COMPOSABLE EVENT SOURCING WITH MONADS

Daniel Krzywicki @eleaar

Lambda Days 2018-02-22

// FABERNOVEL TECHNOLOGIES

The following is based on a true story

Outline

01 // Minimal model for event sourcing

02 // The problem of composition

03 // The Functional approach

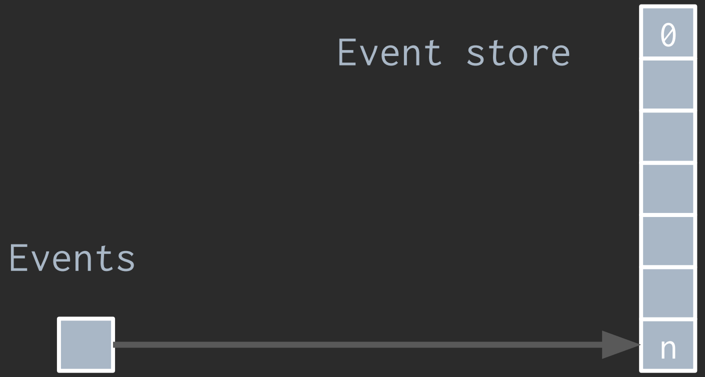
04 // Further possibilities

Introduction to Event Sourcing

Instead of storing state,

store changes to the state

Introduction :: event store



Changes to the system state are reified as events and appended to an event store.

Introduction :: replaying state

The system state is said to be projected/replayed from the store using event handlers

Introduction :: partial replays

We can easily replay only a part of the events to know the state of the system at any point in time

What the talk is not about

- Event sourcing frameworks
- Infrastructure (Kafka, MongoDB, …)
- Architecture (Event Store, sharding, partitioning)
- Error handling

What the talk is about

```
Functional  => Focus on composability
Programming => Focus on domain modeling and dev API
```

01 //
Minimal Model for Event Sourcing

01.1 //

Modeling the domain - Turtles!

Domain model

```scala
case class Turtle(id: String, pos: Position, dir: Direction)

object Turtle {

  def create(id: String, pos: Position, dir: Direction): Either[String, Turtle]

  def turn(rot: Rotation)(turtle: Turtle): Either[String, Turtle]

  def walk(dist: Int)(turtle: Turtle): Either[String, Turtle]

}
```

Domain logic

```scala
def create(id: String, pos: Position, dir: Direction): Either[String, Turtle] =
  if (tooFarAwayFromOrigin(pos)) Left("Too far away")
  else Right(Turtle(id, pos, dir))
```

Basic model - demo

```scala
def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5

state shouldBe Right(Turtle("123", Position(-1, -1), North))
```

Basic model - demo

```scala
def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5

state shouldBe Right(Turtle("123", Position(-1, -1), North))
```

Basic model - demo

```scala
def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5

state shouldBe Right(Turtle("123", Position(-1, -1), North))
```

Basic model - demo

```scala
def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5

state shouldBe Right(Turtle("123", Position(-1, -1), North))
```

Basic model - demo

```
val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5
```

Basic model - demo

```scala
// We have to propagate the state manually - verbose and error-prone

val state = for {
  state1 <- Turtle.create("123", Position.zero, North)
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
  state4 <- walkRight(2)(state3)
  state5 <- walkRight(2)(state4)
} yield state5
```

Basic model - demo

```scala
// We can flatMap to avoid passing the state explicitly

val state =
  Turtle.create("123", Position.zero, North)
    .flatMap(walkRight(1))
    .flatMap(walkRight(1))
    .flatMap(walkRight(2))
    .flatMap(walkRight(2))
```

01.2 //
Event sourcing the domain

Modeling events

```scala
// We can represent the result of our commands as events
sealed trait TurtleEvent { def id: String }

case class Created(id: String, pos: Position, dir: Direction) extends TurtleEvent
case class Turned(id: String, rot: Rotation) extends TurtleEvent
case class Walked(id: String, dist: Int) extends TurtleEvent
```

