends StateMachine[HvacState, HVac](HvacTransitions.all, Set(heater)) {
  var temperature = 40

  def update(temperature : Int) = {
    this.temperature = temperature
    act
  }
}  // %$#! it, let's go bowling
```

This train departs from *Mutation City*

All Aboard!

First Stop:

The State Monad!

Note: We will be using cats, however scalaz also has extensive support for the State Monad, and the implementation is very similar.

`State[S, A]` wraps `f: S => (S, A)`

- **`S`** defines "our world" wrt the State
- **`A`** represents some projection (computation result) off the state at some point in time
- **`(S, A)`** can be thought of as a state *snapshot* or *absolute state*
- **`State[S, A]`** can be thought of as a state computation or *relative state*

A Simple example (part 1/2)

```
def addSample(i: Int) : State[List[Int], Unit] =
  State.get[List[Int]].modify( _ :+ i ).map(_ => ())

def makeSample(i: Int) : State[List[Int], Unit] =
  State[List[Int], Unit] { l => (l :+ i) -> () }

def mean : State[List[Int], Double] =
  State.get[List[Int]].map(l => l.sum / l.size)

def max : State[List[Int], Int] =
    State.get[List[Int]].map(l => l.max)
```

A Simple example (part 2/2)

```scala
val result =
  for {
    _ <- addSample(1)
    _ <- addSample(2)
    _ <- addSample(3)
    _ <- addSample(4)
    avg <- mean
    largest <- max
  } yield (avg, largest)

scala> result.run(List.empty[Int]).run
res1: (List[Int], (Double, Int)) = (List(1, 2, 3, 4),(2.0,4))
```

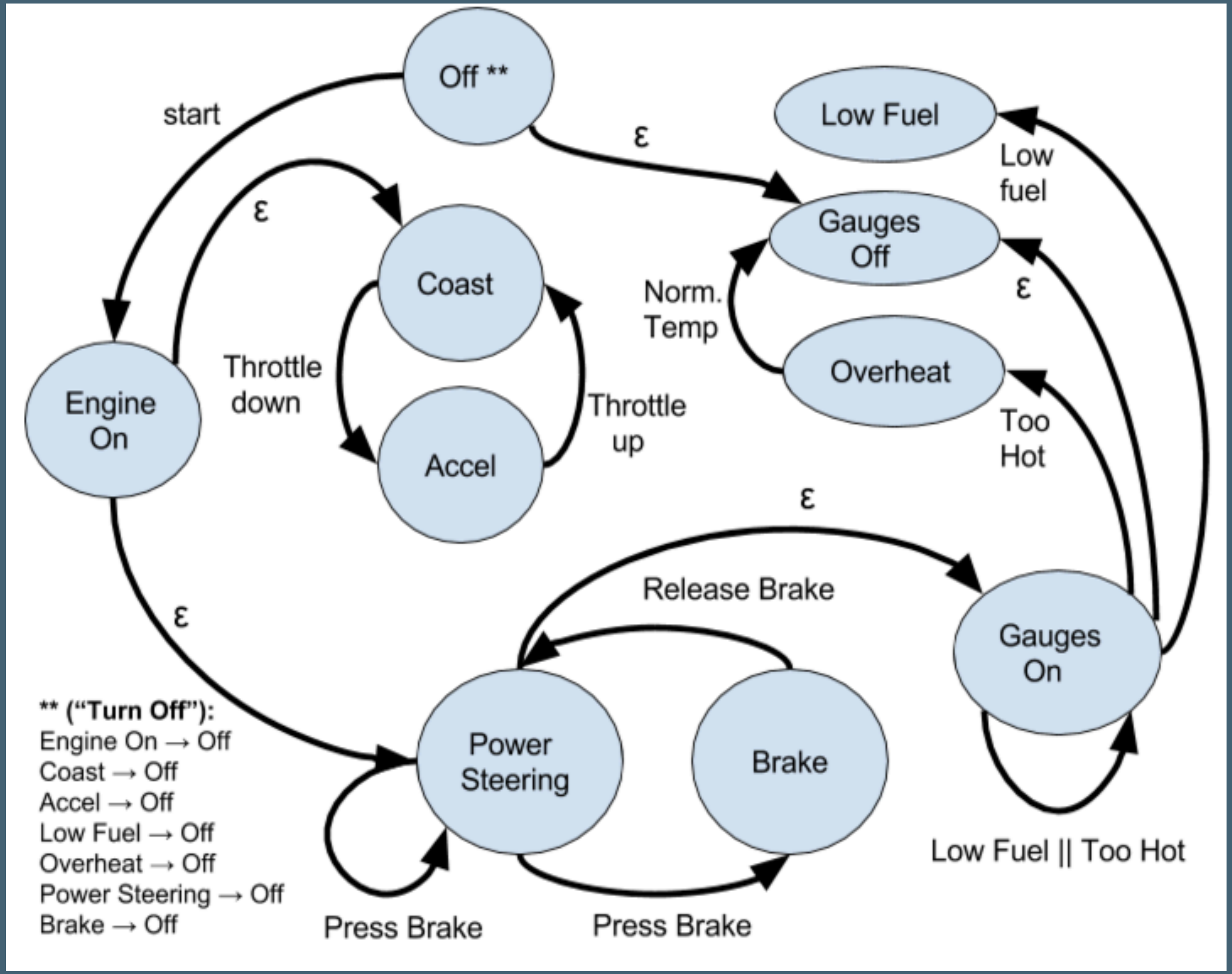
The **State Monad** leverages the power of *Trampolines* for stack-safety
as well as the **Free Monad** for execution over an **initial condition**

e.g. You can:

- define your own *free algebra* of state read/write operations,
- compose state operations together
- And now we have: purely functional immutable state!

Extraordinarily powerful!

A More Interesting State Machine



