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

Philipp Haller
Typesafe, Inc.
philipp.haller@typesafe.com

Heather Miller
EPFL
heather.miller@epfl.ch

ABSTRACT

Languages like F#, C#, and recently also Scala, provide "async" extensions which aim to make asynchronous programming easier by avoiding an inversion of control that is inherent in traditional callback-based programming models (for the purpose of this paper called the "Async" model). This paper outlines a novel approach to integrate the Async model with observable streams of the Reactive Extensions model which is best-known from the .NET platform, and of which popular implementations exist for Java, Ruby, and other widespread languages. We outline the translation of "Reactive Async" programs to efficient state machines, in a way that generalizes the state machine translation of regular Async programs. Finally, we sketch a formalization of the Reactive Async model in terms of a small-step operational semantics.

1. INTRODUCTION

Asynchronous programming has been a challenge for a long time. A multitude of programming models have been proposed that aim to simplify the task. Interestingly, there are elements of a convergence arising, at least with respect to the basic building blocks: futures and promises have begun to play a more and more important role in a number of languages like Java, C++, ECMAScript, and Scala.

The Async extensions of F# [12], C# [1], and Scala [6] provide language support for programming with futures (or "tasks"), by avoiding an inversion of control that is inherent in designs based on callbacks. However, these extensions are so far only applicable to futures or future-like abstractions. In this paper we present an integration of the Async model with a richer underlying abstraction, the observable streams of the Reactive Extensions model. [9] A reactive stream is a stream of *observable events* which an arbitrary number of *observers* can subscribe to. The set of possible event patterns of observable streams is strictly greater than those of futures. A stream can (a) produce zero or more regular events, (b) complete normally, or (c) complete with an error (it's even possible for a stream to never complete.) Given the richer substrate of reactive streams, the Async model has to be generalized in several dimensions.

We call our model RAY, inspired by its main constructs, `rasync`, `await`, `yield`, introduced later in the paper.

This paper makes the following contributions:

- A design of a new programming model, RAY, which integrates the Async model and the Reactive Extensions model in the context of the Scala Async project [5] (proposed for adoption in mainline Scala [6]);
- An operational semantics of the proposed programming model. Our operational semantics extends the formal model presented in [1] for C#'s async/await to observable streams.

2. BACKGROUND

2.1 Scala Async

Scala Async provides constructs that aim to facilitate programming with asynchronous events in Scala. The introduced constructs are inspired to a large extent by extensions that have been introduced in C# version 5 [7] in a similar form. The goal is to enable expressing asynchronous code in "direct style", i.e., in a familiar blocking style where suspending operations look as if they were blocking while at the same time using efficient non-blocking APIs under the hood.

In Scala, an immediate consequence is that non-blocking code using Scala's futures API [4] does not have to resort to (a) low-level callbacks, or (b) higher-order functions like `map` and `flatMap`. While the latter have great composability properties, they can appear unnatural when used to express the regular control flow of a program.

For example, an efficient non-blocking composition of asynchronous web service calls using futures can be expressed as follows in Scala:

```
1  val futureDOY: Future[Response] =
2    WS.url("http://api.day-of-year/today").get
3
4  val futureDaysLeft: Future[Response] =
5    WS.url("http://api.days-left/today").get
6
```

```
 7  futureDOY.flatMap { doyResponse =>
 8    val dayOfYear = doyResponse.body
 9    futureDaysLeft.map { daysLeftResponse =>
10      val daysLeft = daysLeftResponse.body
11      Ok("" + dayOfYear + ": " +
12              daysLeft + " days left!")
13    }
14  }
```

Line 1 and 4 define two futures obtained as results of asynchronous requests to two hypothetical web services using an API inspired by the Play! Framework (for the purpose of this example, the definition of type `Response` is unimportant).

