STATEFUL FUNCTIONAL PROGRAMMING

CASE STUDY: POINTER GESTURE RECOGNITION

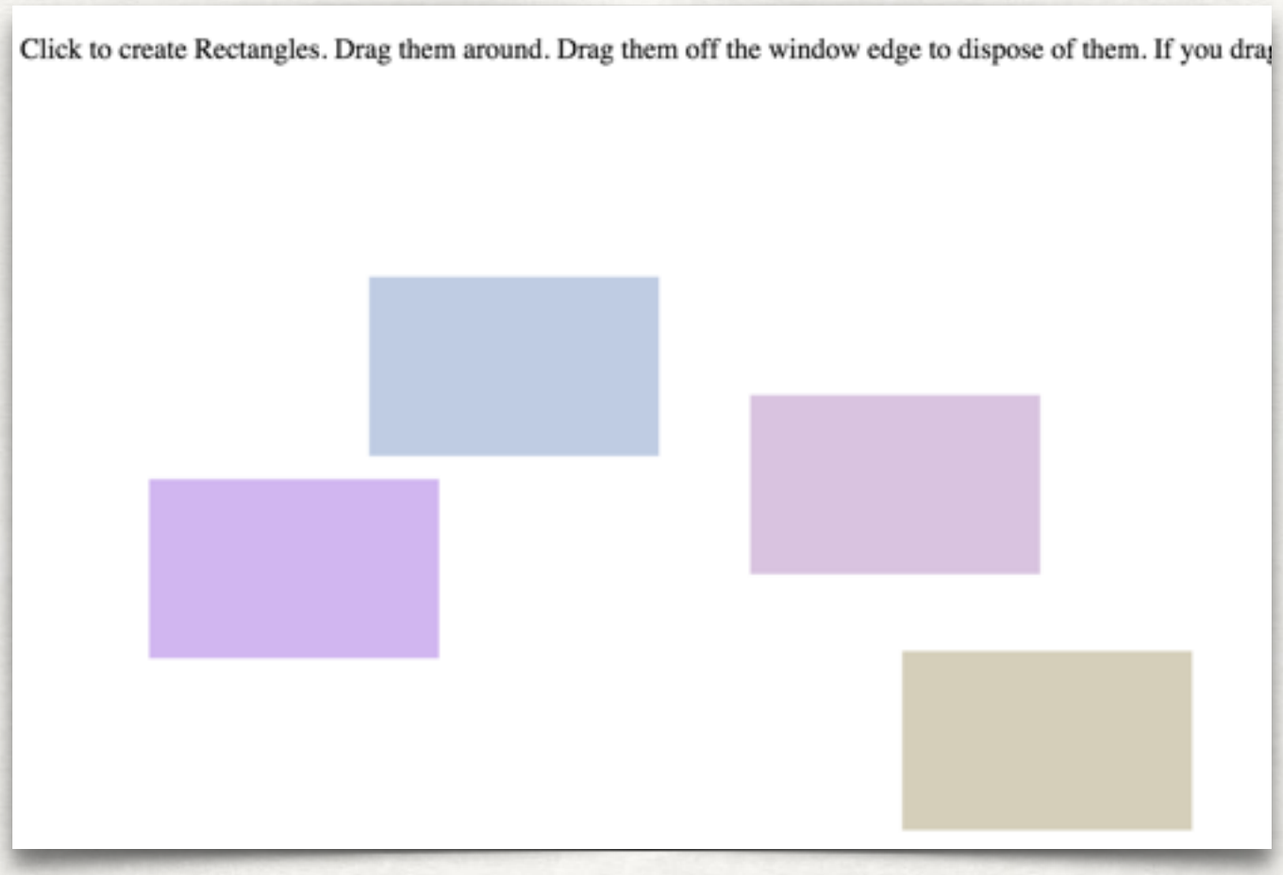
BEN HUTCHISON, V2.0, FEB 2022

PART 1

THE PROBLEM

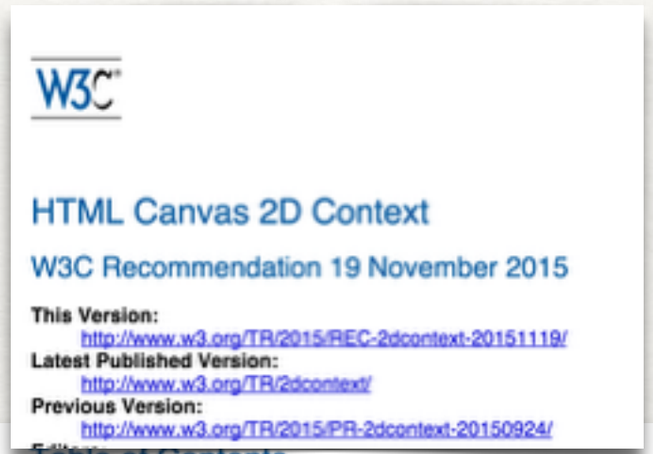
Dragging On An Html Canvas

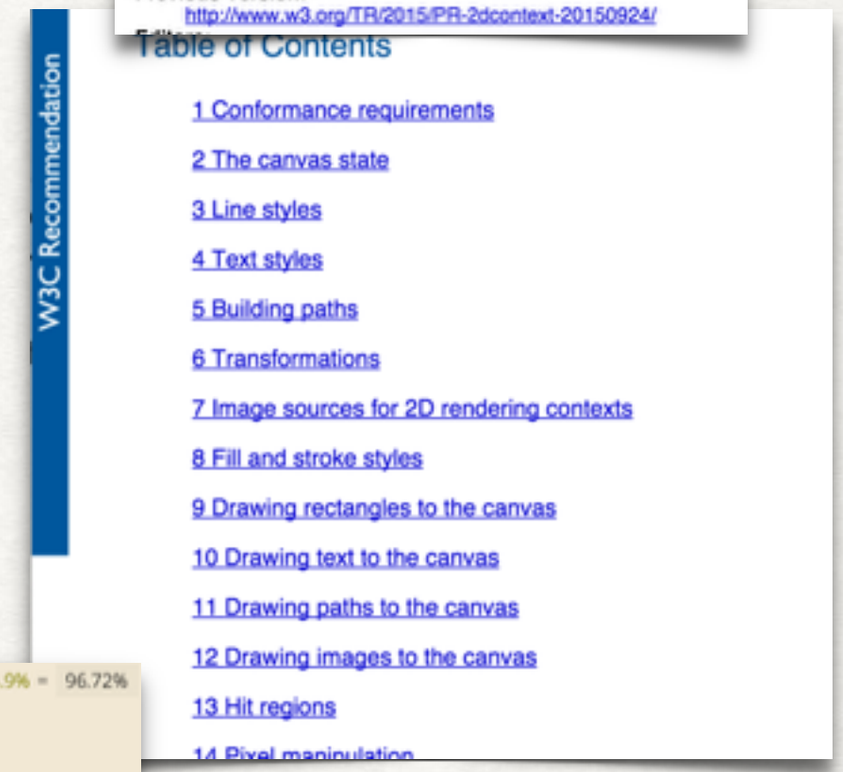
- I wanted to support drag and drop type behaviors using Scala.js and the HTML Canvas

