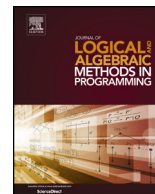


Contents lists available at [ScienceDirect](https://www.sciencedirect.com)

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp


A reduction semantics for direct-style asynchronous observables

 Philipp Haller ^{a,*}, Heather Miller ^b
^a KTH Royal Institute of Technology, Sweden

^b Carnegie Mellon University, USA


ARTICLE INFO

Article history:

Received 31 January 2016

Received in revised form 6 March 2019

Accepted 6 March 2019

Available online 18 March 2019

ABSTRACT

Asynchronous programming has gained in importance, not only due to hardware developments like multi-core processors, but also due to pervasive asynchronicity in client-side Web programming and large-scale Web applications. However, asynchronous programming is challenging. For example, control-flow management and error handling are much more complex in an asynchronous than a synchronous context. Programming with asynchronous event streams is especially difficult: expressing asynchronous stream producers and consumers requires explicit state machines in continuation-passing style when using widely-used languages like Java.

In order to address this challenge, recent language designs like Google's Dart introduce asynchronous generators which allow expressing complex asynchronous programs in a familiar blocking style while using efficient non-blocking concurrency control under the hood. However, several issues remain unresolved, including the integration of analogous constructs into statically-typed languages, and the formalization and proof of important correctness properties.

This paper presents a design for asynchronous stream generators for Scala, thereby extending previous facilities for asynchronous programming in Scala from tasks/futures to asynchronous streams. We present a complete formalization of the programming model based on a reduction semantics and a static type system. Building on the formal model, we contribute a complete type soundness proof, as well as the proof of a subject reduction theorem which establishes that the programming model enforces an important state transition protocol for asynchronous streams.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

There is an increasing need for software systems to respond to asynchronous events in an efficient and scalable way. This need is driven by developments in both hardware and software. On the hardware side, the rise of multi-core processors has made concurrent and, thereby, asynchronous programming indispensable. Furthermore, the increasing gap between the latency of main memory access and I/O [49] requires asynchronous programming techniques for scalability. On the software side, both client-side Web programming and large-scale Web applications are faced with pervasive asynchronous events [42, 13].

* Corresponding author.

E-mail addresses: phaller@kth.se (P. Haller), heather.miller@cs.cmu.edu (H. Miller).

However, asynchronous programming on a large scale is a major challenge in widely-used programming languages which are designed and optimized for synchronous computation. Handling asynchronous events is well-known to be difficult [55, 47] due to an inversion of control [37], forcing programmers to write code in continuation-passing style (CPS). To further exacerbate the situation, (a) exception handling constructs like try-catch no longer work with asynchronous code [7,42], and (b) working with *streams* of asynchronous events is even more difficult due to the manual construction of CPS state machines.

Several programming models have been proposed to simplify asynchronous programming:

- **Futures and promises.** Widely-used languages provide library-based abstractions based on futures [38] or promises [40] to mitigate some of the challenges (e.g., [14,25]). However, these abstractions do not integrate with language-provided exception handling constructs [7,42], but instead provide separate methods for error handling. Moreover, library-based abstractions, while greatly benefitting from lambda expressions [43], cannot avoid CPS completely.
- **The *async-await* model.** The *async* and *await* constructs of languages like F[#] [54] and C[#] [7] are designed to liberate programmers from writing explicit CPS code. Moreover, these constructs enable the use of exception handling facilities like try-catch also in asynchronous code. However, when programming with *streams* of asynchronous events, programmers are “left to their own devices” [42]. As a result, the behavior of stream producers and consumers must still be expressed using explicit CPS state machines. Moreover, the programming model cannot enforce API protocols typically imposed on the correct usage of asynchronous streams (e.g., [41]); API usage protocols are an important source of bugs [11,5].

In summary, programming models and constructs simultaneously addressing the variety of requirements discussed above have been missing. In a response to this, Google’s Dart language [33] has recently been extended with asynchronous functions that extend the *async-await* model to asynchronous streams [42]. The proposed *async** functions address the two main issues of the plain *async-await* model: first, producers and consumers of asynchronous streams can be written in a familiar blocking style while using efficient non-blocking concurrency control under the hood; explicit CPS code is no longer required. Second, the language constructs enforce the non-trivial usage protocol of asynchronous streams [41].

While Dart’s *async** functions are part of an original and practical language design, several important issues are left unresolved: first, given that Dart is a dynamic language, how could a similar programming model be integrated into a statically-typed language? Second, the foundations of Dart’s *async** functions are not well understood. Meijer et al. [42] define a continuation semantics for a “featherweight” subset of Dart’s asynchronous functions. However, correctness properties are neither proved nor formalized.

In this article we attempt to address both of these issues. First, we present the design of a programming model which, like Dart, extends the *async-await* model to asynchronous streams, but, unlike Dart, in a *statically-typed* language, Scala [46]. Second, we provide the first small-step reduction semantics for this “*async** model,” complete with a type soundness proof. Finally, we prove a subject reduction theorem which shows that the programming model enforces a typical usage protocol of asynchronous streams.

1.1. Contributions

This article makes the following contributions:

- A design for asynchronous stream generators for Scala, thereby extending previous facilities for asynchronous programming in Scala from tasks/futures to asynchronous streams. Unlike Dart’s asynchronous generators [42], our programming model is integrated into a *statically-typed* language.¹
- The first reduction semantics and static type system for Dart-style asynchronous generators with a complete and detailed type soundness proof.
- The proof of a subject reduction theorem which establishes, besides type preservation, that the programming model enforces an important state transition protocol for asynchronous streams. This result suggests that the presented formalization could be a suitable basis for providing a formal foundation also for widely-used APIs for asynchronous streams, such as Reactive Extensions [41] and the proposed Reactive Streams [21] JVM standard (see Section 5.3).

The rest of the paper is organized as follows. The following Section 2 provides background on futures, the *async-await* model, and the Reactive Extensions model. We present the background in the context of Scala, since the proposed programming model is designed as a Scala extension. Section 3 introduces RAY, the design of our unified asynchronous programming model. In Section 4 we present a formalization of RAY in the context of an object-based core calculus. Section 5 presents correctness results including subject reduction and type soundness. Section 6 provides an overview of our implementation as a Scala extension. Section 7 discusses related work in more detail, and Section 8 concludes.

¹ An open-source prototype implementation is available at: <https://github.com/phaller/scala-async-flow>.

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 [27] 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 [25] 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 style using Scala Async as follows (this example is adopted from the SIP proposal [26]):

```

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 invocation of the `await` pseudo-method 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 pseudo-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`

```

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 onComplete(): Unit
}

```

Fig. 1. The Observable and Observer traits.

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 Fig. 1.

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 (`onComplete`). Thus, the two traits constitute a variation of the classic subject/observer pattern [15]. 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 `onComplete` 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 typically 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 [41]):

```

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 `SelectMany` in C^\sharp) 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 [41] which motivates the use of an additional `Switch` combinator which is omitted here for brevity.)

3. The RAY asynchronous programming model

This section provides an (example-driven) overview of our asynchronous programming model called RAY. RAY can be seen as an integration of the `async-await` model of C^\sharp and the Reactive Extensions Model.

The basic idea is to generalize the `async-await` 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 a variety of events produced by an observable stream.

3.1. First example

The following first example shows how to await a fixed number of events of a stream in RAY²:

```

1  def awaitFixedNumber(stream: Observable[Int], num: Int) = rasync(stream) {
2    var events: List[Int] = List()
3    while (events.size < 5) {
4      val event = await(stream)
5      events = event.get :: events
6    }
7    events
8  }

```

The above method returns an observable by using the `rasync` construct; it is a generalized version of the `async` construct of Section 2.1 which additionally supports methods to await and yield events of observable streams. Importantly, `rasync` takes a variable number of observable arguments that the newly created observable subscribes to. In the above example, the new observable subscribes only to the `stream` observable. Note that it is not necessary to enclose the entire body of a method in an `rasync` block. However, we are using this style, so that code listings shown in this section correspond more closely to the formal model (see Section 4) where `rasync` is a modifier on method declarations. The above style is equivalent to the hypothetical use of `rasync` as a method modifier:

```

def rasync awaitFixedNumber(stream: Observable[Int], num: Int) = {
  ...
}

```

In the above example, the invocation of `await` (line 4) 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 `await`, wrapped in an optional value. Optional values are used to handle stream termination: an invocation of `await` returns an empty optional value if its argument stream terminates before publishing another event. In Scala, optional values are implemented as a simple ADT `Option[T]` with two cases `Some(x)`, a *full* option with value `x` of type `T`, and `None`, an *empty* option. The value of a full option is extracted using the `get` method (line 5). The result of `rasync` has type `Observable[List[Int]]`. Once the body of `rasync` has been fully evaluated, the created observable publishes two final events: first, an `onNext` event which carries `events` (the list with five elements), and second, an `onComplete` event. In this example, these are the only published events; it is, however, possible to publish other events beforehand, as shown in the following.

The following is an improved version of the above example where it is not necessary to know statically how many events a stream might publish. Stream termination is detected using the optional value returned by `await`:

```

1  def awaitTermination(stream: Observable[Int]) = rasync(stream) {
2    var events: List[Int] = List()
3    var next: Option[Int] = await(stream)
4    while (next.nonEmpty) {
5      events = next.get :: events
6      next = await(stream)
7    }
8    events
9  }

```

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. As long as `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).

² For consistency with the formal model of Section 4 we use the name `Observable` to refer to the same type that is called `Publisher` in our implementation.

```

1  def forwarder[T <: AnyRef](stream: Observable[T]) = rasync(stream) {
2    var next: Option[T] = await(stream)
3    var last: Option[T] = None
4
5    while (next.nonEmpty) {
6      last = await(stream)
7      if (last.nonEmpty) {
8        yieldNext(next.get)
9        next = last
10     } else {
11       last = next
12       next = None
13     }
14   }
15
16   last.get
17 }

```

Fig. 2. A simple forwarder stream.

```

1  def forwarder2[T <: AnyRef](stream: Observable[T]) = rasync(stream) {
2    var next: Option[T] = await(stream)
3    while (next.nonEmpty) {
4      yieldNext(next.get)
5      next = await(stream)
6    }
7    yieldDone()
8  }

```

Fig. 3. A simpler forwarder stream.

3.2. Creating 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 RAY.

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 Fig. 2.

Note that in the above example, we are forced to remember the last event that the created observable should publish before terminating, `last.get` (line 16). It would be much simpler to terminate the created observable as soon as there are no more events to publish. This pattern is supported using the new method `yieldDone`: invoking `yieldDone` emits an `onComplete` event and then terminates the current observable.

Fig. 3 shows how the forwarder example can be simplified significantly using `yieldDone`. In this new version, the created observable is terminated unconditionally on line 7 by invoking `yieldDone`. As a result, it is not necessary to maintain a last event to be published after finishing the iteration. In order to enable the use of `yieldDone` for any type of observable, `yieldDone` has a generic type:

```
def yieldDone[A](): A = ...
```

Note that invoking `yieldDone` publishes an `onComplete` event, and then discards the continuation until the end of the `rasync` block.

A challenge problem. In order to illustrate the expressive power of RAY in comparison with the Reactive Extensions model introduced earlier, consider the following “challenge problem.” Given two input streams `stream1` and `stream2`, our task is to create a new output stream that

- yields, for each value of `stream1`, the sum of the previous three values of `stream1`,
- except if the sum is greater than some threshold, in which case the next value of `stream2` should be subtracted.

Note that this composition of `stream1` and `stream2` is *stateful* and involves conditional waiting for, and publishing of, asynchronous events. Before attempting a solution to this challenge, consider the following example input streams and the expected output stream for a threshold of 5:

```

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

val withIndex = three.zipWithIndex

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

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

```

Fig. 4. Challenge solution using Reactive Extensions.

```

val output = rasync[Int](stream1, stream2) {
  var window = List(0, 0, 0)
  var evt = await(stream1)
  while (evt.nonEmpty) {
    window = window.tail :+ evt.get
    val next = window.reduce(_ + _) match {
      case big if big > Threshold =>
        await(stream2).map(x => big - x)

      case small => Some(small)
    }
    evt = if (next.isEmpty) None else { yieldNext(next.get); await(stream1) }
  }
  yieldDone()
}

```

Fig. 5. Challenge solution using RAY.

```

stream1:  7, 1, 0, 2, 3, 1, ...
stream2:  0, 7, 0, 4, 6, 5, ...
output:   7, 1, 8, 3, 5, 2, ...

```

The first value of the output stream is 7: the sum of the previous three values of `stream1` is 7, which is greater than the threshold 5; thus, the next value of `stream2`, which is 0, is subtracted, giving 7. The second value of the output stream is 1: the sum of the previous three values of `stream1` is 8, which is greater than the threshold 5; thus, the next value of `stream2`, which is 7, is subtracted, giving 1; and so on.

We first discuss a solution to this challenge problem using the Reactive Extensions model introduced in Section 2.2; Fig. 4 shows our solution. First, we define a stream `three` which uses a `window` combinator, as well as `map` and `reduce` (on regular arrays) to produce a stream that carries the sum of the previous three values of `stream1`. (The result of invoking `window` is a stream of arrays of size 3.) Then, `withIndex` is simply a stream which pairs each event of stream `three` with its index; this index is used to control the ordering of events subsequently. The `big` stream carries the numbers greater than the threshold reduced by the numbers of `stream2` paired with their index. This index is used in combination with the `merge` combinator (last line) to “insert” those numbers at the right positions when merging with those numbers of `three` that are smaller than the threshold.

In summary, the solution using Reactive Extensions:

- requires complex combinators like `window` and `merge`;
- requires manually ensuring the correct event ordering through explicit management of event indices;
- uses several higher-order functions and lambdas.

The use of several combinators and lambdas indicates that this style of programming is negatively affected by the same inversion of control that motivated the introduction of the `async-await` model in languages like C[#] in the first place. In particular, writing stateful stream combinators is difficult. As a result, Reactive Extensions is likely hard to use for programmers not comfortable with higher-order functions.

Fig. 5 shows a solution to the challenge problem using RAY. Given the `rasync`, `await`, and `yieldNext` constructs introduced earlier, the body of the `rasync` call, which defines the behavior of the output stream can be understood sequentially. The state of the composition logic is managed using a regular variable `window`, which is simply a list of integers. As long as `stream1` carries more values, the `window` is updated. Then, the sum of the window is computed to decide whether a next event should be received from `stream2`. Note that `await` which suspends awaiting the next event from `stream2` is simply called from within a nested pattern matching block within the while loop. As soon as the next value for the output stream is computed, it is published using `yieldNext`.


```

1 def sequenceEqual[T](src1: Observable[T], src2: Observable[T])
2   (compare: (T, T) => Boolean): Observable[Boolean] =
3   rasync(src1, src2) {
4     var result = false
5     var done = false
6
7     while (!done) {
8       val opt1 = await(src1)
9       val opt2 = await(src2)
10      if (opt1.isEmpty && opt2.isEmpty) {
11        result = true
12        done = true
13      } else if (opt1.nonEmpty && opt2.nonEmpty) {
14        if (!compare(opt1.get, opt2.get))
15          done = true
16      } else {
17        done = true
18      }
19    }
20
21    result
22  }

```

Fig. 6. The sequenceEqual combinator.

In summary, the solution using RAY:

- does not require additional complex combinators like `window` or `merge`;
- does not require the explicit management of event ordering using indices as before;
- significantly reduces the importance of higher-order functions;
- enables a simple expression of custom higher-level abstractions (like `zip`), reducing the need for large third-party libraries (the following Section shows two examples).

3.3. Implementing combinators

As shown in the previous section, the Reactive Extensions model gains a lot of flexibility through higher-order functions, or combinators, that combine and transform observables. However, the *implementation* of such combinators can be surprisingly challenging, due to complex internal state machines whose state transitions are triggered by callbacks. Thus, one promising use case of RAY's direct-style constructs is the implementation of such high-level combinators. This section discusses two such combinators.

The sequenceEqual combinator. The first combinator is called `sequenceEqual`: given two observables of type `Observable[T]` and a comparison function of type `(T, T) => Boolean`, the `sequenceEqual` combinator returns an observable of type `Observable[Boolean]` which publishes a single Boolean indicating whether the sequences of events published by the two parameters are equal. Fig. 6 shows a simple implementation using RAY. Essentially, the created observable loops, awaiting the next event of each observable; as long as the event sequences are not shown to be unequal, the iteration continues. The direct-style nature of `await` allows the implementation to focus on just a single pair of optional values at a time, while other events are being buffered in the background.

The zip combinator. The second combinator is called `zip`: given a collection of observables of type `Observable[T]` and a zipper function of type `Array[T] => R`, the `zip` combinator returns an observable of type `Observable[R]` where each `R` event is computed as follows. Whenever each of the observables in the argument collection has published a next event, an array of these events (one event per observable) is passed to the provided zipper function to compute a value of type `R`. This `R` value is published as the next event of the observable returned by `zip`.

