COMPOSABLE EVENT SOURCING WITH MONADS

Daniel Krzywicki @eleaar
Étienne Vallette d'Osia @dohzya

Scala.io 2017-10-03

https://github.com/dohzya/scalaio-2017-esmonad

// FABERNOVEL TECHNOLOGIES

Applidium + Zengularity.

=

// FABERNOVEL
TECHNOLOGIES

The following is based on a true story

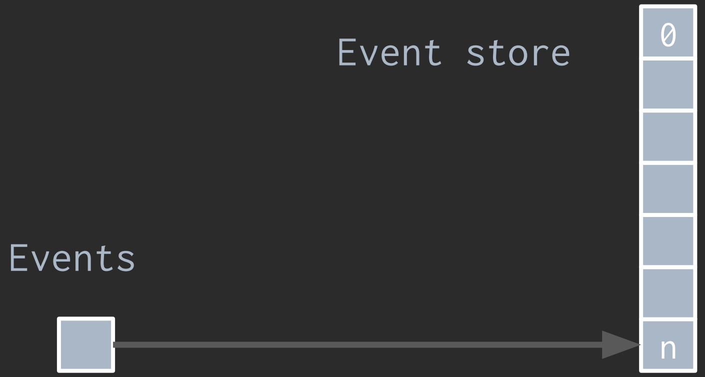
Outline

Introduction to Event Sourcing

Instead of storing state,

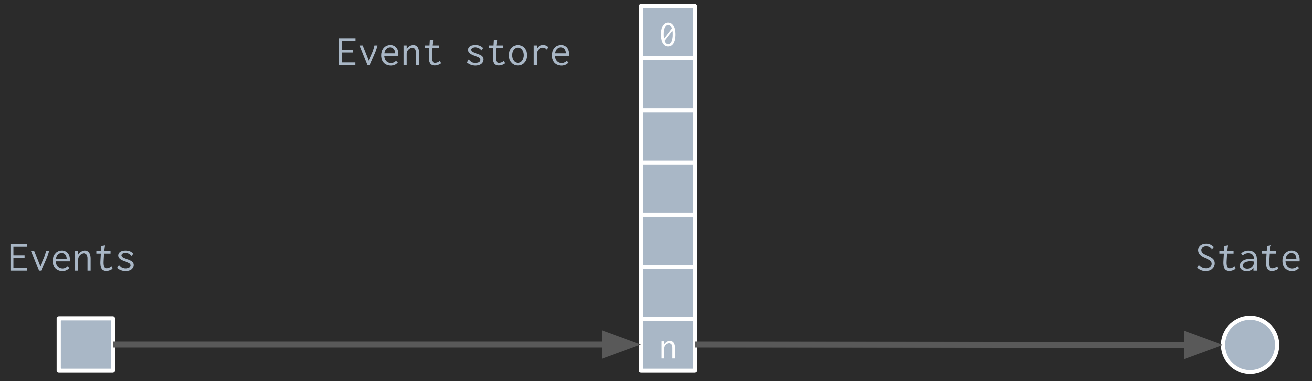
store changes to the state

Introduction :: event store



