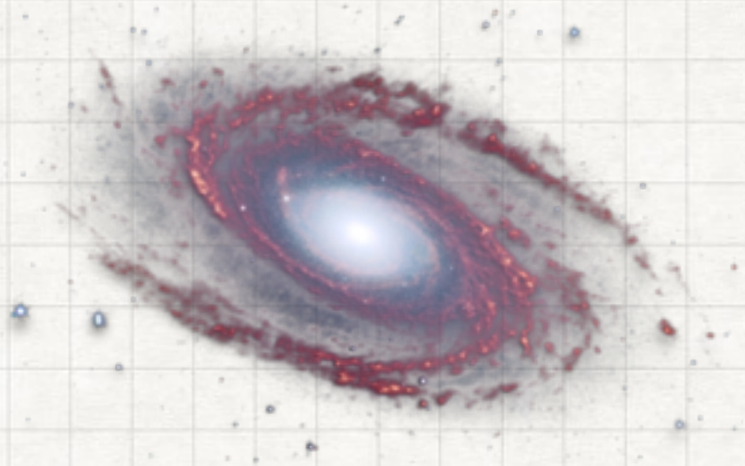
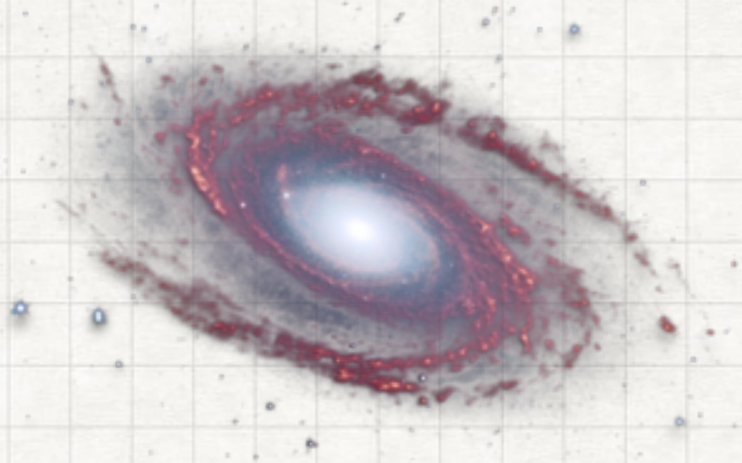
BUILDING STATEFUL SYSTEMS WITH FUNCTIONAL PROGRAMMING

BEN HUTCHISON, V2.0, FEB 2022

IN THE BEGINNING..

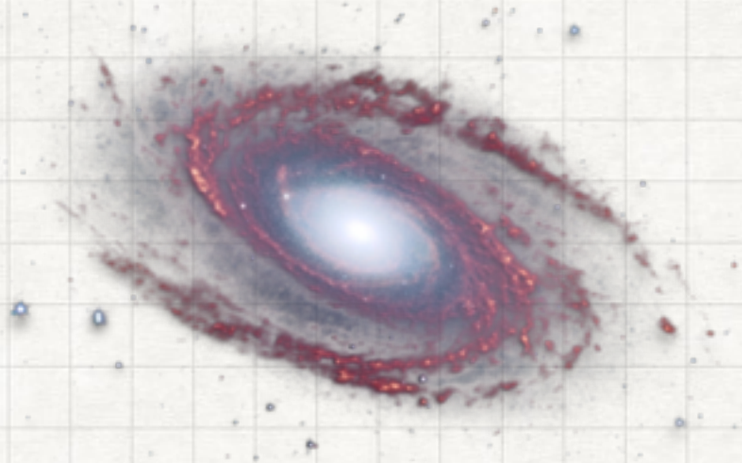


IN THE BEGINNING..

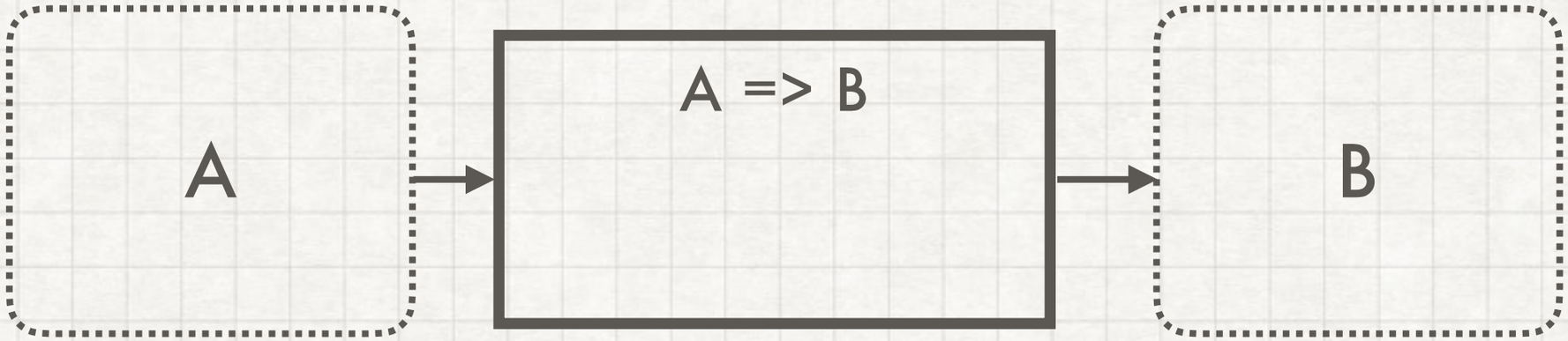


THERE WERE FUNCTIONS

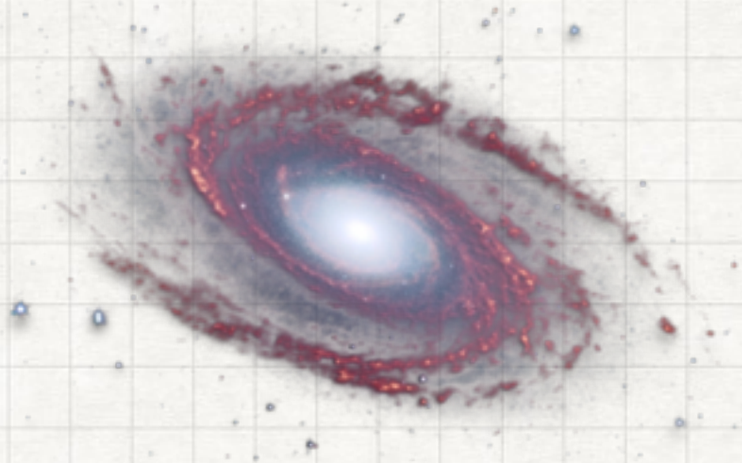
IN THE BEGINNING..



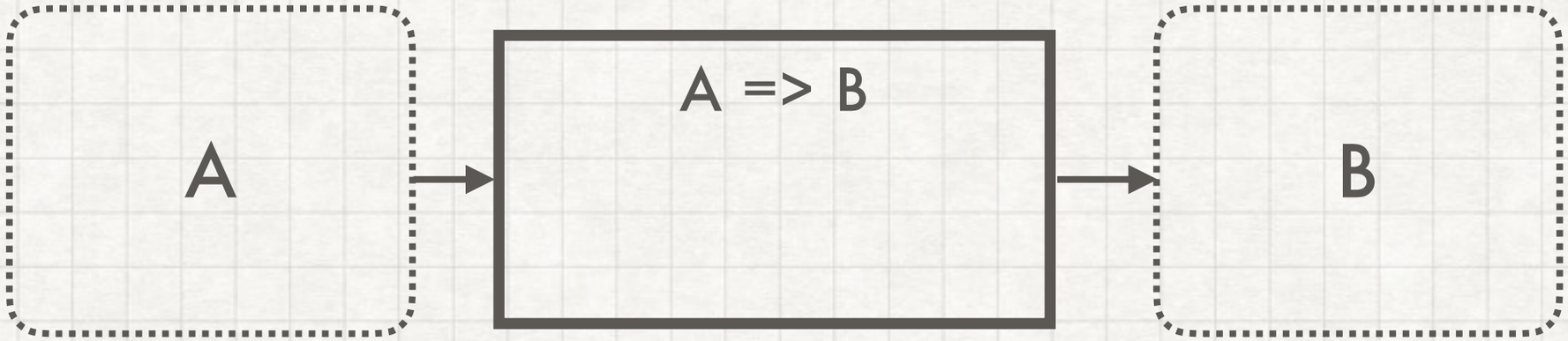
THERE WERE FUNCTIONS



IN THE BEGINNING..

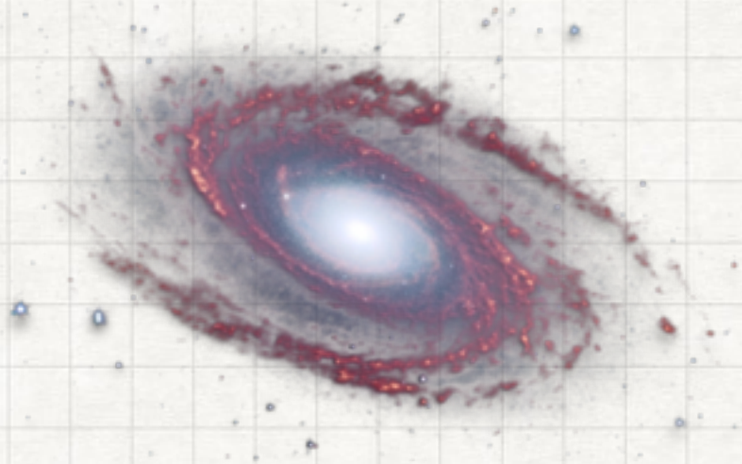


THERE WERE FUNCTIONS

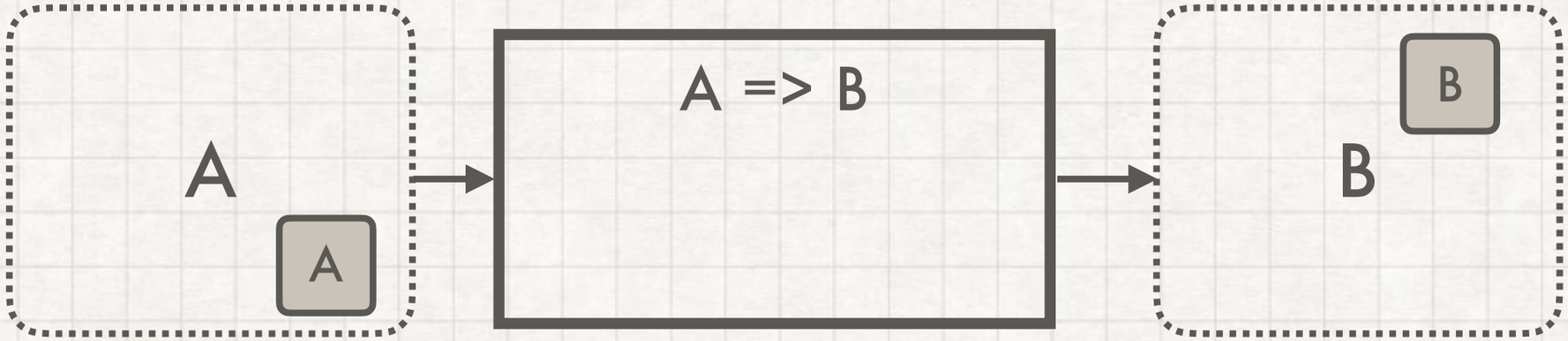


FUNCTIONAL PROGRAMMING = PROGRAMMING WITH FUNCTIONS

IN THE BEGINNING..



