STATE MONAD

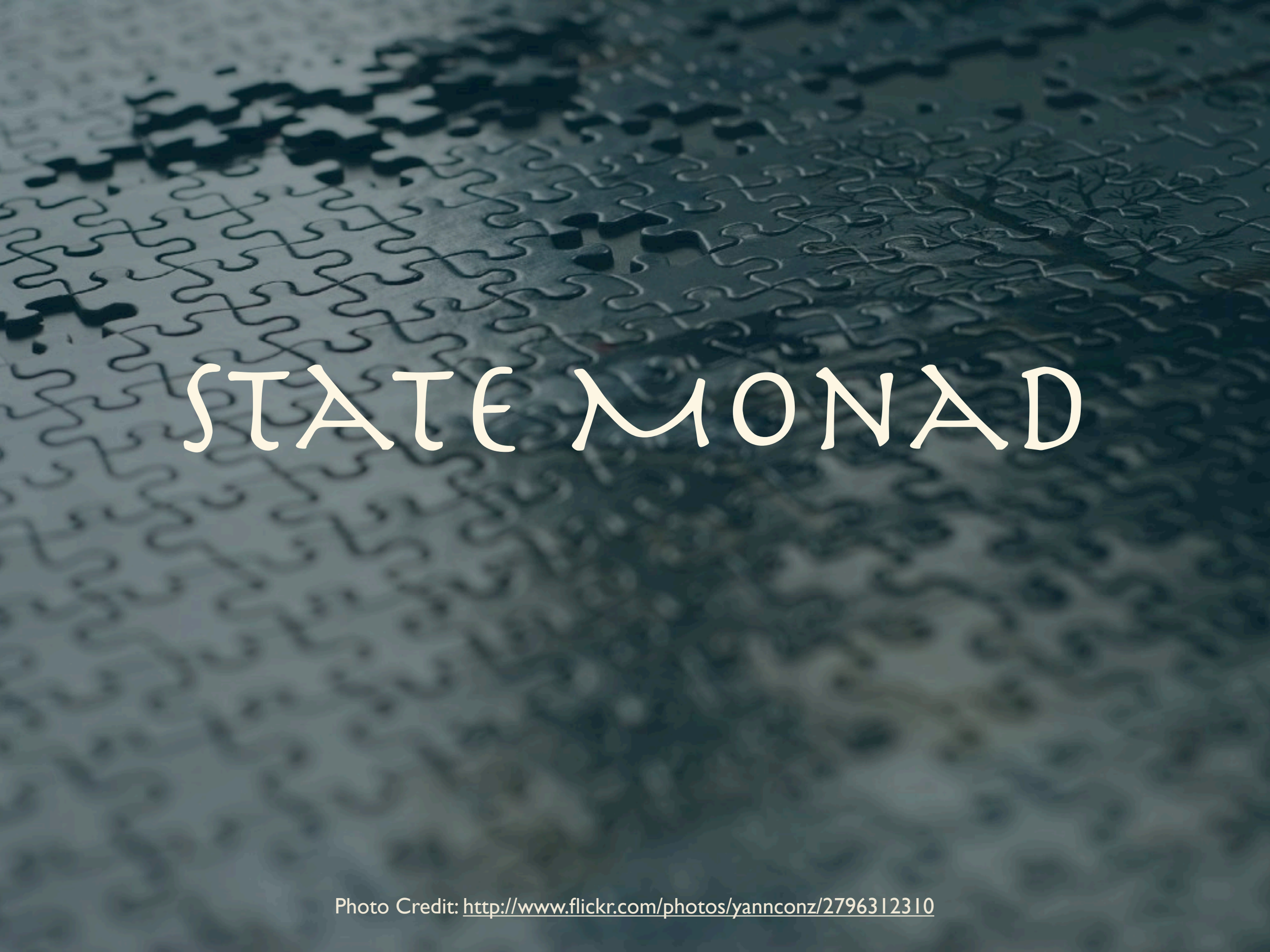

Photo Credit: http://www.flickr.com/photos/yannconz/2796312310

Agenda

- Motivate the *state monad* through simple examples that are familiar to developers with an imperative background

Simple Web Service

- Develop a service that provides statistics of users on a social networking service

Simple Web Service

```
trait SocialService {
  /**
   * Retrieves the follower statistics
   * for the specified user.
   */
  def followerStats(username: String):FollowerStats
}

case class FollowerStats(
  username: String,
  numFollowers: Int,
  numFollowing: Int)
```