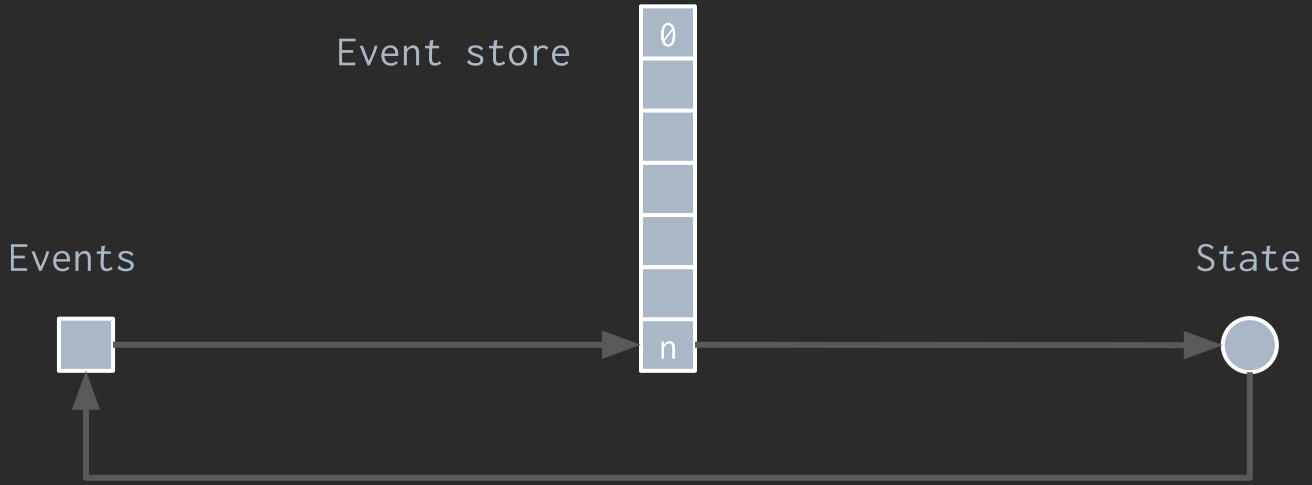
Changes to the system state are reified as events and appended to an event store.

Introduction :: replaying state



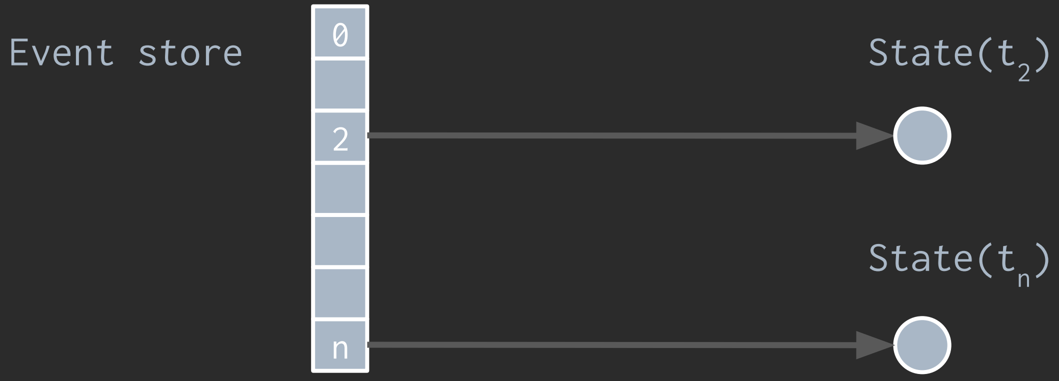
The system state is said to be projected/replayed from the store using event handlers

Introduction :: computing new events



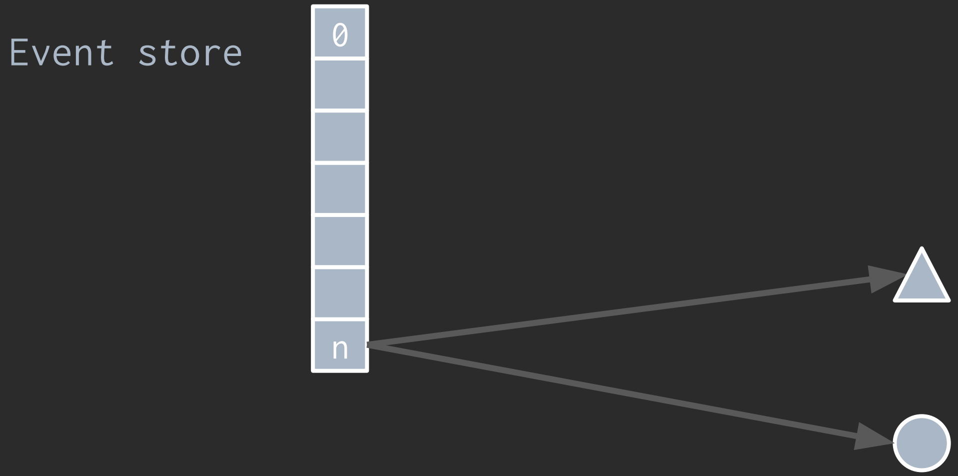
The state can be used to compute new events,
in response to either external signals or internal logic