Event handler for creation events

```scala
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```scala
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```scala
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```
type EventHandler[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(turtle), Turned(id, rot)) if id == turtle.id =>
    Some(turtle.copy(dir = Direction.rotate(turtle.dir, rot)))
  case (Some(turtle), Walked(id, dist)) if id == turtle.id =>
    Some(turtle.copy(pos = Position.move(turtle.pos, turtle.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler usage :: demo

```scala
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Event handler usage :: demo

```
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Event handler usage :: demo

```
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Event handler usage :: demo

```scala
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Event handler usage :: demo

```scala
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Event handler usage :: demo

```scala
val initialState = Option.empty[Turtle]
val events = Seq(
  Created("123", Position.zero, North),
  Walked("123", 1),
  Turned("123", ToRight),
)

val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Syntactic sugar for handler definition

```
// There is some boilerplate when defining the handler
val handler0: EventHandler[Turtle, TurtleEvent] = {
  case (None, Created(id, pos, dir)) =>
    Some(Turtle(id, pos, dir))
  case (Some(t), Turned(id, rot)) if id == t.id =>
    Some(t.copy(dir = Direction.rotate(t.dir, rot)))
  case (Some(t), Walked(id, dist)) if id == t.id =>
    Some(t.copy(pos = Position.move(t.pos, t.dir, dist)))
  case (event, state) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Syntactic sugar for handler definition

```scala
// We can use a factory to reduce boilerplate and have a neat final handler

val handler = EventHandler[Turtle, TurtleEvent] {
  case (None, Created(id, pos, dir)) =>
    Turtle(id, pos, dir)
  case (Some(t), Turned(id, rot)) if id == t.id =>
    t.copy(dir = Direction.rotate(t.dir, rot))
  case (Some(t), Walked(id, dist)) if id == t.id =>
    t.copy(pos = Position.move(t.pos, t.dir, dist))
}
```

Domain logic - revisited

```scala
// Without event sourcing
def create(id: String, pos: Position, dir: Direction): Either[String, Turtle] =
  if (tooFarAwayFromOrigin(pos)) Left("Too far away")
  else Right(Turtle(id, pos, dir))

// With event sourcing
def create(id: String, pos: Position, dir: Direction) =
  if (tooFarAwayFromOrigin(pos)) Left("Too far away")
  else Right(Created(id, pos, dir))

  // in the handler
  case (None, Created(id, pos, dir)) =>
    Turtle(id, pos, dir)
```

What we have seen so far

What we have seen so far

- modeling the domain
- defining events and event handlers

What more could we want?


COMPOSABILITY
imgflip.com

02 //
The problem of composition

Composition in event sourcing

Composing event handlers is easy - they’re just plain functions

Composing commands is less trivial - what events should we create?

Why would we want to compose commands in the first place?

Basic model - demo

```
// Remember this one in the basic model? It's actually a composite command

def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

// How do we event source it?
```

Composing commands

How about these?

```
def turnAround()(turtle: Turtle): Either[String, Turtle] = ???

def makeUTurn(radius: Int)(turtle: Turtle): Either[String, Turtle] = ???
```

Composing commands

```
// The CISC approach: let's just create more event types

// So far we had
create   ---> Created
walk     ---> Walked
turn     ---> Turned

// So that would give us
walkRight    ---> WalkedRight
turnAround   ---> TurnedAround
makeUTurn    ---> MadeUTurn
```

Composing commands

```
// The CISC approach: let's just create more event types

// So far we had
create   ---> Created
walk     ---> Walked
turn     ---> Turned

// So that would give us
walkRight   ---> WalkedRight
turnAround  ---> TurnedAround
makeUTurn   ---> MadeUTurn

// Problem: extensivity
```

Composing commands

```scala
// Consider we might have additional handlers
def turtleTotalDistance(id: String): EventHandler[Int, TurtleEvent] = {
  case (None, Created(turtleId, _, _)) if id == turtleId =>
    Some(0)
  case (Some(total), Walked(turtleId, dist)) if id == turtleId =>
    Some(total + dist)
  case (maybeTotal, _) =>
    maybeTotal
}
```