Fig. 7 shows an implementation of `zip` using RAY. The state machine needed for `zip` is similar to that of `sequenceEqual`. The two main differences are: (1) the observable created by `zip` in general publishes more than one event, and (2) `zip` combines an arbitrary number of observables (the size of the `sources` collection is unknown). Despite these two fundamental generalizations compared to `sequenceEqual`, the implementation of `zip` is not markedly more complex. The use of `yieldNext` is straightforward. More interesting is how to deal with an unbounded number of input observables. Here, the direct-style nature of `await` helps us by allowing us to iterate over the input observables using a standard `while` loop (lines 12–15), awaiting the next event or termination (line 13), in each iteration. As before, buffering of events being published while suspended in an `await` invocation is performed automatically.³

³ The syntax `rasync(sources: _*)` on line 3 converts the `sources` collection to repeated arguments, as expected by `rasync`.


```

1 def zip[T, R](sources: Seq[Observable[T]],
2             zipper: Array[T] => R): Observable[R] =
3   rasync(sources: _*) {
4     val sourcesArr: Array[Observable[T]] = sources.toArray
5     var done = false
6     var lastProduced: Option[R] = None
7
8     while (!done) {
9       // await next event of each source
10      val nextEvents = Array.ofDim[Option[T]](sourcesArr.length)
11      var i = 0
12      while (i < sources.size) {
13        nextEvents(i) = await(sourcesArr(i))
14        i += 1
15      }
16      if (nextEvents.exists(_.isEmpty)) {
17        done = true
18      } else {
19        val arg = nextEvents.map(_.get).toArray
20        if (lastProduced.nonEmpty)
21          yieldNext(lastProduced.get)
22        lastProduced = Some(zipper(arg))
23      }
24    }
25
26    lastProduced.get
27  }

```

Fig. 7. The zip combinator.

```

1 val source = new CancellationTagSource
2 val ct = source.mkTag // create cancellation tag
3 // pass cancellation tag 'ct' to newly created stream:
4 val s = rasync[Int](ct) {
5   await(delay(100))
6   yieldNext(5)
7   ...
8 }
9 ...
10 ct.cancel()

```

Fig. 8. Cancellation of a stream.

3.4. Selective queueing

The implementations of the `sequenceEqual` and `zip` combinators above leverage the fact that an `rasync` block by default buffers all events emitted by streams to which the `rasync` block is subscribed. In some cases, however, the incoming events of certain streams should be dropped while waiting for other events, in order to reduce memory consumption of buffers, or to implement desired runtime semantics. Therefore, RAY provides the following two variants of `await`: `awaitIgnore(stream1, stream2)` waits for the next event from `stream1` while ignoring all values produced by `stream2`; `awaitOnly(stream1)` waits for the next event from `stream1` while ignoring all values produced by any other stream to which the current `rasync` block is subscribed.

3.5. Cancellation

Stream-producing `rasync` blocks may be long-running; `rasync` may be arbitrarily complex and may include `while` loops among others. However, just like with asynchronous tasks, other parts of the program may determine that the events produced by a running stream are no longer needed. In this case, the stream may be cancelled. In our model, cancellation is supported using a special kind of runtime tags: a *cancellation tag* is a runtime capability to cancel one or more streams. Cancellation tags are an adaptation of F^{\sharp} 's cancellation tokens [54]. In the following we summarize how cancellation tags enable cancellation of streams in RAY.

Cancellation tags are unforgeable, and can only be created using cancellation tag sources. To enable cancelling a stream `s`, a cancellation tag must be passed to the creator of `s`. This is illustrated in Fig. 8.

Using a cancellation tag, a stream created as shown in Fig. 8 may be cancelled by invoking the cancellation tag's `cancel` method (line 10). Note that calling `cancel` does not forcibly terminate the stream; it merely signals a request to cancel the stream. Streams check for the presence of cancellation requests whenever RAY operators are invoked. For example, suppose

that `ct.cancel()` on line 10 is executed after the stream suspends using the `await` on line 5. Then, stream `s` notices the cancellation request upon executing `yieldNext(5)` on line 6. As a result of cancelling stream `s`, downstream subscribers receive an `onError` event with a special `CancellationException` that has a reference to the cancellation tag. This enables subscribers to decide how to respond to the cancellation, also based on the identity of the cancellation tag.

Resource clean-up. Like Scala Async, RAY is designed to enable the use of `try-catch-finally` within `rasync` blocks. In particular, it is valid to invoke `await` within the body of `try`; for example:

```

1  def m(s: Observable[Int]) = rasync(s) {
2    e1
3    try {
4      e2
5      val x = await(s)
6      e3
7    } finally {
8      // clean up
9    }
10   e4
11  }
```

Suppose an exception is thrown within expression `e2`, and the closest enclosing `catch` or `finally` is the one on line 7. Then, control is transferred to the body of `finally` (line 8), even though the `finally` clause appears “after” the invocation of `await` on line 5. The rationale behind this behavior is that programmers should be able to reason about the behavior of `await` as if it was a blocking method. It is also possible that the invocation of `await(s)` (line 5) throws an exception, namely when the observable `s` publishes an `onError` event. Also in that case, control is transferred to the body of `finally` (line 8).

3.6. Discussion

The RAY programming model, as introduced above, occupies a point in the design space which is different from related programming models in important ways. The asynchronous programming features of the Dart language [8] and the F^\sharp AsyncSeq [18] library constitute the most closely related work. In the following we discuss these relationships in more detail, as well as design alternatives.

Asynchronous push-based reactive programming. Dart supports creating and consuming asynchronous streams using dedicated language constructs, namely `async*`, `await`, and `yield`, which are similar to the constructs of RAY. The main difference between RAY and Dart is the fact that in Dart, subscribers have the possibility to *pause and resume* streams. When subscribing to a stream, a `StreamSubscription` object is returned which exposes `pause` and `resume` methods. As long as the subscription is paused, it does not emit any events. In contrast, in RAY, subscriptions by default buffer all incoming events, unless `awaitOnly` and `awaitIgnore` are used which enable ignoring (*i.e.*, dropping) events from certain streams while waiting for a specific next event (see Section 3.4). This means that event emission is resumed as soon as an invocation of `awaitOnly` or `awaitIgnore` returns. In Dart, a paused subscription must be resumed explicitly either using a “resume signal” or using the `StreamSubscription`’s `resume` method.

Despite these differences, both Dart and RAY provide an *asynchronous push-based* reactive programming model. Both models are push-based in the sense that consumers have no control over the generation of events, unless this generation is paused (Dart), or events from certain streams are dropped (RAY): events are not retrieved by the consumer on its own terms, but they are pushed downstream by the publisher.

An alternative to RAY’s `awaitOnly` and `awaitIgnore` methods for controlling event buffering would be to (a) use explicit subscription objects, and (b) expose methods for enabling and disabling event buffering. In fact, this would only require exposing parts of the existing implementation (currently utilized by `awaitOnly` and `awaitIgnore`) behind an extended public interface for subscriptions.

A further extension would be to expose methods to request upstream publishers to buffer events until further notice. In turn, such an upstream publisher could request its own upstream publishers to buffer events, thereby realizing a form of flow control.

Asynchronous pull-based reactive programming. The F^\sharp AsyncSeq [18] library provides a sequence abstraction where individual elements are retrieved asynchronously, on demand. Similar to RAY, the library provides syntactic sugar analogous to `rasync` blocks based on computation expressions [50]. Thus, AsyncSeq shows that computation expressions are sufficient to provide this syntactic sugar; the full expressive power of Scala’s macro system is not necessary for realizing a comparable programming model. However, computation expressions are not flexible enough to generate efficient state machines as described in Section 6, and they are restricted to a sublanguage of F^\sharp , whereas RAY is designed to enable using its constructs in regular Scala code.

$p ::= \overline{cd} \ mb$	Program
$cd ::= \text{public class } C \{ \overline{fd} \ \overline{md} \}$	Class declaration
$fd ::= \text{public } \sigma \ f ;$	Field declaration
$md ::= \text{public } \phi \ m(\overline{\sigma} \ \overline{x}) \ mb \mid \text{rasync public } \psi \ m(\overline{\sigma} \ \overline{x}) \ mb$	Method declaration
$mb ::= \{ \overline{\sigma} \ \overline{x}; \overline{s} \}$	Method body
$\phi ::= \sigma \mid \text{void}$	Return type
$\sigma, \tau ::= \gamma \mid \rho$	Type
$\gamma ::= \text{bool} \mid \text{int} \mid \text{Option}\langle\sigma\rangle$	Value type
$\rho ::= C \mid \text{Observable}\langle\sigma\rangle$	Reference type
$\psi ::= \text{Observable}\langle\sigma\rangle$	Observable return type

Fig. 9. Programs and types. C ranges over class names, f , m , \overline{x} range over identifiers.

F^\sharp AsyncSeq implements an *asynchronous pull* semantics: in contrast to RAY, elements are not asynchronously pushed to consumers, but instead they are asynchronously pulled by consumers. Note that it is possible to convert between the two styles. A pull-based source can be converted to a push-based source by first pulling, automatically, and pushing events to downstream consumers as they become available. Conversely, a push-based event source can be converted to a pull-based source by buffering its output until pulled by the consumer. While such conversions are possible in principle, they do not always preserve non-functional properties such as (soft real-time) responsiveness.

3.7. Summary

In this section we have presented RAY, an asynchronous programming model which generalizes the async-await model from tasks/futures to observables. RAY enables an intuitive coordination and composition of streams, with important properties: first, there is no need to use higher-order functions; second, RAY provides a direct style API for awaiting stream events; finally, programmers can leverage their experience using the async-await model known from C^\sharp and F^\sharp .

4. Formalization

We formalize the introduced programming model in the context of an imperative, object-based core language. Our focus is modeling concurrent stream producers and consumers using the `rasync`, `await`, and `yieldNext` constructs informally introduced in Section 3. Therefore, the operational model includes (a) concurrently reduced frame stacks (“threads”) and (b) a shared heap that is used to communicate asynchronous events between frame stacks. Orthogonal features of the core language, such as classes, are as simple as possible. Concretely, our core language extends Featherweight C^\sharp [7], or FC_5^\sharp , which has been used to formalize the features for asynchronous programming in C^\sharp 5.0 (`async/await`). The principles of this core language are similar to well-known, class-based core languages, including Classic Java [17], MJ [6], Creol [36], ABS [35], and Welterweight Java [48]. Consequently, we do not expect any challenges integrating the presented asynchronous features into more complex core languages.

Notation. In our formalization we adopt the overbar notation of Featherweight Java [32]; *i.e.*, we write \overline{x} for a possibly empty sequence x_1, \dots, x_n and ϵ for the empty sequence. We also use the overbar notation to abbreviate operations and expressions involving multiple sequences. For example, we write $\overline{x} : \overline{\sigma}$ for the (possibly empty) sequence $x_1 : \sigma_1, \dots, x_n : \sigma_n$. Similarly, we write $\overline{f} \mapsto L(\overline{y})$ for the sequence $f_1 \mapsto L(y_1), \dots, f_n \mapsto L(y_n)$. Finally, we write $L(\overline{y}) = \overline{p}$ for the sequence $L(y_1) = p_1, \dots, L(y_n) = p_n$.

4.1. Syntax

Figs. 9 and 10 show the syntax of our core language. A RAY program consists of a sequence of class definitions, \overline{cd} , as well as the body of a “main” method, mb . A class C has (a possibly empty) sequence of fields, \overline{fd} , and methods, \overline{md} . Note that our core language does not support any form of subtyping; thus, class declarations do not specify a superclass. This is adopted from [7]; the presented asynchronous features are orthogonal to subtyping.

Types, ranged over by σ, τ , are either value types γ or reference types ρ . Value types only include `bool`, `int`, and the built-in generic type `Option` $\langle\sigma\rangle$. The type `Option` $\langle\sigma\rangle$ is the type of optional values which have one of two forms: `None` $\langle\sigma\rangle$, an empty optional value, or `Some`(v), a “full” optional value containing value v . Reference types are either class types C , introduced by class definitions, or instances of the built-in generic type `Observable` $\langle\sigma\rangle$. The type `Observable` $\langle\sigma\rangle$ is the type of observables, *i.e.*, asynchronous event streams. An object of observable type produces exactly one stream of events which can be consumed by other observables. Conversely, an observable can choose to subscribe to several other observables in order to consume their events. Methods marked with modifier `rasync` must have an observable return type ψ .

In order to simplify the presentation of the operational semantics, programs are written in *statement normal form* [7] (SNF) which requires all subexpressions to be named. Apart from their syntax in SNF, most statements and expressions are standard, including expressions for Boolean and integer constants, variables, conditionals, field selection, field assignment,

$e ::=$	Expressions
c	Constant
$ x$	Variable
$ x \oplus y$	Built-in operator
$ x.f$	Field selection
$ x.m(\bar{y})$	Method invocation
$ \text{new } C()$	Object creation
$ \text{Some}(x)$	Full optional value
$ \text{None}\langle\sigma\rangle$	Empty optional value
$ \text{get } x$	Get optional value
$ \text{await } x$	Await expression
$s, t ::=$	Statements
$x = e;$	Assignment
$ \text{if } (x) \{ \bar{s} \} \text{ else } \{ \bar{t} \}$	Conditional
$ \text{while } (x) \{ \bar{s} \}$	Iteration
$ x.f = y;$	Field assignment
$ x.m(\bar{y});$	Method invocation statement
$ \text{return};$	Return statement
$ \text{return } x;$	Return value statement
$ \text{yieldNext } x;$	Yield value statement
$ \text{yieldDone}();$	Yield finish statement

Fig. 10. Expressions and statements.

$S ::=$	object state:
FM	field map
$ ST$	running state

Fig. 11. Heap object state.

method invocations, and instance creation. The expression $\text{Some}(x)$ creates a non-empty optional value containing x . The expression $\text{None}\langle\sigma\rangle$ creates an empty optional value; the type argument σ is used in the corresponding type rule to assign the expression type $\text{Option}\langle\sigma\rangle$. The expression $\text{get } x$ extracts the value contained in an optional value x (if any). Note that attempting to extract a value from an empty option causes reduction to get stuck; however, this is an allowed stuck state in our system (see case (f) in Theorem 2). Our type system is not intended to prevent incorrect option usages. The yieldNext and yieldDone statements and the await expression are new: $\text{yieldNext } x$ asynchronously emits event x to subscribers of the current observable; $\text{yieldDone}()$ emits a “done” event to subscribers and terminates the current observable; $\text{await } x$ awaits the next event emitted by observable x .

4.2. Dynamic semantics

We define the dynamic semantics of the core language using a small-step operational semantics. The operational semantics is based on three different transition relations for (a) frames, (b) frame stacks, and (c) processes. A frame $\langle L, \bar{s} \rangle^l$ combines a sequence of statements \bar{s} with a variable environment L which maps the free variables in \bar{s} (if any) to their values. The frame label l distinguishes between synchronous and asynchronous frames. For now, we only consider synchronous frames where $l = s$. (Asynchronous frames are discussed below in Section 4.2.2.) A frame stack $F \circ FS$ models a “thread” in our system; the stack of frames is used for both (a) method call/return transitions and (b) creation of asynchronous observables. Finally, a process is a set of frame stacks. Reduction of processes is defined using an interleaving semantics: at each step, the next frame stack to reduce is chosen non-deterministically.

All transition relations include a heap (or store [51]). The heap is necessary for two orthogonal aspects: first, mutation of regular objects; second, communication of asynchronous events between concurrent frame stacks.

Definition 4.1 (Heap). $H \in \text{Oid} \rightarrow \rho \times S$.

A heap, denoted H , partially maps object identifiers $o \in \text{Oid}$ to heap objects $\langle \rho, S \rangle$, pairs of a reference type and an object state. An object state (see Fig. 11) is either a field map FM or a *running state* ST (running states are defined in Section 4.2.2 below). A field map partially maps fields f to values, ranged over by v , where v can be either an integer, a Boolean, an empty option ($\text{None}\langle\sigma\rangle$), a full option ($\text{Some}(v)$), or an object identifier.

4.2.1. Synchronous transition rules

Fig. 12 shows simple frame transition rules. We use \longrightarrow to denote the transition relation for single frames. Note that all transition rules preserve the labels of frames. Rule E-CONSTANT updates variable x in local variable mapping L to map to constant c ; reduction continues with statements \bar{s} . Rule E-VAR looks up the value $L(y)$ of variable y in mapping L . The local

$$\begin{array}{l}
H, \langle L, x=c; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto c], \bar{s} \rangle^l \quad (\text{E-CONSTANT}) \\
H, \langle L, x=y; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto L(y)], \bar{s} \rangle^l \quad (\text{E-VAR}) \\
H, \langle L, x=y \oplus z; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto L(y) \oplus L(z)], \bar{s} \rangle^l \quad (\text{E-OP}) \\
\frac{H(L(y)) = \langle C, FM \rangle \quad f \in \text{dom}(FM)}{H, \langle L, x=y.f; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto FM(f)], \bar{s} \rangle^l} \quad (\text{E-FIELD}) \\
\frac{\bar{r} = \begin{cases} \bar{s} \bar{u} & \text{if } L(x) = \text{true} \\ \bar{t} \bar{u} & \text{otherwise} \end{cases}}{H, \langle L, \text{if } (x) \{ \bar{s} \} \text{ else } \{ \bar{t} \} \bar{u} \rangle^l \longrightarrow H, \langle L, \bar{r} \rangle^l} \quad (\text{E-CONDEQ}) \\
\frac{\bar{r} = \begin{cases} \bar{s} \text{ while } (x) \{ \bar{s} \} \bar{t} & \text{if } L(x) = \text{true} \\ \bar{t} & \text{otherwise} \end{cases}}{H, \langle L, \text{while } (x) \{ \bar{s} \} \bar{t} \rangle^l \longrightarrow H, \langle L, \bar{r} \rangle^l} \quad (\text{E-WHILE}) \\
\frac{L(x) = o \quad H_0(o) = \langle C, FM \rangle \quad f \in \text{dom}(FM) \quad H_1 = H_0[o \mapsto \langle C, FM[f \mapsto L(y)] \rangle]}{H_0, \langle L, x.f=y; \bar{s} \rangle^l \longrightarrow H_1, \langle L, \bar{s} \rangle^l} \quad (\text{E-ASN}) \\
\frac{\text{fields}(C) = \bar{c} \bar{f} \quad o \notin \text{dom}(H_0) \quad H_1 = H_0[o \mapsto \langle C, \bar{f} \mapsto \text{default}(\bar{c}) \rangle]}{H_0, \langle L, x=\text{new } C (); \bar{s} \rangle^l \longrightarrow H_1, \langle L[x \mapsto o], \bar{s} \rangle^l} \quad (\text{E-NEW})
\end{array}$$

Fig. 12. Simple frame transition rules.