Introduction :: partial replays



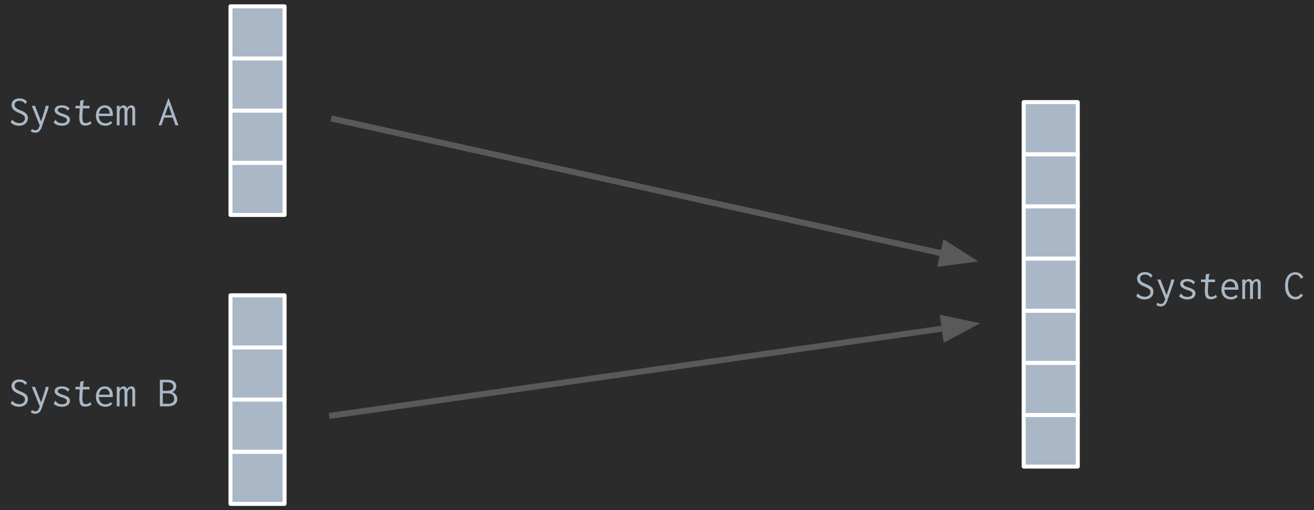
We can easily replay only a part of the events to know the state of the system at any point in time

Introduction :: multiple handlers



We can also use different handlers to interpret the events

Introduction :: distributed integration



By projecting events between distributed systems, we can easily have an architecture which is reactive, fault-tolerant, and scalable.

What the talk is not about

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

What the talk is about

```
Track theme: Type & Functional Programming

Functional  => Focus on composability
Programming => Focus on model and dev API
```

01 //

Minimal Model for Event Sourcing

01.1 //

Modeling the domain - Turtles!

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Basic model

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

case class Position(x: Int, y: Int) {
  val zero = Position(0, 0)
  def move(dir: Direction, distance: Int): Position = { … }
}

sealed trait Direction {
  def rotate(rot: Rotation): Direction = { … }
}
case object North extends Direction ; case object South extends Direction
case object East extends Direction ; case object West extends Direction

sealed trait Rotation
case object ToLeft extends Rotation ; case object ToRight extends Rotation
```

Domain logic

```scala
object Turtle {

  def create(id: String, pos: Position, dir: Direction): Either[String, Turtle] =
    if (tooFarAwayFromOrigin(pos)) Either.left("Too far away")
    else Either.right(Turtle(id, pos, dir))

  def turn(rot: Rotation)(turtle: Turtle): Either[String, Turtle] =
    Right(turtle.copy(dir = turtle.dir.rotate(rot)))

  def walk(dist: Int)(turtle: Turtle): Either[String, Turtle] = {
    val newPos = turtle.pos.move(turtle.dir, dist)
    if (tooFarAwayFromOrigin(newPos)) Either.left("Too far away")
    else Either.right(turtle.copy(pos = newPos))
  }