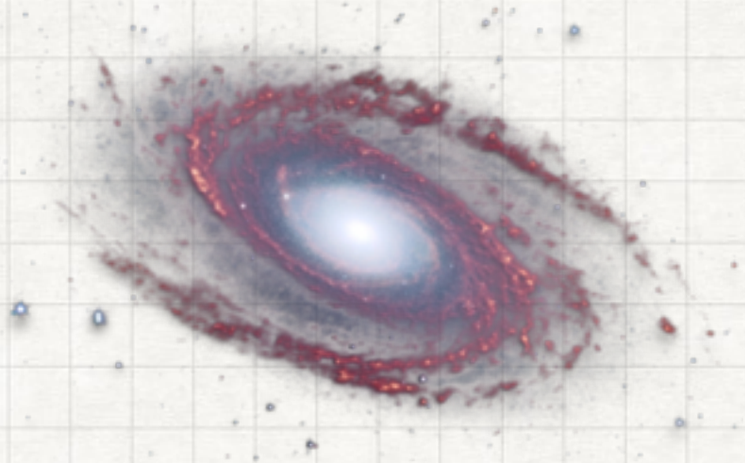
THERE WERE FUNCTIONS



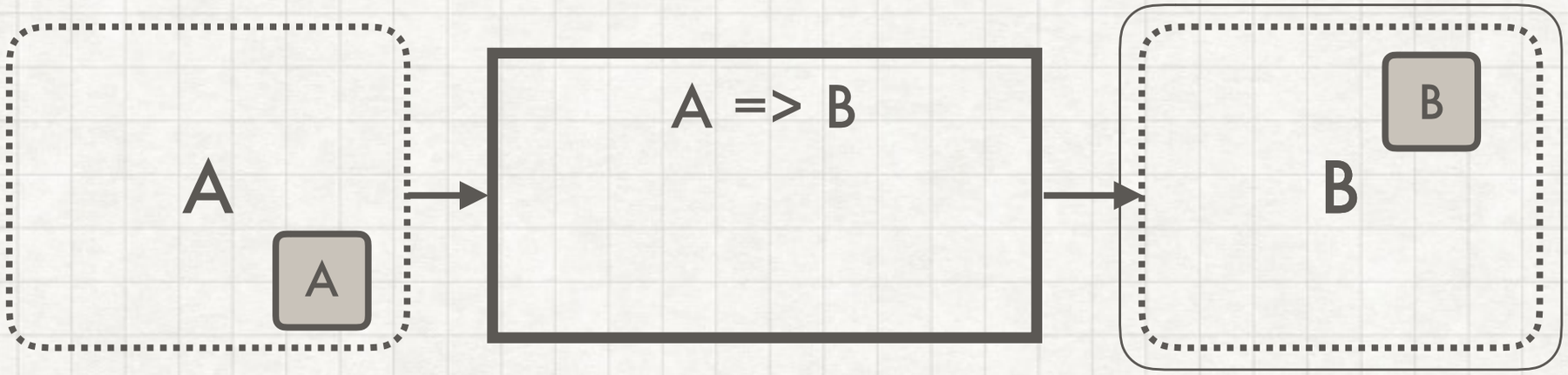
DETERMINISTIC & STATELESS

FUNCTIONAL PROGRAMMING = PROGRAMMING WITH FUNCTIONS

IN THE BEGINNING..



THERE WERE FUNCTIONS

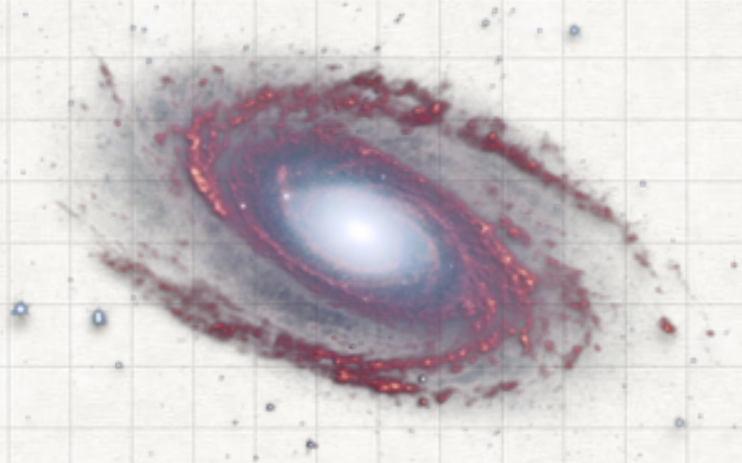


DETERMINISTIC & STATELESS

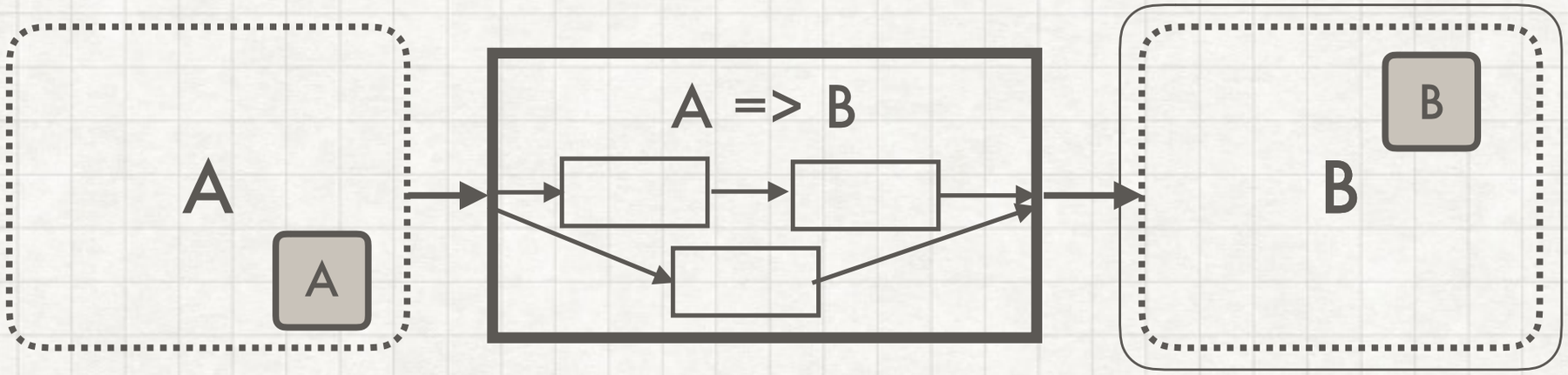
PURE

FUNCTIONAL PROGRAMMING = PROGRAMMING WITH FUNCTIONS

IN THE BEGINNING..



THERE WERE FUNCTIONS



DETERMINISTIC & STATELESS

PURE

COMPOSABLE

FUNCTIONAL PROGRAMMING = PROGRAMMING WITH FUNCTIONS

COMBINING FUNCTIONS WITH STATE

BEN HUTCHISON, V2.0, FEB 2022

STATEFUL SYSTEMS..

STATEFUL SYSTEMS..

REMEMBER THEIR PAST

STATEFUL SYSTEMS..

REMEMBER THEIR PAST

CAN RETURN DIFFERENT VALUES EACH CALL

STATEFUL SYSTEMS..

REMEMBER THEIR PAST

CAN RETURN DIFFERENT VALUES EACH CALL

ESSENTIAL TO MODEL REAL WORLD

STATEFUL SYSTEMS..

REMEMBER THEIR PAST

CAN RETURN DIFFERENT VALUES EACH CALL

ESSENTIAL TO MODEL REAL WORLD

CANNOT BE REPRESENTED BY COMPOSING FUNCTIONS OF ANY COMPLEXITY

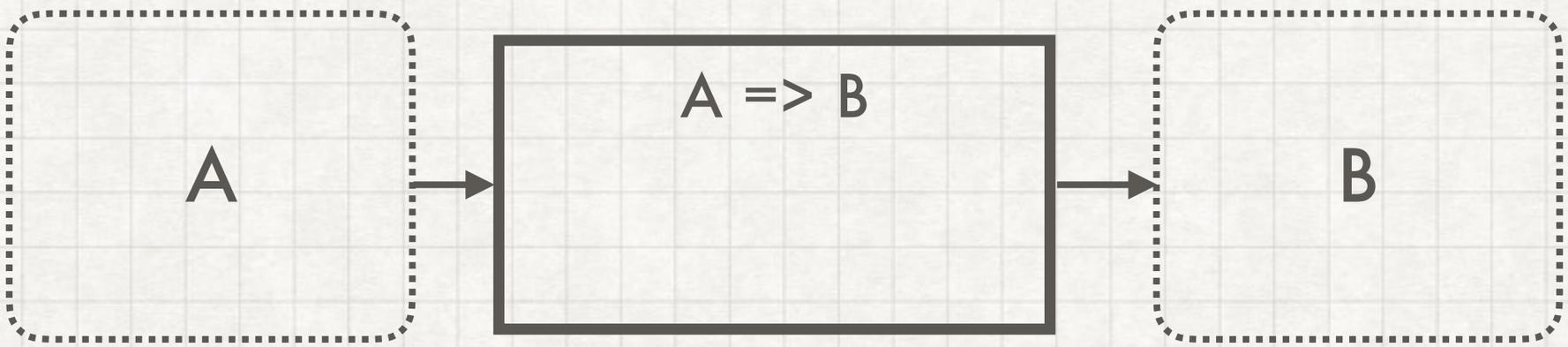
STATEFUL SYSTEMS..

REMEMBER THEIR PAST

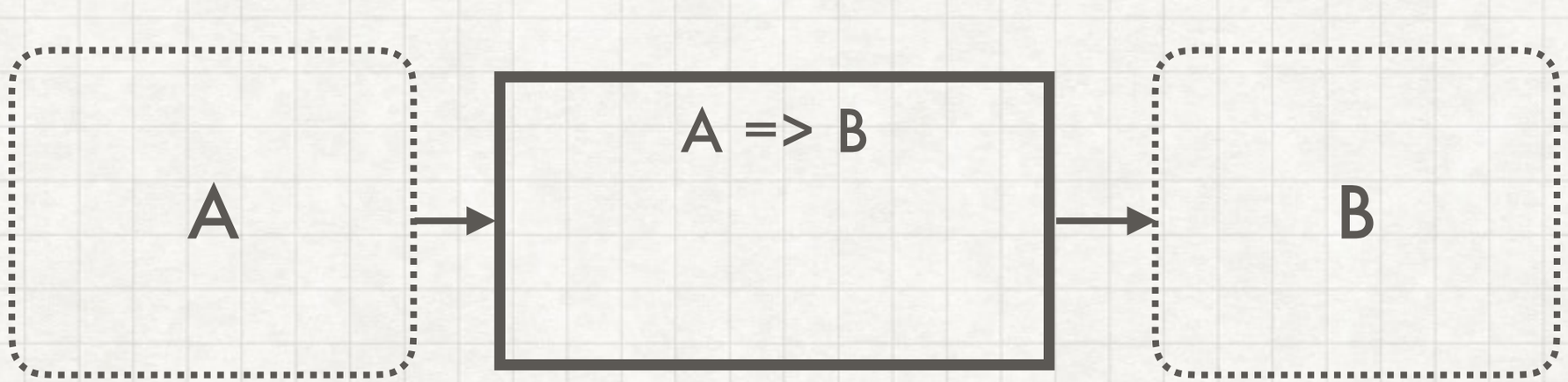
CAN RETURN DIFFERENT VALUES EACH CALL

ESSENTIAL TO MODEL REAL WORLD

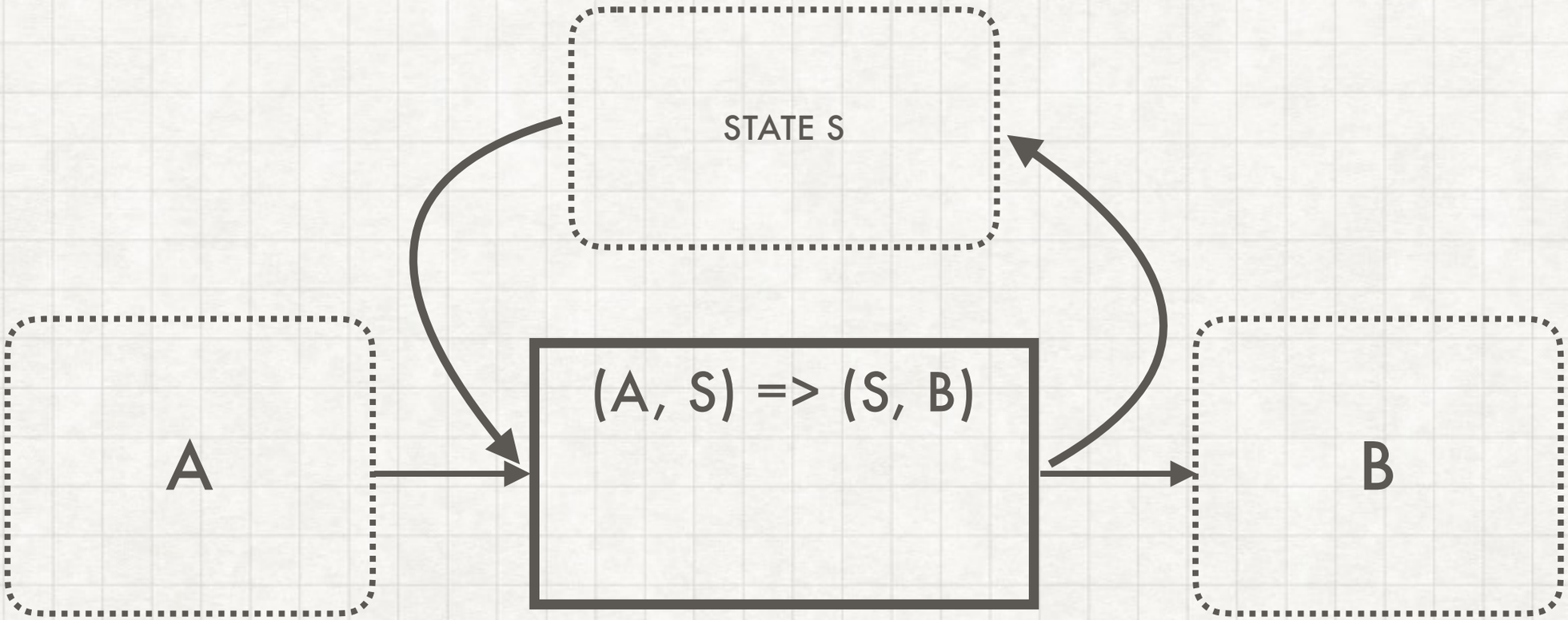
CANNOT BE REPRESENTED BY COMPOSING A FUNCTION OF ANY COMPLEXITY



WHAT DO WE NEED TO ADD TO THIS PICTURE?