Composing commands

```scala
// Adding new event types forces up to update every possible interpreter
def turtleTotalDistance(id: String): EventHandler[Int, TurtleEvent] = {
  case (None, Created(turtleId, _, _)) if id == turtleId =>
    Some(0)
  case (Some(total), Walked(turtleId, dist)) if id == turtleId =>
    Some(total + dist)
  case (Some(total), WalkedRight(turtleId, dist)) if id == turtleId =>
    Some(total + dist)
  case (Some(total), MadeUTurn(turtleId, radius)) if id == turtleId =>
    Some(total + 3 * radius)
  case (maybeTotal, _) =>
    maybeTotal
}
```

Events with overlapping semantics are leaky

How about composition?

Composing commands

```
// The RISC approach: let's compose existing event types

// So far we had
create  ---> Created
walk    ---> Walked
turn    ---> Turned

// So that would give us
walkRight   ---> Walked + Turned
turnAround  ---> Turned + Turned
makeUTurn   ---> Walked + Turned + Walked + Turned + Walked
```

Composing commands

```
// That's what we did without event sourcing: composition

def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2

// Why should it be any different now?
```

02.1 //

Dealing with multiple events

Composing commands

```scala
// So how could we try to compose this:

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  event2 <- Turtle.turn(ToRight)(???)
} yield ???
```

Composing commands

```
// We need a state here

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  event2 <- Turtle.turn(ToRight)(???)
} yield ???
```

Composing commands

```scala
// We can use our handler to replay the first event

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state2 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state2)
} yield ???
```

Composing commands

```
// We can use our handler to replay the first event

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state2 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state2)
} yield ???
```

Composing commands

```
// We’ll need to return both events

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state2 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state2)
} yield Seq(event1, event2)
```

Persisting multiple events atomically

Event journal - revisited

```scala
// Obviously, we'll need to be able to persist multiple events together

trait WriteJournal[EVENT] {
  // Saving the batch of events must be atomic
  def persist(events: Seq[EVENT]): Future[Unit]
}
```

Persisting multiple events

Persisting multiple events may seem odd to some.

Others do that as well:
Greg Young's Event Store has a concept of atomic "commits" which contain multiple events.
Akka Persistence API allows to persist multiple events at once, as long as the journal supports it

Are we good already?

02.2 //

The limits of an imperative approach

An imperative approach problems

```scala
// This imperative approach...
for {
  event1 <- Turtle.create("123", zero, North)
  state1 = Turtle.handler(None, event1).value
  event2 <- Turtle.walk(1)(state1)
} yield Seq(event1, event2)
```

An imperative approach problems:: does not scale

```
// This imperative approach… does not scale!
for {
  event1 <- Turtle.create("123", zero, North)
  state1 = Turtle.handler(None, event1).value
  event2 <- Turtle.walk(1)(state1)
  state2 = Turtle.handler(Some(state1), event2).value
  event3 <- Turtle.walk(1)(state2)
  state3 = Turtle.handler(Some(state2), event3).value
  event4 <- Turtle.walk(1)(state3)
  state4 = Turtle.handler(Some(state3), event4).value
  event5 <- Turtle.walk(1)(state4)
  state5 = Turtle.handler(Some(state4), event5).value
  event6 <- Turtle.walk(1)(state5)
} yield Seq(event1, event2, event3, event4, event5, event6)
```

An imperative approach problems :: replaying events

```
// We need to manually replay at each step
for {
  event1 <- Turtle.create("123", zero, North)
  state1 = Turtle.handler(None, event1).value
  event2 <- Turtle.walk(1)(state1)
  state2 = Turtle.handler(Some(state1), event2).value
  event3 <- Turtle.walk(1)(state2)
  state3 = Turtle.handler(Some(state2), event3).value
  event4 <- Turtle.walk(1)(state3)
  state4 = Turtle.handler(Some(state3), event4).value
  event5 <- Turtle.walk(1)(state4)
  state5 = Turtle.handler(Some(state4), event5).value
  event6 <- Turtle.walk(1)(state5)
} yield Seq(event1, event2, event3, event4, event5, event6)
```

