RAY: Integrating Rx and Async for Direct-Style Reactive Streams

Philipp Haller, Typesafe
Heather Miller, EPFL

Typesafe

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Topic of this Talk

- Integration of two well-known, widely-used programming models
- Goal: simplify programming with asynchronous streams of observable events

Outline

- Review Async and Rx Models
- A Challenge Problem
- RAY
- The Paper
- Conclusion

The Async Model

- This work focuses on one recent proposal to simplify asynchronous programming: the Async Model
- The essence of the Async Model:
  1. A way to spawn an asynchronous computation (*async*), returning a (first-class) future
  2. A way to suspend an asynchronous computation (*await*) until a future is completed
- Result: a *direct-style API for non-blocking futures*
- Practical relevance: F#, C# 5.0, Scala 2.11

Example

- Setting: Play Web Framework
- Task: Given two web service requests, when both are completed, return response with the results of both:

```
val futureDOY: Future[Response] =
  WS.url("http://api.day-of-year/today").get
val futureDaysLeft: Future[Response] =
  WS.url("http://api.days-left/today").get
```

Example

Using plain Scala futures

```
futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok("" + dayOfYear + ": " + daysLeft + " days left!")
  }
}
```

Example

Using plain Scala futures

```
futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok("" + dayOfYear + ": " + daysLeft + " days left!")
  }
}
```

Using Scala Async

```
val respFut = async {
  val dayOfYear = await(futureDOY).body
  val daysLeft  = await(futureDaysLeft).body
  Ok("" + dayOfYear + ": " + daysLeft + " days left!")
}
```

Example

Using plain Scala futures

```
futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok("" + dayOfYear + ": " + daysLeft + " days left!")
  }
}
```

Using Scala Async

```
val respFut = async {
  val dayOfYear = await(futureDOY).body
  val daysLeft  = await(futureDaysLeft).body
  Ok("" + dayOfYear + ": " + daysLeft + " days left!")
}
```

RAY adopts direct-style await but for observables instead of futures!

Reactive Extensions (Rx)

- Asynchronous event streams and push notifications: a fundamental abstraction for web and mobile apps
- Typically, event streams have to be scalable, robust, and composable
  - Examples: Netflix, Twitter, ...
- Most popular framework: Reactive Extensions (Rx)
  - Based on the duality of iterators and observers (Meijer'12)
  - Cross-platform framework (RxJava, RxJS, ...)
  - Composition using higher-order functions