Simple Web Service

```
object Stateless extends SocialService {
  def followerStats(username: String) = {
    // Make a call to a remote web service
  }
}
```

Simple Web Service

```
object Stateless extends SocialService {
  def followerStats(username: String) = {
    // Make a call to a remote web service
  }
}
```

Let's cache the responses for up to 5 minutes

Simple Web Service

```
object Stateful extends SocialService {
  private val mutableCache = ...

  def followerStats(username: String) = {
    // Check cache
    // If cached response exists and
    // it isn't over 5 minutes old, return it;
    // otherwise, call web service, update
    // cache with response, and return response
  }
}
```

Simple Web Service

```
object Stateful extends SocialService {
  private val mutableCache = ...

  def followerStats(username: String) = {
    // Check cache
    // If cached response exists and
    // it isn't over 5 minutes old, return it;
    // otherwise, call web service, update
    // cache with response, and return response
  }
}
```

How can we do this immutably?

Simple Web Service

```
object Stateful extends SocialService {
  private val mutableCache = ...
                                    Get from cache
  def followerStats(username: String) = {
    // Check cache
    // If cached response exists and
    // it isn't over 5 minutes old, return it;
    // otherwise, call web service, update
    // cache with response, and return response
  }
}
```

Write to cache

Return value

Simple Web Service

```
object Stateful extends SocialService {
  private val mutableCache = ...

  def followerStats(username: String) = {
    // Check cache
    // If cached response exists and
    // it isn't over 5 minutes old, return it;
    // otherwise, call web service, update
    // cache with response, and return response
  }
}
```

Get from cache

Create copy of cache with new value

Return new cache and value

Simple Web Service

Pass Cache to function

```
trait SocialService {
  def followerStats(u: String, c: Cache):
    (Cache, FollowerStats)
}
```

Return a new cache and the value

Simple Web Service

```
def followerStats(u: String, c: Cache) = {
  val (c1, ofs) = checkCache(u, c)
  ofs match {
    case Some(fs) => (c1, fs)
    case None => retrieve(u, c1)
  }
}

def checkCache(u: String, c: Cache):
  (Cache, Option[FollowerStats]) = ...

def retrieve(u: String, c: Cache):
  (Cache, FollowerStats) = ...
```

Simple Web Service

```
def checkCache(u: String, c: Cache):
  (Cache, Option[FollowerStats]) = {
  c.get(u) match {
    case Some(Timestamped(fs, ts))
      if !stale(ts) =>
        (c.copy(hits = c.hits + 1), Some(fs))
    case other =>
        (c.copy(misses = c.misses + 1), None)
  }
}

def stale(ts: Long): Boolean = {
  now - ts > (5 * 60 * 1000L)
}
```

Simple Web Service

```
def retrieve(u: String, c: Cache):
    (Cache, FollowerStats) = {
  val fs = callWebService(u)
  val tfs = Timestamped(fs, now)
  (c.update(u, tfs), fs)
}
```

Explicit State Passing

- Pros
  - Simple
  - Immutable
- Cons
  - state must be manually wired through all calls - error prone!

State Wiring Bugs

```
def followerStats(u: String, c: Cache) = {
  val (c1, ofs) = checkCache(u, c)
  ofs match {
    case Some(fs) => (c1, fs)
    case None => retrieve(u, c1)
  }
}
```

- Easy to pass the wrong value through (e.g., passing `c` instead of `c1`)
- Refactoring makes this an easy mistake to make

Immutable State Functions

Immutable State Functions

- Represent state as a value (immutable)

Immutable State Functions

- Represent state as a value (immutable)
- Pure functions need ways to read state, write state, and compute values

Immutable State Functions

- Represent state as a value (immutable)
- Pure functions need ways to read state, write state, and compute values
- Simplest way to represent this:

```
S ⇒ (S, A)
```

State datatype

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

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

State datatype

Runs the function, passing initial as input

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

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

State datatype

Runs the function, passing initial as input

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

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

Wraps a state function in to a State[S, A]

State datatype

Runs the function, passing initial as input

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

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

Wraps a state function in to a State[S, A]

State partially applied with one type S is a monad

i.e., State[S, ?] is a monad

State datatype

```
object State {
  def apply[S, A](f: S => (S, A)): State[S, A] =
    new State[S, A] {
      def run(i: S) = f(i)
    }
}
```