This can be expressed more intuitively in direct-styling using Scala Async as follows (this example is adopted from the SIP proposal [6]):

```
1  val respFut = async {
2    val dayOfYear = await(futureDOY).body
3    val daysLeft = await(futureDaysLeft).body
4    Ok("" + dayOfYear + ": " +
5            daysLeft + " days left!")
6  }
```

The `await` on line 2 causes the execution of the `async` block to suspend until `futureDOY` is completed (with a successful result or with an exception). When the future is completed successfully, its result is bound to the `dayOfYear` local variable, and the execution of the `async` block is resumed. When the future is completed with an exception (for example, because of a timeout), the invocation of `await` re-throws the exception that the future was completed with. In turn, this completes future `respFut` with the same exception. Likewise, the `await` on line 3 suspends the execution of the `async` block until `futureDaysLeft` is completed.

The main methods provided by Scala Async, `async` and `await`, have the following type signatures:

```
def async[T](body: => T): Future[T]
def await[T](future: Future[T]): T
```

Given the above definitions, `async` and `await` "cancel each other out:"

```
await(async { <expr> }) = <expr>
```

This "equation" paints a grossly over-simplified picture, though, since the actual operational behavior is much more complicated: `async` typically schedules its argument expression to run asynchronously on a thread pool; moreover, `await` may only be invoked within a syntactically enclosing `async` block.

2.2 Reactive Extensions

The Rx programming model is based on two interface traits: `Observable` and `Observer`. `Observable` represents observable streams, i.e., streams that produce a sequence of events. These events can be observed by registering an `Observer` with the `Observable`. The `Observer` provides methods which are invoked for each of the kinds of events produced by the `Observable`. In Scala, the two traits can be defined as shown in Figure 1.

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

trait Observer[T] extends (Try[T] => Unit) {
  def apply(tr: Try[T]): Unit
  def onNext(v: T) = apply(Success(v))
  def onFailure(t: Throwable) = apply(Failure(t))
  def onDone(): Unit
}
```

Figure 1: The `Observable` and `Observer` traits.

The idea of the `Observer` is that it can respond to three different kinds of events, (1) the next regular event (`onNext`), (2) a failure (`onFailure`), and (3) the end of the observable stream (`onDone`). Thus, the two traits constitute a variation of the classic subject/observer pattern [3]. Note that `Observable`'s `subscribe` method returns a `Closable`; it has only a single abstract `close` method which removes the subscription from the observable. The next listing shows an example implementation.

Note that in our Scala version the `Observer` trait extends the function type `Try[T] => Unit`. `Try[T]` is a simple container type which supports heap-based exception handling (as opposed to the traditional stack-based exception handling using expressions like `try-catch-finally`.) There are two subclasses of `Try[T]`: `Success` (encapsulating a value of type `T`) and `Failure` (encapsulating an exception). Given the above definition, a concrete `Observer` only has to provide implementations for the `apply` and `onDone` methods. Since `apply` takes a parameter of type `Try[T]` its implementation handles the `onNext` and `onFailure` events all at once (in Scala, this is tyically done by pattern matching on `tr` with cases for `Success` and `Failure`).

The `Observer` and `Observable` traits are used as follows. For example, here is a factory method for creating an observable from a text input field of typical GUI toolkits (this example is adapted from [9]):

```
def textChanges(tf: JTextField): Observable[String] =
  new ObservableBase[String] {
    def subscribe(o: Observer[String]) = {
      val l = new DocumentListener {
        def changedUpdate(e: DocumentEvent) = {
          o.onNext(tf.getText())
        }
      }
      tf.addDocumentListener(l)
      new Closable() {
        def close() = {
          tf.removeDocumentListener(l)
        }
      }
    }
  }
```

This newly-defined `textChanges` combinator can be used

with other Rx combinators as follows:

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

We start with the observable created using the `textChanges` method from above. Then we use the `flatMap` combinator (called `Select` in C#) to transform the observable into a new observable which is a stream of completions for a given word (a string). On the resulting observable we call `subscribe` to register a consumer: `observeChanges` creates an observer which outputs all received events to the `output` stream. (The shown example suffers from a problem explained in [9] which motivates the use of an additional `Switch` combinator which is omitted here for brevity.)

3. THE REACTIVE ASYNC MODEL

This Section provides an (example-driven) overview of the Reactive Async Model which integrates the Async Model and the Reactive Extensions Model.

The basic idea is to generalize the Async model, so that it can be used not only with futures, but also with observable streams. This means, we need constructs that can create observables, as opposed to only futures (like `async`), and we need ways to wait for more events than just the completion of a future. Essentially, it should be possible to await all kinds of events produced by an observable stream. Analogous to `await` which waits for the completion event of a future, we introduce variations like `awaitNext` and `awaitNextOrDone` to express waiting for the events of an observable stream.

3.1 A first example

The following example shows how to await a fixed number of events of a stream in the Reactive Async Model:

```
val obs = rasync {
  var events = List[Int]()
  while (events.size < 5) {
    val event = awaitNext(stream)
    events = event :: events
  }
  Some(events)
}
```

Note that we are using the `rasync` construct; it is a generalized version of the `async` construct of Section 2.1 which additionally supports methods to await events of observable streams.

In the above example, the invocation of `awaitNext` suspends the `rasync` block until the producer of `stream` calls `onNext` on its observers. The argument of this `onNext` call (the next event) is returned as a result from `awaitNext`. The result of `rasync`, `obs`, has type `Observable[List[Int]]`. Once the body of `rasync` has been fully evaluated, `obs` publishes two events: first, an `onNext` event which carries `events` (the list with five elements), and second, an `onDone` event; it is not possible for `obs` to publish further events.

Note that the result of an `rasync` block has a type of the form `Option[T]`; in the case where this optional value is empty (`None`), only an `onDone` event is published as a result of fully evaluating the `rasync` block. (It is, however, possible to publish other events beforehand, as shown in the following sections.) Otherwise, the semantics of `rasync` is analogous to the behavior of a regular `async` block: when its body has been fully evaluated, the future, which is the result of `async`, is completed and further changes to the state of the future are impossible.

3.2 Awaiting the end of a stream

Sometimes it is not known statically how many events a stream might still publish. One might want to collect all events until the stream is done (finished publishing events). In this case it is necessary to have a way to wait for either of two events: the stream publishes a next event, or the stream is done. This can be supported using a method `awaitNextOrDone` which returns an `Option[T]` when applied to an `Observable[T]`:

```
rasync {
  var events: List[Int] = List()
  var next: Option[Int] = awaitNextOrDone(stream)
  while (next.nonEmpty) {
    events = next.get :: events
    next = awaitNextOrDone(stream)
  }
  Some(events)
}
```

In the above example, the body of `rasync` repeatedly waits for the given `stream` to publish either a next event or to reach its end, using `awaitNextOrDone`. As long as the `stream` continues to publish events (in which case `next` of type `Option[Int]` is non-empty), each event is prepended to the `events` list; this list is the single event that the observable which is, in turn, created by `rasync` publishes (once the body of `rasync` has been fully evaluated).

3.3 Creating more complex streams

The streams created by `rasync` in the previous sections are rather simple: after consuming events from other streams only a single interesting event is published on the created stream (by virtue of reaching the end of the `rasync` block). In this section, we explain how more complex streams can be created in the Reactive Async Model.

```
val forwarder = rasync[Int] {
  var next: Option[Int] =
    awaitNextOrDone(stream)

  while (next.nonEmpty) {
    yieldNext(next.get)
    next = awaitNextOrDone(stream)
  }
  None
}
```

Figure 2: A simple forwarder stream.

3.3.1 A simple forwarder stream

Suppose we would like to create a stream which simply publishes an event for each event observed on another `stream`.

In this case, the constructs we have seen so far are not sufficient, since an arbitrary number of events have to be published from within the `rasync` block. This is where the new method `yieldNext` comes in: it publishes the next event to the stream returned by `rasync`. Our simple forwarder example can then be expressed as shown in Figure 2.

Note that in the above example, the result of the body of the `rasync` block is `None`; consequently, the resulting `forwarder` stream only publishes an `onDone` event when `rasync`'s body has been fully evaluated. In this case, it is assumed that the only "interesting" non-done events of `forwarder` are published using `yieldNext`.

4. TRANSLATION

In this section we describe the translation of the introduced constructs. Before considering the translation of `rasync` blocks in combination with methods like `awaitNext`, we first give a short overview of the translation of regular `async` blocks as it is implemented in Scala Async. As before, the exposition is driven by concrete code examples.

Consider the following simple use of `async`/`await`:

```
val fut1: Future[T] = async {
  <expr1>
  val res = await(fut2)
  <expr2>
}
```

In general, `async` blocks are translated into state machines [6]. The above example is simple enough, though, that we can illustrate how it maps to Scala's futures API without the complexities associated with a state machine. For simplicity, we assume that `<expr2>` does not contain another invocation of `await`. Then, the above `async` block can be translated, intuitively, as follows:

```
val fut1: Future[T] = {
  val p = Promise[T]()
  future {
    <expr1>
    fut2 onComplete {
      case Success(res) => p.success(<expr2>)
      case Failure(t)   => p.failure(t)
    }
  }
  p.future
}
```

In the resulting program the body of the `async { ... }` block is contained within `future { ... }` which asynchronously executes it on a thread pool. Before starting this future, a new promise `p` is created. A promise is a placeholder for a result that becomes available asynchronously. A promise can be resolved either with a successful result (using the `success` method) or with an exception (using the `failure` method), *at most once.* Each promise has a future associated with it which provides a *read-only* interface to the asynchronous result. The future associated with `p` is the result of the original `async` block.

One of the critical parts of the translation is the replacement of invocations of `await`. Instead of blocking the current thread until the awaited future is completed, a completion callback is registered with the awaited future. The completion callback is invoked when `fut2` is completed; it handles two cases for the successful (`case Success(res)`) and the failed (`case Failure(t)`) completion of the future, respectively. In case of a failure, promise `p` is immediately