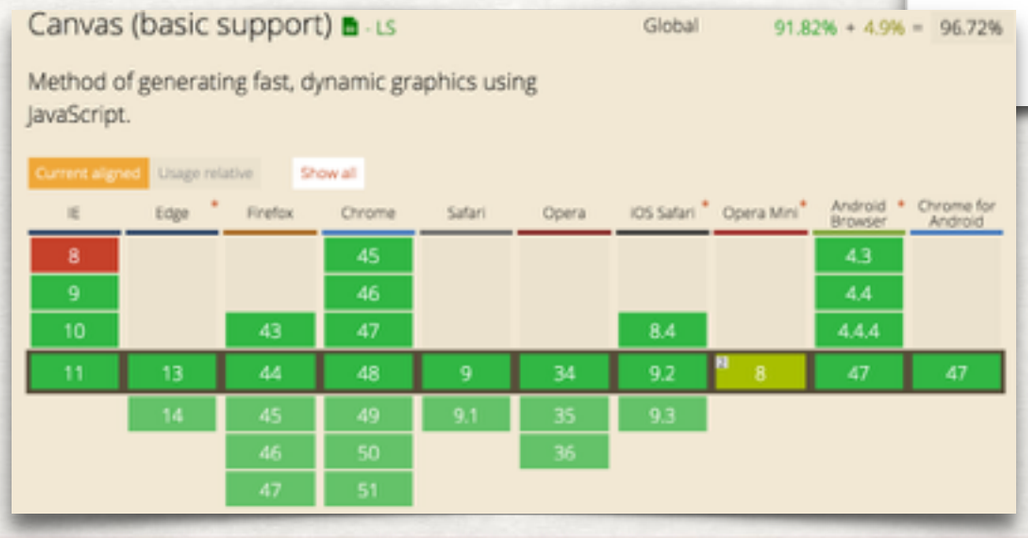


Html Canvases

- 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

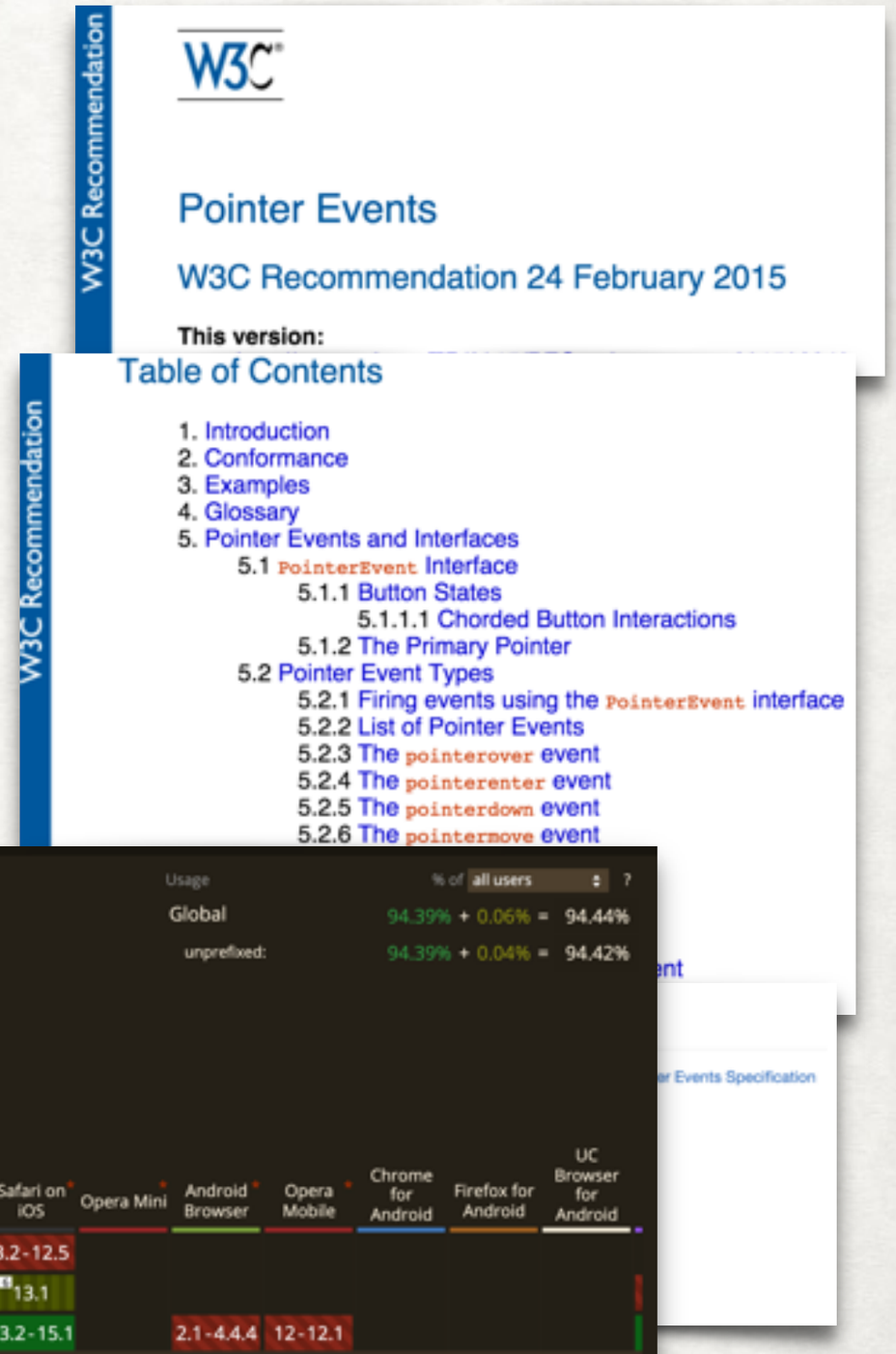






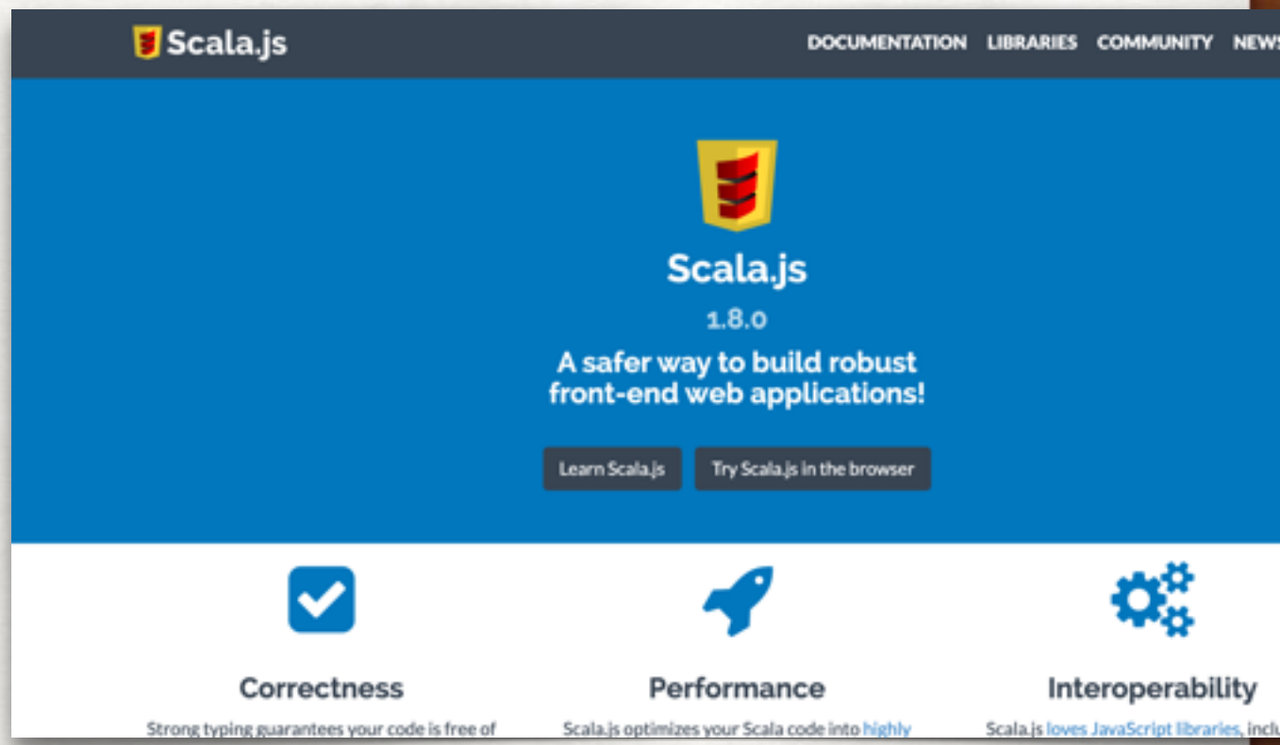