State datatype

```
trait State[S, +A] {
  def run(initial: S): (S, A)
  def map[B](f: A => B): State[S, B] =
    ???

  def flatMap[B](f: A => State[S, B]): State[S, B] =
    ???


}
```

State datatype

```
trait State[S, +A] {
  def run(initial: S): (S, A)
  def map[B](f: A => B): State[S, B] =
    State { s =>
      val (s1, a) = run(s)
      (s1, f(a))
    }
  def flatMap[B](f: A => State[S, B]): State[S, B] =
    ???


}
```

State datatype

```
trait State[S, +A] {
  def run(initial: S): (S, A)
  def map[B](f: A => B): State[S, B] =
    State { s =>
      val (s1, a) = run(s)
      (s1, f(a))
    }
  def flatMap[B](f: A => State[S, B]): State[S, B] =
    State { s =>
      val (s1, a) = run(s)
      f(a).run(s1)
    }
}
```

Let's refactor these...

Goal: use State datatype to simplify this code

```
def followerStats(u: String, c: Cache):
  (Cache, FollowerStats) = ...

def checkCache(u: String, c: Cache):
  (Cache, Option[FollowerStats]) = ...

def retrieve(u: String, c: Cache):
  (Cache, FollowerStats) = ...
```

Let's refactor these...

Step 1: make these have the shape S => (S, A)

```
def followerStats(u: String, c: Cache):
  (Cache, FollowerStats) = ...
```

```
def checkCache(u: String, c: Cache):
  (Cache, Option[FollowerStats]) = ...

def retrieve(u: String, c: Cache):
  (Cache, FollowerStats) = ...
```

Let's refactor these...

Step 1: make these have the shape S => (S, A)

```
                                                S
def followerStats(u: String, c: Cache):
  (Cache, FollowerStats) = ...
    S        A                  S
def checkCache(u: String, c: Cache):
  (Cache, Option[FollowerStats]) = ...
    S        A                  S
def retrieve(u: String, c: Cache):
  (Cache, FollowerStats) = ...
    S        A
```

Let's refactor these...

Step 1: make these have the shape S => (S, A)

```
                                                  S
def followerStats(u: String)(c: Cache):
  (Cache, FollowerStats) = ...
    S          A                  S
def checkCache(u: String)(c: Cache):
  (Cache, Option[FollowerStats]) = ...
    S          A                  S
def retrieve(u: String)(c: Cache):
  (Cache, FollowerStats) = ...
    S          A
```

Let's refactor these...

Step 2: rewrite followerStats to use State

```
def followerStats(u: String, c: Cache) = {
  val (c1, ofs) = checkCache(u, c)
  ofs match {
    case Some(fs) => (c1, fs)
    case None => retrieve(u, c)
  }
}
```

Let's refactor these...

Step 2: rewrite followerStats to use State

```
def followerStats(u: String)(c: Cache) = {
  State(checkCache(u)) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        State(retrieve(u))
    }
  }.run(c)
}
```

Let's refactor these...

Step 3: don't run the state

```
def followerStats(u: String)(c: Cache) = {
  State(checkCache(u)) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        State(retrieve(u))
    }
  }.run(c)
}
```

Let's refactor these...

Step 3: don't run the state

```
def followerStats(u: String) = {
  State(checkCache(u)) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        State(retrieve(u))
    }
  }
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def checkCache(u: String)(c: Cache):
  (Cache, Option[FollowerStats]) = {
  c.get(u) match {
    case Some(Timestamped(fs, ts))
      if !stale(ts) =>
      (c.copy(hits = c.hits + 1), Some(fs))
    case other =>
      (c.copy(misses = c.misses + 1), None)
  }
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def checkCache(u: String):
State[Cache, Option[FollowerStats]] = State { c =>
  c.get(u) match {
    case Some(Timestamped(fs, ts))
      if !stale(ts) =>
      (c.copy(hits = c.hits + 1), Some(fs))
    case other =>
      (c.copy(misses = c.misses + 1), None)
  }
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def retrieve(u: String)(c: Cache):
    (Cache, FollowerStats) = {
  val fs = callWebService(u)
  val tfs = Timestamped(fs, now)
  (c.update(u, tfs), fs)
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def retrieve(u: String):
    State[Cache, FollowerStats] = State { c =>
    val fs = callWebService(u)
    val tfs = Timestamped(fs, now)
    (c.update(u, tfs), fs)
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def followerStats(u: String) = {
  State(checkCache(u)) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        State(retrieve(u))
    }
  }
}
```

Let's refactor these...