```scala
val exec = for {
  a <- step[CarEvent, CarState](Start)
  b <- step[CarEvent, CarState](ThrottleDown)
  c <- step[CarEvent, CarState](ThrottleUp)
  d <- step[CarEvent, CarState](PressBrake)
  e <- step[CarEvent, CarState](ReleaseBrake)
  f <- step[CarEvent, CarState](ThrottleDown)
  g <- step[CarEvent, CarState](TooHot)
  h <- step[CarEvent, CarState](ThrottleUp)
  i <- step[CarEvent, CarState](NormalTemp)
  j <- step[CarEvent, CarState](PressBrake)
  k <- step[CarEvent, CarState](TurnOff)
} yield (a, b, c, d, e, f, g, h, i, j, k)

val result = exec.run(car.start).run
```

Start Car ---
From(Off) -> To(EngineOn)
From(EngineOn) -> To(Coast)
From(EngineOn) -> To(PowerSteering)
From(PowerSteering) -> To(GaugesOn)
From(GaugesOn) -> To(GaugesOff)
Throttle Down ---
From(Coast) -> To(Accel)
Throttle Up ---
From(Accel) -> To(Coast)
Press Brake ---
From(PowerSteering) -> To(Brake)
From(PowerSteering) -> To(PowerSteering)

Throttle Down ---

From(Coast) -> To(Accel)

Too Hot ---

From(GaugesOn) -> To(GaugesOn)

From(GaugesOn) -> To(Overheat)

Throttle Up ---

From(Accel) -> To(Coast)

Normal Temp ---

From(Overheat) -> To(GaugesOff)

Press Brake ---

From(PowerSteering) -> To(Brake)

From(PowerSteering) -> To(PowerSteering)

Turn Off ---

From(Brake) -> To(Off)

From(EngineOn) -> To(Off)

From(Coast) -> To(Off)

Now we shall build an NFA using the State Monad

```scala
type F[I] = I => Boolean
case class From[+A](a: A) // "FromOps" def in : To[A]
case class To[+A](a: A)   // "ToOps"   def out : From[A]

type X[A] = (From[A], To[A]) // "Xops" def from, def to

case class Structure[I, A](
  transitions: Map[X[A], F[I]],
  εTransitions: List[X[A]],
  initial: From[A]
)
```

Our Machine Execution State

```
case class Machine[I, A](
  states : Set[To[A]] = Set.empty[To[A]],
  choiceSpace : State[Machine[I, A], Set[(From[A], Map[To[A], F[I]])]],
  εTransitions : Map[From[A], Set[To[A]]],
  initial: From[A],
  εNew: Set[To[A]] = Set.empty[To[A]],
  εCycle: Boolean = false
)

type MX[I, A] = State[Machine[I, A], List[X[A]]]
def emptyX[I, A]: MX[I, A] = State.get[Machine[I, A]].map(_ => List.empty[X[A]])
```

Some Machine operations

```
implicit class MachineOps[I, A](val m: Machine[I, A]) extends AnyVal {
  def +(a: To[A]) : Machine[I, A] = m.copy(states = m.states + a)
  def ++(a: Set[To[A]]) : Machine[I, A] = m.copy(states = m.states ++ a)
  def -(a: From[A]) : Machine[I, A] = m.copy(states = m.states - a.in)
  // ... others listed later
}
```

Building an initial machine from the structure