An imperative approach problems:: accumulating events

```
// Accumulating events - so error-prone!
for {
  event1 <- Turtle.create("123", zero, North)
  state1 = Turtle.handler(None, event1).value
  event2 <- Turtle.walk(1)(state1)
  state2 = Turtle.handler(Some(state1), event2).value
  event3 <- Turtle.walk(1)(state2)
  state3 = Turtle.handler(Some(state2), event3).value
  event4 <- Turtle.walk(1)(state3)
  state4 = Turtle.handler(Some(state3), event4).value
  event5 <- Turtle.walk(1)(state4)
  state5 = Turtle.handler(Some(state4), event5).value
  event6 <- Turtle.walk(1)(state5)
} yield Seq(event1, event2, event3, event4, event5, event6)
```

An imperative approach problems:: propagating events and state

```
// Propagating events and state - repetitive and so error-prone
for {
  event1 <- Turtle.create("123", zero, North)
  state1 = Turtle.handler(None, event1).value
  event2 <- Turtle.walk(1)(state1)
  state2 = Turtle.handler(Some(state1), event2).value
  event3 <- Turtle.walk(1)(state2)
  state3 = Turtle.handler(Some(state2), event3).value
  event4 <- Turtle.walk(1)(state3)
  state4 = Turtle.handler(Some(state3), event4).value
  event5 <- Turtle.walk(1)(state4)
  state5 = Turtle.handler(Some(state4), event5).value
  event6 <- Turtle.walk(1)(state5)
} yield Seq(event1, event2, event3, event4, event5, event6)
```

03 //
A functional approach

Quick recap - Problems left

Problems we need to solve yet when composing commands:

- replaying intermediate events
- accumulating new events
- propagating new state

03.1 //

Replaying events automatically

Replaying events manually - recap

```scala
def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state1 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state1)
} yield Seq(event1, event2)

for {
  event1 <- Turtle.create("123", Position.zero, North)
  state1 = Turtle.handler(None, event1).value
  events2 <- walkRight(1)(state1)
  state2 = events.foldLeft(Some(state1))(Turtle.handler).value
  events3 <- walkRight(1)(state2)
} yield event1 +: events2 ++ events2
```

Replaying events manually - recap

```
def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state1 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state1)
} yield Seq(event1, event2)

for {
  event1 <- Turtle.create("123", Position.zero, North)
  state1 = Turtle.handler(None, event1).value
  events2 <- walkRight(1)(state1)
  state2 = events.foldLeft(Some(state1))(Turtle.handler).value
  events3 <- walkRight(1)(state2)
} yield event1 +: events2 ++ events2
```

What could we do to automate this?

Replaying events automatically with helpers

```
// Let’s use helpers to compute the new state along with every new event

def sourceNew(block: Either[String, TurtleEvent]) =
  block.map { event =>
    event -> Turtle.handler(None, event).value
  }

def source(block: Turtle => Either[String, TurtleEvent]) = (state: Turtle) =>
  block(state).map { event =>
    event -> Turtle.handler(Some(state), event).value
  }
```

Replaying events automatically with helpers

```
// Let's use helpers to compute the new state along with every new event

def sourceNew(block: Either[String, TurtleEvent]) =
  block.map { event =>
    event -> Turtle.handler(None, event).value
  }

def source(block: Turtle => Either[String, TurtleEvent]) = (state: Turtle) =>
  block(state).map { event =>
    event -> Turtle.handler(Some(state), event).value
  }
```

Replaying events automatically with helpers - types

```
// These helpers only "lift" creation and update functions

def sourceNew: Either[String, TurtleEvent] =>
                 Either[String, (TurtleEvent, Turtle)]


def source: (Turtle => Either[String, TurtleEvent]) =>
              (Turtle => Either[String, (TurtleEvent, Turtle)])
```