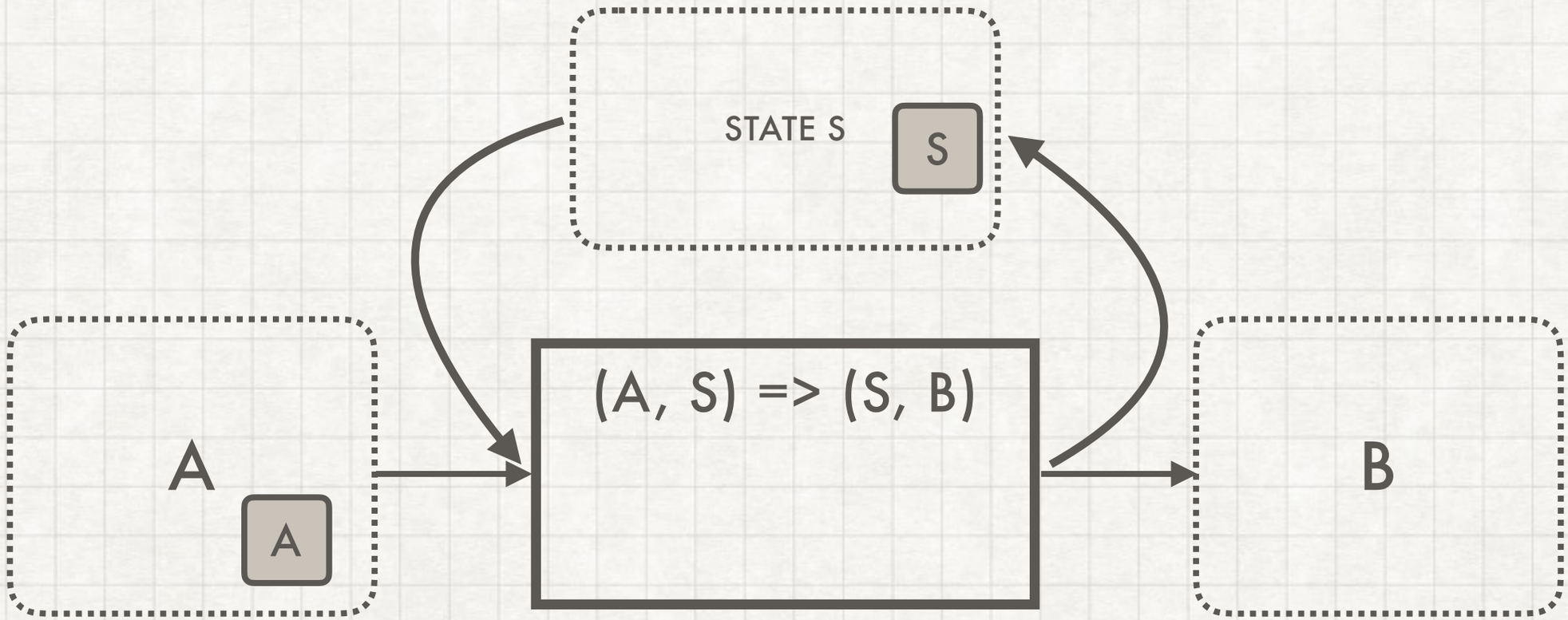

A
A => B
B

PAIR: PURE FUNCTION + STATE STORE



PAIR: PURE FUNCTION + STATE STORE

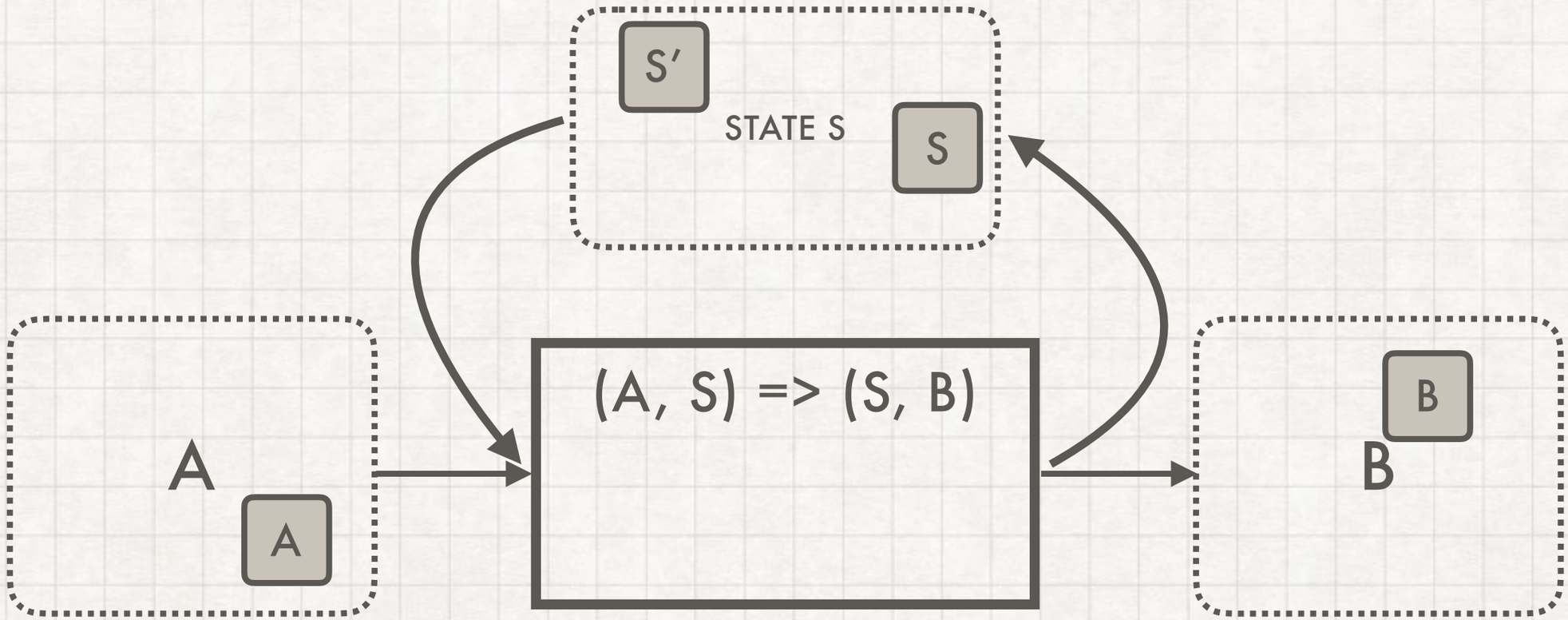
IN: INPUT A + CURRENT STATE S



PAIR: PURE FUNCTION + STATE STORE

IN: INPUT A + CURRENT STATE S

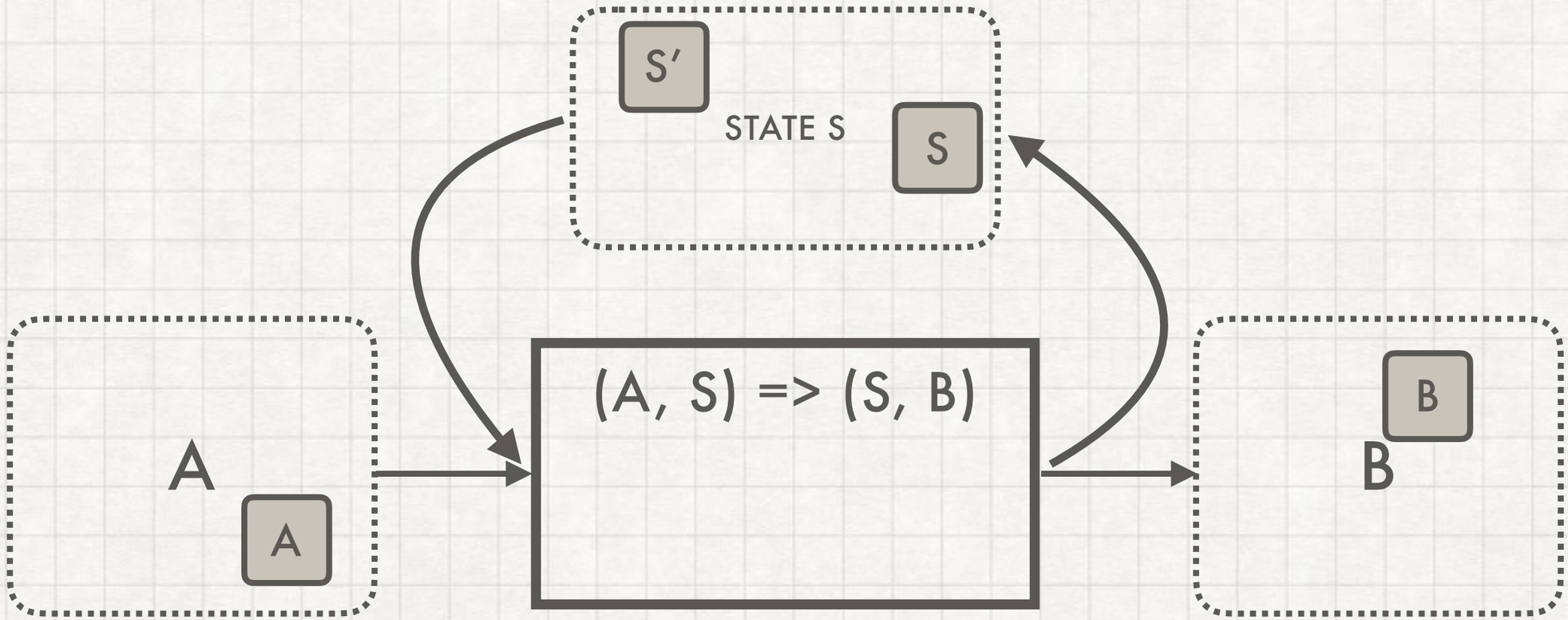
OUT: OUTPUT B + NEXT STATE S′



PAIR: PURE FUNCTION + STATE STORE

IN: INPUT A + CURRENT STATE S

OUT: OUTPUT B + NEXT STATE S′

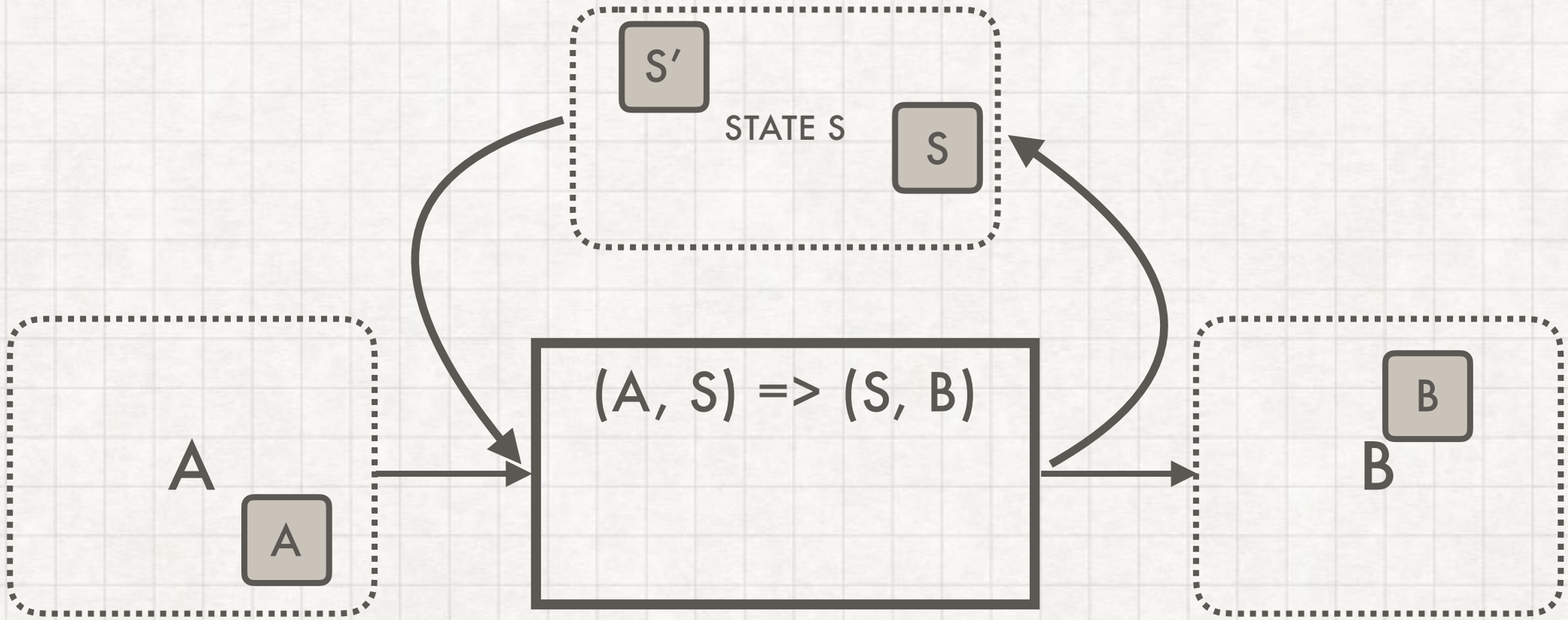


THE KEY IDEA OF THIS TALK

PAIR: PURE FUNCTION + STATE STORE

IN: INPUT A + CURRENT STATE S

OUT: OUTPUT B + NEXT STATE S′



THE KEY IDEA OF THIS TALK

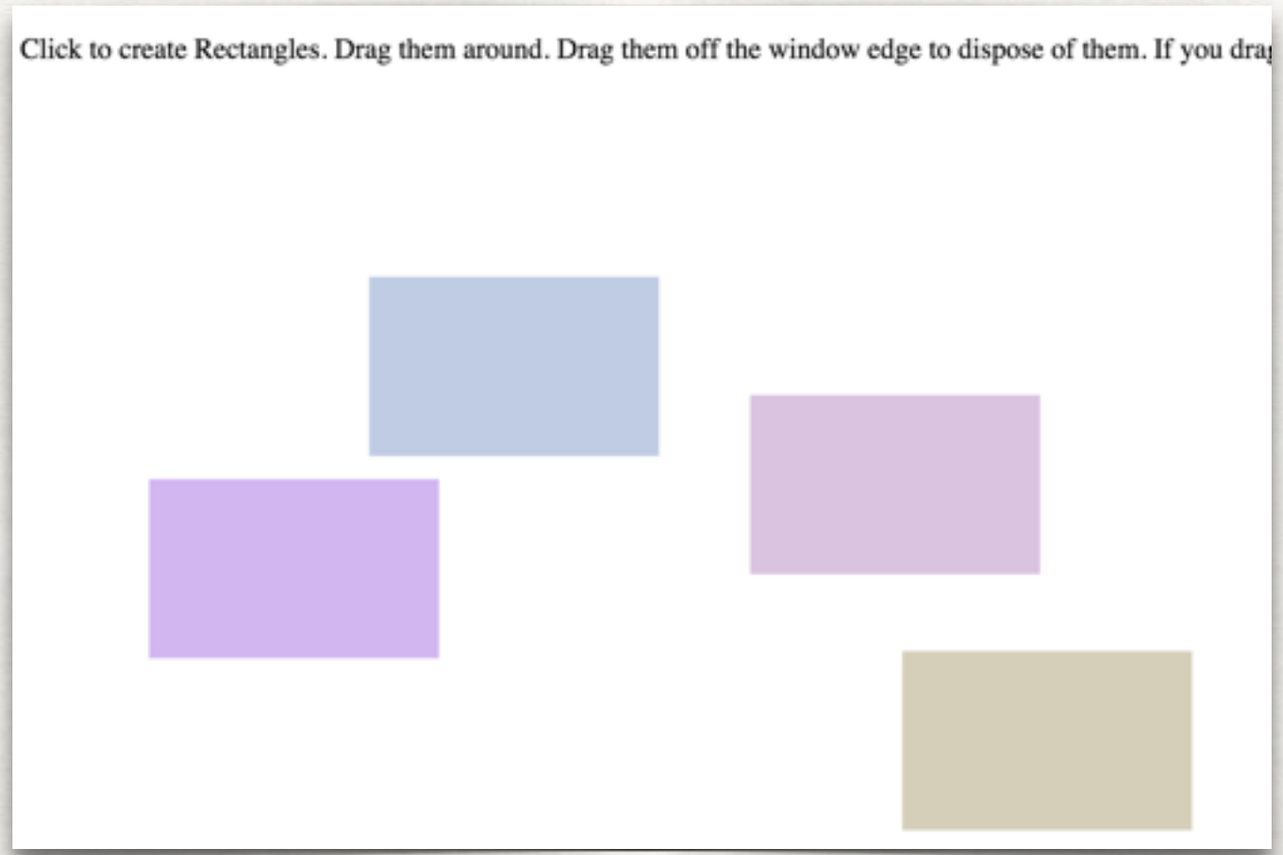