Step 4: perform the same refactoring on helpers

```
def followerStats(u: String) = {
 checkCache(u) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        retrieve(u)
    }
  }
}
```

Let's refactor these...

Step 5: take advantage of rewritten helpers

```
def followerStats(u: String) = {
 checkCache(u) flatMap { ofs =>
    ofs match {
      case Some(fs) =>
        State { s => (s, fs) }
      case None =>
        retrieve(u)
    }
  }
}
```

Let's refactor these...

Step 5: take advantage of rewritten helpers

```
def followerStats(u: String) = for {
  ofs <- checkCache(u)
  fs <- ofs match {
    case Some(fs) =>
      State { s => (s, fs) }
    case None =>
      retrieve(u)
  }
} yield fs
```

State datatype

- Pros
  - simple
  - immutable
  - automatic state wiring
  - for-comprehensions let us write imperative looking code while maintaining these benefits

State Combinators

State Combinators

Constructs a State[S, A] for some value of A

```
def followerStats(u: String) = for {
  ofs <- checkCache(u)
  fs <- ofs match {
    case Some(fs) =>
      State { s => (s, fs) }
    case None =>
      retrieve(u)
  }
} yield fs
```

State Combinators

Constructs a State[S, A] for some value of A

```
def state[S, A](a: A): State[S, A] =
  State { s => (s, a) }
```

Let's refactor these...

```
def followerStats(u: String) = for {
  ofs <- checkCache(u)
  fs <- ofs match {
    case Some(fs) =>
      State.state(fs)
    case None =>
      retrieve(u)
  }
} yield fs
```

Let's refactor these...

```
def followerStats(u: String) = for {
  ofs <- checkCache(u)
  fs <- ofs.
    map(State.state[Cache, FollowerStats]).
    getOrElse(retrieve(u))
} yield fs
```

State Combinators

State Combinators

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

State Combinators

```scala
def get[S]: State[S, S] =
  State { s => (s, s) }
```

```scala
def gets[S, A](f: S => A): State[S, A] =
  State { s => (s, f(s)) }
```

State Combinators

```scala
def get[S]: State[S, S] =
  State { s => (s, s) }

def gets[S, A](f: S => A): State[S, A] =
  State { s => (s, f(s)) }

def put[S](s: S): State[S, Unit] =
  State { _ => (s, ()) }
```

State Combinators

```scala
def get[S]: State[S, S] =
  State { s => (s, s) }

def gets[S, A](f: S => A): State[S, A] =
  State { s => (s, f(s)) }

def put[S](s: S): State[S, Unit] =
  State { _ => (s, ()) }

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

Let's refactor these...

```
def checkCache(u: String):
State[Cache, Option[FollowerStats]] = State { c =>
  c.get(u) match {
    case Some(Timestamped(fs, ts))
      if !stale(ts) =>
      (c.copy(hits = c.hits + 1), Some(fs))
    case other =>
      (c.copy(misses = c.misses + 1), None)
  }
}
```

Let's refactor these...

```
def checkCache(u: String):
  State[Cache, Option[FollowerStats]] = for {
  c <- State.get[Cache]
  ofs <- State.state {
    c.get(u).collect {
      case Timestamped(fs, ts) if !stale(ts) =>
        fs
    }
  }
  _ <- State.put(ofs ? c.recordHit | c.recordMiss)
} yield ofs
```

Let's refactor these...

```
def checkCache(u: String):
  State[Cache, Option[FollowerStats]] = for {
  c <- State.get[Cache]
  ofs <- State.state {
    c.get(u).collect {
      case Timestamped(fs, ts) if !stale(ts) =>
        fs
    }
  }
  _ <- State.put(ofs ? c.recordHit | c.recordMiss)
} yield ofs
```

What's the potential bug with get at top and put at bottom?

Let's refactor these...

```
def checkCache(u: String):
  State[Cache, Option[FollowerStats]] = for {
  ofs <- State.gets { c: Cache =>
    c.get(u).collect {
      case Timestamped(fs, ts) if !stale(ts) =>
        fs
    }
  }
  _ <- State.modify { c: Cache =>
    ofs ? c.recordHit | c.recordMiss
  }
} yield ofs
```

Let's refactor these...