Pointer Events

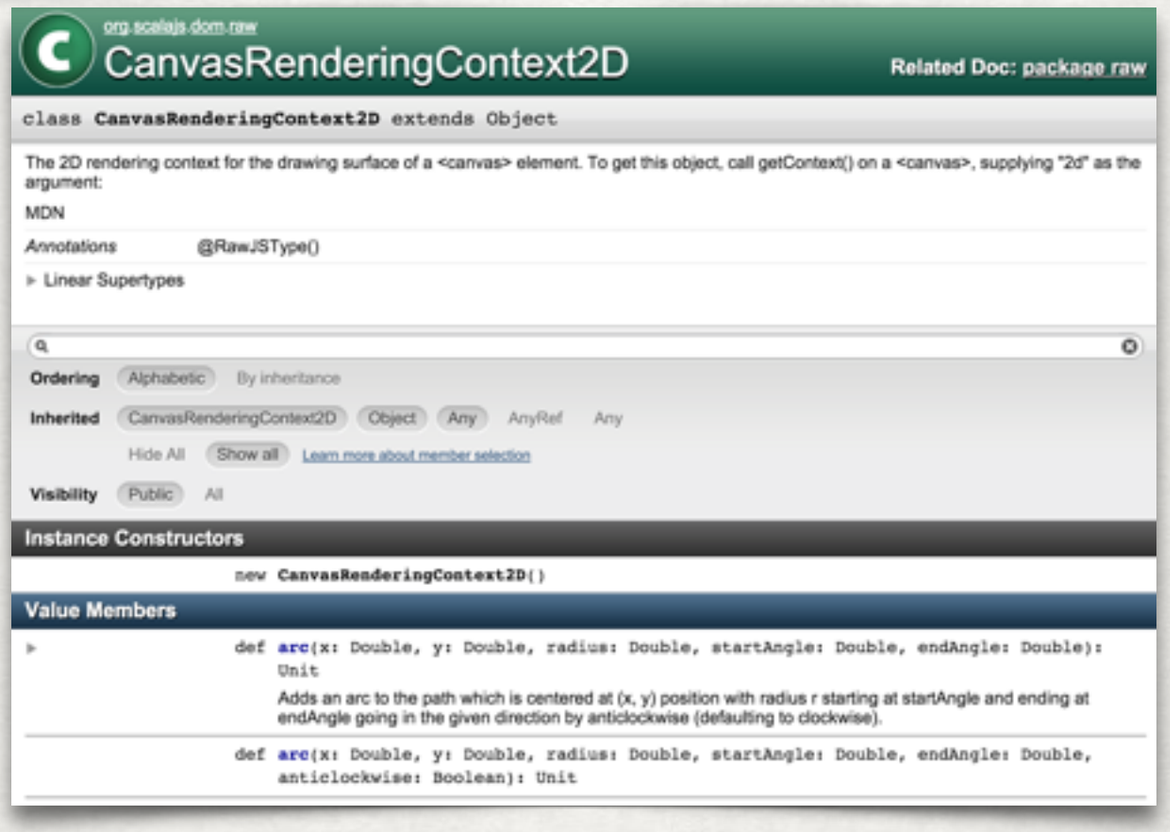
- I want to target all web-capable devices, whether touch or mouse based
- The W3C Pointer Events API abstracts over mouse & touch events
- Because a canvas has no internal structure, the only events we'll get inside are "raw" e.g.
  - pointerdown
  - pointerup
  - pointermove
  - pointerleave