```
1  val forwarder = {
2    val ofp = ObservableFlowPool[T]()
3    val sm  = new StateMachine {
4      var state: Int = 0
5      var next: Option[Int] = None
6      var channels = Map[Observable, Channel[_]]()
7      var subs = List[Closable]()
8      def apply(): Unit = {
9        val c = channels.get(stream) match {
10         case None =>
11           val channel = new Channel[Option[Int]]
12           channels += (stream -> channel)
13           subs ::= stream.subscribe(new Observer {
14             def apply(tr: Try[Int]) = tr match {
15               case Success(res) =>
16                 channel.put(Some(res))
17               case Failure(t) =>
18                 ofp.failure(t)
19             }
20             def onDone() =
21               channel.put(None)
22           })
23           channel
24
25         case Some(c) =>
26           c.asInstanceOf[Channel[Option[Int]]]
27       }
28
29       c get { res =>
30         next = res
31         state = if (res.isEmpty) 2 else 1
32         resume()
33       }
34     }
35
36     def resume(): Unit = state match {
37       case 1 =>
38         ofp << next.get
39         apply()
40
41       case 2 =>
42         subs.foreach(c => c.close())
43         ofp.done()
44     }
45   }
46
47   future {
48     sm.apply()
49   }
50
51   ofp.observable
52 }
```

Figure 3: Result of reactive async translation.

completed with the exception `t` of the observed failure event. Otherwise, the rest of the `async` block, which in this case is just `<expr2>`, is executed, and its result used to complete `p` successfully. Apart from the missing state machine logic (which is not required in this simple case), this example translation does not handle exceptions that are thrown within `<expr2>`.

With this simplified overview of the translation of regular `async` blocks, we provide a sketch explaining the translation of `rasync` blocks.

4.1 Reactive Async Translation

The translation of `rasync` in combination with the `await*` and `yield*` methods is very similar to the previous translation. The main changes are the implementation of observables as the result of an `rasync` block, as well as the replacement of `await*` invocations.

Consider the forwarder example from Figure 2. Our extended translation produces the program shown in Figure 3 (simplified). The first interesting change is that instead of creating a promise, an instance of `ObservableFlowPool` is created. Like Scala's promises, flowpools [10] are a non-blocking data structure which can be completed programmatically. Instead of carrying at most one result, though, flowpools can carry an unbounded number of elements. Flowpools are observable: callbacks can be registered which are called whenever a new element is added to the flowpool. Moreover, it is possible to "seal" a flowpool which means that no more elements can be added. This event, too, can be observed; it plays the role of the `onDone` event of reactive streams. The observable associated with the flowpool is returned as the result of an `rasync` block (this is an extension of the original design of [10]).

For each observable that the `rasync` block is awaiting events from, the state machine maintains a non-blocking channel (in the `channels` map). A channel supports a non-blocking `put` operation, as well as a non- blocking `get` operation. Each invocation of `awaitNextOrDone` is translated as follows. First, we check whether there is already a channel for `stream`. If not, we create a channel and subscribe an observer which puts new events into the channel. Moreover, we call `get` on the channel, passing the current continuation. Whenever the channel receives the first event the continuation is called, which executes the corresponding state of the state machine. The continuation is (atomically) deregistered, so that the following events are just enqueued in the channel. When the consumer reaches the original `awaitNextOrDone` invocation it gets either a queued element from the channel, or, if the channel is empty, it registers its continuation once again using `get`. At the end of the `rasync` block, all collected subscriptions (`subs`) are closed; this turns all created channels into garbage.

5. FORMALIZATION

One of the contributions of this paper is a sketch of the operational semantics of the proposed programming model. Our operational semantics generalizes the formal model presented in [1]. To make it easier to pinpoint the essential semantic differences between our models, we will re-use their formal model.

$p ::= \overline{cd}\ mb$	program