$$\begin{array}{l}
H, \langle L, x=\text{Some}(y); \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto \text{Some}(L(y))], \bar{s} \rangle^l \quad (\text{E-SOME}) \\
H, \langle L, x=\text{None}\langle \sigma \rangle; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto \text{None}\langle \sigma \rangle], \bar{s} \rangle^l \quad (\text{E-NONE}) \\
\frac{L(y) = \text{Some}(v)}{H, \langle L, x=\text{get } y; \bar{s} \rangle^l \longrightarrow H, \langle L[x \mapsto v], \bar{s} \rangle^l} \quad (\text{E-GET})
\end{array}$$

Fig. 13. Frame transition rules for options.

$$\begin{array}{l}
\frac{H, F \longrightarrow H', F'}{H, F \circ FS \longrightarrow H', F' \circ FS} \quad (\text{E-FRAME}) \\
\frac{H(L(y)) = \langle \rho, FM \rangle \quad \text{mbody}(\rho, m) = \text{mb} : (\bar{\sigma} \bar{x}) \rightarrow^s \sigma_1, \text{mb} = \bar{c} \bar{y}; \bar{t} \\ L' = [\bar{x} \mapsto L(\bar{z}), \bar{y} \mapsto \text{default}(\bar{c}), \text{this} \mapsto L(y)]}{H, \langle L, x=y.m(\bar{z}); \bar{s} \rangle^l \circ FS \longrightarrow H, \langle L', \bar{t} \rangle^s \circ \langle L, \bar{s} \rangle_x^l \circ FS} \quad (\text{E-METHOD-EXP}) \\
\frac{H(L(x)) = \langle \rho, FM \rangle \quad \text{mbody}(\rho, m) = \text{mb} : (\bar{\sigma} \bar{x}) \rightarrow^s \sigma_1, \text{mb} = \bar{c} \bar{y}; \bar{t} \\ L' = [\bar{x} \mapsto L(\bar{y}), \bar{y} \mapsto \text{default}(\bar{c}), \text{this} \mapsto L(x)]}{H, \langle L, x.m(\bar{y}); \bar{s} \rangle^l \circ FS \longrightarrow H, \langle L', \bar{t} \rangle^s \circ \langle L, \bar{s} \rangle^l \circ FS} \quad (\text{E-METHOD-STMT}) \\
H, \langle L, \text{return } y; \bar{s} \rangle^s \circ \langle L', \bar{t} \rangle_x^l \circ FS \longrightarrow H, \langle L'[x \mapsto L(y)], \bar{t} \rangle^l \circ FS \quad (\text{E-RETURN-VAL}) \\
H, \langle L, \text{return}; \bar{s} \rangle^s \circ \langle L', \bar{t} \rangle^l \circ FS \longrightarrow H, \langle L', \bar{t} \rangle^l \circ FS \quad (\text{E-RETURN})
\end{array}$$

Fig. 14. Synchronous method call/return transition rules.

variable mapping of the target frame maps x to $L(y)$. Rule E-OP is analogous. Rule E-FIELD looks up the value of field $y.f$ using L and H ; as before, reduction continues with statements \bar{s} . Rules E-CONDEQ and E-WHILE are straightforward; they are adopted unchanged from Featherweight C^\sharp [7], the basis for our formal model. Rule E-ASN combines the heap H , local variable mapping L , and field mapping FM in the natural way for field assignment. Rule E-NEW creates a new instance of class C , assigning type-specific default values to the fields of the new instance. For a type τ , the default value $\text{default}(\tau)$ is defined in the obvious way, such that $\text{default}(\text{Option}\langle \sigma \rangle) = \text{None}\langle \sigma \rangle$ and $\text{default}(\rho) = \text{null}$ if ρ is a reference type.

Fig. 13 shows the frame transition rules for options. Rule E-SOME creates a “full” option instance. Analogously, rule E-NONE creates an “empty” option instance. Rule E-GET extracts the wrapped value of a full option instance.

Fig. 14 shows the transition rules for method call and return. We use \rightarrow to denote the transition relation for frame stacks (\longrightarrow transitions single frames). Rule E-FRAME transitions a *frame stack* $F \circ FS$ by transitioning frame F . Rule E-METHOD-EXP evaluates a result-returning method invocation. The run-time type of the receiver, ρ , is looked up in heap H . Using the auxiliary function mbody we look up the body of method m in ρ . Note that this transition rule only applies if the method is a *synchronous* method (i.e., not marked with modified `async`); this requirement is indicated using function arrow \rightarrow^s in the type of the method. To evaluate the method body, a new frame with synchronous label s is created and pushed

$$\begin{array}{c}
H(L(y)) = \langle \rho, FM \rangle \quad mbody(\rho, m) = mb : (\bar{\sigma} \bar{x}) \rightarrow^a \psi, mb = \bar{\tau} \bar{y}; \bar{t} \\
L' = [\bar{x} \mapsto L(\bar{z}), \bar{y} \mapsto default(\bar{\tau}), this \mapsto L(y)] \\
H'' = H[o \mapsto \langle \psi, running(\epsilon, \epsilon) \rangle] \quad o \notin dom(H) \\
\bar{p} = \{L(z_i) \mid z_i \in \bar{z} \wedge \sigma_i = \psi_i\} \quad H' = subscribe(o, \bar{p}, H'') \\
\hline
H, \langle L, x=y.m(\bar{z}); \bar{s} \rangle^l \circ FS \rightarrow H', \langle L', \bar{t} \rangle^{a(o, \bar{p})} \circ \langle L[x \mapsto o], \bar{s} \rangle^l \circ FS \quad (E\text{-RASync-METHOD})
\end{array}$$

$$\begin{array}{c}
H(o) = \langle Observable\langle \sigma \rangle, running(\bar{F}, \bar{S}) \rangle \\
(\bar{o}, \bar{R}) = resume(\bar{F}, Some(L(z))) \quad Q = \{R \circ \epsilon \mid R \in \bar{R}\} \\
\bar{S}' = \{ \langle o', q :: L(z) \mid \langle o', q \rangle \in \bar{S} \} :: \{ \langle o_i, [] \mid o_i \in \bar{o} \} \\
H' = H[o \mapsto \langle Observable\langle \sigma \rangle, running(\epsilon, \bar{S}') \rangle] \\
\hline
H, \{ \langle L, yieldNext z; \bar{s} \rangle^{a(o, \bar{p})} \circ FS \} \cup P \rightsquigarrow H', \{ \langle L, \bar{s} \rangle^{a(o, \bar{p})} \circ FS \} \cup P \cup Q \quad (E\text{-YIELD})
\end{array}$$

$$\begin{array}{c}
H(o) = \langle Observable\langle \sigma \rangle, running(\bar{F}, \bar{S}) \rangle \\
(\bar{o}, \bar{R}) = resume(\bar{F}, Some(L(x))) \quad Q = \{R \circ \epsilon \mid R \in \bar{R}\} \\
\bar{S}' = \{ \langle o', q :: L(x) \mid \langle o', q \rangle \in \bar{S} \} :: \{ \langle o_i, [] \mid o_i \in \bar{o} \} \\
H_0 = H[o \mapsto \langle Observable\langle \sigma \rangle, done(\bar{S}') \rangle] \\
\forall i \in 1 \dots n. H_i = H_{i-1}[p_i \mapsto unsub(o, p_i, H)] \\
\hline
H, \{ \langle L, return x; \bar{s} \rangle^{a(o, \bar{p})} \circ FS \} \cup P \rightsquigarrow H_n, \{FS\} \cup P \cup Q \quad (E\text{-RASync-RETURN})
\end{array}$$

$$\begin{array}{c}
H(o) = \langle Observable\langle \sigma \rangle, running(\bar{F}, \bar{S}) \rangle \\
(\bar{o}, \bar{R}) = resume(\bar{F}, None\langle \sigma \rangle) \quad Q = \{R \circ \epsilon \mid R \in \bar{R}\} \\
\bar{S}' = \bar{S} \cup \{ \langle o_i, [] \mid o_i \in \bar{o} \} \\
H_0 = H[o \mapsto \langle Observable\langle \sigma \rangle, done(\bar{S}') \rangle] \\
\forall i \in 1 \dots n. H_i = H_{i-1}[p_i \mapsto unsub(o, p_i, H)] \\
\hline
H, \{ \langle L, yieldDone () ; \bar{s} \rangle^{a(o, \bar{p})} \circ FS \} \cup P \rightsquigarrow H_n, \{FS\} \cup P \cup Q \quad (E\text{-YIELDDONE})
\end{array}$$

Fig. 15. Asynchronous method call/return transition rules.

on top of the frame stack. Importantly, the caller frame (with statements \bar{s}) is annotated with variable x ; this annotation is used for the transfer of the return value as follows. Rule E-RETURN-VAL shows how a value is returned from a method invocation to the caller. A method call returns when the statements of its frame have been reduced to a sequence beginning with `return y`; . The method's frame is popped off the frame stack, and the frame of the caller is replaced with a frame that maps variable x to the value of y . Rules E-METHOD-`STMT` and E-RETURN are analogous for the cases where a method invocation or a `return` statement does not return result values, respectively.

4.2.2. Asynchronous transition rules

In contrast to synchronous transition rules, asynchronous transition rules involve *asynchronous frames*, in addition to the synchronous frames used for method call/return transitions. An asynchronous frame has the form $\langle L, \bar{t} \rangle^l$ with an *asynchronous label* $l = a(o, \bar{p})$. In this label, o is the object identifier of a corresponding *observable object* $\langle Observable\langle \sigma \rangle, ST \rangle$. Instead of a field map FM , an observable object has a running state ST . ST has one of two forms:

- $ST = running(\bar{F}, \bar{S})$: this state indicates that observable o is *running*, i.e., its behavior has not yet been reduced to a value. Observable o may still await events of other observables and/or yield events itself. \bar{F} is a list of asynchronous frames, namely, all observables that are currently suspended awaiting o to publish a new event. \bar{S} is a list of *subscribers*. A subscriber $S \in \bar{S}$ is a pair $S = \langle o', q \rangle$ where o' is an observable that has expressed interest in awaiting events published by o , and q is a queue of events published by o , but not yet consumed by o' .
- $ST = done(\bar{S})$: this state indicates that observable o is *done*, i.e., its behavior cannot be reduced further. Observable o may no longer await or yield events. However, subscribers $\langle o', q \rangle \in \bar{S}$ may still consume events from their queue q .

The second component of an asynchronous label $a(o, \bar{p})$ is a sequence of identifiers \bar{p} of observable objects that observable o is subscribed to. As explained above, subscriptions are used for managing asynchronicity: receivers of asynchronous events may not always be ready to receive new events; therefore, subscriptions are used for setting up queues within the sending/publishing observables for buffering events until they can be consumed.

Notation. Given a sequence q , we use $v :: q$ for denoting the sequence that prepends a single element v to q . Conversely, we use $q :: v$ for denoting the sequence that appends a single element v to q . Given two sequences q and p , we use $q :: p$ for denoting the sequence that concatenates the two sequences. The binary operator \oplus is used to express the destructuring of a sequence such that $\bar{Q} = \bar{R} \oplus \bar{S}$ if $\forall r \in \bar{R}. r \in \bar{Q} \wedge r \notin \bar{S}$ and $\forall s \in \bar{S}. s \in \bar{Q} \wedge s \notin \bar{R}$. Finally, given sets Q , R , and S , we use $Q = R \uplus S$ to express the fact that Q is equal to the disjoint union of R and S .

Fig. 15 shows the asynchronous transition rules. These rules transition either between frame stacks (\rightarrow) or processes (\rightsquigarrow).

Rule E-RASync-METHOD evaluates the invocation of an `rasync` method. The look-up of such a method is identical to that of a regular, synchronous method. However, the method type looked up using *mbody* indicates using function arrow \rightarrow^a that the method is marked as `rasync`. Thus, the method's result type is guaranteed to be an observable type

$$\frac{\forall p_i \in \bar{p}. H_0(p_i) = \langle \psi_i, \text{running}(\bar{F}_i, \bar{S}_i) \rangle \quad |\bar{p}| = n}{\forall i \in 1 \dots n. H_i = H_{i-1}[p_i \mapsto \langle \psi_i, \text{running}(\bar{F}_i, \langle o, [] \rangle :: \bar{S}_i) \rangle]} \quad \text{subscribe}(o, \bar{p}, H_0) = H_n$$

Fig. 16. Function *subscribe* computes, starting from a heap H_0 , a heap H_n where a given observable o is subscribed to a set of observables \bar{p} .

```

rasync public Observable<int> fwd(Observable<int> s) {
  int x;
  x = await s;
  yieldNext x;
}

```

Fig. 17. Example *rasync* method.

$$\begin{aligned} \text{resume}(\bar{F}, v) &= [(o, \langle L[x \mapsto v], \bar{t} \rangle^{a(o, \bar{p})}) \mid \langle L, x = \text{await } y; \bar{t} \rangle^{a(o, \bar{p})} \in \bar{F}] \\ \text{unsub}(\bar{S}, o) &= \{ \langle o', q \rangle \mid \langle o', q \rangle \in \bar{S} \wedge o' \neq o \} \\ \text{unsub}(o, p, H) &= \begin{cases} \langle \psi, \text{running}(\bar{F}, \text{unsub}(\bar{S}, o)) \rangle & \text{if } H(p) = \langle \psi, \text{running}(\bar{F}, \bar{S}) \rangle \\ \langle \psi, \text{done}(\text{unsub}(\bar{S}, o)) \rangle & \text{if } H(p) = \langle \psi, \text{done}(\bar{S}) \rangle \end{cases} \end{aligned}$$

Fig. 18. Auxiliary functions.

$\psi = \text{Observable} \langle \sigma \rangle$ (ensured by rule *ASYNCMETHOD-OK*; see Fig. 23). Rule *E-RASYNCMETHOD* allocates a new observable o of type ψ with running state $\text{running}(\epsilon, \epsilon)$. Furthermore, this new observable o subscribes itself to all other observables \bar{p} passed as arguments in the method invocation. The actual subscription is performed using the *subscribe* function which is defined in Fig. 16; the function adds o as a subscriber (with an empty queue) to each observable $p_i \in \bar{p}$. Finally, back in rule *E-RASYNCMETHOD*, a new asynchronous frame $\langle L', \bar{t} \rangle^{a(o, \bar{p})}$ is pushed onto the frame stack. The asynchronous label $a(o, \bar{p})$ indicates that the frame belongs to the behavior of observable o , which is subscribed to observables \bar{p} .

Example. Consider an invocation of the *rasync* method *fwd* shown in Fig. 17. Suppose the executing process has thus a frame stack $\langle L, w = y.\text{fwd}(z); \bar{s} \rangle^l \circ FS$ for some mapping L , variables w, y, z , statements \bar{s} , label l , and frame stack FS such that $\{w, y, z\} \subseteq \text{dom}(L)$. Furthermore, suppose $H(L(y)) = \langle C, FM \rangle$, $\text{mbody}(C, \text{fwd}) = mb : (\text{Observable} \langle \text{int} \rangle \rightarrow^a \text{Observable} \langle \text{int} \rangle)$, and mb as shown in Fig. 17. Then, by rule *E-RASYNCMETHOD*, $H, \langle L, w = y.\text{fwd}(z); \bar{s} \rangle^l \circ FS \rightarrow H', \langle L', \bar{t} \rangle^{a(o, \bar{p})} \circ \langle L[w \mapsto o], \bar{s} \rangle^l \circ FS$ where

- (1) $H'' = H[o \mapsto \langle \text{Observable} \langle \text{int} \rangle, \text{running}(\epsilon, \epsilon) \rangle]$, $o \notin \text{dom}(H)$
- (2) $H' = \text{subscribe}(o, \bar{p}, H'')$, $\bar{p} = \{L(z)\}$
- (3) $L' = [s \mapsto L(z), x \mapsto 0, \text{this} \mapsto L(y)]$
- (4) $\bar{t} = x = \text{await } s; \text{yieldNext } x;$

(1) allocates the new stream o in heap H'' ; o 's running state has empty sets of waiters and subscribers. (2) adds the newly created stream o as a subscriber to stream $L(z)$ (since $\bar{p} = \{L(z)\}$): $\text{subscribe}(o, \bar{p}, H'') = H''[L(z) \mapsto \langle \text{Observable} \langle \text{int} \rangle, \text{running}(\bar{F}, \langle o, [] \rangle :: \bar{S}) \rangle]$ where $H''(L(z)) = \langle \text{Observable} \langle \text{int} \rangle, \text{running}(\bar{F}, \bar{S}) \rangle$. (3) initializes the mapping L' of the new asynchronous frame $\langle L', \bar{t} \rangle^{a(o, \bar{p})}$. Finally, (4) initializes the statements of the new frame which are equal to the body of the *fwd* method.

Differences to the informal semantics. In the formal model, the invocation of an *rasync* method $y.m(\bar{z})$ such that $H(L(y)) = \langle \rho, FM \rangle$ and $\text{mbody}(\rho, m) = mb : (\bar{\sigma} \bar{x}) \rightarrow^a \psi$ corresponds to the invocation of a method of the following form in the informal description:

```

def m( $\bar{x} : \bar{\sigma}$ ) :  $\psi = \text{rasync}(\bar{x'})$  {
  ...
}

```

Here, $\bar{x}' = [x_i \in \bar{x} \mid \sigma_i = \psi_i]$ is a list of all parameters with an observable type. The implementation does not enforce the above form, though; thus, not passing an observable parameter as an argument to *rasync* would lead to a semantic difference between the practical implementation described in Section 3 and the formal model described here.

Rule *E-YIELD* models the built-in *yieldNext* statement which publishes a new event to all waiters and subscribers of observable o . Waiters $F \in \bar{F}$ are resumed by creating a set of new frame stacks Q based on frames created using the *resume* auxiliary function (defined in Fig. 18). Rule *E-YIELD* resumes waiters with value $L(z)$. The state of observable o is updated such that the set of waiters is empty. Furthermore, for each resumed waiter, an empty queue is added to the new subscriber set \bar{S}' .