Replaying events automatically with helpers - types

```
// These helpers only "lift" creation and update functions

def sourceNew: Either[String, TurtleEvent] =>
                 Either[String, (TurtleEvent, Turtle)]



def source: (Turtle => Either[String, TurtleEvent]) =>
              (Turtle => Either[String, (TurtleEvent, Turtle)])
```

Replaying events automatically with helpers - comparison

```scala
// Before: manually replaying state

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state1 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state1)
} yield Seq(event1, event2)
```

Replaying events automatically with helpers - comparison

```
// Before: manually replaying state

def walkRight(dist: Int)(state: Turtle) = for {
  event1 <- Turtle.walk(dist)(state)
  state1 = Turtle.handler(Some(state), event1).value
  event2 <- Turtle.turn(ToRight)(state1)
} yield Seq(event1, event2)

// After: automatically replaying state

def walkRight(dist: Int)(state: Turtle) = for {
  (event1, state1) <- source(Turtle.walk(dist))(state)
  (event2, state2) <- source(Turtle.turn(ToRight))(state1)
} yield (Seq(event1, event2), state2)
```

Replaying events automatically with helpers - demo

```
// Our example rewritten using the helper functions

def walkRight(dist: Int)(state: Turtle) = for {
  (event1, state1) <- source(Turtle.walk(dist))(state)
  (event2, state2) <- source(Turtle.turn(ToRight))(state1)
} yield (Seq(event1, event2), state2)

for {
  (event1, state1) <- sourceNew(Turtle.create("123", Position.zero, North))
  (events2, state2) <- walkRight(1)(state1)
  (events3, state3) <- walkRight(1)(state2)
} yield (event1 +: events2 ++ events2, state3)
```

Problems left

Problems we need to solve yet when composing commands:

- ~~replaying previous events~~
- accumulating new events
- propagating new state

```scala
// We still need to emit events in the right order at the end

for {
  (event1, state1) <- sourceNew(Turtle.create("123", Position.zero, North))
  (events2, state2) <- walkRight(1)(state1)
  (events3, state3) <- walkRight(1)(state2)
} yield (event1 +: events2 ++ events2, state3)

// What if we could accumulate them at each step of the for-comprehension?
```

03.2 //
Accumulating events automatically

Sourced class

```
// Remember our helper?

def sourceNew: Either[String, TurtleEvent] =>
                 Either[String, (TurtleEvent, Turtle)]
```

Sourced class

```scala
// Remember our helper?

def sourceNew: Either[String, TurtleEvent] =>
                 Either[String, (TurtleEvent, Turtle)]

// We wrap our result into a case class, so that we try to write a flatMap

case class Sourced[STATE, EVENT](run: Either[String, (Seq[EVENT], STATE)])

// We use a Seq as we will be accumulating events
```

Sourced class

```scala
case class Sourced[STATE, EVENT](run: Either[String, (Seq[EVENT], STATE)] {

  def events: Either[String, Seq[EVENT]] = run.map { case (events, _) => events }






}
```

Sourced class

```scala
case class Sourced[STATE, EVENT](run: Either[String, (Seq[EVENT], STATE)] {

  def events: Either[String, STATE] = run.map { case (events, _) => events }

  def flatMap[B](fn: STATE => Sourced[B, EVENT]): Sourced[B, EVENT] =
    Sourced[B, EVENT](
      for {
        (currentEvents, currentState) <- this.run
        (newEvents,     newState    ) <- fn(currentState).run
      } yield (currentEvents ++ newEvents, newState)
    )

}
```

Sourced class

```scala
object Sourced {

  def pure[STATE, EVENT](state: STATE): Sourced[STATE, EVENT] =
    Sourced[STATE, EVENT](Right(Nil -> state))

}
```

Writer monad

```
Sourced[STATE, EVENT]

// is equivalent to

WriterT[Either[String, ?], Seq[EVENT], STATE]
```

Sourced monad

```
// We can update our helpers. They really feel like “lifting” now.

def sourceNew: Either[String, TurtleEvent] =>
                 Sourced[Turtle, TurtleEvent]

def source: (Turtle => Either[String, TurtleEvent]) =>
              (Turtle => Sourced[Turtle, TurtleEvent])
```