$cd ::= \texttt{class}\ C\ \{\overline{fd}\ \overline{md}\}$	class declaration
$fd ::= \texttt{var}\ f : \sigma$	field
$md ::=$	method declaration
$\quad \mid \texttt{def}\ m(\overline{x : \sigma}) : \phi = mb$	sync method
$\quad \mid \texttt{def}\ m(\overline{x : \sigma}) : \psi = \texttt{rasync}\ \{\ mb\ \}$	async method
$mb ::= \{\ \overline{\texttt{var}\ x : \sigma}; \overline{e}\ \}$	method body
$\phi ::= \sigma$	return type
$\sigma, \tau ::=$	type
$\quad \mid \gamma$	value type
$\quad \mid \texttt{C}$	class type *(including a family of types* $\texttt{Observable}[\sigma]$*)*
$\gamma ::=$	value type
$\quad \mid \texttt{Boolean}$	boolean
$\quad \mid \texttt{Int}$	integer
$\psi ::= \texttt{Observable}[\sigma]$	observable return type

Figure 4: Core language syntax. C is a class name, f, m are field and method names.

$t ::= \texttt{let}\ x = e\ \text{in}\ t$	let binding
$\quad \mid x.f = y$	assignment
$\quad \mid \texttt{yieldNext}(x)$	yield event
$\quad \mid x$	variable
$e ::=$	expressions
$\quad \mid \underline{b}$	boolean
$\quad \mid \underline{i}$	integer
$\quad \mid x$	variable
$\quad \mid \texttt{null}$	null
$\quad \mid x.f$	selection
$\quad \mid x.m(\overline{y})$	invocation
$\quad \mid \texttt{new C()}$	instance creation
$\quad \mid \texttt{awaitNextOrDone}(x)$	await next event
$\quad \mid t$	term

Figure 5: RAY expressions and terms.

5.1 Syntax

Figure 4 and Figure 5 show the syntax of our core language. The core language is taken virtually unchanged from [1]. To make it more uniform with the rest of the paper we use a Scala-like syntax, however. Like in the original paper, programs are written in *statement normal form* (SNF) which forces all subexpressions to be named; this simplifies the presentation of the operational semantics. Note that our core language does not support any form of subtyping, so class declarations do not specify a superclass. This is again adopted from [1]; the presented reactive features are orthogonal to subtyping.

A RAY program consists of a collection of class definitions, as well as the definition of a (`main`) method body. A class `C` has (a possibly empty) sequence of public fields and methods, $\overline{f}$ with types σ, and $\overline{md}$, respectively. Importantly, method declarations may be either synchronous, or asynchronous. In the synchronous case, methods are public with return type ϕ, a type which may represent either a class or value type, and they may contain local variables and/or a list of expressions $\overline{e}$. Asynchronous methods on the other hand are marked with `rasync` and are expected to have return type $\texttt{Observable}[\sigma]$, but are otherwise syntactically the same as synchronous methods.

The `Observable[σ]` family of types is used to model the generic nature of observables. They represent observables such that an async method can choose to await their next event using `awaitNextOrDone`. Conversely, inside the body of an async method with result type `Observable[σ]`, `yield-Next` can be used to publish an event of type σ.

Expressions in RAY include constants, which can be either an integer i, boolean b, or the `null` literal. They may also be represented by class declarations, selections, invocations, or terms. Here, x and y represent variable names, while f ranges over field names and m ranges over method names. In order to enforce the above-mentioned SNF, we include a second syntactic category for terms.

5.2 Operational Semantics

5.2.1 Notation

A heap, denoted H, partially maps object identifiers (ranged over by o) to heap objects, denoted $\langle \mathtt{C}, FM \rangle$, representing a pair of type `C` and a field map, FM. A field map partially maps fields f to values (ranged over by v), where v can be either an integer, a boolean, `null`, or an object identifier (the address of an object in the heap).

Frames have the form $\langle L, \bar{e} \rangle^l$ where L maps local variables to their values, $\bar{e}$ is a sequence of expressions, and l is a label. A label is either s denoting a regular, synchronous frame, or $a(o)$ denoting an asynchronous frame; in this case, o is the heap address of a corresponding observable object $\langle \mathtt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D}) \rangle$. $\bar{F}$ is a set of asynchronous frames, namely, all observables that are currently suspended awaiting o to publish a new event. $\bar{D}$ is a set of so-called "dormant queues" which are explained below. Invoking a synchronous method pushes a synchronous frame to the current frame stack; invoking an asynchronous method pushes an asynchronous frame to the current frame stack.