In rule *E-RASYNCRETURN* a process has an asynchronous frame $\langle L, \text{return } x; \bar{s} \rangle^{a(o, \bar{p})}$ on top of its frame stack. Since the frame's statements have been reduced to a sequence beginning with *return* x ;, this means the corresponding observable

$$\begin{array}{c}
\frac{L(y) = o' \quad H(o') = \langle \text{Observable}\langle \sigma \rangle, \text{running}(\overline{F}, \overline{S}) \rangle \quad \overline{S} = \overline{R} \uplus \{(o, [])\}}{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \circ FS \rightarrow H', FS} \quad (\text{E-AWAIT1}) \\
\\
\frac{L(y) = o' \quad H(o') = \langle \text{Observable}\langle \sigma \rangle, \text{running}(\overline{F}, \overline{S}) \rangle \quad \overline{S} = \overline{R} \uplus \{(o, q :: v)\}}{H' = H[o' \mapsto \langle \text{Observable}\langle \sigma \rangle, \text{running}(\overline{F}, \overline{R} \cup \{(o, q)\}) \rangle]} \\
\frac{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \rightarrow H', \langle L[x \mapsto \text{Some}(v)], \overline{s} \rangle^{a(o, \overline{p})}}{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \rightarrow H', \langle L[x \mapsto \text{Some}(v)], \overline{s} \rangle^{a(o, \overline{p})}} \quad (\text{E-AWAIT2}) \\
\\
\frac{L(y) = o' \quad H(o') = \langle \text{Observable}\langle \sigma \rangle, \text{done}(\overline{S}) \rangle \quad \overline{S} = \overline{R} \uplus \{(o, q :: v)\}}{H' = H[o' \mapsto \langle \text{Observable}\langle \sigma \rangle, \text{done}(\overline{R} \cup \{(o, q)\}) \rangle]} \\
\frac{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \rightarrow H', \langle L[x \mapsto \text{Some}(v)], \overline{s} \rangle^{a(o, \overline{p})}}{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \rightarrow H', \langle L[x \mapsto \text{Some}(v)], \overline{s} \rangle^{a(o, \overline{p})}} \quad (\text{E-AWAIT3}) \\
\\
\frac{L(y) = o' \quad H(o') = \langle \text{Observable}\langle \sigma \rangle, \text{done}(\overline{S}) \rangle \quad \overline{S} = \overline{R} \uplus \{(o, [])\}}{H, \langle L, x = \text{await } y; \overline{s} \rangle^{a(o, \overline{p})} \rightarrow H, \langle L[x \mapsto \text{None}\langle \sigma \rangle], \overline{s} \rangle^{a(o, \overline{p})}} \quad (\text{E-AWAIT4})
\end{array}$$

Fig. 19. Asynchronous transition rules for await.

$$\begin{array}{c}
H, \{\epsilon\} \cup P \rightsquigarrow H, P \quad (\text{E-EXIT}) \\
\\
\frac{H, FS \rightarrow H', FS'}{H, \{FS\} \cup P \rightsquigarrow H', \{FS'\} \cup P} \quad (\text{E-SCHEDULE})
\end{array}$$

Fig. 20. Process transition rules.

o is about to publish $L(x)$ as its very last event. Therefore, any waiters \overline{F} are resumed with result $\text{Some}(L(x))$. The state of observable o is updated to $\langle \text{Observable}\langle \sigma \rangle, \text{done}(\overline{S}) \rangle$, indicating that o has transitioned to the terminated state; the (updated) subscribers \overline{S}' are defined in exactly the same way as in the case of yielding an event (see above). Importantly, even though o has terminated, events previously published to its subscribers \overline{S}' remain available for consumption (see rules for `await` below). Finally, o unsubscribes from all observables \overline{p} using the `unsub` auxiliary function defined in Fig. 18.

Rule E-YIELDDONE is analogous to rule E-RASYNC-RETURN, except that the waiting frames \overline{F} are resumed with an empty option value $\text{None}\langle \sigma \rangle$ instead of a full option value $\text{Some}(L(x))$. The statement sequence \overline{s} following the `yieldDone()` invocation is discarded.

Fig. 19 shows the transition rules for the `await` expression. Rule E-AWAIT1 adds the asynchronous frame F of observable o to the waiters of observable o' in the case where there is no event from o' ready to be consumed by o . Rule E-AWAIT2 handles the dual case where observable o immediately receives an event from observable o' ; the subscribers of o' are updated accordingly in the target heap H' . Rule E-AWAIT3 handles the case where observable o' is in a terminated state $\text{done}(\overline{S})$. Importantly, a subscriber queue in \overline{S} may contain an event that can be consumed by the `await`-invoking observable o . In case the corresponding subscriber queue is empty (E-AWAIT4), `await` reduces to $\text{None}\langle \sigma \rangle$, since observable o' is in a terminated state and will therefore not yield any further events. Note that rules E-AWAIT2-4 use the \rightarrow transition relation, since their transition depends only on a single frame and the state of the heap. In contrast, rule E-AWAIT1 replaces the entire frame stack (by popping the top frame); therefore, it is necessary to use a different transition relation, \rightsquigarrow .

Process transition rules enable reducing frame stacks, *i.e.*, threads; Fig. 20 shows the transition rules. We use an interleaving semantics. Rule E-SCHEDULE non-deterministically selects and transitions a thread; note that the transition may have side effects on the heap. Rule E-EXIT removes threads with empty frame stacks from the soup of threads.

4.3. Static semantics

The typing relation for expressions and statements is defined using a judgement of the form $\Gamma \vdash e : \sigma$ where Γ is a standard type environment consisting of bindings $x : \rho$, e is an expression or a statement, and σ is a type. Fig. 21 shows the typing rules for expressions.

Most typing rules are standard and identical to those of FC_5^\sharp . Rule C-AWAIT is different from the rule of the same name in FC_5^\sharp : in FC_5^\sharp `await` is used to wait for the completion of a task that produces a single result. In RAY, `await` is used to wait for the next emitted event of an observable. For an expression `await x` to be well-typed, x must be an observable of type $\text{Observable}\langle \sigma \rangle$. The result of `await x` is an optional value, since it is possible that observable x has already finished emitting events when `await` is invoked.

Fig. 22 shows the type checking of statements. All type rules except for C-YIELD and C-YIELDDONE are unchanged compared to FC_5^\sharp . Note that for type checking statements, the type σ in the judgement $\Gamma \vdash s ; : \sigma$ indicates that statement s may only return (or yield) values of type σ ; however, statement s may also not return (or yield) anything, *e.g.*, in rule C-ASN.

Recall that a program is a sequence of class declarations followed by the body of a “main” method. Fig. 23 shows the rules for method and class typing. A class is well-typed if its methods are well-typed (CLASS-OK); note that fields are well-formed thanks to their syntax. A method is well-formed if its body is well-typed under the type environment constructed from the current `this`, the method’s formal parameters \overline{x} , and the method’s local variables \overline{y} (METH-OK and

$\frac{}{\Gamma \vdash \underline{b} : \text{bool}}$	(C-BOOL)
$\frac{}{\Gamma \vdash \underline{i} : \text{int}}$	(C-INT)
$\frac{}{\Gamma \vdash \text{null} : \rho}$	(C-NULL)
$\frac{\Gamma \vdash x : \sigma_0 \quad \Gamma \vdash y : \sigma_1 \quad \oplus : \sigma_0 \times \sigma_1 \rightarrow \tau}{\Gamma \vdash x \oplus y : \tau}$	(C-OP)
$\frac{}{\Gamma \vdash \text{new } C() : C}$	(C-NEW)
$\frac{}{\Gamma, x : \tau \vdash x : \tau}$	(C-VAR)
$\frac{f\text{type}(\sigma, f) = \tau}{\Gamma, x : \sigma \vdash x.f : \tau}$	(C-FIELD)
$\frac{m\text{type}(\sigma_0, m) = (\bar{\tau}) \rightarrow \sigma_1 \quad \Gamma, x : \sigma_0 \vdash \bar{y} : \bar{\tau}}{\Gamma, x : \sigma_0 \vdash x.m(\bar{y}) : \sigma_1}$	(C-METHINV)
$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \text{Some}(x) : \text{Option}\langle\sigma\rangle}$	(C-SOME)
$\frac{}{\Gamma \vdash \text{None}\langle\sigma\rangle : \text{Option}\langle\sigma\rangle}$	(C-NONE)
$\frac{\Gamma \vdash x : \text{Option}\langle\sigma\rangle}{\Gamma \vdash \text{get } x : \sigma}$	(C-GET)
$\frac{}{\Gamma, x : \text{Observable}\langle\sigma\rangle \vdash \text{await } x : \text{Option}\langle\sigma\rangle}$	(C-AWAIT)

Fig. 21. Expression type checking.

$\frac{\Gamma, x : \sigma \vdash e : \sigma}{\Gamma, x : \sigma \vdash x = e ; : \phi}$	(C-ASN)
$\frac{\Gamma, x : \text{bool} \vdash \bar{s} : \phi \quad \Gamma, x : \text{bool} \vdash \bar{t} : \phi}{\Gamma, x : \text{bool} \vdash \text{if } (x) \{ \bar{s} \} \text{ else } \{ \bar{t} \} : \phi}$	(C-COND)
$\frac{\Gamma, x : \text{bool} \vdash \bar{s} : \phi}{\Gamma, x : \text{bool} \vdash \text{while } (x) \{ \bar{s} \} : \phi}$	(C-WHILE)
$\frac{f\text{type}(\sigma_0, f) = \sigma_1 \quad \Gamma, x : \sigma_0 \vdash y : \sigma_1}{\Gamma, x : \sigma_0 \vdash x.f = y ; : \phi}$	(C-FASN)
$\frac{m\text{type}(\sigma_0, m) = (\bar{\tau}) \rightarrow \text{void} \quad \Gamma, x : \sigma_0 \vdash \bar{y} : \bar{\tau}}{\Gamma, x : \sigma_0 \vdash x.m(\bar{y}) ; : \phi}$	(C-MINV)
$\frac{}{\Gamma \vdash \text{return} ; : \text{void}}$	(C-RETURN)
$\frac{}{\Gamma, x : \sigma \vdash \text{return } x ; : \sigma}$	(C-RETURNEXP)
$\frac{}{\Gamma, x : \sigma \vdash \text{yieldNext } x ; : \sigma}$	(C-YIELD)
$\frac{}{\Gamma \vdash \text{yieldDone}() ; : \phi}$	(C-YIELDDONE)

Fig. 22. Statement type checking.

ASYNCMETH-OK). The body of a regular, synchronous method may not contain `await` expressions (METH-OK). Note that statements in the body of an `rasync` method with return type `Observable< σ_0 >` are type-checked with expected type σ_0 ; as a result, values emitted using `yieldNext` must have type σ_0 (C-YIELD), as required when creating an observable of type `Observable< σ_0 >`.

$$\frac{C \vdash \overline{m d} \text{ ok}}{\vdash \text{public class } C \{ \overline{f d} \overline{m d} \} \text{ ok}} \quad (\text{CLASS-OK})$$

$$\frac{\overline{x} : \overline{\sigma}, \overline{y} : \overline{\tau}, \text{this} : C \vdash \overline{s} : \phi \quad \forall e \in \overline{s}. e \neq \text{await } _}{C \vdash \text{public } \phi \text{ m}(\overline{\sigma} \overline{x}) \{ \overline{\tau} \overline{y}; \overline{s} \} \text{ ok}} \quad (\text{METH-OK})$$

$$\frac{\overline{x} : \overline{\sigma}, \overline{y} : \overline{\tau}, \text{this} : C \vdash \overline{s} : \sigma_0}{C \vdash \text{rasync public Observable} \langle \sigma_0 \rangle \text{ m}(\overline{\sigma} \overline{x}) \{ \overline{\tau} \overline{y}; \overline{s} \} \text{ ok}} \quad (\text{ASYNCMETH-OK})$$

Fig. 23. Method and class typing.

$$\frac{\Gamma \vdash \overline{s} : \sigma \quad \text{dom}(\Gamma) \subseteq \text{dom}(L) \quad \forall (x : \rho) \in \Gamma. \text{typeof}(L(x), H) = \rho \quad l = a(o, \overline{p}) \implies \text{okObs}(H, o, \sigma)}{H \vdash \langle L, \overline{s} \rangle^l : \sigma} \quad (\text{T-FRAME1})$$

$$\frac{\Gamma, x : \tau \vdash \overline{s} : \sigma \quad \text{dom}(\Gamma) \subseteq \text{dom}(L) \quad \forall (y : \rho) \in \Gamma. \text{typeof}(L(y), H) = \rho \quad l = a(o, \overline{p}) \implies \text{okObs}(H, o, \sigma)}{H \vdash_x^\tau \langle L, \overline{s} \rangle_x^l : \sigma} \quad (\text{T-FRAME2})$$

$$\frac{H \vdash \langle L, \overline{s} \rangle^l : \sigma}{H \vdash \langle L, \overline{s} \rangle^l \circ \epsilon : \sigma} \quad (\text{T-FS1})$$

$$\frac{FS \neq \epsilon \quad \exists \tau. H \vdash \langle L, \overline{s} \rangle^{a(o, \overline{p})} : \tau \quad H \vdash FS : \sigma}{H \vdash \langle L, \overline{s} \rangle^{a(o, \overline{p})} \circ FS : \sigma} \quad (\text{T-FS2})$$

$$\frac{H \vdash_x^\tau G_x : \sigma}{H \vdash_x^\tau G_x \circ \epsilon : \sigma} \quad (\text{T-FS3})$$

$$\frac{GS \neq \epsilon \quad \exists \tau'. H \vdash_x^\tau G_x : \tau' \quad H \vdash GS : \sigma}{H \vdash_x^\tau G_x \circ GS : \sigma} \quad (\text{T-FS4})$$

$$\frac{\exists \tau. H \vdash F^S : \tau \wedge H \vdash_x^\tau G_x \circ GS : \sigma}{H \vdash F^S \circ G_x \circ GS : \sigma} \quad (\text{T-FS5})$$

$$\frac{\exists \tau'. H \vdash_x^\tau F_x : \tau' \wedge H \vdash_y^\tau G_y \circ GS : \sigma}{H \vdash_x^\tau F_x \circ G_y \circ GS : \sigma} \quad (\text{T-FS6})$$

$$\frac{P = \{FS\} \cup Q \quad \exists \sigma. H \vdash FS : \sigma \quad H \vdash Q : \star}{H \vdash P : \star} \quad (\text{T-PROC})$$

Fig. 24. Typing frames, frame stacks, and processes.

Typing frames, frame stacks, and processes. Rules T-FRAME1 and T-FRAME2 shown in Fig. 24 extend statement typing to frames. Rule T-FRAME1 covers the case where a frame F is *not* annotated with a variable eventually carrying the result of a method invocation. Such non-annotated frames are created when applying the E-METHOD-EXP, the E-METHOD-STMT, or the E-RASYNC-METHOD reduction rule. (Rule E-METHOD-EXP creates a new non-annotated frame and adds an annotation to an existing frame.) Note that the T-FRAME1 rule abstracts from the label l of the frame $\langle L, \overline{s} \rangle^l$; thus, l can either be a synchronous label $l = s$ or an asynchronous label $l = a(o, \overline{p})$. The frame's term must be well-typed in a type environment Γ whose domain is a subset of the domain of L . Crucially, for frame F to be well-typed, the types of values $L(x)$ of variables x in heap H , $\text{typeof}(L(x), H)$, must agree with the static type environment Γ .

Rule T-FRAME2 covers the case where a frame F is annotated with a variable x eventually carrying the result of a method invocation. The judgement $H \vdash_x^\tau F : \sigma$ dictates the type of result variable x to be τ . Consequently, the frame's statements \overline{s} must be well-typed in an environment $\Gamma, x : \tau$. Note that the dynamic variable mapping L is undefined for x . The annotated judgement \vdash_x^τ is introduced by rule T-FS5, i.e., whenever a synchronous frame is followed by a non-empty frame stack. (A synchronous frame followed by an empty frame stack is handled by rule T-FS1.) In that case, the frame stack right below the synchronous frame ($G_x \circ GS$ in T-FS5) is checked for well-formedness using the annotated judgement in order to keep track of the expected type τ of variable x . Rules T-FS3 and T-FS4 have annotated judgements for frame stacks in their conclusion and require well-formed (single) frames according to an annotated judgement. In turn, that judgement is defined by T-FRAME2. Otherwise, T-FRAME2 is analogous to T-FRAME1. Both rules include a well-formedness condition $\text{okObs}(H, o, \sigma)$ in case the frame label l is asynchronous, i.e., $l = a(o, \overline{p})$.

$$\begin{array}{c}
\frac{}{H \vdash \epsilon \text{ ok}} \text{ (EMPFS-ok)} \\
\frac{H \vdash F \text{ ok} \quad H \vdash FS \text{ ok} \quad \text{obslds}(F) \# \text{obslds}(FS)}{H \vdash F \circ FS \text{ ok}} \text{ (FS-ok)} \\
\frac{}{H \vdash F^s \text{ ok}} \text{ (SF-ok)} \\
\frac{H \vdash F^s \text{ ok}}{H \vdash F_x^s \text{ ok}} \text{ (CSF-ok)} \\
\frac{\text{Running}(H(o)) \quad \forall o' \in \text{dom}(H). o \notin \text{waiters}(H(o')) \wedge (o \in \text{subscribers}(H(o')) \Leftrightarrow o' \in \bar{p})}{H \vdash F^{a(o, \bar{p})} \text{ ok}} \text{ (AF-ok)} \\
\frac{\forall o \in \text{dom}(H). H \vdash H(o) \text{ ok} \quad \forall o_1 \neq o_2 \in \text{dom}(H). \text{waiters}(H(o_1)) \# \text{waiters}(H(o_2))}{\vdash H \text{ ok}} \text{ (H-ok)} \\
\frac{}{H \vdash \langle C, FM \rangle \text{ ok}} \text{ (HO-ok)} \\
\frac{}{H \vdash \langle \text{Observable} \langle \sigma \rangle, \text{done}(\bar{S}) \rangle \text{ ok}} \text{ (DOHO-ok)} \\
\frac{\forall i \neq j \in \{1..n\}. \text{obslds}(F_i) \# \text{obslds}(F_j) \quad \forall i \in \{1..n\}. \forall o \in \text{obslds}(F_i). \text{Running}(H(o))}{H \vdash \langle \text{Observable} \langle \sigma \rangle, \text{running}(F_1, \dots, F_n, \bar{S}) \rangle \text{ ok}} \text{ (ROHO-ok)} \\
\frac{H \vdash FS_1 \text{ ok} \quad \dots \quad H \vdash FS_n \text{ ok} \quad \forall i \neq j \in \{1..n\}. \text{obslds}(FS_i) \# \text{obslds}(FS_j)}{H \vdash \{FS_1, \dots, FS_n\} \text{ ok}} \text{ (PROC-ok)}
\end{array}$$

Fig. 25. Non-interference properties.