}
```

Domain logic

```scala
object Turtle {
  // commands have validation, so they can return an error
  def create(id: String, pos: Position, dir: Direction): Either[String, Turtle] =
    if (tooFarAwayFromOrigin(pos)) Either.left("Too far away")
    else Either.right(Turtle(id, pos, dir))

  def turn(rot: Rotation)(turtle: Turtle): Either[String, Turtle] =
    Right(turtle.copy(dir = turtle.dir.rotate(rot)))

  def walk(dist: Int)(turtle: Turtle): Either[String, Turtle] = {
    val newPos = turtle.pos.move(turtle.dir, dist)
    if (tooFarAwayFromOrigin(newPos)) Either.left("Too far away")
    else Either.right(turtle.copy(pos = newPos))
  }

}
```

Domain logic

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

  def turn(rot: Rotation)(turtle: Turtle): Either[String, Turtle] =
    Right(turtle.copy(dir = turtle.dir.rotate(rot)))

  def walk(dist: Int)(turtle: Turtle): Either[String, Turtle] = {
    val newPos = turtle.pos.move(turtle.dir, dist)
    if (tooFarAwayFromOrigin(newPos)) Either.left("Too far away")
    else Either.right(turtle.copy(pos = newPos))
  }

}
```

Domain logic

```scala
object Turtle {

  def create(id: String, pos: Position, dir: Direction): Either[String, Turtle] =
    if (tooFarAwayFromOrigin(pos)) Either.left("Too far away")
    else Either.right(Turtle(id, pos, dir))
  // curried command are like already-configured command
  def turn(rot: Rotation)(turtle: Turtle): Either[String, Turtle] =
    Right(turtle.copy(dir = turtle.dir.rotate(rot)))

  def walk(dist: Int)(turtle: Turtle): Either[String, Turtle] = {
    val newPos = turtle.pos.move(turtle.dir, dist)
    if (tooFarAwayFromOrigin(newPos)) Either.left("Too far away")
    else Either.right(turtle.copy(pos = newPos))
  }

}
```

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
// let's focus on this code

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
// (it’s not perfect, but it works for now)
val state =
  Turtle.create("123", Position.zero, North)
    .flatMap(walkRight(1))
    .flatMap(walkRight(1))
    .flatMap(walkRight(2))
    .flatMap(walkRight(2))
```

We have a model now
How can we event source it?

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
```

Modeling events

```scala
// We can represent the result of our commands as events
sealed trait TurtleEvent { def id: String }
// we store the turtle’s id directly in the events
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

// it’s usually done by wrapping events
```

Event handler for creation events

```scala
// an event handler is a function allowing to folf a sequence of events
type EventHandler0[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler0[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 (state, event) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```scala
// we accept an option to handle creation event (where there is no state yet)
type EventHandler0[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler0[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 (state, event) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```scala
// we do know we have a state to return, so let’s return a Some directly
type EventHandler0[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]

val handler1: EventHandler0[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 (state, event) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Event handler for creation events

```scala
type EventHandler0[STATE, EVENT] = (Option[STATE], EVENT) => Some[STATE]
// this handler throws an exception in case of invalid transition
val handler1: EventHandler0[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 (state, event) =>
    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),
)
// note that we use Some.value instead of Option.get
val finalState = events.foldLeft(initialState)(handler0).value
finalState shouldBe Turtle("123", Position(0, 1), Est)
```

Syntactic sugar for handler definition

```scala
// However, there is more boilerplate when defining the handler
val handler0: EventHandler0[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 (state, event) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Syntactic sugar for handler definition

```
// However, there is more boilerplate when defining the handler
val handler0: EventHandler0[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 (state, event) =>
    sys.error(s"Invalid event $event for state $state")
}
```

Syntactic sugar for handler definition

```scala
// Let's reduce boilerplate by creating an event handler from a partial function

case class EventHandler[STATE, EVENT](
  fn: PartialFunction[(Option[STATE], EVENT), STATE]
) {
  def apply(state: Option[STATE], event: EVENT): Some[STATE] = {
    val input = (state, event)
    if (fn.isDefinedAt(input)) Some(fn(input))
    else sys.error(s"Invalid event $event for state $state")
  }
}
```