```
def retrieve(u: String):
  State[Cache, FollowerStats] = State { c =>
  val fs = callWebService(u)
  val tfs = Timestamped(fs, now)
  (c.update(u, tfs), fs)
}
```

Let's refactor these...

```
def retrieve(u: String):
  State[Cache, FollowerStats] = for {
  fs <- State.state(callWebService(u))
  tfs = Timestamped(fs, now)
  _ <- State.modify[Cache] { _.update(u, tfs) }
} yield fs
```

scalaz.State

scalaz.State

- Scalaz provides scalaz.State datatype

scalaz.State

- Scalaz provides scalaz.State datatype
- All examples so far compile with Scalaz

scalaz.State

- Scalaz provides scalaz.State datatype
- All examples so far compile with Scalaz
- Benefits of using Scalaz version:
  - Typeclass integration
  - Lens integration
  - Monad transformers

eval and exec

eval and exec

- In addition to `run`, there are two other convenient ways to run a stateful computation:

eval and exec

- In addition to `run`, there are two other convenient ways to run a stateful computation:
  - `eval` - Ignore output state (`S`), resulting in a value (`A`)

```
def eval(initial: S): A
```

eval and exec

- In addition to `run`, there are two other convenient ways to run a stateful computation:
  - `eval` - Ignore output state (`S`), resulting in a value (`A`)
    ```
    def eval(initial: S): A
    ```
  - `exec` - Ignore output value (`A`), resulting in a state (`S`)
    ```
    def exec(initial: S): S
    ```

eval and exec

- Given a monoid for `S`, it's common to use the zero element as the initial state
- Scalaz provides conveniences for this given an implicitly scoped `Monoid[S]`

```
def runZero(implicit S: Monoid[S]): (S, A)

def evalZero(implicit S: Monoid[S]): A

def execZero(implicit S: Monoid[S]): S
```

Typeclass Integration

- `State[Cache, ?]` has default implementations of:
  - `Pointed`
  - `Functor`
  - `Applicative`
  - `Monad`

Typeclass Integration

```
type StateCache[+A] = State[Cache, A]

Pointed[StateCache].point(10)
```

```
import scalaz.syntax.id._
10.point[StateCache]

Functor[StateCache].map(ten) { _.toString }

// etc.
```

Typeclass Integration

```
import scalaz.syntax.traverse._
import scalaz.std.list._
```

```
val listOfState: List[StateCache[FollowerStats]] =
List(s.followerStats("u1"), s.followerStats("u2"),
s.followerStats("u1"))

val stateOfList: StateCache[List[FollowerStats]] =
listOfState.sequence[StateCache, FollowerStats]

stateOfList.run(Cache.empty)
```

Resultant cache has 2 misses and 1 hit

State Transforms

State Transforms

State Transforms

- So far, we don't have a way to manipulate state data

State Transforms

- So far, we don't have a way to manipulate state data
- Suppose application is structured as `State[MyApplicationState, ?]`

State Transforms

- So far, we don't have a way to manipulate state data
- Suppose application is structured as `State[MyApplicationState, ?]`
- We don't want to fix every stateful function in the app to use `MyApplicationState` as state type

State Transforms

- So far, we don't have a way to manipulate state data
- Suppose application is structured as `State[MyApplicationState, ?]`
- We don't want to fix every stateful function in the app to use `MyApplicationState` as state type
- We want to compose state functions of "smaller" state types in to functions of "larger" state types

State Transforms

- So far, we don't have a way to manipulate state data
- Suppose application is structured as `State[MyApplicationState, ?]`
- We don't want to fix every stateful function in the app to use `MyApplicationState` as state type
- We want to compose state functions of "smaller" state types in to functions of "larger" state types
- Problem: given `State[T, ?]` how can we treat it as `State[S, ?]`

State Transforms

```
def lift[T,S,A](
  s: State[T, A]
): State[S, A] = {
  ???
}
```

State Transforms

```
def lift[T,S,A](
  s: State[T, A]
): State[S, A] = {
  State { inputS =>
    val inputT = ???
    val (outputT, a) = s.run(inputT)
    val outputS = ???
    (outputS, a)
  }
}
```

State Transforms