THE REST IS “COLOURING IN” THIS PICTURE

CASE STUDY: POINTER GESTURE RECOGNITION

BEN HUTCHISON, V2.0, FEB 2022

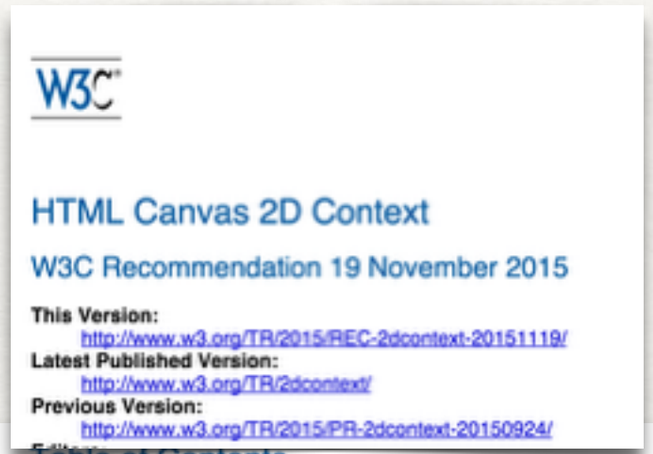
Dragging On An Html Canvas

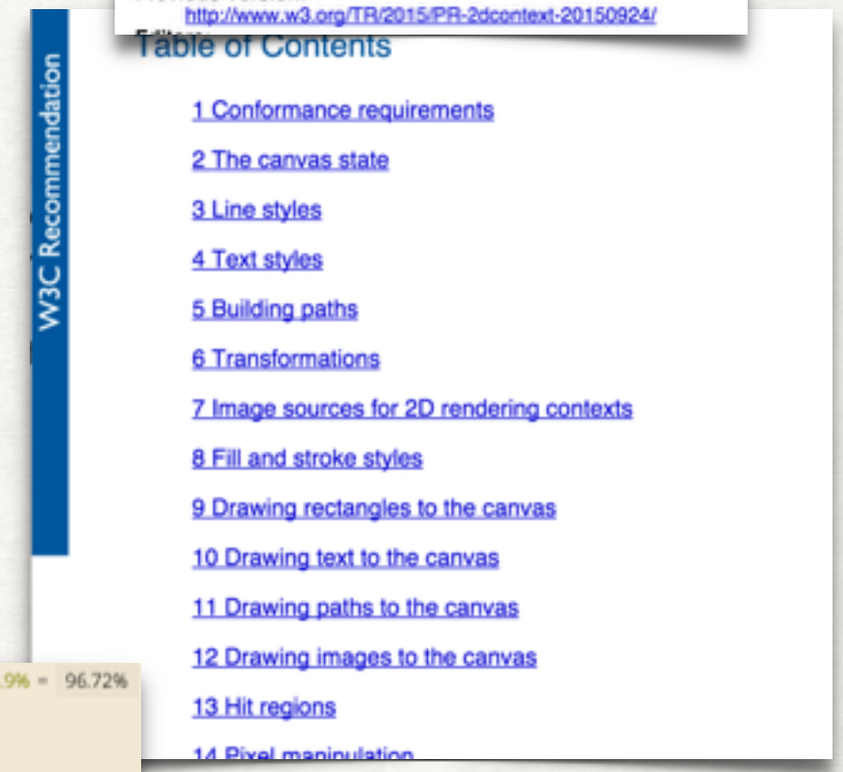
- To support drag and drop type behaviors using Scala.js and the HTML Canvas

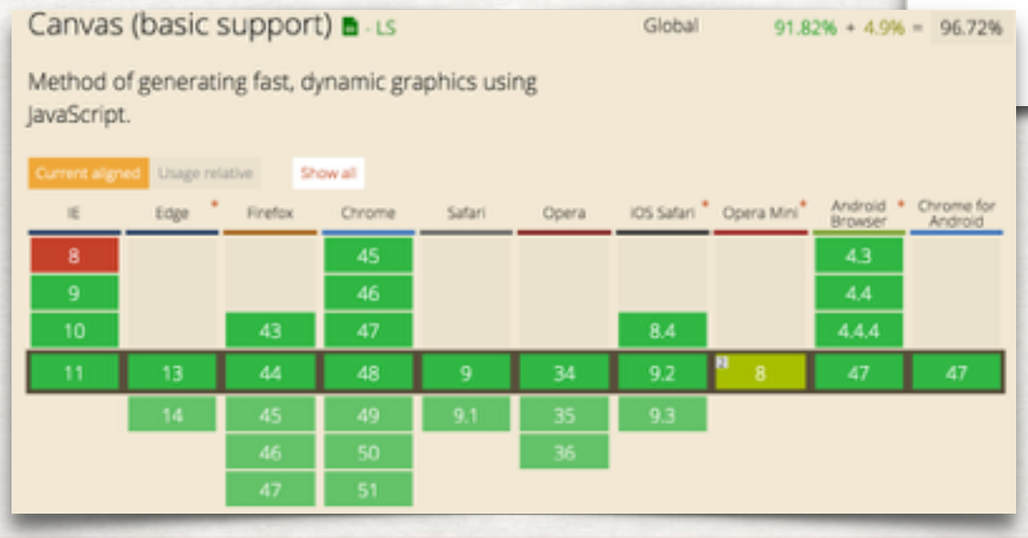


Html Canvases

- Performant, portable and versatile API for drawing vector graphics
  - Its imperative, based on commands e.g.
    - fillRect(x, y, w, h)
    - lineTo(x, y)
    - fillStyle=(cssColor)
    - transform(a, b, c, d, e, f)
- A canvas has no internal structure other than what the programmer creates
- No drag support