Syntactic sugar for handler definition

```scala
// 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
// The commands now return events
object Turtle {
  def create(id: String, pos: Position, dir: Direction) =
    if (tooFarAwayFromOrigin(pos)) Left("Too far away")
    else Right(Created(id, pos, dir))

  def turn(rot: Rotation)(turtle: Turtle): Either[String, TurtleEvent] =
    Right(Turned(turtle.id, rot))

  def walk(dist: Int)(turtle: Turtle): Either[String, TurtleEvent] = {
    val newPos = turtle.pos.move(turtle.dir, dist)
    if (tooFarAwayFromOrigin(newPos)) Left("Too far away")
    else Right(Walked(turtle.id, dist))
  }
}
```

01.3 //

Persisting and replaying events

Event Journal

```scala
// journal to write events
trait WriteJournal[EVENT] {
  def persist(event: EVENT): Future[Unit]
}
def persist[EVENT: WriteJournal](events:EVENT)): Future[Unit]

trait Hydratable[STATE] {
  def hydrate(id: String): Future[Option[STATE]]
}
def hydrate[STATE: Hydratable](id: String): Future[Option[STATE]]


implicit object TurtleJournal
  extends WriteJournal[TurtleEvent] with Hydratable[Turtle] { … }
```

Event Journal

```
trait WriteJournal[EVENT] {
  def persist(events: EVENT): Future[Unit]
}
def persist[EVENT: WriteJournal](events:EVENT)): Future[Unit]
// hydratable fetch the events and use them to build the state
trait Hydratable[STATE] {
  def hydrate(id: String): Future[Option[STATE]]
}
def hydrate[STATE: Hydratable](id: String): Future[Option[STATE]]


implicit object TurtleJournal
  extends WriteJournal[TurtleEvent] with Hydratable[Turtle] { … }
```

Event Journal

```scala
trait WriteJournal[EVENT] {
  def persist(events: EVENT): Future[Unit]
}
def persist[EVENT: WriteJournal](events:EVENT)): Future[Unit]

trait Hydratable[STATE] {
  def hydrate(id: String): Future[Option[STATE]]
}
def hydrate[STATE: Hydratable](id: String): Future[Option[STATE]]

// let’s assume we have instance for those
implicit object TurtleJournal
  extends WriteJournal[TurtleEvent] with Hydratable[Turtle] { … }
```

Wrapping up

does not compile

```
// Simple example which creates and retrieve a turtle using the journal

for {
  event = Turtle.create("123", Position(0, 1), North)
  _ <- persist(event)

  state <- hydrate[Turtle]("123")
} yield state shouldBe Turtle("123", Position(0, 1), North)
```

```scala
// The same example but which actually compiles :-)
// (thanks to ScalaZ/cats's monad transformer)
(for {
  event <- EitherT.fromEither(Turtle.create("123", zero, North))
  _ <- EitherT.right(persist(event))

  state <- OptionT(hydrate[Turtle]("123")).toRight("not found")
} yield state).value.map {
  _ shouldBe Right(Turtle("123", zero, North))
}
```

What we have seen so far

What we have seen so far

- modeling the domain
- defining events and event handlers
- persisting events and replaying state

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

```scala
// 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?