Scala.Js And Scala-Js-Dom

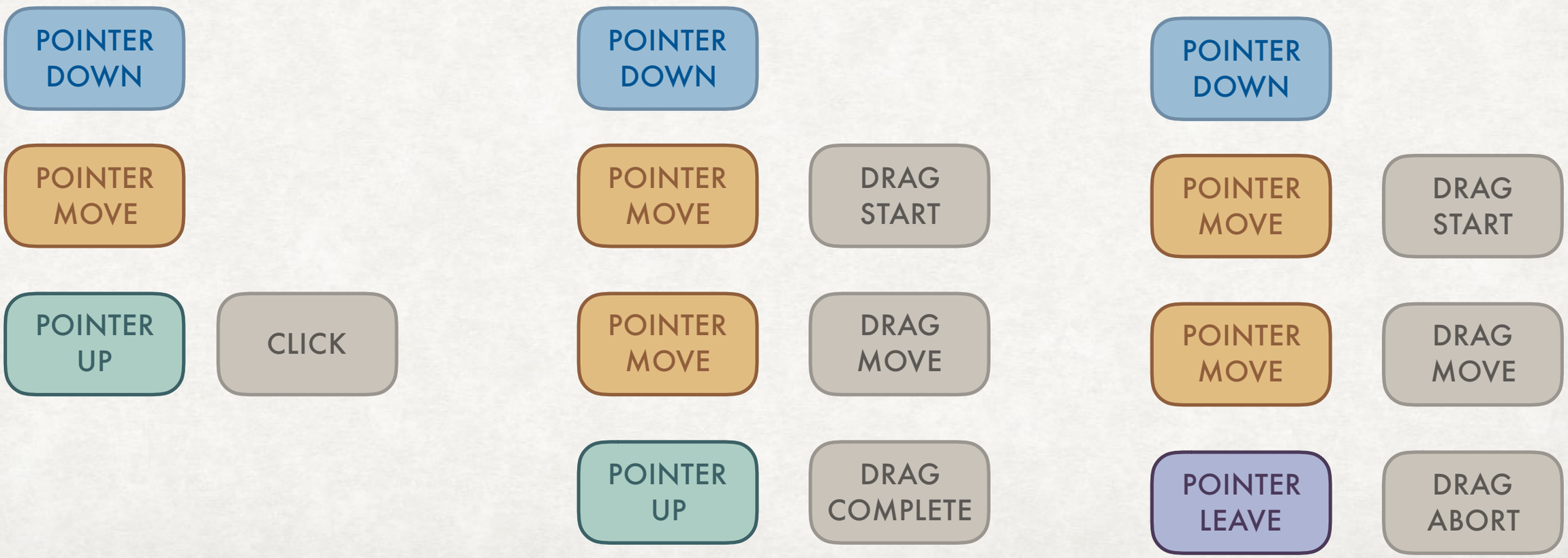
- Scala.js compiles Scala code to Javascript
  - Near-transparent interop with native JS code
- Facade libraries put a typed interface over JS APIs
  - scala-js-dom covers a lot of core W3C browser APIs including canvas
  - Not Pointer Events (*yet) - Gesture lib defines them directly





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



States

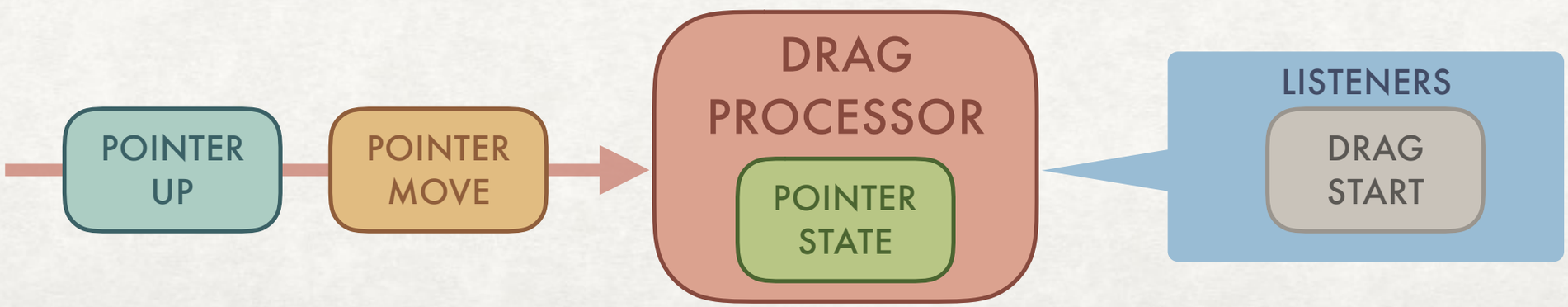
```
sealed trait PointerState

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