Definition 1 (*Well-formed Observable*). An observable o yielding values of type σ is well-formed in heap H , written $okObs(H, o, \sigma)$, iff

$$\begin{aligned}
H(o) &= \langle \text{Observable} \langle \sigma \rangle, \text{running}(\bar{F}, \bar{S}) \rangle \\
&\implies \forall F \in \bar{F}. H \vdash F : \tau \wedge F = \langle K, x = \text{await } y; \bar{t} \rangle^l \wedge \text{typeof}(K(x), H) = \text{Option} \langle \sigma \rangle
\end{aligned}$$

Essentially, $okObs(H, o, \sigma)$ requires waiter frames of running observables to be well-typed and the result of the suspended `await` must be an option of a type matching the type of values yielded by observable o .

Rules T-FS₁ to T-FS₆, shown in Fig. 24, extend frame typing to frame stacks. Essentially, pushing a well-typed frame F onto a well-typed frame stack GS with type σ (in heap H) preserves type σ for the extended frame stack $F \circ GS$ in H . Importantly, frame F may have a type different from σ in H . The precise formulation of these rules is critical for the proof of subject reduction (e.g., in case E-METHOD-EXP in the proof of Lemma 2 in Section A.2).

Finally, rule T-PROC extends frame stack typing to process typing. Note that processes are simply sets of frame stacks.

5. Correctness properties

We show that well-typed programs satisfy desirable properties:

1. *Observable protocol*. For example, a terminated observable never publishes events again; this protocol property is captured by a *heap evolution* invariant, Definition 2 (see below).
2. *Subject reduction*. Reduction of well-typed programs preserves types.

The proofs of these properties are based on the typing relation defined in Section 4.3, as well as invariants preserved by reduction. To establish the correctness properties we have to consider non-interference properties for processes, frame stacks, frames, and heaps; these properties are shown in Fig. 25.

The application $obslds(FS)$ of the $obslds$ auxiliary function returns the set of all object addresses o in labels $a(o, \bar{p})$ of the *asynchronous frames* FS (similarly for a single asynchronous frame F). In particular, $obslds(F) = \emptyset$ if F is a synchronous frame. The $waiters$ function returns the observable ids of the waiting frames of the running state of a given observable heap object. For an observable heap object $H(o) = \langle \text{Observable} \langle \sigma \rangle, \text{running}(\bar{F}, \bar{S}) \rangle$,

$waiters(H(o)) = obsIds(\overline{F})$. The *subscribers* function returns the set of subscribers of an observable o . For an observable heap object $H(o) = \langle Observable\langle\sigma\rangle, running(\overline{F}, \overline{(p, q)}) \rangle$, $subscribers(H(o)) = \overline{p}$. For an observable heap object $H(o) = \langle Observable\langle\sigma\rangle, done(\overline{(p, q)}) \rangle$, $subscribers(H(o)) = \overline{p}$. To test whether an observable is currently running (as opposed to done) we use a simple predicate, *Running*. Finally, to express disjointness of (sets of) heap addresses we use the symbol $\#$.

To enforce non-interference during evaluation we define a relation between heaps. The following relation also (a) preserves the types of heap objects and (b) bounds the observable ids of new running states.

Definition 2 (Heap Evolution). Heap H evolves to H' in one step with respect to a set of observable ids B , written $H \leq_B H'$, iff

- (i) $\forall o \in dom(H')$. if $o \notin dom(H)$ and $H'(o) = \langle \psi, running(\overline{F}, \overline{S}) \rangle$ then $\overline{F} = \overline{S} = \epsilon$, and
- (ii) $\forall o \in dom(H)$.
 - if $H(o) = \langle C, FM \rangle$ then $H'(o) = \langle C, FM' \rangle$,
 - if $H(o) = \langle \psi, done(\overline{S}) \rangle$ and $\overline{S} = \overline{R} \oplus [\langle o', q \rangle]$ then $H'(o) = \langle \psi, done(\overline{R} :: \langle o', q' \rangle) \rangle$ or $H'(o) = H(o)$, and
 - if $H(o) = \langle \psi, running(\overline{F}, \overline{S}) \rangle$ then
 - (a) $H'(o) = \langle \psi, running(\overline{F}, \langle o', [] \rangle :: \overline{S}) \rangle$ for some $o' \in dom(H')$; or
 - (b) $H'(o) = \langle \psi, running(\overline{F}, \overline{R}) \rangle$ where $\overline{S} = \overline{R} \oplus [\langle o', q' \rangle]$; or
 - (c) $H'(o) = \langle \psi, running(\epsilon, \overline{S} :: \overline{R}) \rangle$ where $\overline{R} = [\langle o', [] \rangle \mid F^{a(o', \overline{F})} \in \overline{F}]$; or
 - (d) $H'(o) = \langle \psi, running(\overline{F}, \overline{R} :: \langle o', q \rangle) \rangle$ where $\overline{S} = \overline{R} \oplus [\langle o', q :: v \rangle]$; or
 - (e) $\overline{S} = \overline{R} \oplus [\langle o', [] \rangle \mid G^{a(o', \overline{F})} \in \overline{G}] \implies H'(o) = \langle \psi, running(\overline{F} :: \overline{G}, \overline{R}) \rangle \wedge obsIds(\overline{F}) \# obsIds(\overline{G}) \wedge obsIds(\overline{G}) \subseteq B$; or
 - (f) $H'(o) = \langle \psi, done(\overline{S} :: [\langle o', [] \rangle \mid F^{a(o', \overline{F})} \in \overline{F}]) \rangle$.

The above heap evolution property specifies the state transition protocol of observables in RAY. Informally, newly created observables have empty sets of waiters and subscribers; the set of subscribers may increase (a) or decrease (b) such that new subscribers are added with empty event queues; the set of waiters may be converted to subscribers with empty event queues (c); subscribers may consume queued events using *await* (d); subscribers with empty event queues may become waiters (e); an observable may transition from a ‘running’ to a ‘done’ state, converting waiters to subscribers with empty event queues (f).

Example 1. Consider the reduction of the frame stack $\langle L, x=y.m(\overline{z}); \overline{s} \rangle \circ FS$ in heap H where $H(L(y)) = \langle \rho, FM \rangle$ and $mbody(\rho, m) = mb : (\overline{\sigma} \ \overline{x}) \rightarrow^a \psi, mb = \overline{y} \ \overline{t}$. The arrow \rightarrow^a indicates that m is an asynchronous method in class type ρ . Then, according to reduction rule E-RASYNC-METHOD (see Fig. 15) H evolves to H' in one step, such that $dom(H') \setminus dom(H) = \{o\}$ where $H'(o) = \langle \psi, running(\epsilon, \epsilon) \rangle$. Thus, case (i) of Definition 2 holds.

Example 2. Consider the reduction of the frame $\langle L, x=await \ y; \overline{s} \rangle^{a(o, \overline{p})}$ in heap H where $L(y) = o'$, $H(o') = \langle Observable\langle\sigma\rangle, running(\overline{F}, \overline{S}) \rangle$, and $\overline{S} = \overline{R} \oplus [\langle o, q :: v \rangle]$. This means that subscriber o has a value v ready to be consumed from the publishing observable o' . Then, by reduction rule E-AWAIT2 (see Fig. 19) H evolves to H' in one step, such that $H'(o') = \langle Observable\langle\sigma\rangle, running(\overline{F}, \overline{R} :: \langle o, q \rangle) \rangle$ and $H'(o'') = H(o'') \ \forall o'' \in dom(H') \setminus \{o'\}$. Thus, case (ii.d) of Definition 2 holds.

In the following section we prove that the protocol stipulated by Definition 2 is preserved by reduction using a subject reduction theorem.

5.1. Subject reduction

The following subject reduction theorem is based on the typing relation defined in Section 4.3. Following a standard approach, frames, frame stacks, and processes must be reduced in well-typed heaps, which are defined as follows.

Definition 3 (Well-typed Heap). A heap H is well-typed, written $\vdash H : \star$ iff

$$\forall o \in dom(H). H(o) = \langle \sigma, FM \rangle \implies \\ (dom(FM) = fields(\sigma) \wedge \forall f \in dom(FM). typeof(FM(f), H) = ftype(\sigma, f))$$

where

$$typeof(o, H) = \begin{cases} \rho & \text{if } o \in dom(H) \wedge H(o) = \langle \rho, FM' \rangle \\ \text{Option}\langle\sigma\rangle & \text{if } o = \text{None}\langle\sigma\rangle \vee (o = \text{Some}(o') \wedge typeof(o', H) = \sigma) \\ \text{bool} & \text{if } o = \underline{b} \\ \text{int} & \text{if } o = \underline{i} \end{cases}$$

Theorem 1 (Subject Reduction). *If $\vdash H : \star$ and $\vdash H \text{ ok}$ then:*

1. *If $H \vdash F : \sigma$, $H \vdash F \text{ ok}$ and $H, F \longrightarrow H', F'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash F' : \sigma$, $H' \vdash F' \text{ ok}$, and $\forall B. H \leq_B H'$.*
2. *If $H \vdash FS : \sigma$, $H \vdash FS \text{ ok}$ and $H, FS \longrightarrow H', FS'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash FS' : \sigma$, $H' \vdash FS' \text{ ok}$ and $H \leq_{\text{obslds}(FS)} H'$.*
3. *If $H \vdash P : \star$, $H \vdash P \text{ ok}$ and $H, P \rightsquigarrow H', P'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash P' : \star$ and $H' \vdash P' \text{ ok}$.*

Proof. Part (1) is proved by induction on the derivation of $H, F \longrightarrow H', F'$ (see Appendix A.1). Part (2) is proved by induction on the derivation of $H, FS \longrightarrow H', FS'$ and part (1) (see Appendix A.2). Part (3) is proved by induction on the derivation of $H, P \rightsquigarrow H', P'$ and part (2) (see Appendix A.3). \square

5.2. Soundness

Using a standard syntactic approach [56] we prove soundness of the type system as a corollary of subject reduction and a progress theorem.

The progress theorem states that in a well-formed heap, a well-typed process can either be reduced (according to \rightsquigarrow) or each of its frame stacks satisfies one of two conditions: either the statements in its only remaining frame begin with a return statement, or it is in a permitted stuck state (see below for a discussion of the permitted stuck states).

Theorem 2 (Progress). *If $\vdash H : \star$ and $\vdash H \text{ ok}$ then:*

If $H \vdash P : \star$ and $H \vdash P \text{ ok}$ then

1. *$H, P \rightsquigarrow H', P'$ for some H', P' ; or*
2. *$\forall FS \in P$, one of the following holds:*
 - (a) *$FS = \langle L, \text{return}; \bar{t} \rangle^s \circ \epsilon$ or $FS = \langle L, \text{return } x; \bar{t} \rangle^s \circ \epsilon$*
 - (b) *$FS = \langle L, s; \bar{t} \rangle^l \circ FS'$ where $(s = y.x.m(\bar{z})$ or $s = x.m(\bar{z})$ or $s = y.x.f$ or $s = x.f = y$) and $L(x) = \text{null}$*
 - (c) *$FS = \langle L, y.x.m(\bar{z}); \bar{t} \rangle^l \circ FS'$ where $H(L(x)) = \langle \rho, FM \rangle$, $\text{mbody}(\rho, m) = \text{mb} : (\bar{\sigma} \bar{x}) \rightarrow^a \psi$, and $\exists p_i \in \{L(z_i) \mid z_i \in \bar{z} \wedge \sigma_i = \psi_i\}$ such that $H(p_i) = \langle \psi_i, \text{done}(\bar{S}) \rangle$*
 - (d) *$FS = \langle L, y.\text{await } x; \bar{t} \rangle^{a(o, \bar{p})} \circ FS'$ where $L(x) = \text{null}$ or $L(x) \notin \bar{p}$*
 - (e) *$FS = \langle L, \epsilon \rangle^l \circ FS'$*
 - (f) *$FS = \langle L, y.\text{get } x; \bar{t} \rangle^l \circ FS'$ where $L(x) = \text{None} \langle \sigma \rangle$*

Proof. By induction on the derivation of $H \vdash P : \star$ (see Appendix A.5). \square

The definition of permitted stuck states is mostly standard. For example, attempting to get the value of an empty option is a typical permitted stuck state; avoiding such stuck states would require a more complex type system able to statically ensure initialization (e.g., [45,34,16,12]).

The following permitted stuck states are specific to our system. The first stuck state is state 2.c) where an `rasync` method is invoked which attempts to subscribe to an observable that already terminated. The reduction semantics does not buffer all published events; therefore, it would not be clear which events to publish from a terminated observable to a new subscriber. Thus, in order to simplify the reduction semantics, we decided to require subscribing to running observables only. The second stuck state is state 2.d) where an observable o attempts to await the next event of another observable o' , but o did not subscribe to o' . This permitted stuck state is due to the only run-time type check of our system that can fail: awaiting the next event of an observable o requires the awaiting observable to be a subscriber of o . Crucially, failures of this run-time check cannot invalidate the heap evolution invariant which guarantees that the state of all observables in the system evolves correctly.

Soundness of the type system follows from Theorem 1 and Theorem 2.

Example. We illustrate stuck state 2.d) using an example. Consider the reduction of the frame stack $FS = \langle L, x=y.m(z); w=\text{await } x; \bar{s} \rangle^{a(o, \bar{p})} \circ FS'$ in heap H and soup of processes P where $H(L(y)) = \langle \rho, FM \rangle$ and $\text{mbody}(\rho, m) = \text{mb} : (\text{int } v) \rightarrow^a \psi, \text{mb} = \text{return } v$; . Then, by reduction rule E-RASync-METHOD, $H, FS \longrightarrow H', FS''$ where $FS'' = \langle L, \text{return } v; \bar{s} \rangle^{a(o', \epsilon)} \circ \langle L[x \mapsto o'], w=\text{await } x; \bar{s} \rangle^{a(o, \bar{p})} \circ FS'$ for some $o' \notin \text{dom}(H)$ and thus $o' \notin \bar{p}$. Note that the label of the top-most frame of FS'' is $a(o', \epsilon)$, because the type of the parameter of method m is `int` and not an observable type. Then, by reduction rule E-RASync-RETURN, $H', \{FS''\} \cup P \rightsquigarrow H'', \{\langle L[x \mapsto o'], w=\text{await } x; \bar{s} \rangle^{a(o, \bar{p})} \circ FS'\} \cup P$ where $H''(o') = \langle \psi, \text{done}(\epsilon) \rangle$. We have thus produced the stuck state 2.d), since $L[x \mapsto o'](x) = o' \notin \bar{p}$.

Note that this stuck state does not break type soundness or cause a problem for the preservation of the heap evolution invariant: all involved observables (o and o' in the above example) evolve according to Definition 2. Stuck state 2.d) merely expresses the fact that our type system does not guarantee deadlock freedom for communication between observables. Such a guarantee is outside the scope of the present paper, and would require a more powerful type system, such as *session types* (see, e.g. [31]).

```

1 def fwd(s: Observable[Int]) = rasync(s) {
2   var x: Option[Int] = None
3   x = await(s)
4   x.get
5 }

```

Fig. 26. A simple `rasync` expression.

5.3. Discussion

Ideally, the presented formalization could provide a formal foundation also for APIs for asynchronous streams, such as Reactive Streams [21]. This would enable formal reasoning about programs using such APIs. One approach to provide such a formal foundation would be to formulate a weaker variant of the heap evolution relation (see Definition 2) which applies to the semantics provided by an API. For example, heap evolution for RAY states that new subscribers are added with empty event queues. A similar property is provided also by widely-used asynchronous stream APIs. Similarly, possible transitions between waiting and non-waiting subscribers as formulated in Definition 2 would likely have corresponding formulations for asynchronous stream APIs. In contrast, certain invariants enforced by RAY would not be possible to enforce in asynchronous APIs. For example, RAY ensures that terminated streams never publish events again (see Theorem 1). In an asynchronous API where (attempted) publication of events is typically unrestricted (programmers may call “emit” without constraints). As a result, such an asynchronous API could not prevent runtime exceptions that are thrown when a method is invoked to emit an event from a terminated stream.

In summary, while the provided guarantees would be weaker compared to RAY or other language-based approaches, we believe that our formal model could serve as a starting point for developing formal operational semantics that could help make informal specifications more precise.⁴

6. Implementation

We have implemented RAY in terms of two components: (1) a *macro* which extends Scala Async [26],⁵ and (2) a *library*, `scala-async-flow`,⁶ which provides abstractions for creating observables using the introduced constructs.

The macro component leverages Scala’s (experimental) support for macros [9] to analyze and expand expressions of the form `rasync(...)` { `<block>` }. Invocations of the pseudo-method `await` are treated as markers that are used to transform the block of code (`<block>`) into a state machine that can be paused and resumed at each invocation site of `await`. The block of code is first converted to statement normal form (SNF) [7]. Like A-Normal Form (ANF), SNF requires all subexpressions to be named. In addition, `if`, `while`, `match`, and other control-flow constructs are only used as statements whose results are ignored. Note that our formal model is based on the same normal form (see Section 4).