```
def lift[T,S,A](
  s: State[T, A]
): State[S, A] = {
  State { inputS =>
    val inputT = get(inputS)
    val (outputT, a) = s.run(inputT)
    val outputS = set(s, outputT)
    (outputS, a)
  }
}
```

State Transforms

```
def lift[T, S, A](
    s: State[T, A],
    get: S => T,
    set: (S, T) => S
  ): State[S, A] = {
    State { inputS =>
      val inputT = get(inputS)
      val (outputT, a) = s.run(inputT)
      val outputS = set(inputS, outputT)
      (outputS, a)
    }
  }
```

State Transforms

```
def lift[T, S, A](
    s: State[T, A],
    l: Lens[S, T]
  ): State[S, A] = {
    State { inputS =>
      val inputT = l.get(inputS)
      val (outputT, a) = s.run(inputT)
      val outputS = l.set(inputS, outputT)
      (outputS, a)
    }
  }
```

State Transforms

```
def lift[T, S, A](
  s: State[T, A],
  l: Lens[S, T]
): State[S, A] = {
  l.lifts(s)
}
```

State Transforms

```
sealed trait LensT[F[+_], A, B] {
  ...
  def lifts[C](
    s: StateT[F, B, C]
  )(implicit M: Bind[F]): StateT[F, A, C]
}
```

Integrating State and Either

Either Integration

```
trait Model
trait StatsQuery
trait QueryResult

def runQuery(s: String, m: Model):
  String \/ QueryResult = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res

def parseQuery(s: String):
  String \/ StatsQuery = "TODO".left
def performQuery(q: StatsQuery, m: Model):
  String \/ QueryResult = "TODO".left
```

Either Integration

```
trait Model
trait StatsQuery
trait QueryResult
```

Requirements changed and now parse and peform need to be stateful!

```
def runQuery(s: String, m: Model):
  String \/ QueryResult = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res

def parseQuery(s: String):
  String \/ StatsQuery = "TODO".left
def performQuery(q: StatsQuery, m: Model):
  String \/ QueryResult = "TODO".left
```

Either Integration

Wrap result types in State

```
trait QueryState
type QueryStateS[+A] = State[QueryState, A]

def runQuery(s: String, m: Model):
  QueryStateS[String \/ QueryResult] = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res

def parseQuery(s: String):
  QueryStateS[String \/ StatsQuery] = ...
def performQuery(q: StatsQuery, m: Model):
  QueryStateS[String \/ QueryResult] = ...
```

Either Integration

```
trait QueryState
type QueryStateS[+A] = State[QueryState, A]

def runQuery(s: String, m: Model):
  QueryStateS[String \/ QueryResult] = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res

def parseQuery(s: String):
  QueryStateS[String \/ StatsQuery] = ...
def performQuery(q: StatsQuery, m: Model):
  QueryStateS[String \/ QueryResult] = ...
```

Either Integration

```
trait QueryState
type QueryStateS[+A] = State[QueryState, A]

def runQuery(s: String, m: Model):
  QueryStateS[String \/ QueryResult] = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res
```

query is String \/ StatsQuery now!

```
def parseQuery(s: String):
  QueryStateS[String \/ StatsQuery] = ...
def performQuery(q: StatsQuery, m: Model):
  QueryStateS[String \/ QueryResult] = ...
```

Either Integration

We need to combine the effects of Either and State!

EitherT

```
trait EitherT[F[+_], +A, +B] {
  val run: F[A \/ B]
... }
```

- `EitherT` represents a value `F[A \/ B]`
- `EitherT` is a *monad transformer*, which lets us compose effects of Either with effects of an *arbitrary monad*
- We won't cover the theory of monad transformers See this fantastic talk: http://goo.gl/D6rt1

Full URL: http://marakana.com/s/video_monad_tranformers_in_scalamachine_scaliak,1232/index.html

EitherT

```
trait EitherT[F[+_], +A, +B] { ... }
```

- `EitherT` represents a value `F[A \/ B]`
- `EitherT` is a *monad transformer*, which lets us compose effects of Either with effects of an *arbitrary monad*
- We won't cover the theory of monad transformers See this fantastic talk: http://goo.gl/D6rt1

Full URL: http://marakana.com/s/video_monad_tranformers_in_scalamachine_scaliak,1232/index.html

EitherT