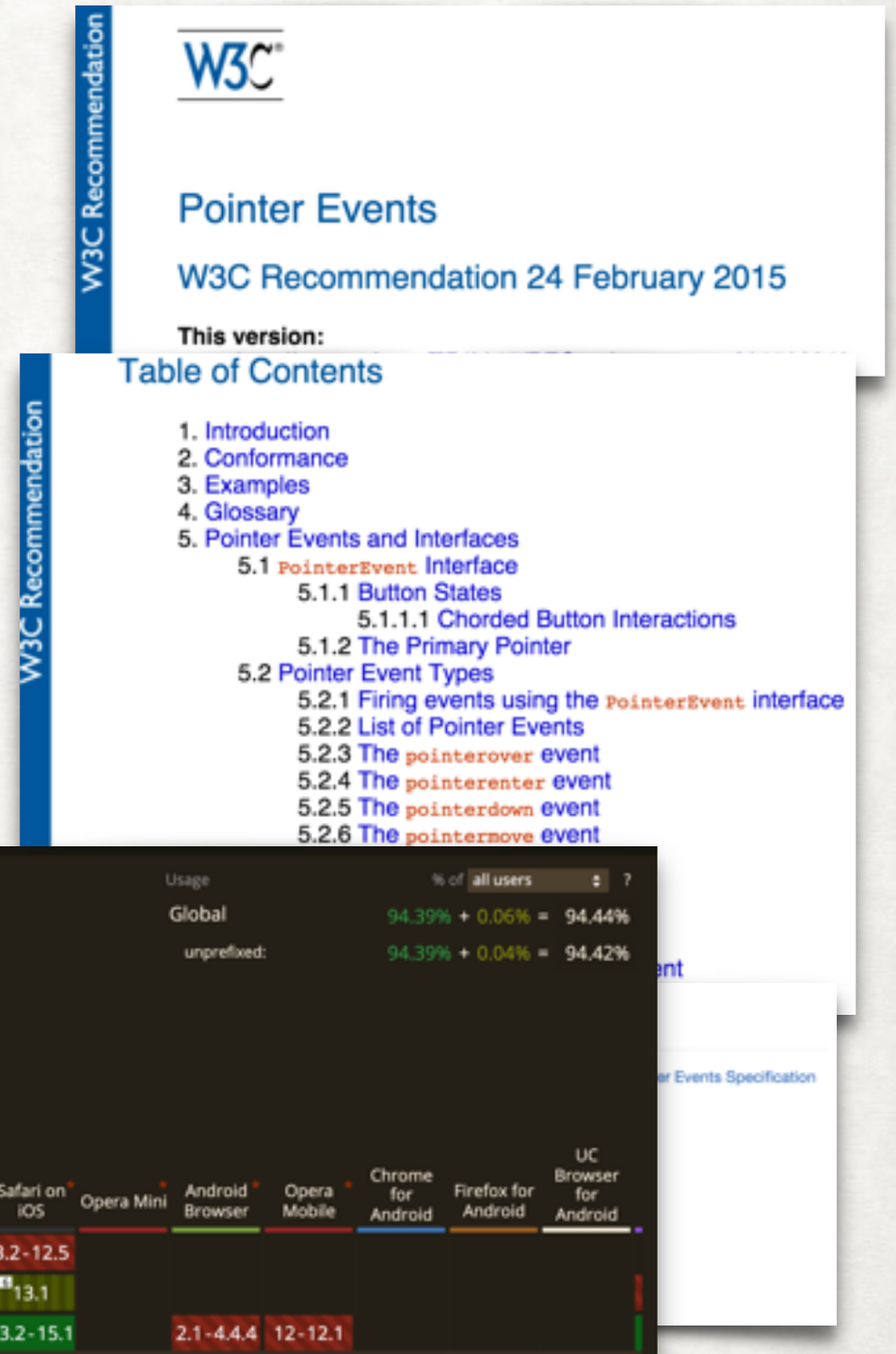






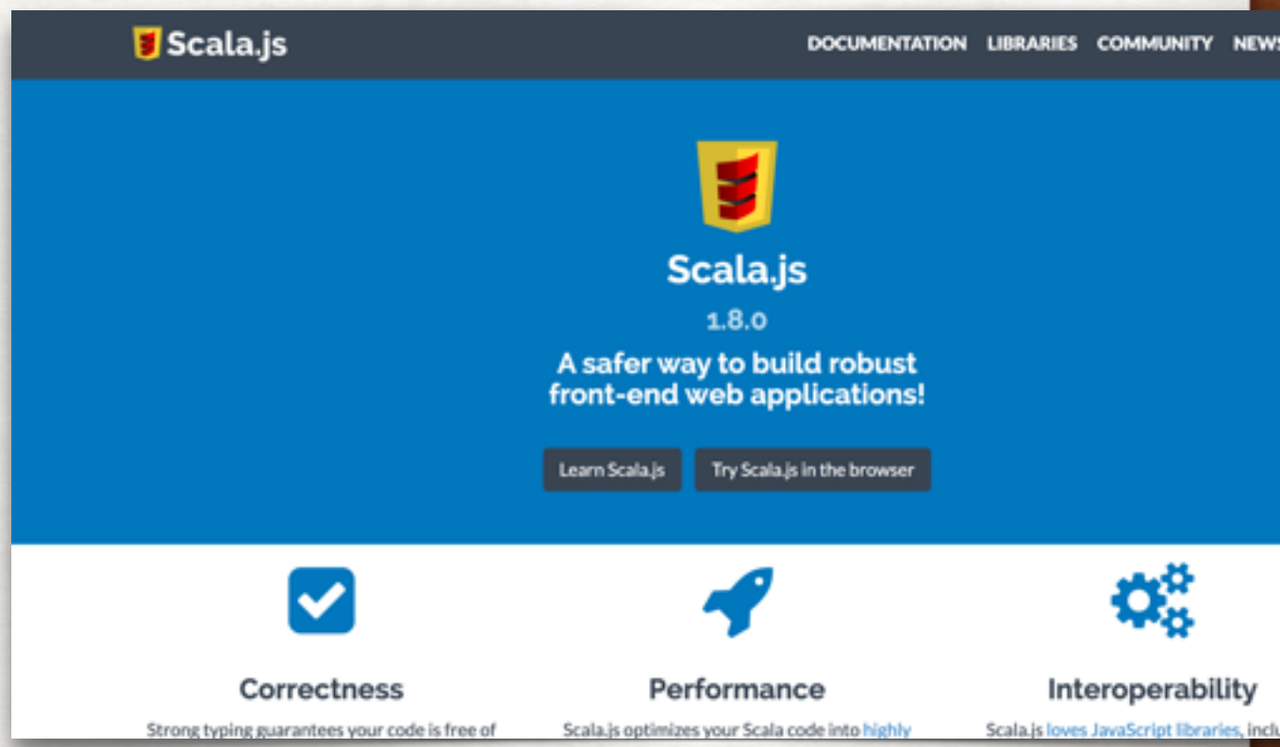
Pointer Events

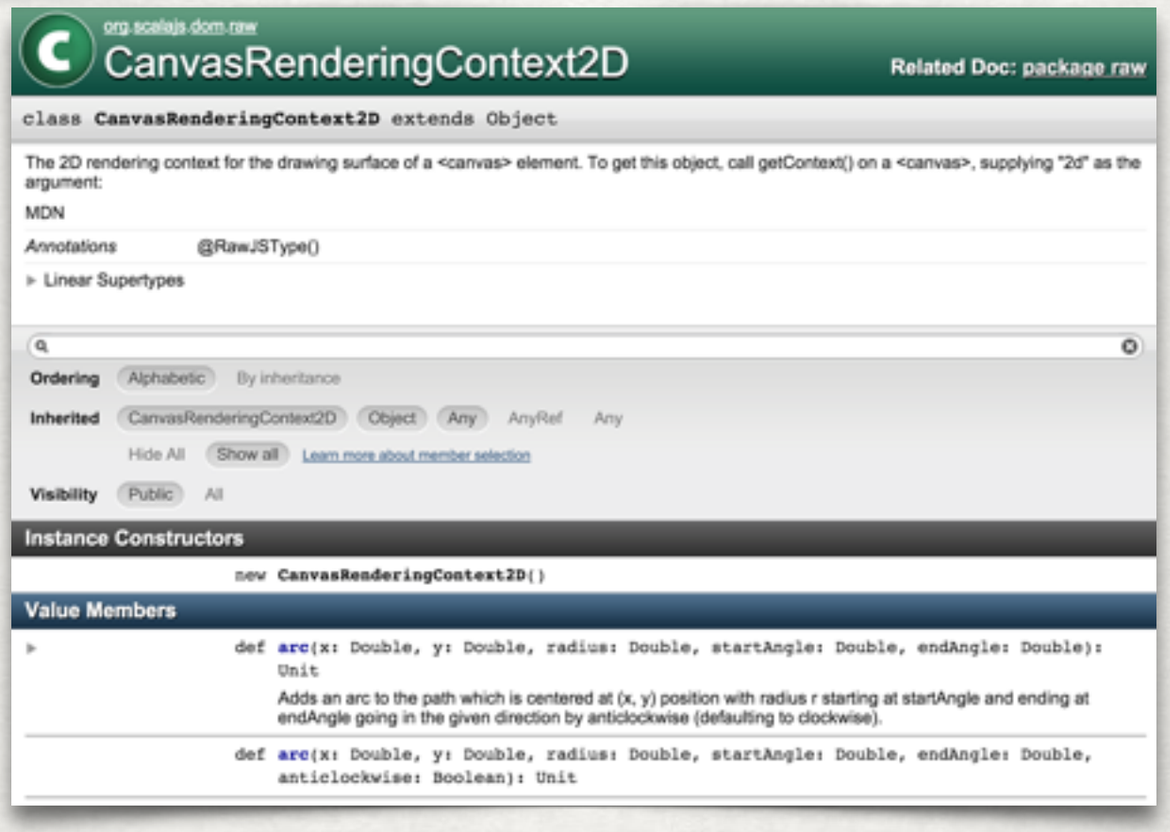
- I want to target all web-capable devices, whether touch or mouse based
- The W3C Pointer Events API abstracts over mouse & touch events
- Because a canvas has no internal structure, the only events we'll get inside are "raw" e.g.
  - pointerdown
  - pointerup
  - pointermove
  - pointerleave



Scala.Js And Scala-Js-Dom

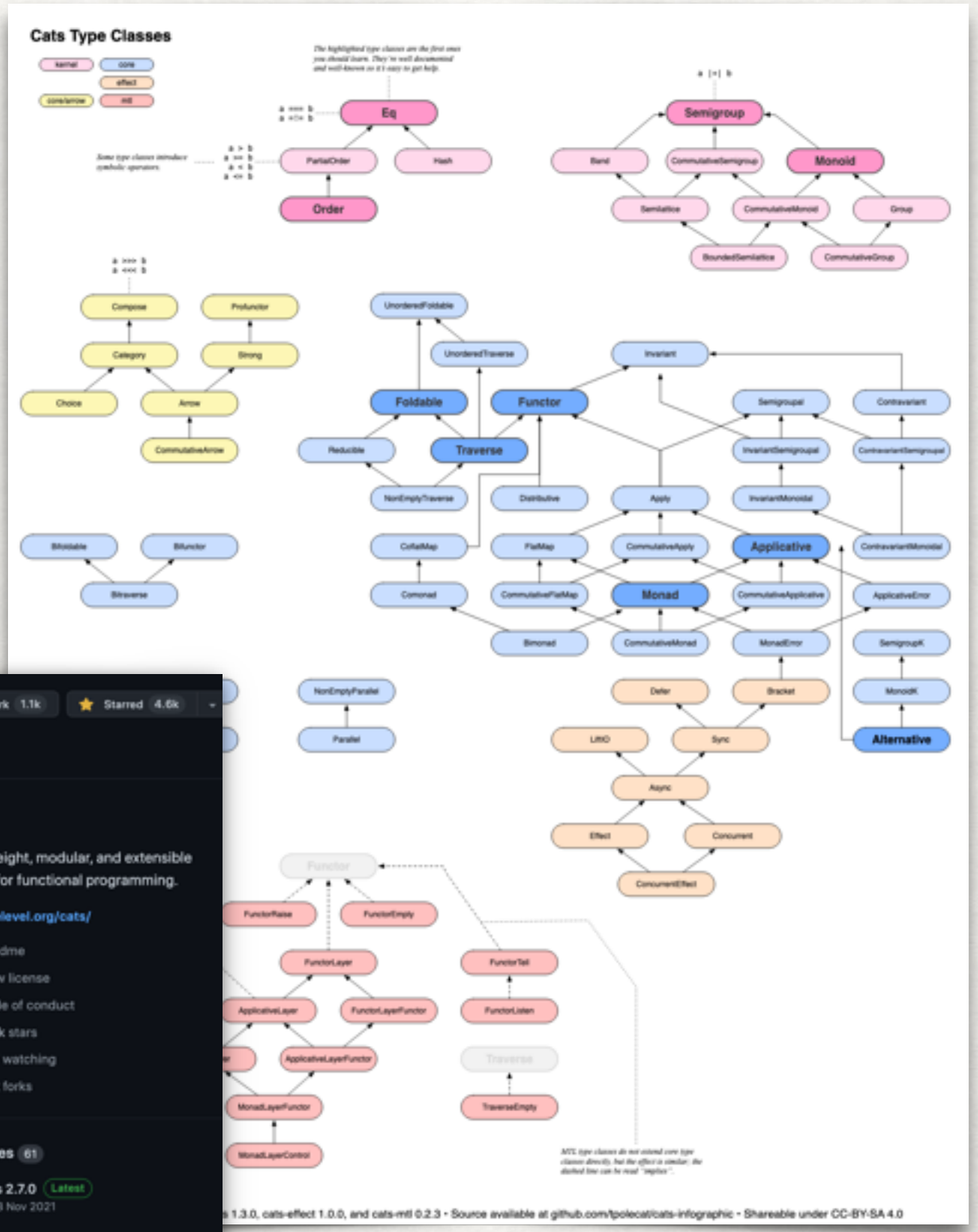
- Scala.js compiles Scala code to Javascript
  - Near-transparent interop with native JS code
- Facade libraries put a typed interface over JS APIs
  - scala-js-dom covers a lot of core W3C browser APIs including canvas and pointer events