For example, consider the simple `rasync` expression shown in Fig. 26. The body of the `rasync` block is already SNF-normalized; all subexpressions are named. The single `await` on line 3 divides the block of code into two logical parts: the part before `await` and the part after `await`. The RAY macro thus generates a state machine with two logical states, as well as code to execute the body starting from these two different states. The first and initial state enables executing the body from the beginning, *i.e.*, from line 2. The second state enables executing the body from line 4, given a value for variable `x`.

Fig. 27 shows a sketch of the generated state machine.⁷ The state machine is implemented by a class with several fields: the `state` field (line 2) maintains the current state of the FSM; initially, `state = 0`. The `result` field (line 3) refers to a promise which is asynchronously completed with the result of the `rasync` block; this result is the last event emitted by the `rasync` block’s observable. The `state` and `result` fields are common to all state machines. In addition, this particular state machine has a field `x` which maintains the value of variable `x` across all states where `x` is used.

Invoking the `apply` method (line 6) executes the body of the corresponding `rasync` block from the beginning. Since `state = 0`, the branch on line 10 is evaluated. This branch contains the code before the first (and only) `await` expression; in this case, the code only consists of the initialization `x = None`. The original `await` expression is replaced by the code on lines 12–25. First, `flow.pubToSub(s)` looks up the `subscription` object corresponding to the observable `s` (`flow` is a constructor parameter of the `StateMachine` class; below we describe its role in more detail). The subscription object exposes methods to query if an event is available for consumption, and to obtain the event if available; this extends the push-based interface of observables as required for `await`. These pull-based methods are used as follows.

The call `sub.getCompleted` returns the next (regular) event emitted by `s`, or `null` if there is currently no event ready to be consumed. In the latter case, the state machine registers its unary `apply` method as the on-completion handler

⁴ See, for instance, the informal specification of the Reactive Streams API: <https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md#specification>.

⁵ See <https://github.com/phaller/async> (branch `async-flow`).

⁶ See <https://github.com/phaller/scala-async-flow>.

⁷ The shown listing is a sketch, because we omit aspects such as implicit execution contexts [25] which are not essential for the presentation, but would obscure it.


```

1  class StateMachine(flow: Flow[Int]) {
2    var state = 0
3    val result = Promise[Int]()
4    var x: Option[Int] = _
5
6    def apply(): Unit =
7      apply(null)
8
9    def apply(tr: Try[Int]): Unit = state match {
10     case 0 =>
11       x = None
12       val sub = flow.pubToSub(s)
13       val completed = sub.getCompleted
14       if (completed == null) {
15         state = 2
16         sub.onComplete(evt => apply(evt))
17         return
18       } else if (completed.isFailure) {
19         result.complete(completed)
20         return
21       } else {
22         x = completed.get
23         state = 1
24         apply() // recursive call
25       }
26     case 1 =>
27       result.complete(x.get)
28       return
29     case 2 =>
30       if (tr.isFailure) {
31         result.complete(tr)
32         return
33       } else {
34         x = tr.get
35         state = 1
36         apply() // recursive call
37       }
38   }
39 }

```

Fig. 27. Sketch of state machine generated for `rasync` block of Fig. 26.

```

1  val forwarder = new Flow[Int]
2  forwarder.subscribe(s)
3  forwarder.init((flow: Flow[Int]) => {
4    val stateMachine = new StateMachine(flow)
5    Future { stateMachine.apply() }
6    stateMachine.result.future
7  })
8  return forwarder

```

Fig. 28. Expansion of the body of the `fwd` method in Fig. 26.

with `sub` (line 16) after setting `state` to 2. Therefore, as soon as `s` emits the next event or completes with an exception, the state machine's `apply` method is invoked (line 9). Since `state` is equal to 2 when this happens, the branch on line 29 is evaluated. In case `s` was completed with an exception, the `result` promise is completed with the same failure object (line 31). Otherwise, the result is assigned to `x` (line 34), and the state machine resumed immediately, but in state 1. This results in the execution of the final part of the `rasync` body starting on line 26. Here, it simply consists of evaluating `x.get` which is also the result of the `rasync` block; this result is used to complete the `result` promise, which completes the evaluation of the `rasync` block.

The library component. The state machine described above is augmented with a library component. The body of the `fwd` method in Fig. 26 is expanded as shown in Fig. 28. Each `rasync` block is represented by an instance of the `Flow` class whose main purpose is to manage subscriptions. First, the `forwarder` instance is used to subscribe to observable `s` (line 2). Then, the invocation of `init` registers the body for execution. Actual execution of the body only starts once the first observer subscribes to the created observable. The body itself consists of instantiating the state machine (line 4) of Fig. 27 and creating a new asynchronous task that runs the state machine from the beginning (line 5). The future corresponding to the state machine's `result` promise (see above) is used to communicate the final emitted event to `forwarder`. Finally, the

forwarder instance is returned as the observable created by the original `rasync` block (class `Flow[T]` also implements `Observable[T]`).

7. Related work

Google's Dart programming language [33] has recently been extended with asynchronous functions and generator functions. Asynchronous functions carrying the modifier `async*` asynchronously produce streams of values, similar to the `rasync` construct introduced in this article. Meijer et al. [42] define a continuation semantics for a “featherweight” subset of Dart's asynchronous and generator functions in Scott-Strachey style [53]. However, correctness properties are neither proved nor formalized. An important contribution of this article is a formalization of the main correctness property of constructs like `rasync` and `async*`, as well as a complete formal proof.

Our formalization of the introduced programming model builds on the formalization of asynchronous C^\sharp by Bierman et al. [7]; in particular, we adopt the approach of formulating a heap evolution property that encodes valid “protocol transitions” of the programming model. Our main contribution is a generalization of this approach from a future-like programming model to a programming model based on *streams* of asynchronous events. To enable this generalization we introduce technical novelties in the underlying formal model absent from the more basic model of [7], such as the concept of *subscribers* which enable the emission and reception of a potentially unbounded number of asynchronous events.

The proposed programming model, RAY, is related to previous work on object-based concurrency. The programming model of Creol [36,10] is based on concurrent objects, asynchronous method calls, and processor release points. The primary motivation for Creol is an integration of object orientation and distributed programming. Processor release points enable methods to suspend execution until pending asynchronous method calls have returned replies. A guard statement `await g`, where `g` is a guard, declares a release point; guards may check whether replies to asynchronous invocations are available. The main difference between Creol, and related languages including ABS [35] and JCoBox [52], and RAY is that in Creol asynchronous methods have a bounded number of output parameters, whereas in RAY an `rasync` block may yield an unbounded sequence of events/results. Furthermore, in Creol each call of an asynchronous method creates its own process within the receiver object; in contrast, in RAY there is only a single “thread” (frame stack) associated with each observable, and all clients are served concurrently. We believe that building on the design of RAY (and its formal model), languages like Creol or ABS could be extended to enable asynchronous methods to yield a statically unbounded number of results, while enforcing an observable-like communication protocol.

The abstraction of observables provided by RAY is related to channel-based communication in progress algebras like CSP [29] and π -calculus [44]. Both CSP and π -calculus are based on synchronous communication. The asynchronous π -calculus [30] encodes asynchronous communication in a synchronous framework using dummy processes. In contrast, RAY's communication model is purely asynchronous. The actor model [28,1] is a powerful foundation for asynchronous programming [2] due to fair, asynchronous communication without “artificial” message ordering constraints. Recently, implementations of the actor model in languages like Erlang [4], Scala [24,22], and, more generally, on the JVM [39] have gained increasing interest with numerous commercial applications in telecommunications, internet commerce, and others [3]. The concurrency model of RAY is less general in the sense that RAY imposes stricter ordering constraints on asynchronous messages; conversely, RAY provides stronger guarantees such as the observable stream protocol, which cannot be enforced by “general-purpose” actor languages. The message queues of observables in RAY are reminiscent of asynchronous channels in JoCaml [19], Scala Joins [23], or other implementations of the join-calculus [20]. The main difference is that join-calculus style channels allow multiple concurrent senders on the same channel, whereas in RAY only a single “thread” yields *all* events of a given observable. Communication in RAY is similar to publish/subscribe systems where objects subscribing to an event are notified through anonymous method invocation [15].

8. Conclusions

Asynchronous programming is at the core of an increasingly important class of software systems, including large-scale web applications running on cloud computing platforms, providing rich, client-side interfaces. However, programming with pervasive asynchronous events is challenging using mainstream languages, which provide only limited support for asynchronicity.

In this paper we have presented a design for asynchronous stream generators, called RAY, which builds on the popular `async-await` model, known from F^\sharp and C^\sharp , to simplify programming with asynchronous event streams. Importantly, RAY enables expressing asynchronous stream computations in direct style, *i.e.*, in a familiar blocking style while using efficient non-blocking concurrency control under the hood.

The recent addition of asynchronous generators to Google's Dart programming language suggests a strong developer need for such programming facilities. Unlike Dart, our design targets statically-typed languages. Furthermore, we have presented a reduction semantics of the programming model, including a static type system with a complete type soundness proof. Crucially, we have shown that type soundness implies that RAY enforces an important state transition protocol for asynchronous streams. Establishing this result has required a novel technical treatment of “subscriptions” in the reduction semantics and type system.

Finally, given the presented state transition protocol, we believe that our formal development could be refined to provide a foundation also for popular stream-based APIs, including Reactive Extensions [41] and the proposed Reactive Streams [21] JVM standard.

Appendix A. Full proofs

A.1. Proof of Lemma 1

Lemma 1. *If $\vdash H : \star$ and $\vdash H \text{ ok}$ then:*

If $H \vdash F : \sigma$, $H \vdash F \text{ ok}$ and $H, F \longrightarrow H', F'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash F' : \sigma$, $H' \vdash F' \text{ ok}$, and $\forall B. H \leq_B H'$.

Proof. By induction on the derivation of $H, F \longrightarrow H', F'$.

- Case (E-Var)

1. By the assumptions

- (a) $\vdash H : \star$
- (b) $\vdash H \text{ ok}$
- (c) $H \vdash F : \sigma$
- (d) $H \vdash F \text{ ok}$
- (e) $H, F \longrightarrow H', F'$

2. By (E-Var)

- (a) $F = \langle L, x=y; \bar{s} \rangle^l$
- (b) $H' = H$
- (c) $F' = \langle L[x \mapsto L(y)], \bar{s} \rangle^l$

3. By 1.c), 2.a), and (T-Frame1)

- (a) $\Gamma \vdash x=y; : \sigma$
- (b) $\Gamma \vdash \bar{s} : \sigma$
- (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
- (d) $\forall (z : \rho') \in \Gamma. \text{typeof}(L(z), H) = \rho'$
- (e) $l = a(o, \bar{p}) \implies \text{okObs}(H, o, \sigma)$

4. By 3.a), (C-Asn), and (C-Var), $\{x : \tau, y : \tau\} \subseteq \Gamma$.

5. By 2.b), 3.d), and 4. we have $\forall (z : \tau') \in \Gamma. \text{typeof}(L[x \mapsto L(y)](z), H') = \tau'$

6. By 2.b-c), 3.a-c), 3.e), 5., and (T-Frame1) we have $H' \vdash F' : \sigma$

7. By 1.d), 2.b-c), (SF-ok), (CSF-ok), and (AF-ok) we have $H' \vdash F' \text{ ok}$

8. 1.a-b), 2.b), 6., and 7. conclude this case

- Case (E-Field)

1. By the assumptions

- (a) $\vdash H : \star$
- (b) $\vdash H \text{ ok}$
- (c) $H \vdash F : \sigma$
- (d) $H \vdash F \text{ ok}$
- (e) $H, F \longrightarrow H', F'$

2. By (E-Field)

- (a) $F = \langle L, x=y.f; \bar{s} \rangle^l$
- (b) $H' = H$
- (c) $F' = \langle L[x \mapsto FM(f)], \bar{s} \rangle^l$ where $H(L(y)) = \langle \rho, FM \rangle$

3. By 1.c), 2.a), and (T-Frame1)

- (a) $\Gamma \vdash x=y.f : \sigma$
- (b) $\Gamma \vdash \bar{s} : \sigma$
- (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
- (d) $\forall (z : \tau') \in \Gamma. \text{typeof}(L(z), H) = \tau'$
- (e) $l = a(o, \bar{p}) \implies \text{okObs}(H, o, \sigma)$

4. By 3.a) and (C-Asn)

- (a) $\Gamma \vdash y.f : \tau$
- (b) $(x : \tau) \in \Gamma$

5. By 4.a) and (C-Field)

- (a) $(y : \sigma') \in \Gamma$
- (b) $f\text{type}(\sigma', f) = \tau$

6. By 3.d) and 5.a), $\text{typeof}(L(y), H) = \sigma'$.

7. By 1.a) and 6.

- (a) $H(L(y)) = \langle \sigma', FM \rangle$

- (b) $\text{dom}(FM) = \text{fields}(\sigma')$
- (c) $\forall g \in \text{dom}(FM). \text{typeof}(FM(g), H) = \text{ftype}(\sigma', g)$
- 8. By 5.b) and 7.c), $\text{typeof}(FM(f), H) = \tau$
- 9. By 3.d), 4.b), and 8., $\forall(z : \tau') \in \Gamma. \text{typeof}(L[x \mapsto FM(f)](z), H) = \tau'$.
- 10. By 2.b-c), 3.b-c), 3.e), 9. and (T-Frame1), $H \vdash F' : \sigma$.
- 11. By 1.d), 2.b-c), (SF-ok), (CSF-ok), and (AF-ok) we have $H \vdash F' \mathbf{ok}$
- 12. 1.a-b), 2.b), 10., and 11. conclude this case
- Case (E-Asn)
 1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \mathbf{ok}$
 - (c) $H \vdash F : \sigma$
 - (d) $H \vdash F \mathbf{ok}$
 - (e) $H, F \longrightarrow H', F'$
 2. By (E-Asn)
 - (a) $F = \langle L, x.f=y; \bar{s} \rangle^l$
 - (b) $L(x) = o$
 - (c) $H(o) = \langle \rho, FM \rangle$
 - (d) $H' = H[o \mapsto \langle \rho, FM[f \mapsto L(y)] \rangle]$
 - (e) $F' = \langle L, \bar{s} \rangle^l$
 3. By 1.b), 2.d), (HO-ok), and (H-ok) we have $\vdash H' \mathbf{ok}$
 4. By 1.c), 2.a), and (T-Frame1)
 - (a) $\Gamma \vdash x.f=y; : \sigma$
 - (b) $\Gamma \vdash \bar{s} : \sigma$
 - (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
 - (d) $\forall(z : \tau') \in \Gamma. \text{typeof}(L(z), H) = \tau'$
 - (e) $l = a(o, \bar{p}) \implies \text{okObs}(H, o, \sigma)$
 5. 4.a), (C-FAsn), and (C-Var)
 - (a) $(x : \rho) \in \Gamma$
 - (b) $\text{ftype}(\rho, f) = \tau$
 - (c) $(y : \tau) \in \Gamma$
 6. By 2.d) and 4.d) we have $\forall(z : \tau') \in \Gamma. \text{typeof}(L(z), H') = \tau'$
 7. By 2.e), 4.b-c), 4.e), 6., and (T-Frame1) we have $H' \vdash F' : \sigma$
 8. By 1.d), 2.d), (SF-ok), (CSF-ok), and (AF-ok) we have $H' \vdash F' \mathbf{ok}$
 9. By 2.d) we have $\forall B. H \leq_B H'$
 10. 1.a), 2.d), 3., 7., 8., and 9. conclude this case
- Case (E-Await4)
 1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \mathbf{ok}$
 - (c) $H \vdash F : \sigma$
 - (d) $H \vdash F \mathbf{ok}$
 - (e) $H, F \longrightarrow H', F'$
 2. By (E-Await4)
 - (a) $H' = H$
 - (b) $F = \langle L, x=\text{await } y; \bar{s} \rangle^{a(o, \bar{p})}$
 - (c) $F' = \langle L[x \mapsto \text{None}\langle \sigma' \rangle], \bar{s} \rangle^{a(o, \bar{p})}$
 3. By 1.c), 2.b), and (T-Frame1)
 - (a) $\Gamma \vdash x=\text{await } y : \tau$
 - (b) $\Gamma \vdash \bar{s} : \sigma$
 - (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
 - (d) $\forall(z : \tau') \in \Gamma. \text{typeof}(L(z), H) = \tau'$
 - (e) $\text{okObs}(H, o, \sigma)$
 4. By 3.a), (C-Asn), and (C-Await)
 - (a) $x : \text{Option}\langle \sigma' \rangle \in \Gamma$
 - (b) $\text{dom}(\Gamma) \subseteq \text{dom}(L[x \mapsto \text{None}\langle \sigma' \rangle])$
 5. By 3.d), 4.a-b), and def. *typeof*, we have $\forall(z : \tau') \in \Gamma. \text{typeof}(L[x \mapsto \text{None}\langle \sigma' \rangle](z), H) = \tau'$
 6. By 2.a), 2.c), 3.b), 3.e), 4.b), 5., and (T-Frame1), we have $H' \vdash F' : \sigma$
 7. By 1.d), 2.a), and (AF-ok), we have $H' \vdash F' \mathbf{ok}$
 8. By 2.a) we have $\forall B. H \leq_B H'$
 9. 1.a), 1.b), 2.a), 6., 7., and 8. conclude this case

- Cases (E-Constant), (E-Op), (E-New), (E-While), (E-CondEq), (E-Some), (E-None), (E-Get), and (E-Await2-3) follow analogously. \square

A.2. Proof of Lemma 2

Lemma 2. *If $\vdash H : \star$ and $\vdash H \text{ ok}$ then:*

If $H \vdash FS : \sigma$, $H \vdash FS \text{ ok}$ and $H, FS \rightarrow H', FS'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash FS' : \sigma$, $H' \vdash FS' \text{ ok}$ and $H \leq_{\text{obsIds}(FS)} H'$.