The Essence of Rx

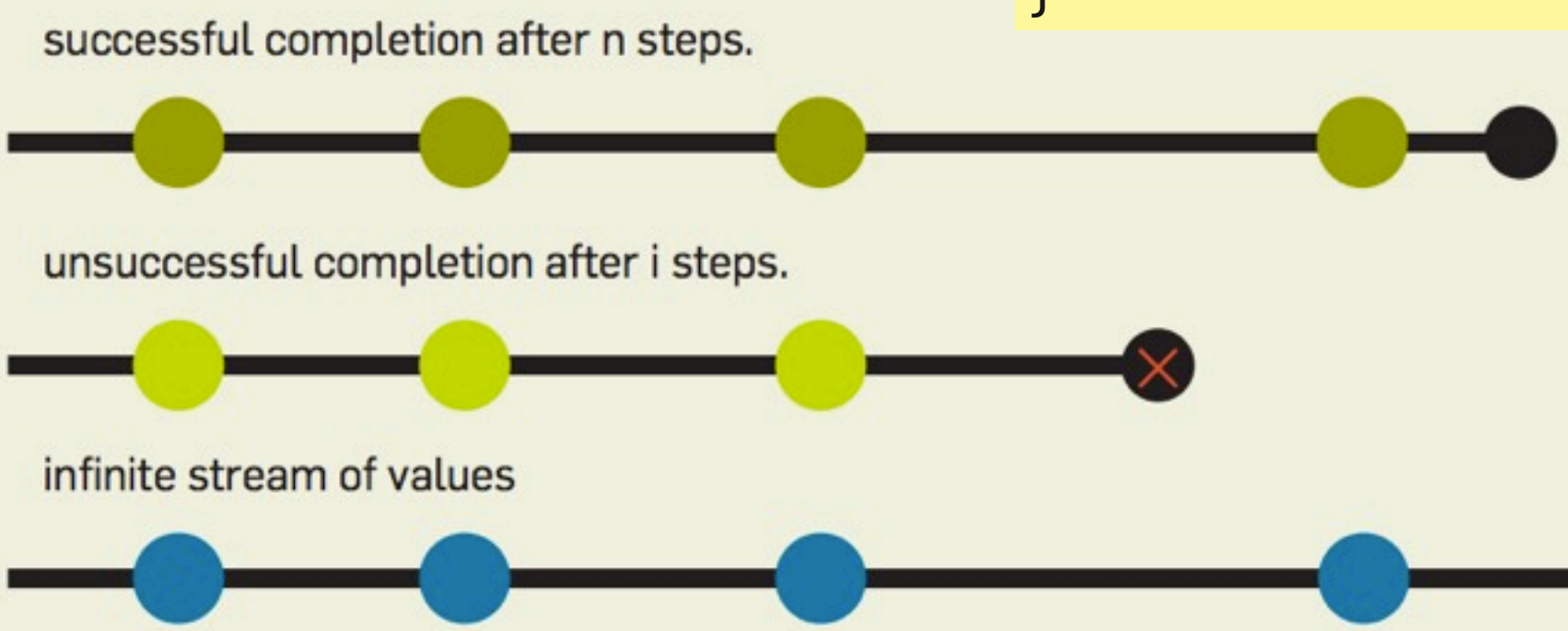
```
trait Observable[T] {
  def subscribe(obs: Observer[T]): Closable
}
```

```
trait Observer[T] {
  def onNext(v: T): Unit
  def onFailure(t: Throwable): Unit
  def onDone(): Unit
}
```

Observer[T]: Interactions

```
trait Observer[T] {
  def onNext(v: T): Unit
  def onFailure(t: Throwable): Unit
  def onDone(): Unit
}
```

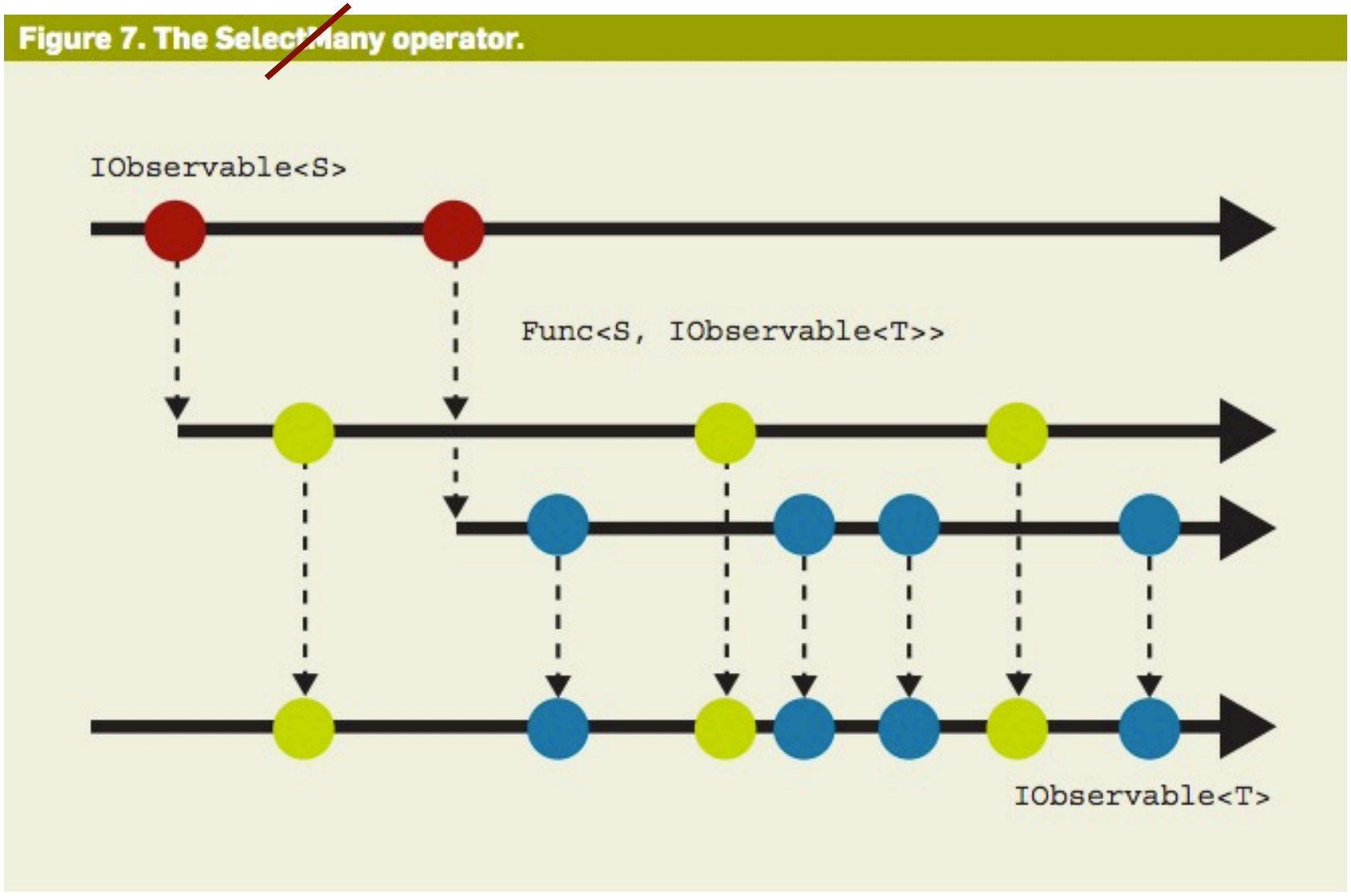
Figure 3. Possible sequences of interaction whe