```
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
```

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

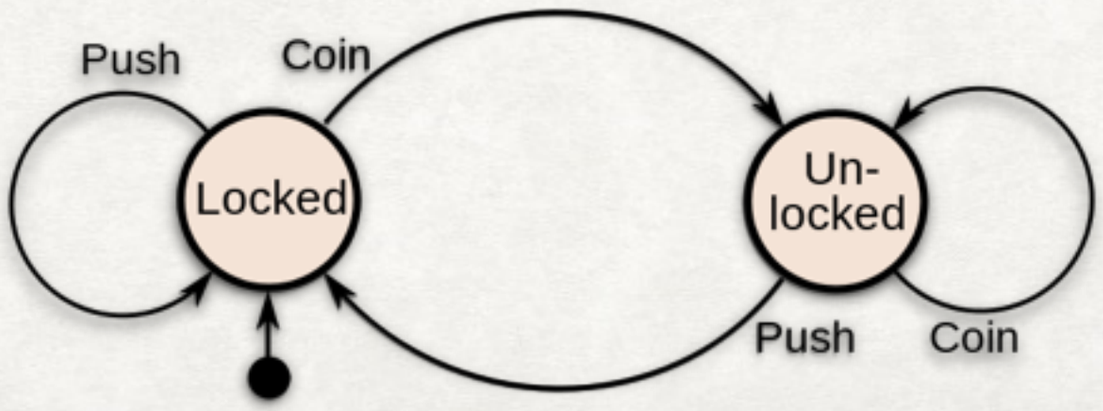
Input, Current State => (Next State, Gesture)