Proof. By induction on the derivation of $H, FS \rightarrow H', FS'$.

- Case (E-Frame)
 1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \text{ ok}$
 - (c) $H \vdash FS : \sigma$
 - (d) $H \vdash FS \text{ ok}$
 - (e) $H, FS \rightarrow H', FS'$
 2. By (E-Frame)
 - (a) $FS = F \circ GS$
 - (b) $FS' = F' \circ GS$
 - (c) $H, F \rightarrow H', F'$
 3. By 1.d), 2.a) and (FS-ok)
 - (a) $H \vdash F \text{ ok}$
 - (b) $H \vdash GS \text{ ok}$
 - (c) $\text{obsIds}(F) \# \text{obsIds}(GS)$
 4. We show $H \vdash F : \tau$ for some τ
 - (a) Case $GS = \epsilon$: by 1.c), 2.a) and (T-FS1), $H \vdash F : \sigma$
 - (b) Case $GS \neq \epsilon \wedge l = s$ where $F = K^l$: by 1.c), 2.a) and (T-FS5), $H \vdash F : \tau$ for some τ
 - (c) Case $GS \neq \epsilon \wedge l = a(o, \bar{p})$ where $F = K^l$: by 1.c), 2.a) and (T-FS2), $H \vdash F : \tau$ for some τ
 5. By 1.a-b), 3.a), 4., 2.c) and Lemma 1
 - (a) $\vdash H' : \star$
 - (b) $\vdash H' \text{ ok}$
 - (c) $H' \vdash F' : \tau$
 - (d) $H' \vdash F' \text{ ok}$
 - (e) $\forall B. H \leq_B H'$
 6. By 5.e) and Definition 2 we have $\forall o \in \text{dom}(H). H(o) = \langle C, FM \rangle \implies H'(o) = \langle C, FM' \rangle$.
 7. We show $H' \vdash FS' : \sigma$
 - (a) Case $GS = \epsilon$
 - i. By 1.c), 2.a) and (T-FS1), $H \vdash F : \sigma$
 - ii. By 5.c), i. and (T-FS1), $H' \vdash F' \circ \epsilon : \sigma$
 - iii. By 2.b) and ii., $H' \vdash FS' : \sigma$
 - (b) Case $GS \neq \epsilon \wedge l = s$ where $F = K^l$
 - i. By 1.c), 2.a), 4. and (T-FS5)
 - (a) $GS = G_x \circ GS'$
 - (b) $H \vdash_x^{\tau} G_x \circ GS' : \sigma$
 - ii. By 6., i.b), (T-FS1-6), (T-Frame1) and (T-Frame2), $H' \vdash_x^{\tau} G_x \circ GS' : \sigma$
 - iii. By 5.c), ii. and (T-FS5), $H' \vdash F' \circ G_x \circ GS' : \sigma$
 - iv. By 2.b), i.a) and iii., $H' \vdash FS' : \sigma$
 - (c) Case $GS \neq \epsilon \wedge l = a(o, \bar{p})$ where $F = K^l$
 - i. By 1.c), 2.a), 4. and (T-FS2), $H \vdash GS : \sigma$
 - ii. By 6., i., (T-FS1-6), (T-Frame1) and (T-Frame2), $H' \vdash GS : \sigma$
 - iii. By 5.c), ii. and (T-FS2), $H' \vdash F' \circ GS : \sigma$
 - iv. By 2.b) and iii., $H' \vdash FS' : \sigma$
 8. By 2.c) and the frame transition rules we have $\text{obsIds}(F') = \text{obsIds}(F)$.
 9. By 3.c) and 8. we have $\text{obsIds}(F') \# \text{obsIds}(GS)$.
 10. By 3.b) and (FS-ok) we have $H \vdash G \text{ ok} \forall G \in GS$.
 11. Let $G = K^{a(o, \bar{p})} \in GS$.
 12. By 2.c) and the transition rules we have $\forall o' \in \text{dom}(H). o' \in \text{dom}(H') \wedge \text{Running}(H(o')) = \text{Running}(H'(o'))$.
 13. By 11. and 12. we have $\text{Running}(H'(o))$.
 14. By 2.c), 3.b) and the transition rules we have $\forall o' \in \text{dom}(H'). o \notin \text{waiters}(H'(o'))$.
 15. By 2.c) and the transition rules we have $\forall o' \in \text{dom}(H') \cap \text{dom}(H). \text{subscribers}(H'(o')) = \text{subscribers}(H(o'))$.

16. By 10., 11. and (AF-ok)
- (a) $\text{Running}(H(o))$
 - (b) $\forall o' \in \text{dom}(H). o \notin \text{waiters}(H(o')) \wedge (o \in \text{subscribers}(H(o')) \Leftrightarrow o' \in \bar{p})$
17. By 13., 14., 15., 16.b) and (AF-ok) we have $H' \vdash G \text{ ok}$.
18. By 11., 17., 3.b) and (FS-ok) we have $H' \vdash GS \text{ ok}$.
19. By 2.b), 5.d), 9., 18. and (FS-ok) we have $H' \vdash FS' \text{ ok}$.
20. 5.a), 5.b), 7., 19. and 5.e) conclude this case.
- Case (E-Method-Exp)
1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \text{ ok}$
 - (c) $H \vdash FS : \sigma$
 - (d) $H \vdash FS \text{ ok}$
 - (e) $H, FS \rightarrow H', FS'$
 2. By (E-Method-Exp)
 - (a) $FS = \langle L, x=y.m(\bar{z}); \bar{s} \rangle^l \circ GS$
 - (b) $H' = H$
 - (c) $H(L(y)) = \langle \rho, FM \rangle$
 - (d) $\text{mbody}(\rho, m) = \text{mb} : (\bar{\tau} \bar{x}) \rightarrow^s \tau', \text{mb} = \bar{\sigma} \bar{y}; \bar{t}$
 - (e) $L' = [\bar{x} \mapsto L(\bar{z}), \bar{y} \mapsto \text{default}(\bar{\sigma}), \text{this} \mapsto L(y)]$
 - (f) $FS' = \langle L', \bar{t} \rangle^s \circ \langle L, \bar{s} \rangle_x^l \circ GS$
 - (g) $F = \langle L, x=y.m(\bar{z}); \bar{s} \rangle^l$
 3. By 1.a), 1.b), and 2.b)
 - (a) $\vdash H' : \star$
 - (b) $\vdash H' \text{ ok}$
 - (c) We show $H \vdash F : \tau \implies \exists \tau'. H \vdash \langle L', \bar{t} \rangle^s : \tau' \wedge H \vdash_{x'}^{\tau'} \langle L, \bar{s} \rangle_x^l : \tau$
 - i. By 2.g) and (T-Frame1)
 - (a) $\Gamma \vdash x=y.m(\bar{z}) : \tau$
 - (b) $\Gamma \vdash \bar{s} : \tau$
 - (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
 - (d) $\forall (z : \rho') \in \Gamma. \text{typeof}(L(z), H) = \rho'$
 - (e) $l = a(o, \bar{p}) \implies \text{okObs}(H, o, \tau)$
 - ii. By i.a) and (C-Asn)
 - (a) $(x : \tau') \in \Gamma$
 - (b) $\Gamma \vdash y.m(\bar{z}) : \tau'$
 - iii. By ii.b), 2.d), and (C-MethInv)
 - (a) $\Gamma \vdash y : \rho$ for some ρ
 - (b) $\text{mtype}(\rho, m) = (\bar{\tau}) \rightarrow \tau'$
 - (c) $\Gamma \vdash \bar{z} : \bar{\tau}$
 - iv. By 2.d), 2.e), iii.b) and (Meth-OK)
 - (a) $\text{this} : \rho, \bar{x} : \bar{\tau}, \bar{y} : \bar{\sigma} \vdash \bar{t} : \tau'$
 - (b) $\Gamma' = \text{this} : \rho, \bar{x} : \bar{\tau}, \bar{y} : \bar{\sigma}$
 - (c) $\text{dom}(\Gamma') \subseteq \text{dom}(L')$
 - v. By 2.e) and i.d), $\forall (z : \rho') \in \Gamma'. \text{typeof}(L'(z), H) = \rho'$
 - vi. By iv.a), iv.b), iv.c), v. and (T-Frame1), $H \vdash \langle L', \bar{t} \rangle^s : \tau'$
 - vii. By i.b-e), ii.a), and (T-Frame2), $H \vdash_{x'}^{\tau'} \langle L, \bar{s} \rangle_x^l : \tau$
 4. We show $H' \vdash FS' : \sigma$
 - (a) Case $GS = \epsilon$
 - i. By 1.c), 2.a), 2.g) and (T-FS1), $H \vdash F : \sigma$
 - ii. By 3.c) and i.
 - (a) $H \vdash \langle L', \bar{t} \rangle^s : \tau'$ for some τ'
 - (b) $H \vdash_{x'}^{\tau'} \langle L, \bar{s} \rangle_x^l : \sigma$
 - iii. By ii.b) and (T-FS3), $H \vdash_{x'}^{\tau'} \langle L, \bar{s} \rangle_x^l \circ \epsilon : \sigma$
 - iv. By 2.b), 2.f), ii.a), iii. and (T-FS5), $H' \vdash FS' : \sigma$
 - (b) Case $GS \neq \epsilon \wedge l = s$
 - i. By 1.c), 2.a), 2.g) and (T-FS5)
 - (a) $FS = F \circ G_y \circ GS'$
 - (b) $H \vdash F : \tau$ for some τ
 - (c) $H \vdash_y^{\tau} G_y \circ GS' : \sigma$
 - ii. By 3.c) and i.b)
 - (a) $H \vdash \langle L', \bar{t} \rangle^s : \tau'$ for some τ'

- (b) $H \vdash_x^{\tau'} \langle L, \bar{s} \rangle_x^l : \tau$
- iii. By i.c), ii.b) and (T-FS6), $H \vdash_x^{\tau'} \langle L, \bar{s} \rangle_x^l \circ G_y \circ GS' : \sigma$
- iv. By ii.a), iii. and (T-FS5), $H \vdash \langle L', \bar{t} \rangle^s \circ \langle L, \bar{s} \rangle_x^l \circ G_y \circ GS' : \sigma$
- v. By 2.b), 2.f) and iv., $H' \vdash FS' : \sigma$
- (c) Case $GS \neq \epsilon \wedge l = a(o, \bar{p})$
 - i. By 1.c), 2.a), 2.g) and (T-FS2)
 - (a) $H \vdash F : \tau$ for some τ
 - (b) $H \vdash GS : \sigma$
 - ii. By 3.c) and i.a)
 - (a) $H \vdash \langle L', \bar{t} \rangle^s : \tau'$ for some τ'
 - (b) $H \vdash_x^{\tau'} \langle L, \bar{s} \rangle_x^l : \tau$
 - iii. By i.b), ii.b) and (T-FS4), $H \vdash_x^{\tau'} \langle L, \bar{s} \rangle_x^l \circ GS : \sigma$
 - iv. By ii.a), iii. and (T-FS5), $H \vdash \langle L', \bar{t} \rangle^s \circ \langle L, \bar{s} \rangle_x^l \circ GS : \sigma$
 - v. By 2.b), 2.f) and iv., $H' \vdash FS' : \sigma$
- 5. By (SF-ok), $H' \vdash \langle L', \bar{t} \rangle^s$ **ok**
- 6. By 1.d), 2.a), and (FS-ok)
 - (a) $H \vdash F$ **ok**
 - (b) $H \vdash GS$ **ok**
 - (c) $obsIds(F) \# obsIds(GS)$
- 7. By 2.b), 6.a), (CSF-ok) and (AF-ok), $H' \vdash \langle L, \bar{s} \rangle_x^l$ **ok**
- 8. By 2.b) and 6.b), $H' \vdash GS$ **ok**
- 9. By 2.g), 6.c) and def. $obsIds$, $obsIds(\langle L, \bar{s} \rangle_x^l) \# obsIds(GS)$
- 10. By 7., 8., 9. and (FS-ok), $H' \vdash \langle L, \bar{s} \rangle_x^l \circ GS$ **ok**
- 11. By 2.f), 5., 10., (FS-ok) and def. $obsIds$, $H' \vdash FS'$ **ok**
- 12. By 2.b), $H \leq_{obsIds(FS)} H'$
- 13. 3.a), 3.b), 4., 11. and 12. conclude this case.
- Case (E-Method-Stmt) follows analogously.
- Case (E-Return-Val)
 1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H$ **ok**
 - (c) $H \vdash FS : \sigma$
 - (d) $H \vdash FS$ **ok**
 - (e) $H, FS \twoheadrightarrow H', FS'$
 2. By (E-Return-Val)
 - (a) $FS = \langle L, \text{return } y; \bar{s} \rangle^s \circ \langle L', \bar{t} \rangle_x^l \circ GS$
 - (b) $FS' = \langle L'[x \mapsto L(y)], \bar{t} \rangle^l \circ GS$
 - (c) $H' = H$
 3. By 1.a), 1.b) and 2.c)
 - (a) $\vdash H' : \star$
 - (b) $\vdash H'$ **ok**
 4. By 1.c), 2.a) and (T-FS5)
 - (a) $H \vdash \langle L, \text{return } y; \bar{s} \rangle^s : \tau$ for some τ
 - (b) $H \vdash_x^{\tau} \langle L', \bar{t} \rangle_x^l \circ GS : \sigma$
 5. By 4.a) and (T-Frame1)
 - (a) $\Gamma \vdash \text{return } y; : \tau$
 - (b) $\Gamma \vdash \bar{s} : \tau$
 - (c) $dom(\Gamma) \subseteq dom(L)$
 - (d) $\forall (z : \rho) \in \Gamma. \text{typeof}(L(z), H) = \rho$
 6. By 5.a) and (C-ReturnExp), $y : \tau \in \Gamma$.
 7. By 5.d) and 6., $\text{typeof}(L(y), H) = \tau$.
 8. We show: $H \vdash FS' : \sigma$
 - (a) case $GS \neq \epsilon$
 - i. By 4.b) and (T-FS4)
 - (a) $H \vdash_x^{\tau} \langle L', \bar{t} \rangle_x^l : \tau'$ for some τ'
 - (b) $H \vdash GS : \sigma$
 - ii. By i.a) and (T-Frame2)
 - (a) $\Gamma', x : \tau \vdash \bar{t} : \tau'$
 - (b) $dom(\Gamma') \subseteq dom(L')$
 - (c) $\forall (z : \rho) \in \Gamma'. \text{typeof}(L'(z), H) = \rho$
 - (d) $l = a(o, \bar{p}) \implies okObs(H, o, \tau')$

- iii. By ii.b), $\text{dom}(\Gamma', x : \tau) \subseteq \text{dom}(L'[x \mapsto L(y)])$.
 - iv. By ii.c) and 7., $\forall(z : \rho) \in (\Gamma', x : \tau)$. $\text{typeof}(L'[x \mapsto L(y)](z), H) = \rho$.
 - v. By ii.a), ii.d), iii., iv., and (T-Frame1), $H \vdash \langle L'[x \mapsto L(y)], \bar{t} \rangle^l : \tau'$.
 - vi. By i.b), v. and (T-FS2), $H \vdash FS' : \sigma$.
- (b) case $GS = \epsilon$
- i. By 4.b) and (T-FS3), $H \vdash_x^{\tau} \langle L', \bar{t} \rangle_x^l : \sigma$.
 - ii. By i. and (T-Frame2)
 - (a) $\Gamma', x : \tau \vdash \bar{t} : \sigma$
 - (b) $\text{dom}(\Gamma') \subseteq \text{dom}(L')$
 - (c) $\forall(z : \rho) \in \Gamma'$. $\text{typeof}(L'(z), H) = \rho$
 - (d) $l = a(o, \bar{p}) \implies \text{okObs}(H, o, \sigma)$
 - iii. By ii.b), $\text{dom}(\Gamma', x : \tau) \subseteq \text{dom}(L'[x \mapsto L(y)])$.
 - iv. By ii.c) and 7., $\forall(z : \rho) \in (\Gamma', x : \tau)$. $\text{typeof}(L'[x \mapsto L(y)](z), H) = \rho$.
 - v. By ii.a), ii.d), iii., iv., and (T-Frame1), $H \vdash \langle L'[x \mapsto L(y)], \bar{t} \rangle^l : \sigma$.
 - vi. By v. and (T-FS1), $H \vdash FS' : \sigma$.
9. By 1.d) and (FS-ok)
- (a) $H \vdash \langle L', \bar{t} \rangle_x^l \mathbf{ok}$
 - (b) $\text{obsIds}(\langle L', \bar{t} \rangle_x^l) \# \text{obsIds}(GS)$
10. By 9.a), (SF-ok), (CSF-ok) and (AF-ok)
- (a) $H \vdash \langle L'[x \mapsto L(y)], \bar{t} \rangle^l \mathbf{ok}$
 - (b) $\text{obsIds}(\langle L'[x \mapsto L(y)], \bar{t} \rangle^l) \# \text{obsIds}(GS)$
11. By 10.a-b) and (FS-ok), $H \vdash FS' \mathbf{ok}$.
12. 2.c), 3.a), 3.b), 8. and 11. conclude this case.
- Case (E-Return) follows analogously.
- Case (E-RAsync-Method)
1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \mathbf{ok}$
 - (c) $H \vdash FS : \sigma$
 - (d) $H \vdash FS \mathbf{ok}$
 - (e) $H, FS \rightarrow H', FS'$
 2. By (E-RAsync-Method)
 - (a) $FS = F \circ GS$
 - (b) $F = \langle L, x=y.m(\bar{z}); \bar{s} \rangle^l$
 - (c) $FS' = \langle L', \bar{t} \rangle^{a(o, \bar{p})} \circ \langle L[x \mapsto o], \bar{s} \rangle^l \circ GS$
 - (d) $H(L(y)) = \langle \rho, FM \rangle$
 - (e) $\text{mbody}(\rho, m) = mb : (\bar{\sigma} \bar{x}) \rightarrow^a \psi, mb = \bar{\tau} \bar{y}; \bar{t}$
 - (f) $L' = [\bar{x} \mapsto L(\bar{z}), \bar{y} \mapsto \text{default}(\bar{\tau}), \text{this} \mapsto L(y)]$
 - (g) $H_0 = H[o \mapsto \langle \psi, \text{running}(\epsilon, \epsilon) \rangle]$
 - (h) $o \notin \text{dom}(H)$
 - (i) $\bar{p} = [L(z_i) \mid z_i \in \bar{z} \wedge \sigma_i = \text{Observable} \langle \rho_i \rangle]$
 - (j) $H' = \text{subscribe}(o, \bar{p}, H_0)$
 3. By 2.j) and Definition *subscribe* (see Fig. 16)
 - (a) $\forall p_i \in \bar{p}$. $H_0(p_i) = \langle \psi_i, \text{running}(\bar{F}_i, \bar{S}_i) \rangle$
 - (b) $|\bar{p}| = n$
 - (c) $\forall i \in 1 \dots n$. $H_i = H_{i-1}[p_i \mapsto \langle \psi_i, \text{running}(\bar{F}_i, \langle o, [] \rangle :: \bar{S}_i) \rangle]$
 - (d) $H' = H_n$
 4. By 2.d-j), 3.a-d), and Definition 2, $\vdash H' : \star$.
 5. By 1.b), (H-ok) and 3.d), $\forall o' \in \text{dom}(H') \setminus (\{o\} \cup \bar{p})$. $H' \vdash H'(o') \mathbf{ok}$.
 6. By 2.g) and (ROHO-ok), $H' \vdash H'(o) \mathbf{ok}$.
 7. By 2.h), 3.a-d) and (ROHO-ok), $\forall p_i \in \bar{p}$. $H' \vdash H'(p_i) \mathbf{ok}$.
 8. By 1.b) and (H-ok)
 - (a) $\forall o \in \text{dom}(H)$. $H \vdash H(o) \mathbf{ok}$
 - (b) $\forall o_1 \neq o_2 \in \text{dom}(H)$. $\text{waiters}(H(o_1)) \# \text{waiters}(H(o_2))$.
 9. By 5., 6. and 7., $\forall o' \in \text{dom}(H')$. $H' \vdash H'(o') \mathbf{ok}$.
 10. By 2.g), 3.a-c), 8.b), $\forall o_1 \neq o_2 \in \text{dom}(H')$. $\text{waiters}(H'(o_1)) \# \text{waiters}(H'(o_2))$.
 11. By 9., 10. and (H-ok), $\vdash H' \mathbf{ok}$.
 12. We show $H \vdash F : \tau$ for some τ
 - (a) Case $GS = \epsilon$: by 1.c), 2.a) and (T-FS1), $H \vdash F : \sigma$
 - (b) Case $GS \neq \epsilon \wedge l = s$: by 1.c), 2.a), 2.b) and (T-FS5), $H \vdash F : \tau$ for some τ
 - (c) Case $GS \neq \epsilon \wedge l = a(o, \bar{p})$: by 1.c), 2.a), 2.b) and (T-FS2), $H \vdash F : \tau$ for some τ