Erik Meijer: Your mouse is a database. CACM'12

The Real Power: Combinators

flatMap



Combinators: Example

def textChanges(tf: JTextField):
Observable[String]

Observable[Array[String]]

```
textChanges(textField)
.flatMap(word => completions(word))
.subscribe(observeChanges(output))
```

Combinators: Example

def textChanges(tf: JTextField):
Observable[String]

Observable[Array[String]]

```
textChanges(textField)
.flatMap(word => completions(word))
.subscribe(observeChanges(output))
```

RAY makes it easy to create new combinators!

Outline

- Review Async and Rx Models
- A Challenge Problem
- RAY
- The Paper
- Conclusion

Challenge

Two input streams with the following values:

stream1: 7, 1, 0, 2, 3, 1, ...

stream2: 0, 7, 0, 4, 6, 5, ...

Task: Create a new output stream that
- yields, for each value of stream1, the sum of the previous 3 values of stream1,
- except if the sum is greater than some threshold in which case the next value of stream2 should be subtracted.

For a threshold of 5, the output stream has the following values:

output: 7, 1, 8, 3, 5, 2, ...

Solution using Rx


```
val three     = stream1.window(3).map(w => w.reduce(_ + _))
val withIndex = three.zipWithIndex

val big       = withIndex.filter(_._1 >= 5).zip(stream2).map {
  case ((l, i), r) => (l - r, i)
}

val output    = withIndex.filter(_._1 < 5).merge(big)
```

Solution using Rx


```
val three     = stream1.window(3).map(w => w.reduce(_ + _))
val withIndex = three.zipWithIndex

val big       = withIndex.filter(_._1 >= 5).zip(stream2).map {
  case ((l, i), r) => (l - r, i)
}

val output    = withIndex.filter(_._1 < 5).merge(big)
```

Requires “window” and “merge” combinators!