```
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)))
```

PART 2
FINITE STATE MACHINES
STATE EFFECTS

Finite State Machines

- FSMs are my favourite way to think about state
  - What are all the states the system can be in?
  - 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

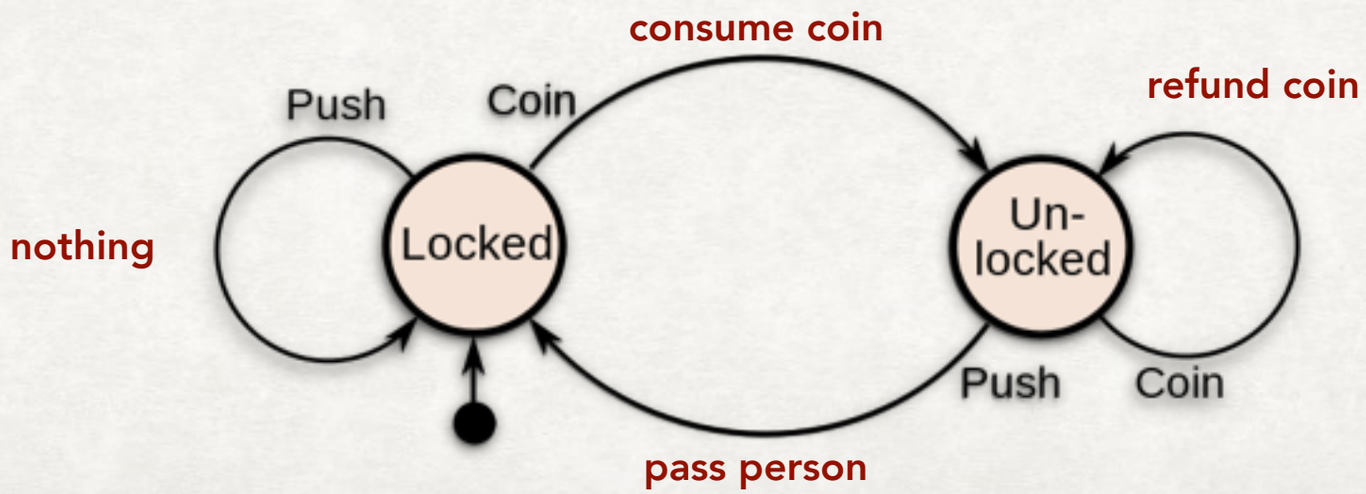
```
sealed trait TurnstileState
case object Locked extends TurnstileState
case object Unlocked extends TurnstileState
```

- Transitions could be modelled as

  f: (Input, TurnstileState) => TurnstileState

- ..but we also want output actions to occur upon state change, so

  f: (Input, TurnstileState) => (TurnstileState, Output)



State Effect Are State Transitions

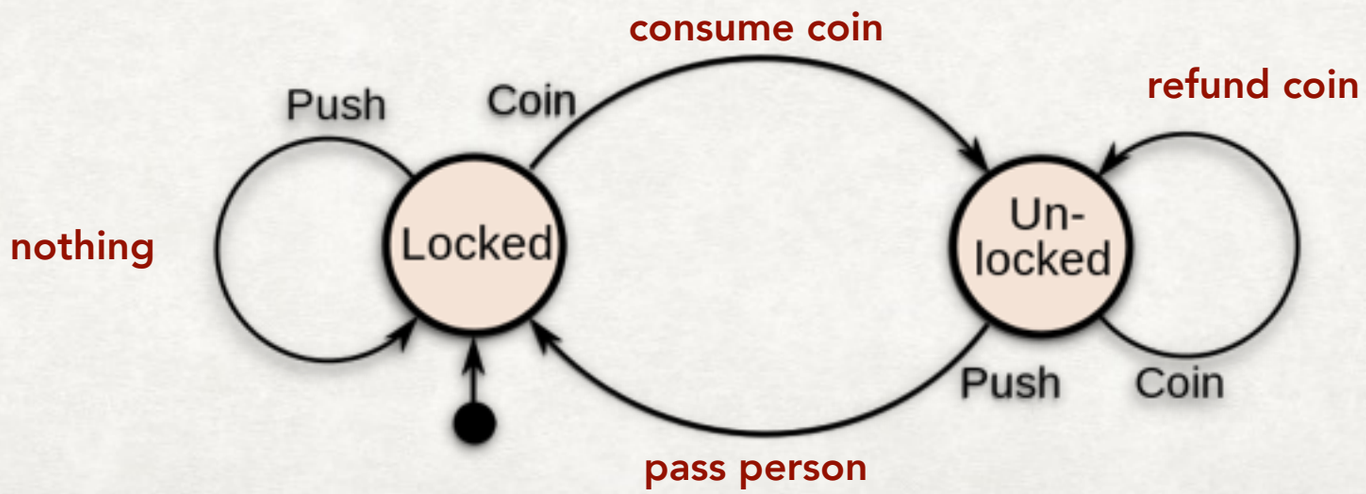
- cats.data.State is essentially a function f: **S**tate => (**S**tate, **A**ction)
- It represents a *transition path* in a FSM, *not a state*. Be warned the terminology can be confusing because Cats library wraps f in a data type called State. But its **not** a state.
- Any inputs are assumed to have already been provided. Using currying, we can write functions that accept required input and yield the State monad

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

- Monads are characterised by their join (aka flatten) operation:

  def join: State[S, State[S, A]] => State[S, A]

  For state monads, join means "chain" the state transitions together



PART 3
USING STATEFUL FUNCTIONAL PROGRAMMING IN GESTURE

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)
})
```

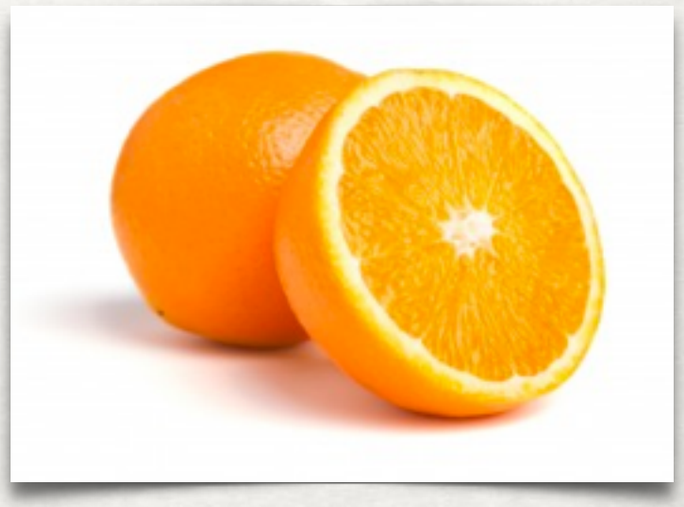
Responding To Pointermove

- 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 updated state computed by a state monad needs to be stored somewhere mutable
- The action emitted by the state monad needs to be executed