There are three kinds of transition rules. The first kind goes from a heap and a frame to a new heap and a new frame (simple right arrow). The second kind goes from a heap and a frame stack to a new heap and a new frame stack (double right arrow). The third kind goes from a heap and a set of frame stacks to a new heap and a new set of frame stacks (squiggly right arrow).

5.2.2 Transition Rules

Rule (E-RAsync-Yield) is an extended version of rule (E-Async-Return) in [1]. It shows how awaiters (frames $\bar{F}$) are resumed when the current asynchronous frame yields a next event. An awaiter has the form $\langle L_2, \mathtt{x = y.GetResult();} \bar{s_2} \rangle^{a(o_2)}$. The call `y.GetResult()` is just a placeholder indicating that the awaiter should be resumed with the next event that stream `y` yields. Note that the state of an observable is $running(\bar{F}, \bar{D})$ as opposed to $running(\bar{F})$ in the simpler async case. The $\bar{D}$ are "dormant" queues, which are queues of awaiters that are currently not suspended, but that might call `awaitNextOrDone` again. To prevent these awaiters from skipping events, we record them in this list of dormant queues, so that new events can be queued up in the dormant queues, until `awaitNextOrDone` is called again. Another difference from the async formalization is that `GetResult` returns an option, which means upon yielding a next result, the awaiter is resumed with value $\mathtt{Some}(L(z))$.

Rule (E-RAsync-Return) applies when an asynchronous frame has been evaluated to the end (the expressions in the frame are just ϵ). In this case all awaiters $\bar{F}$ of observable o are resumed with result `None` to indicate that the observable is done publishing events. Like before, the result is added to existing dormant queues yielding $\bar{D}'$. The state of observable o is changed to $done(\bar{D}')$. As a result, subsequent calls to `awaitNextOrDone` can immediately return `None` (not shown); at the same time, events in dormant queues can be consumed until the last element, `None`, is removed from the queue.

Rule (E-AwaitNextOrDone-1) only applies if in observable o_1 (the observable of `y`) there is no dormant queue for observable o (the observable that is waiting for the next event) or the dormant queue is empty. In the former case, a new awaiter is created and added to observable o_1 in H_1. In the latter case, the empty dormant queue is removed and a regular awaiter is added to the awaiters of observable o_1.

Rule (E-AwaitNextOrDone-2) applies if the observable awaiting an event has a non- empty dormant queue; in this case, the observable can continue while taking an element out of the dormant queue.

6. RELATED WORK

There is a large body of work investigating the relationship of events and threads. Several proposals attempt to reconcile the flexibility and efficiency of event-based programming with the simpler reasoning afforded by direct- style, thread-based programming. Although similar in spirit, the present work proposes an integration of two existing programming models: async/await in the style of C# and Reactive Extensions. Consequently, we refer to other recent publications (e.g., [1]) for a discussion of events and threads, and how they relate to the async/await model.

The implementation of our RAY model is related to Scala's CPS compiler plugin [11] which provides first-class delimited continuations. However, there are important differences. Both Scala Async and RAY are implemented using the macro system [2] introduced in Scala 2.10, which is a more lightweight approach compared to a compiler plugin. The translation avoids a complex interaction with Scala's type checker; the `rasync`, `await*`, and `yield*` constructs are purely syntactic instead of type-driven. The *Scala.React* programming framework [8] builds on Scala's CPS plugin to avoid the inversion of control inherent in the observer pattern. Like other FRP frameworks it provides first-class time-varying signals which support the automatic propagation of updates to other signals. This power comes at the cost of support for concurrent signal propagation. In contrast, RAY is designed for concurrency, but does not feature first-class signals.

7. CONCLUSION

This paper proposes RAY, a programming model and macro-based library for Scala, which integrates the Async model of C# and Scala with the Reactive Extensions model. RAY supports both consuming and creating observable streams