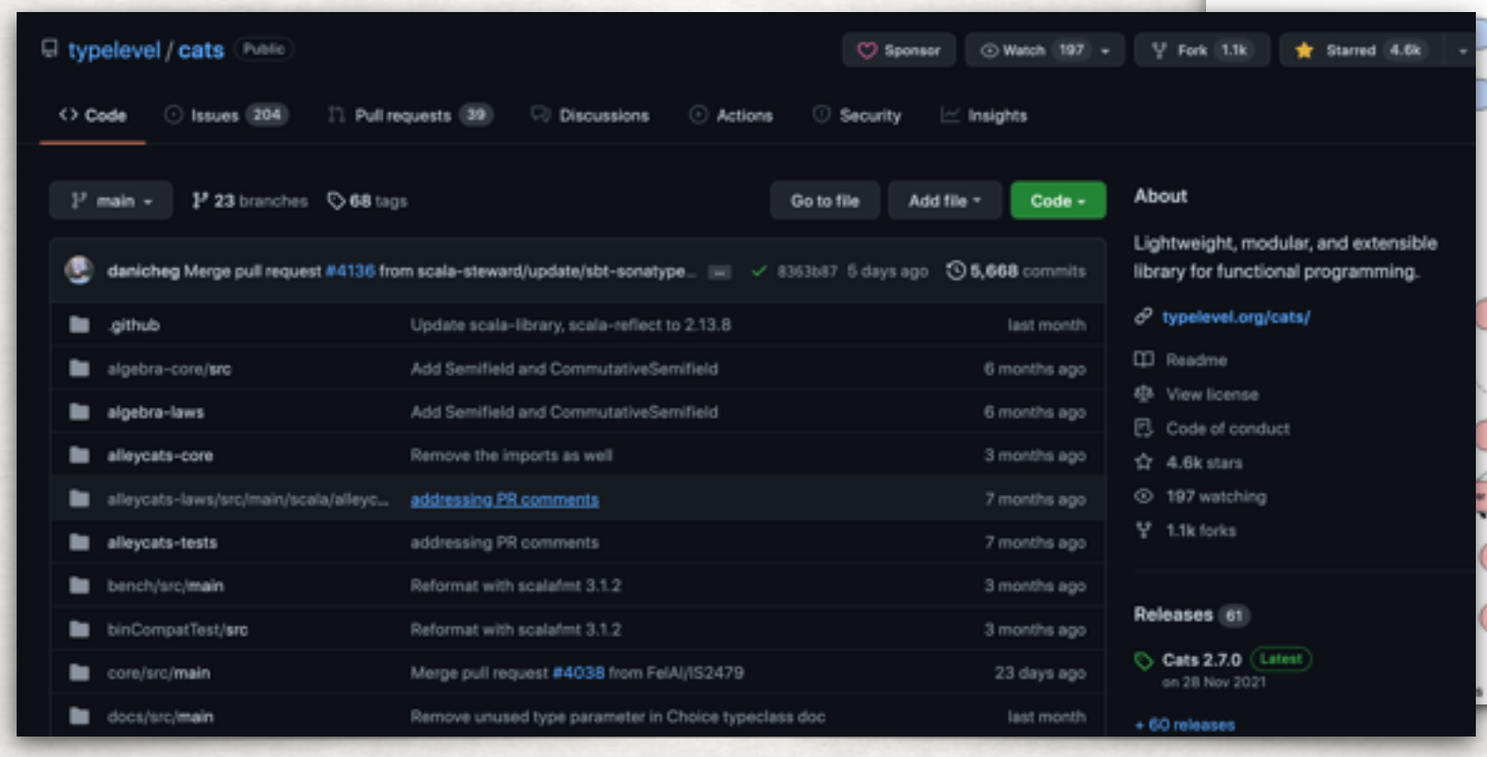




Cats: A Functional Programming Library For Scala

- Representations of State transitions
- Operators for composing stateful computations
  - eg Monad & Applicative type classes



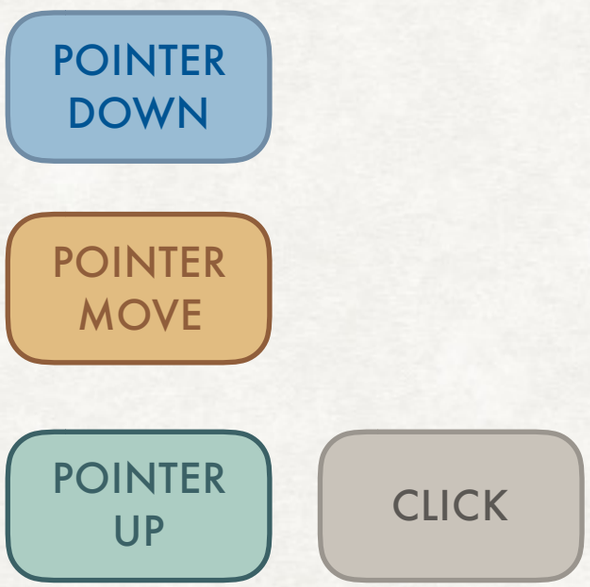


Gesture Recognition

- The process of recognising patterns in a stream of low-level events and emitting higher-level gesture events
  - Inherently *stateful* process
  - An incomplete gesture may be ambiguous

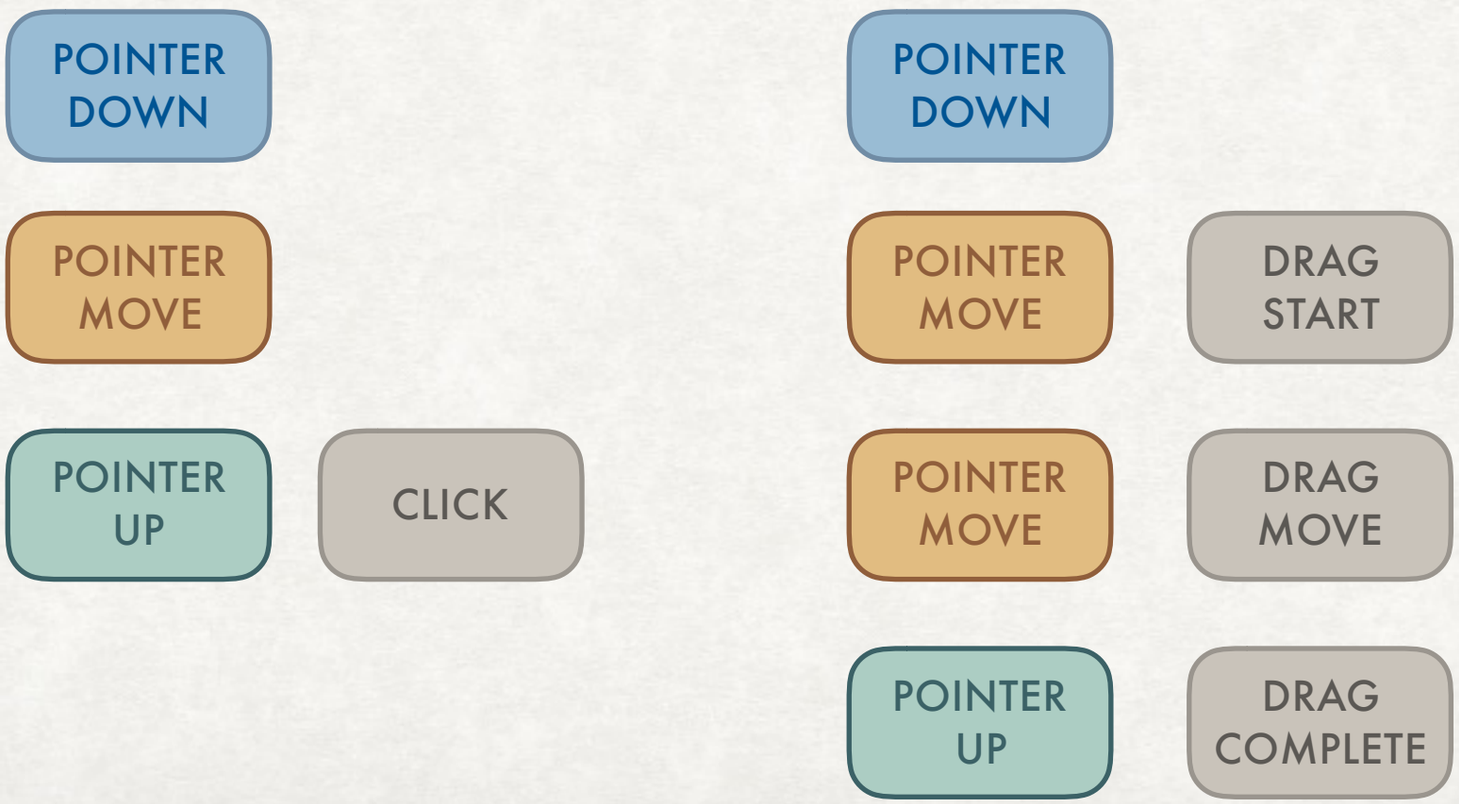
Gesture Recognition

- The process of recognising patterns in a stream of low-level events and emitting higher-level gesture events
  - Inherently *stateful* process
  - An incomplete gesture may be ambiguous



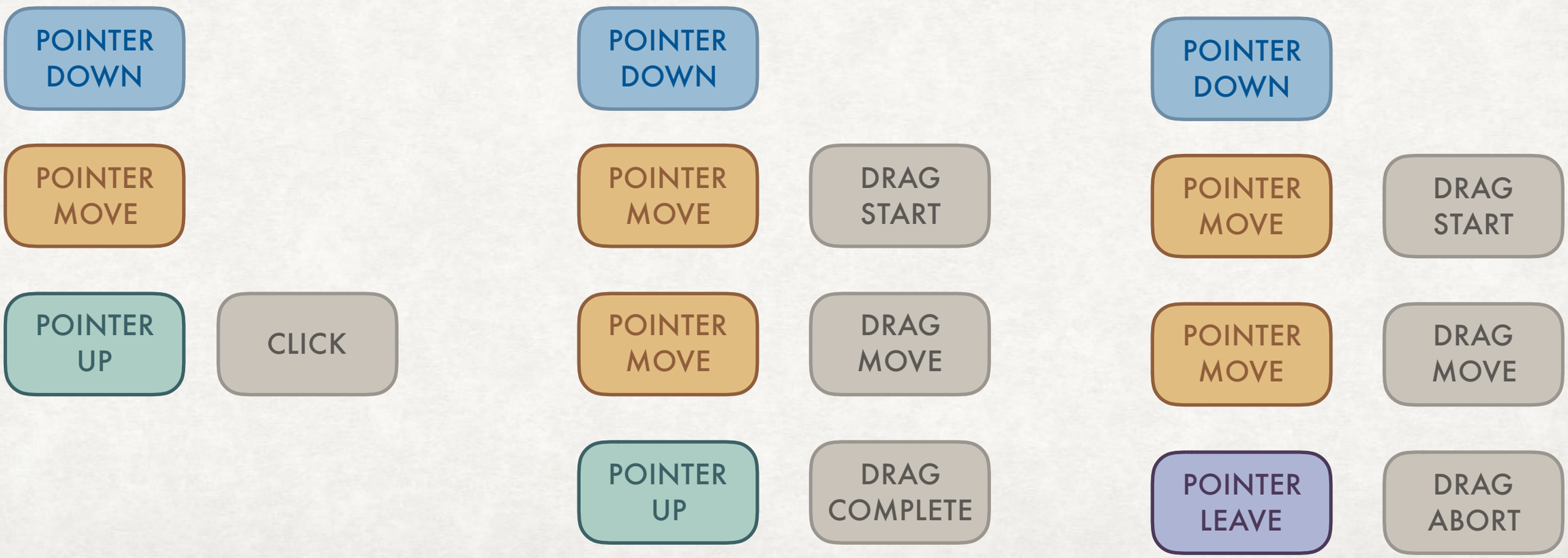
Gesture Recognition

- The process of recognising patterns in a stream of low-level events and emitting higher-level gesture events
  - Inherently *stateful* process
  - An incomplete gesture may be ambiguous



Gesture Recognition

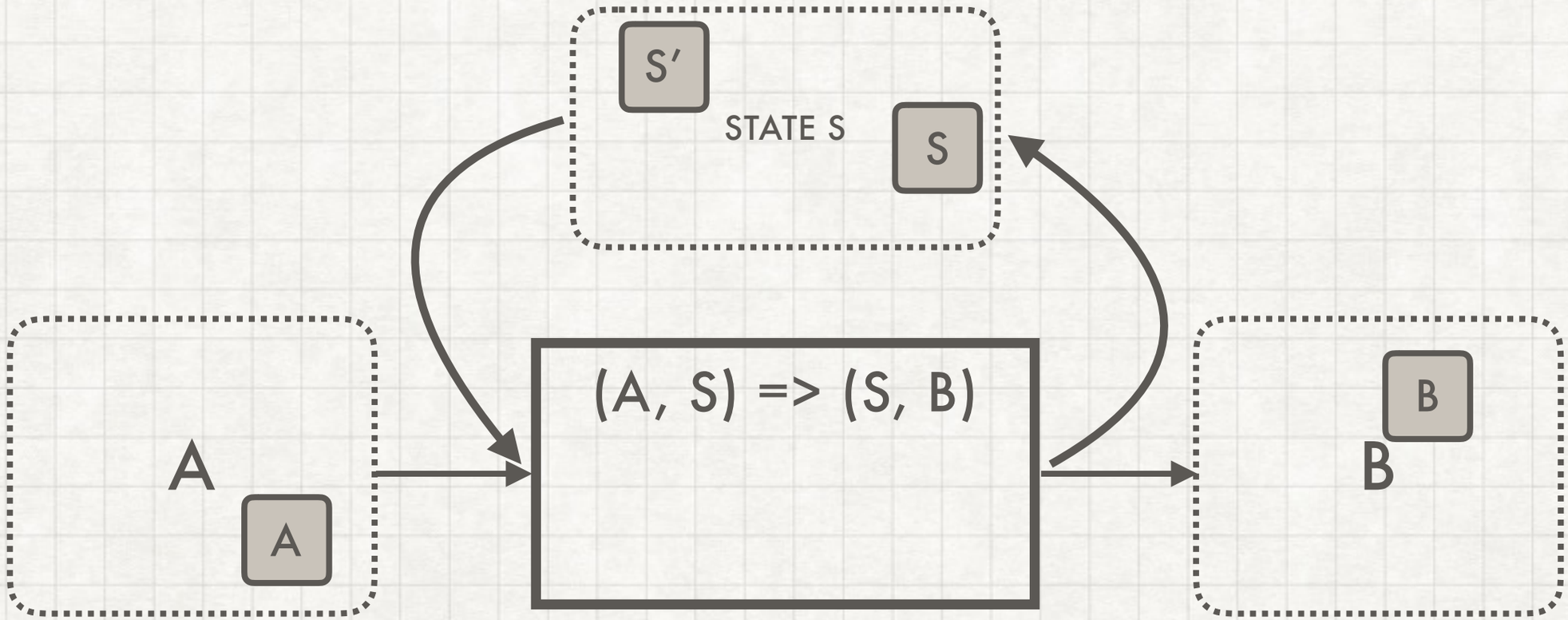
- The process of recognising patterns in a stream of low-level events and emitting higher-level gesture events
  - Inherently *stateful* process
  - An incomplete gesture may be ambiguous