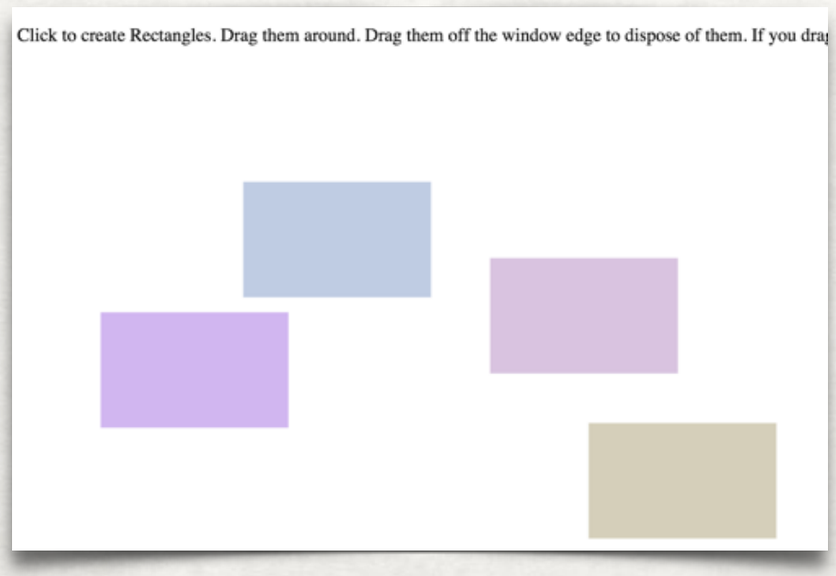


Gesture Demo

```
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()
}
```

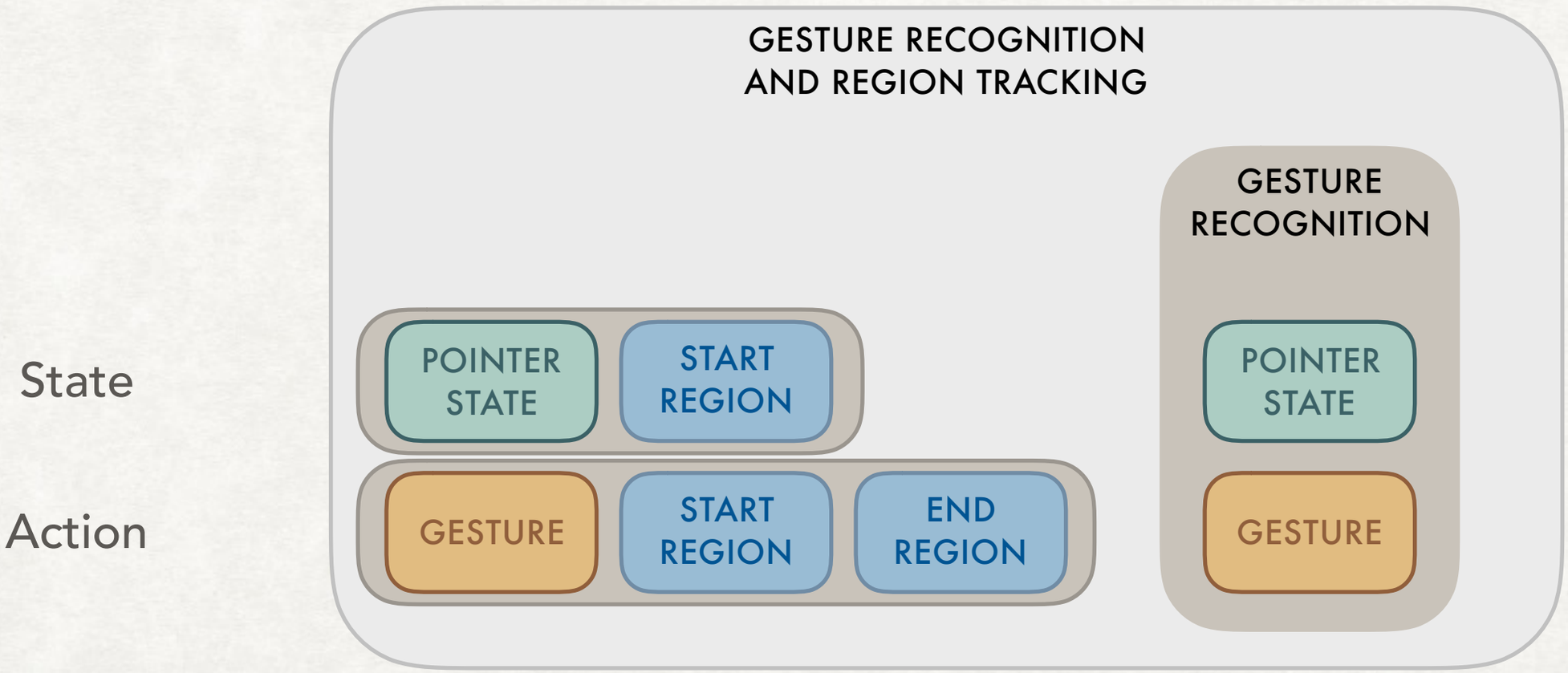
Tracking Drag Regions

- 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 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