$$\frac{\begin{array}{c}H_0(o) = \langle \texttt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D})\rangle \\ \bar{R} = \{\langle L_2[x \mapsto \texttt{Some}(L(z))], \bar{s_2}\rangle^{a(o_2)} \mid \langle L_2, \texttt{x = y.GetResult();}\bar{s_2}\rangle^{a(o_2)} \in \bar{F}\} \\ \bar{N} = \{\langle o_2, []\rangle \mid \langle \_, \_\rangle^{a(o_2)} \in \bar{R}\} \\ \bar{D'} = \{\langle o_3, \texttt{Some}(L(z)) :: q\rangle \mid \langle o_3, q\rangle \in \bar{D}\} \\ H_1 = H_0[o \mapsto \langle \texttt{Observable}[\sigma], state \mapsto running(\epsilon, \bar{D'} \cup \bar{N})\rangle]\end{array}}{\begin{array}{c}H_0, \{\langle L, \texttt{yieldNext(z);}\bar{s}\rangle^{a(o)} \circ FS\} \cup P \\ \rightsquigarrow\ H_1, \{\langle L, \bar{s}\rangle^{a(o)} \circ FS\} \cup \{R \circ \epsilon \mid R \in \bar{R}\} \cup P\end{array}} \text{(E-RAsync-Yield)}$$

$$\frac{\begin{array}{c}H_0(o) = \langle \texttt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D})\rangle \\ \bar{R} = \{\langle L_2[x \mapsto \texttt{None}], \bar{s_2}\rangle^{a(o_2)} \mid \langle L_2, \texttt{x = y.GetResult();}\bar{s_2}\rangle^{a(o_2)} \in \bar{F}\} \\ \bar{D'} = \{\langle o_3, \texttt{None} :: q\rangle \mid \langle o_3, q\rangle \in \bar{D}\} \\ H_1 = H_0[o \mapsto \langle \texttt{Observable}[\sigma], state \mapsto done(\bar{D'})\rangle]\end{array}}{\begin{array}{c}H_0, \{\langle L, \epsilon\rangle^{a(o)} \circ FS\} \cup P \\ \rightsquigarrow\ H_1, \{FS\} \cup \{R \circ \epsilon \mid R \in \bar{R}\} \cup P\end{array}} \text{(E-RAsync-Return)}$$

$$\frac{\begin{array}{c}L(y) = o_1 \\ H_0(o_1) = \langle \texttt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D})\rangle \\ \langle o, []\rangle \in \bar{D} \vee \forall \langle o_2, q\rangle \in \bar{D}.\ o_2 \neq o \\ H_1 = H_0[o_1 \mapsto \langle \texttt{Observable}[\sigma], state \mapsto running(\langle L, \texttt{x = y.GetResult();}\bar{s}\rangle^{a(o)} :: \bar{F}, \bar{D} \setminus \{\langle o, []\rangle\})\rangle]\end{array}}{\begin{array}{c}H_0, \langle L, \texttt{x = awaitNextOrDone(y);}\bar{s}\rangle^{a(o)} \circ FS \\ \twoheadrightarrow\ H_1, FS\end{array}}$$

(E-AwaitNextOrDone-1)

$$\frac{\begin{array}{c}L(y) = o_1 \\ H_0(o_1) = \langle \texttt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D})\rangle \\ \bar{D} = \bar{D'} \cup \{\langle o, q :: \texttt{Some}(e)\rangle\} \\ H_1 = H_0[o_1 \mapsto \langle \texttt{Observable}[\sigma], state \mapsto running(\bar{F}, \bar{D'} \cup \{\langle o, q\rangle\})\rangle]\end{array}}{\begin{array}{c}H_0, \langle L, \texttt{x = awaitNextOrDone(y);}\bar{s}\rangle^{a(o)} \circ FS \\ \twoheadrightarrow\ H_1, \langle L[x \mapsto e], \bar{s}\rangle^{a(o)} \circ FS\end{array}} \text{(E-AwaitNextOrDone-2)}$$

Figure 6: Transition rules for reactive async features.

in a familiar direct style, avoiding higher-order functions and low-level callbacks in many cases. Moreover, it integrates Scala's widely-adopted futures library into a unified programming model. Our goal with this integration is to simplify reactive programming with Scala, async/await, and reactive streams significantly.

8. REFERENCES

[1] G. M. Bierman, C. V. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: Formalizing asynchronous C#. In *ECOOP*, volume 7313, pages 233–257. Springer, 2012.

[2] E. Burmako. Scala macros: Let our powers combine! In *Proceedings of the 4th Workshop on Scala*. ACM, 2013.

[3] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv*, 35(2):114–131, 2003.

[4] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Scala Improvement Proposal 14 - Futures and Promises. `http://docs.scala-lang.org/sips/pending/futures-promises.html`, 2012.

[5] P. Haller and J. Zaugg. Scala Async. `https://github.com/scala/async`, 2013.

[6] P. Haller and J. Zaugg. Scala Improvement Proposal 22 - Async. `http://docs.scala-lang.org/sips/pending/async.html`, 2013.

[7] A. Hejlsberg, M. Togersen, S. Wiltamuth, and P. Golde, editors. *The C# Programming Language*. Addison-Wesley, fourth edition, 2011.

[8] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In G. Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 707–731. Springer, 2013.

[9] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[10] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In H. Kasahara and K. Kimura, editors, *LCPC*, volume 7760 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2012.

[11] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 317–328. ACM, 2009.

[12] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In R. Rocha and J. Launchbury, editors, *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011.