REPRESENTING STATE: FINITE STATE MACHINES

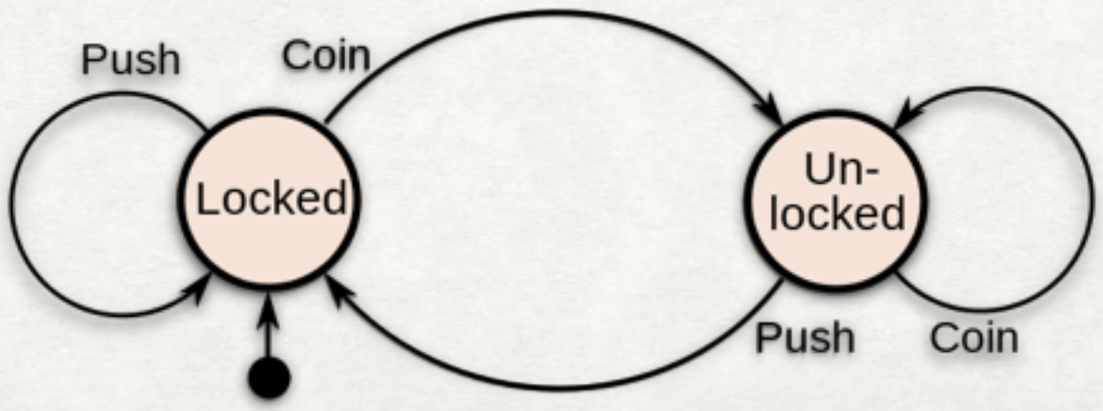
BEN HUTCHISON, V2.0, FEB 2022

PAIR: PURE FUNCTION + STATE STORE



Finite State Machines

- FSMs are my favourite way to think about state
  - What are all the states the system can be in?
    - (at least two, or it's stateless)
  - How does it transition between states?
  - What should happen upon transition?
  - State diagram for a turnstile



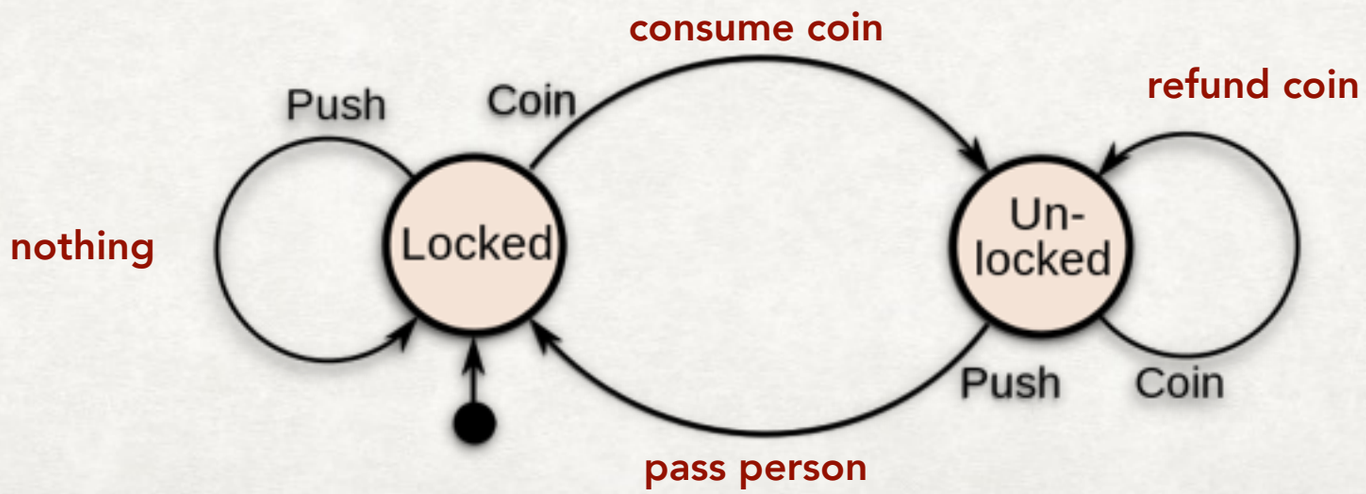
Modelling States & Transitions

- A finite set of state are well modelled using a case class hierarchy

```
sealed trait TurnstileState
case object Locked extends TurnstileState
case object Unlocked extends TurnstileState
```

- While transitions *could* be modelled as
  f: (Input, TurnstileState) => TurnstileState

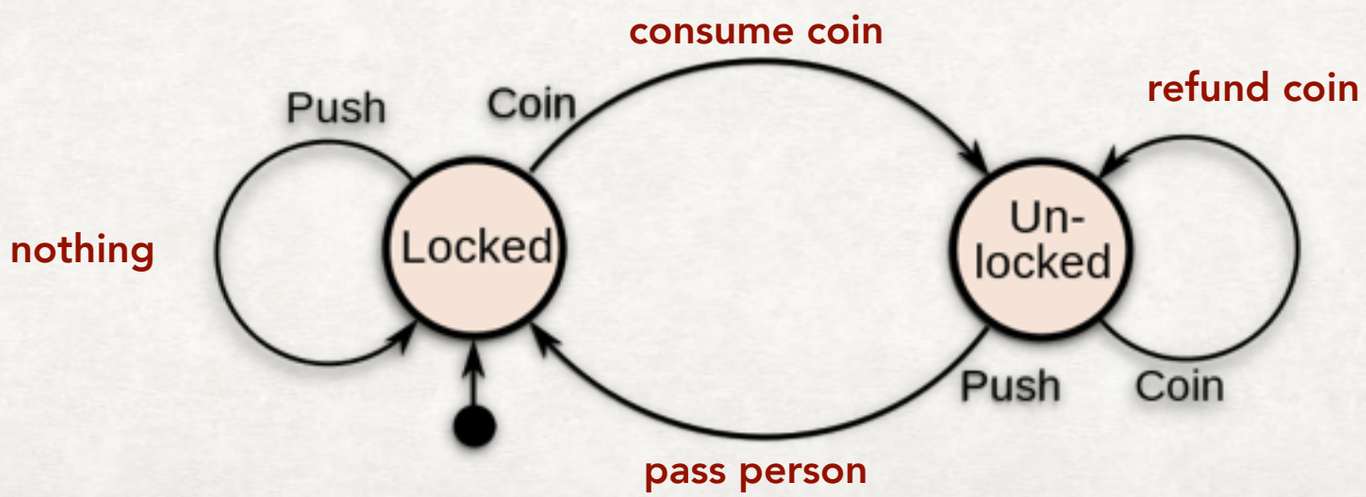
- ..typically we also want output actions to occur upon state change, so
  f: (Input, TurnstileState) => (TurnstileState, Output)



State Transitions

- **cats.data.State** models a state-transition function **S**tate => (**S**tate, **A**ction)
  - represents a *transition path* in a FSM.
  - (terminology can be confusing: **cats.data.State** is not a state, but a *transition*).
- Using currying, we can write functions that accept required input and yield the State

```
def insertCoin(c: Coin): TurnstileState => (TurnstileState, Action)
```



States

```
sealed trait PointerState

case class Up() extends PointerState

case class Down(p: Vec2d, timestamp: Long) extends PointerState

case class Drag(from: Vec2d, fromTimestamp: Long,
                to: Vec2d, toTimestamp: Long) extends
PointerState
```

Gestures: Representing The Output

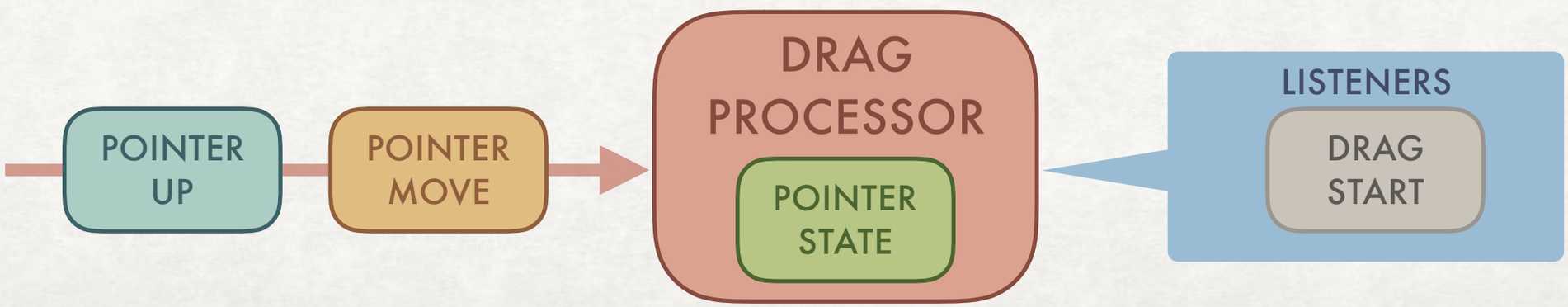
```
sealed trait GestureEvent

case class Click(p: Vec2d, timestamp: Long) extends GestureEvent

case class DragStart(from: Vec2d, fromTimestamp: Long,
  to: Vec2d, toTimestamp: Long, delta: Vec2d) extends GestureEvent

case class DragMove(from: Vec2d, fromTimestamp: Long,
  to: Vec2d, toTimestamp: Long, delta: Vec2d) extends GestureEvent

case class DragComplete(from: Vec2d, fromTimestamp: Long,
  to: Vec2d, toTimestamp: Long, delta: Vec2d) extends GestureEvent

case class DragAbort(from: Vec2d, fromTimestamp: Long,
  to: Vec2d, toTimestamp: Long) extends GestureEvent

case class Invalid(msg: String, pointerEvent2: PointerEvent)
extends
  GestureEvent

case object Noop extends GestureEvent
```

STATEFUL FUNCTIONAL PROGRAMMING IN GESTURE

BEN HUTCHISON, V2.0, FEB 2022

Why Not The Classic Object-Oriented Approach?

- Functional design was in fact my *second* attempt
- Initially I used a DragProcessor written in object-oriented style
- Main problem: it was harder to test
- PointerState encapsulated away inside processor
- Need to mock out listeners to verify emitted events



What Does "Easy To Test" Look Like?

- IMO the easiest code to test would be
  - Specify the current state
  - Specify the input PointEvent
- and it returns
  - The new state
  - The recognised gesture, if any

Input, Current State => (Next State, Gesture)

What Does "Easy To Test" Look Like?