```scala
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 =>
    0
  case (Some(total), Walked(turtleId, dist)) if id == turtleId =>
    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 =>
    0
  case (Some(total), Walked(turtleId, dist)) if id == turtleId =>
    total + dist
  case (Some(total), WalkedRight(turtleId, dist)) if id == turtleId =>
    total + dist
  case (Some(total), MadeUTurn(turtleId, radius)) if id == turtleId =>
    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?
```

Gimme some real-life example, will you?

Use case: appointments app (doctoLib, etc)

Main domain object: **appointments** between users.

Use case: appointments app (doctoLib, etc)

Main domain object: **appointments** between users.

Users with a pending common appointment can send messages to each other:

`postMessage` ---> *MessagedPosted*

Use case: appointments app (doctoLib, etc)

Main domain object: **appointments** between users.

Users with a pending common appointment can send messages to each other:

`postMessage` ---> *MessagedPosted*

New messages update a chat-like view

Use case: appointments app (doctoLib, etc)

New feature: users can now cancel the meeting and provide an optional custom message when doing so.

Use case: appointments app (doctoLib, etc)

New feature: users can now cancel the meeting and provide an optional custom message when doing so.

- We don’t want to deliver the message if the cancellation is not persisted.
- We don’t want the message to be lost

Use case: appointments app (doctoLib, etc)

New feature: users can now cancel the meeting and provide an optional custom message when doing so.

- We don't want to deliver the message if the cancellation is not persisted.
- We don't want the message to be lost
- We don't want (or can't) update the chat handler to handle a new type of event

```
cancelAppointment(Option[Message])  ---> AppointmentCanceled [+ MessagedPosted]
```

Use case: appointments app (doctoLib, etc)

New feature: users can now cancel the meeting and provide an optional custom message when doing so.

- We don’t want to deliver the message if the cancellation is not persisted.
- We don’t want the message to be lost
- We don’t want (or can’t) update the chat handler to handle a new type of event

`cancelAppointment(Option[Message])` ---> *AppointmentCanceled [+ MessagedPosted]*

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

```scala
// 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

```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

```scala
// 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]
}

def persist[EVENT: WriteJournal](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.

Persisting multiple events :: demo

```
for {
  state <- OptionT(hydrate[Turtle]("123")).toRight("not found")
  events <- EitherT.fromEither(Turtle.walkRight(1)(state))
  _ <- EitherT.right(persist(events))
} yield ()
```

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

```scala
// 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 previous 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
```

What could we do

to automate this?

Replaying events automatically with helpers

```scala
// 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

```scala
// 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)

// 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)
```

```
value withFilter is not a member of Either[...]
```

Replaying events automatically with helpers - demo

```scala
// 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

```scala
// 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 }

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

Sourced class

```
// 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 class

```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 class

```
(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

Rings a bell?

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

  def flatMap[B](fn: A => Sourced[B, EVENT]): Sourced[B, EVENT] =
    Sourced[B, EVENT](run.flatMap { case (oldEvents, oldState) =>
      fn(oldState).run.map { case (newEvents, newState) =>
        (oldEvents ++ newEvents, newState)
      }
    })

}
```

Writer monad

```
Sourced[STATE, EVENT]

// is equivalent to

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

// 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 new state (?)**

Writer monad

```
// this code might seem fine (no visible state)...

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

Writer monad

```
// Limitation: no guarantee that we are actually using the event sourced state

sourceNew[Turtle](Turtle.create("123", Position.zero, North))
  .flatMap(walkRight(1))
  .flatMap(walkRight(1))
  .map(state => state.copy(pos = Position(99, 99)))
```

Problems left

Problems we need to solve yet when composing commands:

- ~~replaying previous events~~
- ~~accumulating new events~~
- **propagating state (kind of, but in a very unsafe way)**

Creations and updates are different beasts

Updates are actually monadic and can be composed:

Update THEN Update = Update

Creation can only be composed with later updates:

Create THEN Update = Create

03.4 //

Beyond monads

Propagating state - once again

```
// Let's go back to our original helpers for the Sourced result monad.

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

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

Propagating state - once again

```scala
// Let’s go back to our original helpers for the Sourced result monad.
// What if we lifted creation and updating commands into types with appropriate
// combination operators?

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

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

Reifying commands

```scala
case class SourcedCreation[STATE, EVENT](run: Sourced[STATE, EVENT]) {
  def events: Either[String, Seq[EVENT]] = run.events
  def andThen[A](other: SourcedUpdate[STATE, EVENT, A]) =
    SourcedCreation[A, EVENT] { this.run.flatMap(other.run) }
}

case class SourcedUpdate[STATE, EVENT, A](run: (STATE) => Sourced[A, EVENT]) {
  def events(initialState: STATE): Either[String, Seq[EVENT]] = run(state).events
  def andThen[B](other: SourcedUpdate[A, EVENT, B]) =
    SourcedUpdate[STATE, EVENT, B] { initialState =>
      this.run(initialState).flatMap(other.run)
}
```

Reifying commands