```
implicit class Starter[I, A](val struct: Structure[I, A]) extends AnyVal {
    def start : Machine[I, A] =
      Machine[I, A](states = Set(struct.initial.in),
        choiceSpace = State[Machine[I, A], Set[(From[A], Map[To[A], F[I]])]] { m =>
          m -> m.states.map(to => to.out -> struct.transitions
            .groupBy(getFrom2) // ((from, _), _)
            .mapValues(_ map getToAndF).getOrElse(to.out, Map())) // ((_, to), f)
            .filter { case (_, inF) => inF.nonEmpty }
        },
        εTransitions = struct.εTransitions.groupBy(getFrom2) // (from, _)
          .mapValues(_.toSet.map(getTo)),  // (_, to)
            initial = struct.initial)
  }
```

Defining a single State Machine "step"

```
def step[I, A](input: I) : MX[I, A] = State.get[Machine[I, A]]
    .map(_.states).map(_.map(_.out)) flatMap {
      _.foldLeft(emptyX[I, A]) { case (m, from) =>
        m.flatMap { x =>
          drainMany[I, A](from, input).map(_ ++ x)
        }
      }
    }
```

Conditionally "draining" from states

```
def drainCond[I, A](from: From[A], to: To[A], f: F[I], input: I) : MX[I, A] =
    Some(f(input)).filter(identity).map(_ => drain[I, A](from, to)) getOrElse emptyX

def drainMany[I, A](from: From[A], input: I): MX[I, A] =
  State.get[Machine[I, A]] flatMap (_.choiceSpace) flatMap {
    _.collect{ case (f, toF) if f == from => toF}
      .foldLeft(emptyX[I, A]) { case (state, toF) =>
        toF.foldLeft(state) { case (fromState, (to, f)) =>
          fromState.flatMap { x =>
            drainCond[I, A](from, to, f, input).map(_ ++ x)
          }
        }
      }
  }
```

Following individual transitions

```
def enter[I, A](to: To[A]) = State[Machine[I, A], To[A]] { _ + to -> to }
def exit[I, A](from: From[A]) = State[Machine[I, A], From[A]] { _ - from -> from }

def followX[I, A](fromA: From[A], toA: To[A]) : MX[I, A] = for {
  from <- exit(fromA)
  to <- enter(toA)
} yield (from -> to) :: Nil

def drain[I, A](from: From[A], to: To[A]) : MX[I, A] = for {
  gt <- followX(from, to)
  εt <- ε(to) // Follow epsilon transitions from here
} yield gt ++ εt
```

We also handle epsilon transitions more intelligently

This time we terminate ε-cycles

```scala
def ε[I, A](a: To[A]) : State[Machine[I, A], List[X[A]]] =
    State.get[Machine[I, A]].modify(_.εReset) flatMap ( _ ε a.out )

implicit class MachineOps[I, A](val m: Machine[I, A]) extends AnyVal {
    // ...
    def εReset: Machine[I, A] = m.copy(εNew = Set.empty[To[A]], εCycle = false)

    def ε(from: From[A]) : MX[I, A] = m.εTransitions.getOrElse(from, Set.empty[To[A]])
        .foldLeft(emptyX[I, A])(εExpand(from))

    def εExpand(from: From[A])(state: MX[I, A], to: To[A]) : MX[I, A] = εSafeExpand (
      εFollow(state, from -> to).transform { case (mt, x) =>
        Option(mt.εCycle).filter(identity).map(_ => mt -> List.empty[X[A]]).getOrElse(mt -> x)
      }, to
    )
    // ...
```

```scala
def εSafeExpand(state: MX[I, A], to: To[A]) : MX[I, A] =
  state.flatMap(px => Option(px.isEmpty).filter(identity).map(_ => emptyX[I, A])
    .getOrElse(ε(to.out).map(x => px ++ x).modify(_ ++ ends(px))))

def εFollow(state: MX[I, A], x: X[A]) : MX[I, A] = state.map(_ :+ x) modify { m =>
  Option(m.εNew.contains(x.to)).filter(identity).map(_ => m.εCycleDetected)
    .getOrElse(m εAdd x.to)
}

def toX(from: From[A], e: Set[To[A]]) : List[X[A]] = e.toList.map(t => from -> t)
def ends(t: List[X[A]]) : Set[To[A]] = t.map{ case (_, e) => e }.toSet

def εCycleDetected: Machine[I, A] = m.copy(εCycle = true)
def εAdd(a: To[A]) : Machine[I, A] = m.copy(states = m.states + a, εNew = m.εNew + a)
```

Thank You!

Questions?

Source:

https://github.com/ryanonsrc/catfarm/blob/master
/src/main/scala/io/nary/catfarm/state/nfa.scala

This code is solely for instructional/demonstration purposes and shouldn't be used for production. It is also likely that this code may be changed and/or moved