```
trait EitherT[F[+_], +A, +B] { ... }
```

```
F[A \/ B]
```

EitherT

```
trait EitherT[F[+_], +A, +B] { ... }
```

```
F[A \/ B]
```

We have:

```
QueryStateS[String \/ QueryResult]
```

EitherT

```
trait EitherT[F[+_], +A, +B] { ... }
```

```
F[A \/ B]
```

We have:

```
QueryStateS[String \/ QueryResult]
```

So we can use:

```
EitherT[QueryStateS, String, QueryResult]
```

EitherT Example

```
type QueryStateS[+A] = State[QueryState, A]

type ET[F[+_], A] = EitherT[F, String, A]

type QueryStateES[A] = ET[QueryStateS, A]
```

```
object QueryStateES {
  def apply[A](s: QueryStateS[String \/ A]):
    QueryStateES[A] = EitherT(s)
  def liftE[A](e: String \/ A): QueryStateES[A] =
    apply(Pointed[QueryStateS].point(e))
  def liftS[A](s: QueryStateS[A]): QueryStateES[A] =
    MonadTrans[ET].liftM(s)
}
```

EitherT Example

```
type QueryStateS[+A] = State[QueryState, A]

type ET[F[+_], A] = EitherT[F, String, A]

type QueryStateES[A] = ET[QueryStateS, A]
```

```
object QueryStateES {
  def apply[A](s: QueryStateS[String \/ A]):
    QueryStateES[A] = EitherT(s)
  def liftE[A](e: String \/ A): QueryStateES[A] =
    apply(Pointed[QueryStateS].point(e))
  def liftS[A](s: QueryStateS[A]): QueryStateES[A] =
    MonadTrans[ET].liftM(s)
}
```

def liftM[G[_] : Monad, A](a: G[A]): F[G, A]

EitherT Example

```
def runQuery(s: String, m: Model):
  QueryStateES[QueryResult] = for {
  query <- parseQuery(s)
  res <- performQuery(query, m)
} yield res

def parseQuery(s: String):
  QueryStateES[StatsQuery] = ...
def performQuery(q: StatsQuery, m: Model):
  QueryStateES[QueryResult] = ...

runQuery(s, m).run.run(initialQueryState)
```

StateT

Looking Closer

```
type State[S, +A] = StateT[Id, S, A]

object State extends StateFunctions {
  def apply[S, A](f: S => (S, A)): State[S, A] =
    new StateT[Id, S, A] {
      def apply(s: S) = f(s)
    }
}
```

Looking Closer

```
trait StateT[F[+_], S, +A] { ... }
type Id[+X] = X
type State[S, +A] = StateT[Id, S, A]
```

- `StateT` represents a function `S => F[(S, A)]`
- `Id` type alias lets us use `StateT` as `State`
- `StateT` is a *monad transformer*

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

getAndIncrement.replicateM(10).evalZero
```

StateT Example

```scala
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

getAndIncrement.replicateM(10).evalZero
```

def replicateM(n: Int): F[List[A]]

Think: flatMap(flatMap(flatMap(...)))

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

getAndIncrement.replicateM(10).evalZero
```

List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

getAndIncrement.replicateM(100000).evalZero
```

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

getAndIncrement.replicateM(100000).evalZero
```

StackOverflowError!

StateT Example

- Stack overflows because there are 10,000 nested flatMap calls occurring
- Scalaz provides the `Free` datatype, which when used with `Function0`, trades heap for stack

```
type Trampoline[+A] = Free[Function0, A]
```

- How can we combine the effects of `Trampoline` with `State`?

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

import scalaz.Free.Trampoline
getAndIncrement.lift[Trampoline].
  replicateM(100000).evalZero.run
```

For details on Free/Trampoline: http://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }

import scalaz.Free.Trampoline
getAndIncrement.lift[Trampoline].
  replicateM(100000).evalZero.run
```

```
def lift[M[_]: Pointed]: StateT[M[F[?]], S, A]
```

For details on Free/Trampoline: http://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf

StateT Example

```
val getAndIncrement: State[Int, Int] =
  State { s => (s + 1, s) }
```

StateT[Trampoline[Id], Int, Int]

```
import scalaz.Free.Trampoline
getAndIncrement.lift[Trampoline].
  replicateM(100000).evalZero.run
```

def lift[M[_]: Pointed]: StateT[M[F[?]], S, A]

Questions?

Photo Credit: http://www.flickr.com/photos/yannconz/2795462449