```scala
case class SourcedCreation[STATE, EVENT](run: Sourced[STATE, EVENT]) {
  def events: Either[String, Seq[EVENT]] = run.events
  def andThen[A](other: SourcedUpdate[STATE, EVENT, A]) =
    SourcedCreation[A, EVENT] { this.run.flatMap(other.run) }
}

case class SourcedUpdate[STATE, EVENT, A](run: (STATE) => Sourced[A, EVENT]) {
  def events(initialState: STATE): Either[String, Seq[EVENT]] = run(state).events
  def andThen[B](other: SourcedUpdate[A, EVENT, B]) =
    SourcedUpdate[STATE, EVENT, B] { initialState =>
      this.run(initialState).flatMap(other.run)
}
```

Reifying commands

```scala
case class SourcedCreation[STATE, EVENT](run: Sourced[STATE, EVENT]) {
  def events: Either[String, Seq[EVENT]] = run.events
  def andThen[A](other: SourcedUpdate[STATE, EVENT, A]) =
    SourcedCreation[A, EVENT] { this.run.flatMap(other.run) }
}

case class SourcedUpdate[STATE, EVENT, A](run: (STATE) => Sourced[A, EVENT]) {
  def events(initialState: STATE): Either[String, Seq[EVENT]] = run(state).events
  def andThen[B](other: SourcedUpdate[A, EVENT, B]) =
    SourcedUpdate[STATE, EVENT, B] { initialState =>
      this.run(initialState).flatMap(other.run)
}
```

Reifying commands

```scala
case class SourcedCreation[STATE, EVENT](run: Sourced[STATE, EVENT]) {
  def events: Either[String, Seq[EVENT]] = run.events
  def andThen[A](other: SourcedUpdate[STATE, EVENT, A]) =
    SourcedCreation[A, EVENT] { this.run.flatMap(other.run) }
}

case class SourcedUpdate[STATE, EVENT, A](run: (STATE) => Sourced[A, EVENT]) {
  def events(initialState: STATE): Either[String, Seq[EVENT]] = run(state).events
  def andThen[B](other: SourcedUpdate[A, EVENT, B]) =
    SourcedUpdate[STATE, EVENT, B] { initialState =>
      this.run(initialState).flatMap(other.run)
}
```

Reifying commands

```scala
case class SourcedCreation[STATE, EVENT](run: Sourced[STATE, EVENT]) {
  def events: Either[String, Seq[EVENT]] = run.events
  def andThen[A](other: SourcedUpdate[STATE, EVENT, A]) =
    SourcedCreation[A, EVENT] { this.run.flatMap(other.run) }
}

case class SourcedUpdate[STATE, EVENT, A](run: (STATE) => Sourced[A, EVENT]) {
  def events(initialState: STATE): Either[String, Seq[EVENT]] = run(state).events
  def andThen[B](other: SourcedUpdate[A, EVENT, B]) =
    SourcedUpdate[STATE, EVENT, B] { initialState =>
      this.run(initialState).flatMap(other.run)
}
```

Reifying commands :: demo

```scala
// We no longer need for-comprehension nor flatMaps

def walkRight(dist: Int) = {
  source(Turtle.walk(dist)) andThen
  source(Turtle.turn(ToRight))
}

(
  sourceNew[Turtle](Turtle.create("123", Position.zero, North)) andThen
  walkRight(1) andThen
  walkRight(1) andThen
  source(Turtle.walk(2))
).events
```

Kleisli

```
SourcedUpdate[STATE, EVENT, A]

// is equivalent to

Kleisli[Sourced[?, EVENT], STATE, A]
```

Problems left

Problems we need to solve yet when composing commands:

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

04 //
Further possibilities

Nice syntax

Can we do better?