- IMO the easiest code to test would be
  - Specify the current state
  - Specify the input PointEvent
- and it returns
  - The new state
  - The recognised gesture, if any

Input, Current State => (Next State, Gesture)

```
val (s, g) = eventSequence(initialState = Up())(
  PointerDown((0, 0), 0L), PointerMove((20, 20), 10L), PointerUp((30, 30), 20L))

(s must_== Up()) and (g must_== DragComplete((0, 0), 0L, (30, 30), 20L, (10,
10)))
```

Responding To Pointerdown

- The State[S, A] data type just wraps a function you define with signature f: S => (S, A), providing some useful state monad operations
  - S means “state”, action means “Action”
  - Typically you pattern match on the initial state
- PointerDown rules in prose and then in code
  - “if we’re in an Up state, transition to Down state, recording when and where, and emit no gesture”
  - “a PointerDown event doesn’t make sense if we’re already down or dragging”

```
def pointerDown(pe: PointerDown) = State[PointerState, GestureEvent](ps => ps match {
  case Up() =>
    (Down(pe.p, pe.timestamp), Noop)
  case _ => invalid(pe, ps)
})
```

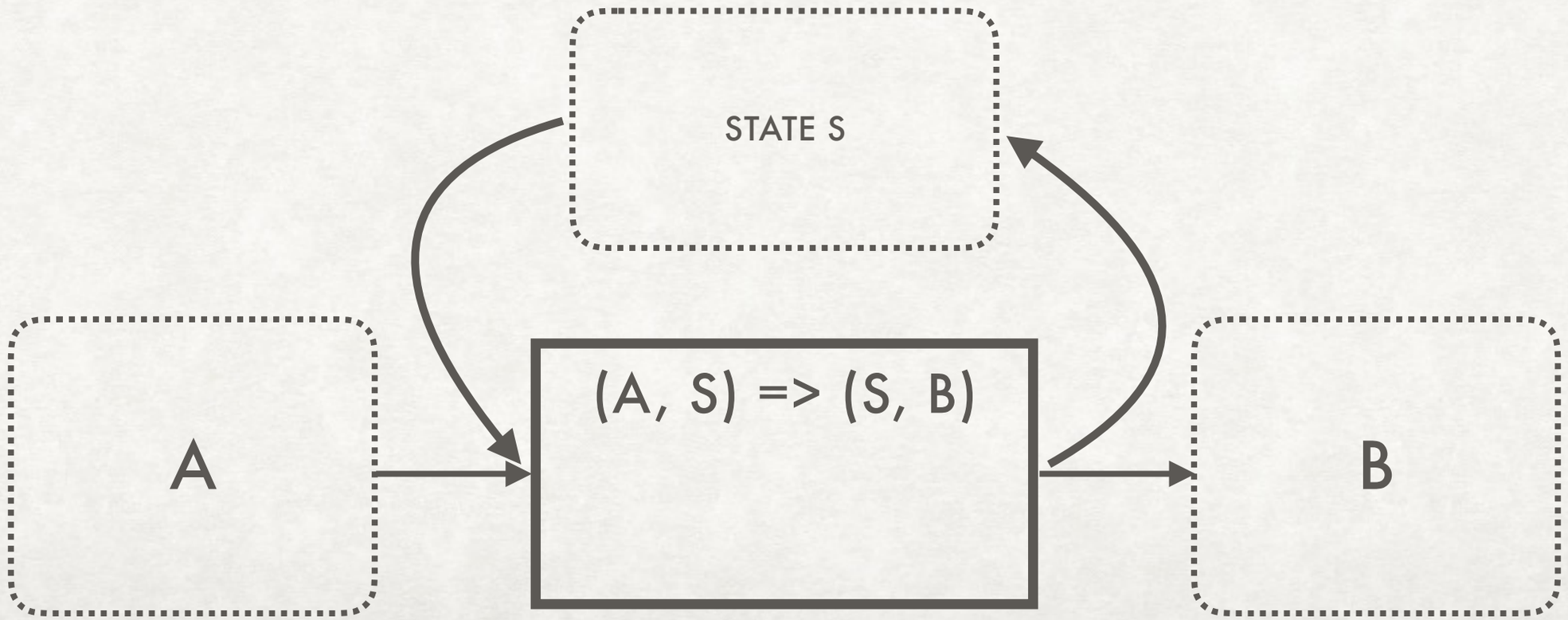
Responding To Pointermove

- PointerMove rules in prose and then in code
  - “if we’re in an Up state, stay there and emit no gesture”
  - “if we’re Down, check how far we’ve travelled since we went down. If its enough to count as a drag, enter Drag state and emit a DragStart gesture”
  - “if we’re already Dragging, extend the Drag to the new location and emit a DragMove”

```
def pointerMove(pe: PointerMove) = State[PointerState, GestureEvent](ps => ps match {
  case Up() => (Up(), Noop)
  case Down(p, timestamp) =>
    if (p.distanceTo(pe.p) > dragThreshold)
      (Drag(p, timestamp, pe.p, pe.timestamp), DragStart(p, timestamp, pe.p, pe.timestamp, pe.p -
p))
    else
      (ps, Noop)
  case Drag(from, fromTimestamp, to, toTimestamp) =>
    (Drag(from, fromTimestamp, pe.p, pe.timestamp), DragMove(from, fromTimestamp, pe.p,
pe.timestamp, pe.p - to))
})
```

The Imperative Rind

- An remark by Simon Peyton Jones early in my FP journey left a mark on me
  - roughly “functional programs have a functional interior and an imperative rind (exterior)”
- The new state computed by a state function needs to be **stored**
- The action emitted by the stateful function needs to be **executed**



Gesture Demo

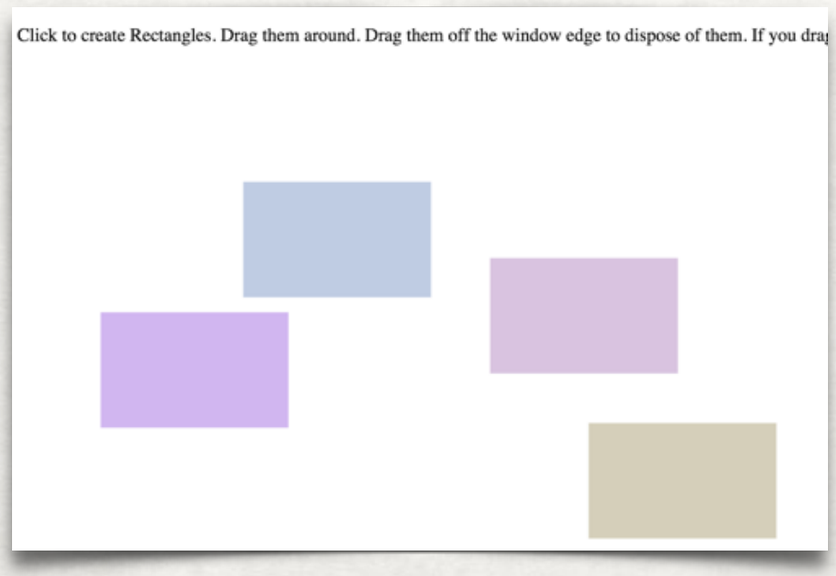
```
def handlePointerEvent(pe: PointerEvent) = {
  val (newState, gestureAndRegions) = gestureRegionProcessor.
    handlePointerEvent(pe, search).run(pointerAndRegionState).run

  pointerAndRegionState = newState
  interpret(gestureAndRegions)
}

def interpret(gr: GestureAndRegions[Rect]) = {
  gr match {
    case GestureAndRegions(Click(p, timestamp), None, None) =>
      def randLightValue = 180 + Random.nextInt(60)
      val randomColor = s"rgb($randLightValue, $randLightValue, $randLightValue)"
      val r = new Rect(p, Width, Height, randomColor)
      rectangles = rectangles :+ r
      draw()
    case GestureAndRegions(d: DragMove, Some(Rect(_, _, _, _, id)), _) =>
      rectangles = rectangles.map(r =>
        if (r.id == id)
          r.copy(topLeft = r.topLeft + d.delta)
        else r)
    case GestureAndRegions(d: DragAbort, Some(Rect(_, _, _, _, id)), _) =>
      rectangles = rectangles.filterNot(_.id == id)
    case GestureAndRegions(d: DragComplete, Some(Rect(_, _, _, _, srcId)), Some(Rect(_, _, _, _,
targetId))) =>
      rectangles = rectangles.map(r =>
        if (srcId != targetId && r.id == targetId)
          r.copy(cssColorString = RedColorString)
        else r)
    case _ => Noop
  }
  draw()
}
```

TRACKING DRAG REGIONS

BEN HUTCHISON, FEB 2022, V2.0

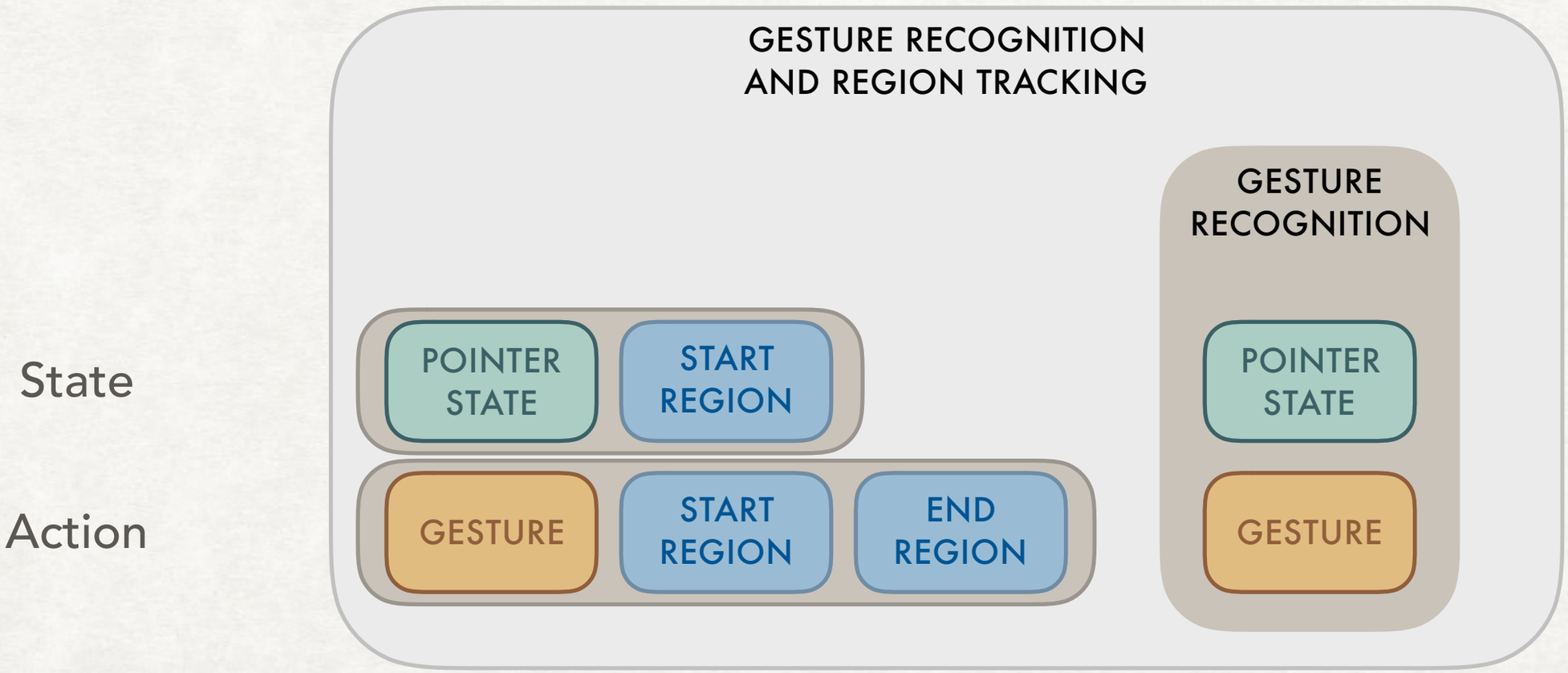
Tracking Drag Regions

- For most applications, its not enough to know that drags have happened
  - We want to know about the object where they began, completed or passed over
  - Gesture keeps track of these objects, which it calls “Regions”, and it actually doesn’t care what they are
  - You give it a function to convert a point into a region of arbitrary type R, and it calls it and tracks the values



State[PointerRegionState, GestureAndRegions[R]]

- The region-tracking State wraps the simpler gesture recognition State



State[PointerRegionState, GestureAndRegions[R]]

- The region-tracking State wraps the simpler gesture recognition State

```
case class GestureAndRegions[R](
    gesture: GestureEvent, from: Option[R], to: Option[R])

type PointerRegionState = (PointerState, Option[R])

def handlePointerEvent(pe: PointerEvent, regionSearch: Vec2d => Option[R]) =
  State[PointerRegionState, GestureAndRegions[R]] {
  case (ps, optRegion) =>
    val (ps2, g) = gestureProcess.handlePointerEvent(pe).run(ps).run
    g match {
      case DragStart(from, _, to, _, _) =>
        val fromR = regionSearch(from)
        val s = (ps2, fromR)
        val a = GestureAndRegions(g, fromR, regionSearch(to))
        (s, a)
      case DragMove(_, _, to, _, _) =>
        val s = (ps2, optRegion)
        val a = GestureAndRegions(g, optRegion, regionSearch(to))
        (s, a)
//more cases for other gestures…
}}
```

SCALING UP WITH STATEFUL FUNCTIONAL PROGRAMMING

BEN HUTCHISON, FEB 2022, V2.0

Scaling Up With Stateful Fp

- What if the whole client application was purely functional?
  - Input: PointerEvent | Server Messages | Time
  - State: ApplicationState(PointerRegionState, AnimationState, ...)
    - Actions..? ..maybe UpdateView | ServerCall | SetCookie ..
- Need a way to compose local State updates (like State[PointerRegionState, GestureAndRegions[R]]) into app-wide State[ApplicationState, AppAction]

Scaling Up With Stateful Fp

- A *Lens* is a pair of functions
  - get: S => T
  - set: (S, T) => S
- Interpretation: if S is global state, T is the subsystem state, the lens *extracts* the local state with get, and *updates* the local state with set

```
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(inputS).run
    val outputS  = l.set(inputS, outputT)
    (outputS, a)
})
```

THE END

BEN HUTCHISON, FEB 2022, V2.0