Sourced monad

```scala
// Event sourcing with the Sourced monad

def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- source(Turtle.walk(dist))(state)
  state2 <- source(Turtle.turn(ToRight))(state1)
} yield state2
```

Sourced monad

```scala
// Event sourcing with the Sourced monad

def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- source(Turtle.walk(dist))(state)
  state2 <- source(Turtle.turn(ToRight))(state1)
} yield state2

// Without event sourcing

def walkRight(dist: Int)(state: Turtle) = for {
  state1 <- Turtle.walk(dist)(state)
  state2 <- Turtle.turn(ToRight)(state1)
} yield state2
```

Sourced monad

```scala
(for {
  state1 <- sourceNew(Turtle.create("123", Position.zero, North))
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
} yield state3).events
```

Problems left

Problems we need to solve yet when composing commands:

- ~~replaying previous events~~
- **~~accumulating new events~~**
- propagating new state

Problems left

Problems we need to solve yet when composing commands:

- ~~replaying previous events~~
- ~~accumulating new events~~
- **propagating new state**

Sourced model demo

```scala
// Using for-comprehension
for {
  state1 <- sourceNew[Turtle](Turtle.create("123", Position.zero, North))
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
} yield state3
```

Sourced model demo

```scala
// Using for-comprehension
for {
  state1 <- sourceNew[Turtle](Turtle.create("123", Position.zero, North))
  state2 <- walkRight(1)(state1)
  state3 <- walkRight(1)(state2)
} yield state3

// Using flatMap
sourceNew[Turtle](Turtle.create("123", Position.zero, North))
  .flatMap(walkRight(1))
  .flatMap(walkRight(1))
```

Problems left

```
Problems we need to solve yet when composing commands:

 -  ~~replaying previous events~~
 -  ~~accumulating new events~~
 -  ~~propagating state~~
```

Spoiler:

There are even better ways to do it

(Kleisli anybody?)

04 //

Further possibilities

Updating multiple instances

Updating multiple aggregates

```
def together(turtle1: Turtle, turtle2: Turtle)
    (update: Turtle => Sourced[Turtle, TurtleEvent])
    : Sourced[(Turtle, Turtle), TurtleEvent] =
  for {
    updated1 <- update(turtle1)
    updated2 <- update(turtle2)
  } yield (updated1, updated2)
```

Updating multiple aggregates

```
def together(turtle1: Turtle, turtle2: Turtle)
    (update: Turtle => Sourced[Turtle, TurtleEvent])
    : Sourced[(Turtle, Turtle), TurtleEvent] =
  for {
    updated1 <- update(turtle1)
    updated2 <- update(turtle2)
  } yield (updated1, updated2)

// Caveat: consistency vs scalability - atomic persistence of events is only
possible within a single shard/partition of the underlying store
```

Handling concurrency

Concurrency

```
// So now we can write declarative programs which reify all the changes we want
// to make to some state.

def myProgram(turtle: Turtle): Sourced[Turtle] = (
  Sourced.pure(turtle)
    .flatMap(walkRight(1))
    .flatMap(walkRight(1))
    .flatMap(walk(2))
)
```

Concurrency

```
// So now we can write declarative programs which reify all the changes we want
// to make to some state.

def myProgram(turtle: Turtle): Sourced[Turtle] = (
  Sourced.pure(turtle)
    .flatMap(walkRight(1))
    .flatMap(walkRight(1))
    .flatMap(walk(2))
)

// It's easy to introduce optimistic locking on top of it
// and achieve something similar to STM
```

Summing up

What we’ve seen today

- Modeling and using events and handlers
- The limitation of an imperative approach
- How a functional approach can help us overcome these limitations
- Event sourcing can become an implementation detail

Gimme some code

Code examples are available at:

https://github.com/eleaar/esmonad

Merci.

Daniel KRZYWICKI

daniel.krzywicki@fabernovel.com

@eleaar