```scala
// boilerplate, let's make the commands return Sourced instances directly
def walkRight(dist: Int) =
  source(Turtle.walk(dist)) andThen
  source(Turtle.turn(ToRight))

(
  sourceNew[Turtle](Turtle.create("123", Position.zero, North)) andThen
  walkRight(1) andThen
  walkRight(1) andThen
  source(Turtle.walk(2))
).events
```

Can we do better?

```scala
// better, is it the best we can do?
def walkRight(dist: Int) =
  Turtle.walk(dist) andThen
  Turtle.turn(ToRight)

(
  Turtle.create("123", Position.zero, North) andThen
  walkRight(1) andThen
  walkRight(1) andThen
  Turtle.walk(2)
).events
```

Can we do better?

```scala
// there is no more difference between walkRight and other commands
def walkRight(dist: Int) =
  Turtle.walk(dist) andThen
  Turtle.turn(ToRight)

(
  Turtle.create("123", Position.zero, North) andThen
  walkRight(1) andThen
  walkRight(1) andThen
  Turtle.walk(2)
).events
```

Pimped commands

```scala
// this code is not only simpler,
// but it does not make any difference (nor should)
// between original commands and composite ones

(
  Turtle.create("123", Position.zero, North) andThen
  Turtle.walkRight(1) andThen
  Turtle.walkRight(1) andThen
  Turtle.walk(2)
).events
```

More combinators

ReaderWriterState :: bonus

```scala
// let's allow optional calls of a command
def when[STATE] = new whenPartiallyApplied[STATE]
final class whenPartiallyApplied[STATE] {
 def apply[EVENT](
   predicate: (STATE) => Boolean,
   block: STATE => SourceUpdated[STATE, EVENT, STATE]
 )(implicit handler: EventHandler[STATE, EVENT]) =
   SourcedUpdate[STATE, EVENT, STATE](Kleisli { state =>
     if (predicate(state)) block(state).run
     else WriterT[Either[String, ?], Vector[EVENT], STATE] {
       Right(Vector.empty -> state) // no-op
     }
   })
}
```

ReaderWriterState :: bonus

```
// the predicate is called only when evaluating commands
// (not when replaying the emitted events)
(
  Turtle.create("123", zero, North) andThen
  Turtle.walkRight(1) andThen
  Turtle.walkRight(1) andThen
  when[Turtle](_.dir == North, Turtle.walk(1)) andThen
  Turtle.walk(2)
) events
```

What more?

Updating multiple instances

Updating multiple aggregates

```scala
// The weakness of the initial monad is also its strength

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

```scala
// The weakness of the initial monad is also its strength

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

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

val myProgram = (
  TurtleCommands.walkRight(1) andThen
  TurtleCommands.walkRight(1) andThen
  TurtleCommands.walk(2)
)
```

Concurrency

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

val myProgram = (
  TurtleCommands.walkRight(1) andThen
  TurtleCommands.walkRight(1) andThen
  TurtleCommands.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

Merci.

Daniel KRZYWICKI

daniel.krzywicki@fabernovel.com

@eleaar

Étienne VALLETTE d'OSIA

etienne.vallette-dosia@fabernovel.com

@dohzya

Appendum

ReaderWriterState monad

```
case class SourcedUpdate[STATE, EVENT, A](
  run: ReaderWriterStateT[Either[String, ?], Unit, Vector[EVENT], STATE, A]
) {
  def events(initialState: STATE): Either[String, Vector[EVENT]] =
    run.runL((), initialState)

  def state(initialState: STATE): Either[String, Option[STATE]] =
    run.runS((), initialState)

  def map[B](fn: A => B): Sourced[STATE, EVENT, B] =
    SourcedState(run.map(fn))

  def flatMap[B](fn: A => Sourced[STATE, EVENT, B]): Sourced[STATE, EVENT, B] =
    SourcedState(run.flatMap(fn(_).run))

}
```

ReaderWriterState monad

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

val result = (
  sourceNew[Turtle](Turtle.create("123", Position.zero, North)) andThen (
    for {
      _ <- sourceNew[Turtle](Turtle.create("123", Position.zero, North))
      _ <- walkRight(1)
      _ <- walkRight(1)
    } yield ()
  )
).run
```