The Problem

- Programming with reactive streams suffers from an inversion of control
  - Requires programming in CPS
  - Example: writing stateful combinators is difficult
- Hard to use for programmers not comfortable with higher-order functions

Outline

- Review Async and Rx Models
- A Challenge Problem
- RAY
- The Paper
- Conclusion

RAY: The Idea

- Integrate Rx and Async: get the best of both worlds
- Introduce variant of `async { }` to create observables instead of futures => `rasync { }`
- Within `rasync { }`: enable *awaiting events of observables in direct-style*
- Creating observables means we need a way to yield events from within `rasync { }`

RAY: Primitives

- `rasync[T] { }` - create `Observable[T]`
- `awaitNextOrDone(obs)` - awaits and returns `Some(`next event of obs`)`, or else if obs has terminated returns `None`
- `yieldNext(evt)` - yields next event of current observable

RAY: First Example

```
val forwarder = rasync[Int] {
  var next: Option[Int] = awaitNextOrDone(stream)
  while (next.nonEmpty) {
    yieldNext(next)
    next = awaitNextOrDone(stream)
  }
}
```

Challenge: Recap

Two input streams with the following values:

stream1: 7, 1, 0, 2, 3, 1, ...

stream2: 0, 7, 0, 4, 6, 5, ...

Task: Create a new output stream that
- yields, for each value of stream1, the sum of the previous 3 values of stream1,
- except if the sum is greater than some threshold in which case the next value of stream2 should be subtracted.

For a threshold of 5, the output stream has the following values:

output: 7, 1, 8, 3, 5, 2, ...

Solution using RAY

```
val output = rasync[Int] {
  var window = List(0, 0, 0)
  var evt = awaitNextOrDone(stream1)
  while (evt.nonEmpty) {
    window = window.tail :+ evt.get
    val next = window.reduce(_ + _) match {
      case big if big > Threshold =>
        awaitNextOrDone(stream2).map(x => big - x)
      case small =>
        Some(small)
    }
    yieldNext(next)
    evt =
      if (next.isEmpty) None else awaitNextOrDone(stream1)
  }
}
```

Solution using RAY

```
val output = rasync[Int] {
  var window = List(0, 0, 0)
  var evt = awaitNextOrDone(stream1)
  while (evt.nonEmpty) {
    window = window.tail :+ evt.get
    val next = window.reduce(_ + _) match {
      case big if big > Threshold =>
        awaitNextOrDone(stream2).map(x => big - x)
      case small =>
        Some(small)
    }
    yieldNext(next)
    evt =
      if (next.isEmpty) None else awaitNextOrDone(stream1)
  }
}
```

No additional combinators required!

RAY: Summary

- Generalize Async from futures to observables
- Enables more intuitively creating and composing streams
  - No need to use higher-order functions
  - Direct-style API for awaiting stream events
- Programmers can leverage their experience with the Async model

Outline

- Review Async and Rx Models
- A Challenge Problem
- RAY
- The Paper
- Conclusion

The Paper

1. Implementation: extends the existing Async state machine translation
    - Leverage new non-blocking “FlowPools” dataflow collection (LCPC’12)
2. Operational semantics
    - Extends operational semantics of C# Async formalization (ECOOP’12)
    - High-level semantics: reasoning independent of low-level state machines

Conclusion

- RAY generalizes Async from futures to observables
- Enables more intuitively composing observables
  - No need to use higher-order functions
  - Direct-style API for awaiting observable events
- Programmers can leverage their experience with the Async model

ANOTHER EXAMPLE

```
def nameOfMonth(num: Int): Future[String] = ...
val date = """(\d+)/(\d+)""".r

async {
  await(futureDOY).body match {
    case date(month, day) =>
      Ok(s"It's ${await(nameOfMonth(month.toInt))}!")
    case _ =>
      NotFound("Not a date, mate!")
  }
}
```

BACK TO USING FOR

```
def nameOfMonth(num: Int): Future[String] = ...
val date = """(\d+)/(\d+)""".r

for { doyResponse <- futureDOY
      dayOfYear = doyResponse.body
      response <- dayOfYear match {
        case date(month, day) =>
          for (name <- nameOfMonth(month.toInt))
            yield Ok(s"It's $name!")
        case _ =>
          Future.successful(NotFound("Not a..."))
      }
} yield response
```

Haller and Zaugg, Scala Async, ScalaDays'13