13. By 2.b), 12. and (T-Frame1)
 - (a) $\Gamma \vdash x=y.m(\bar{z}) ; : \tau$
 - (b) $\Gamma \vdash \bar{s} : \tau$
 - (c) $dom(\Gamma) \subseteq dom(L)$
 - (d) $\forall(z : \rho) \in \Gamma. typeof(L(z), H) = \rho$
 - (e) $l = a(o, \bar{p}) \implies okObs(H, o, \tau)$
 14. By 13.a) and (C-Asn)
 - (a) $(x : \psi') \in \Gamma$
 - (b) $\Gamma \vdash y.m(\bar{z}) : \psi'$
 15. By 2.d-e), 14.b), and (C-MethInv)
 - (a) $(y : \rho) \in \Gamma$
 - (b) $\Gamma \vdash \bar{z} : \bar{\sigma}$
 - (c) $\psi' = \psi$
 16. By 2.e) and (AsyncMeth-OK)
 - (a) $\Gamma' = \bar{x} : \bar{\sigma}, \bar{y} : \bar{\tau}, this : \rho$
 - (b) $\Gamma' \vdash \bar{t} : \rho'$
 - (c) $\psi = Observable\langle \rho' \rangle$
 17. By 2.d.f), 13.d), 15.b), and 16.a), $\forall(z : \sigma) \in \Gamma'. typeof(L'(z), H') = \sigma$.
 18. By 2.f-j), 16.a-b), 17., and (T-Frame1), $H' \vdash \langle L', \bar{t} \rangle^{a(o, \bar{p})} : \rho'$.
 19. By 13.c-d), and 14.a), $dom(\Gamma) \subseteq dom(L[x \mapsto o])$.
 20. By 2.g), and 3.c-d), $H'(o) = \langle \psi, running(\epsilon, \epsilon) \rangle$.
 21. By 14.a) and 15.c), $\forall(z : \rho) \in \Gamma. typeof(L[x \mapsto o](z), H') = \rho$.
 22. By 2.j), 13.b), 13.e), 19., 21. and (T-Frame1), $H' \vdash \langle L[x \mapsto o], \bar{s} \rangle^l : \tau$.
 23. By 3.a-d), $\forall o' \in dom(H') \cap dom(H). typeof(H'(o')) = typeof(H(o'))$.
 24. We show $H' \vdash FS' : \sigma$
 - (a) Case $GS = \epsilon$
 - i. By 1.c), 2.a) and (T-FS1), $H \vdash F : \sigma$
 - ii. By 22., i. and (T-FS1), $H' \vdash \langle L[x \mapsto o], \bar{s} \rangle^l \circ \epsilon : \sigma$
 - iii. By 2.c), 18., ii. and (T-FS2), $H' \vdash FS' : \sigma$
 - (b) Case $GS \neq \epsilon \wedge l = s$
 - i. By 1.c), 2.a-b), 12. and (T-FS5)
 - (a) $GS = G_y \circ GS'$
 - (b) $H \vdash_y^{\tau} G_y \circ GS' : \sigma$
 - ii. By 23., i.b), (T-FS1-6), (T-Frame1) and (T-Frame2), $H' \vdash_y^{\tau} G_y \circ GS' : \sigma$
 - iii. By 22., ii. and (T-FS5), $H' \vdash \langle L[x \mapsto o], \bar{s} \rangle^l \circ G_y \circ GS' : \sigma$
 - iv. By 2.c), 18., iii. and (T-FS2), $H' \vdash FS' : \sigma$
 - (c) Case $GS \neq \epsilon \wedge l = a(o, \bar{p})$
 - i. By 1.c), 2.a), 2.b), 12. and (T-FS2), $H \vdash GS : \sigma$
 - ii. By 23., i., (T-FS1-6), (T-Frame1) and (T-Frame2), $H' \vdash GS : \sigma$
 - iii. By 22., ii. and (T-FS2), $H' \vdash \langle L[x \mapsto o], \bar{s} \rangle^l \circ GS : \sigma$
 - iv. By 2.c), 18., iii. and (T-FS2), $H' \vdash FS' : \sigma$
 25. By 2. and (AF-ok)
 - (a) $H' \vdash \langle L', \bar{t} \rangle^{a(o, \bar{p})} \mathbf{ok}$
 - (b) $H' \vdash \langle L[x \mapsto o], \bar{s} \rangle^l \mathbf{ok}$
 - (c) $H' \vdash GS \mathbf{ok}$
 26. By 2.h), $obsIds(\langle L', \bar{t} \rangle^{a(o, \bar{p})}) \# obsIds(\langle L[x \mapsto o], \bar{s} \rangle^l \circ GS)$.
 27. By 1.d) and (FS-ok)
 - (a) $H \vdash F \mathbf{ok}$
 - (b) $H \vdash GS \mathbf{ok}$
 - (c) $obsIds(F) \# obsIds(GS)$
 28. By 2.b) and 27.c), $obsIds(\langle L[x \mapsto o], \bar{s} \rangle^l) \# obsIds(GS)$.
 29. By 25.a-c), 26., 28. and (FS-ok), $H' \vdash FS' \mathbf{ok}$.
 30. By 2. and Definition 2, $H \leq_{obsIds(FS)} H'$.
 31. 4., 11., 24., 29. and 30. conclude this case.
- Case (E-Await1)
1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \mathbf{ok}$
 - (c) $H \vdash FS : \sigma$
 - (d) $H \vdash FS \mathbf{ok}$
 - (e) $H, FS \implies H', FS'$

2. By (E-Await1)
 - (a) $FS = F \circ FS'$
 - (b) $F = \langle L, x = \text{await } y; \bar{s} \rangle^{\alpha(o, \bar{p})}$
 - (c) $L(y) = o'$
 - (d) $H(o') = (\text{Observable}\langle \sigma \rangle, \text{running}(\bar{F}, \bar{S}))$
 - (e) $\bar{S} = \bar{R} \oplus [(o, [])]$
 - (f) $H' = H[o' \mapsto (\text{Observable}\langle \sigma \rangle, \text{running}(F :: \bar{F}, \bar{R}))]$
3. By 1.a), 2.d), 2.f) and Definition 3, $\vdash H' : \star$.
4. By 1.b) and (H-ok)
 - (a) $\forall q \in \text{dom}(H). H \vdash H(q) \text{ ok}$
 - (b) $\forall o_1 \neq o_2 \in \text{dom}(H). \text{waiters}(H(o_1)) \# \text{waiters}(H(o_2))$
5. By 1.d), 2.a) and (FS-ok)
 - (a) $H \vdash F \text{ ok}$
 - (b) $H \vdash FS' \text{ ok}$
 - (c) $\text{obsIds}(F) \# \text{obsIds}(FS')$
6. By 5.a), 2.b) and (AF-ok)
 - (a) $\text{Running}(H(o))$
 - (b) $\forall q \in \text{dom}(H). o \notin \text{waiters}(H(q)) \wedge (o \in \text{subscribers}(H(q)) \Leftrightarrow q \in \bar{p})$
7. By 6.b), $\forall i \in \{1..n\}. \text{obsIds}(F) \# \text{obsIds}(F_i)$.
8. By 2.d), 4.a) and (ROHO-ok)
 - (a) $\forall i \neq j \in \{1..n\}. \text{obsIds}(F_i) \# \text{obsIds}(F_j)$
 - (b) $\forall i \in \{1..n\}. \forall o \in \text{obsIds}(F_i). \text{Running}(H(o))$
9. By 8.b) and 2.f), $\forall i \in \{1..n\}. \forall o \in \text{obsIds}(F_i). \text{Running}(H'(o))$.
10. By 7., 8.a), 9. and (ROHO-ok), $H' \vdash H'(o') \text{ ok}$
11. By 10., 2.f) and 4.a), $\forall q \in \text{dom}(H'). H' \vdash H'(q) \text{ ok}$.
12. By 4.b), 6.b) and 2.f), $\forall o_1 \neq o_2 \in \text{dom}(H'). \text{waiters}(H'(o_1)) \# \text{waiters}(H'(o_2))$.
13. By 11., 12. and (H-ok), $\vdash H' \text{ ok}$.
14. Given that $FS' \neq \epsilon$, by 1.c), 2.a-b) and (T-FS2), $H \vdash FS' : \sigma$
15. By 2.f) and 14., $H' \vdash FS' : \sigma$.
16. Let $K^{\alpha(r, \bar{q})} = G \in FS'$.
 - (a) By 5.c), $r \neq o$
 - (b) By 5.b) and (FS-ok), $H \vdash G \text{ ok}$.
 - (c) By 16.b) and (AF-ok), $\text{Running}(H(r)) \wedge \forall o' \in \text{dom}(H). r \notin \text{waiters}(H(o')) \wedge (r \in \text{subscribers}(H(o')) \Leftrightarrow o' \in \bar{q})$
 - (d) By 16.a), 16.c) and 2.f), $H' \vdash G \text{ ok}$.
 - (e) By 5.b), 16.d) and (FS-ok), $H' \vdash FS' \text{ ok}$
17. By 2.d), 2.f) and Definition 2, $H \leq_{\text{obsIds}(FS)} H'$.
18. 17., 16.e), 15., 13. and 3. conclude this case. \square

A.3. Proof of Lemma 3

Lemma 3. *If $\vdash H : \star$ and $\vdash H \text{ ok}$ then:*

If $H \vdash P : \star$, $H \vdash P \text{ ok}$ and $H, P \rightsquigarrow H', P'$ then $\vdash H' : \star$, $\vdash H' \text{ ok}$, $H' \vdash P' : \star$ and $H' \vdash P' \text{ ok}$.

Proof. By induction on the derivation of $H, P \rightsquigarrow H', P'$ and Lemma 2.

- Case (E-Exit) is trivial.
- Case (E-Schedule)
 1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \text{ ok}$
 - (c) $H \vdash P : \star$
 - (d) $H \vdash P \text{ ok}$
 - (e) $H, P \rightsquigarrow H', P'$
 2. By (E-Schedule)
 - (a) $P = \{FS\} \cup Q$
 - (b) $P' = \{FS'\} \cup Q$
 - (c) $H, FS \Rightarrow H', FS'$
 3. By 1.c), 2.a) and (T-Proc) there is σ such that
 - (a) $H \vdash FS : \sigma$
 - (b) $H \vdash Q : \star$

4. By 1.d) and (Proc-ok)
 - (a) $H \vdash FS \text{ ok}$
 - (b) $\forall GS \in Q. H \vdash GS \text{ ok} \wedge \text{obsIds}(GS) \# \text{obsIds}(FS)$
 5. By 1.a-b), 2.c), 3.a), 4.a) and Lemma 2
 - (a) $\vdash H' : \star$
 - (b) $\vdash H' \text{ ok}$
 - (c) $H' \vdash FS' : \sigma$
 - (d) $H' \vdash FS' \text{ ok}$
 - (e) $H \leq_{\text{obsIds}(FS)} H'$
 6. By 2.b), 3.b), 5.c) and (T-Proc) we have $H' \vdash P' : \star$.
 7. By 2.c) and inspection of rules (E-Method-Exp), (E-Method-Stmt), (E-Return-Val), (E-Return), (E-Frame), (E-RASync-Method) and (E-Await1-3) we have $\forall GS \in Q. \text{obsIds}(GS) \# \text{obsIds}(FS')$.
 8. Let $GS \in Q$ where $GS = G \circ GS'$ for some G, GS' . We show $H' \vdash GS \text{ ok}$ by induction on the size of GS .
 - (a) By 4.b), (EmpFS-ok) and (FS-ok) we have $H \vdash G \text{ ok}$, $H \vdash GS' \text{ ok}$ and $\text{obsIds}(G) \# \text{obsIds}(GS')$.
 - (b) Trivially, $(G = F^s \vee G = F_x^s) \implies H' \vdash G \text{ ok}$
 - (c) By 8.b), the definition of obsIds and the IH $(G = F^s \vee G = F_x^s) \implies H' \vdash GS \text{ ok}$.
 - (d) Let $G = F^{a(o, \bar{p})}$. By 4.b) $o \notin \text{obsIds}(FS)$.
 - (e) By 8.a), 8.d) and (AF-ok) we have $\text{Running}(H(o))$ and $\forall o' \in \text{dom}(H). o \notin \text{waiters}(H(o')) \wedge (o \in \text{subscribers}(H(o')) \Leftrightarrow o' \in \bar{p})$
 - (f) By 2.c), 8.e), (E-RASync-Method) and (E-Await1-3) we have $\text{Running}(H'(o))$.
 - (g) By 2.c), 8.d), 8.e), (E-Frame), (E-New) and (E-RASync-Method) we have $\forall o' \in \text{dom}(H'). o \notin \text{waiters}(H'(o'))$
 - (h) Let $o' \in \text{dom}(H')$. By 2.c), (E-RASync-Method), (E-Await1-3) and (E-Frame) we have $o \in \text{subscribers}(H'(o')) \Leftrightarrow o \in \text{subscribers}(H(o'))$.
 - (i) By 8.e) and 8.h) we have $\forall o' \in \text{dom}(H'). o \in \text{subscribers}(H'(o')) \Leftrightarrow o' \in \bar{p}$.
 - (j) By 8.f), 8.g), 8.i) and (AF-ok) we have $H' \vdash G \text{ ok}$.
 - (k) By 8.j), the definition of obsIds and the IH we have $G = F^{a(o, \bar{p})} \implies H' \vdash GS \text{ ok}$.
 - (l) By 8.c) and 8.k) we have $H' \vdash GS \text{ ok}$.
 9. By 2.b), 5.c), 7., 8. and (Proc-ok) we have $H' \vdash P' \text{ ok}$.
 10. 5.a), 5.b), 6. and 9. conclude this case.
- Case (E-Yield)
1. By the assumptions
 - (a) $\vdash H : \star$
 - (b) $\vdash H \text{ ok}$
 - (c) $H \vdash P : \star$
 - (d) $H \vdash P \text{ ok}$
 - (e) $H, P \rightsquigarrow H', P'$
 2. By (E-Yield)
 - (a) $P = \{GS\} \cup T$
 - (b) $GS = \langle L, \text{yieldNext } z; \bar{s} \rangle^{a(o, \bar{p})} \circ FS$
 - (c) $H(o) = \langle \text{Observable} \langle \sigma \rangle, \text{running}(\bar{F}, \bar{S}) \rangle$
 - (d) $(\bar{o}, \bar{R}) = \text{resume}(\bar{F}, \text{Some}(L(z)))$
 - (e) $Q = \{R \circ \epsilon \mid R \in \bar{R}\}$
 - (f) $\bar{S}' = [\langle o', q :: L(z) \mid \langle o', q \rangle \in \bar{S} \rangle :: [\langle o_i, [] \rangle \mid o_i \in \bar{o}]]$
 - (g) $H' = H[o \mapsto \langle \text{Observable} \langle \sigma \rangle, \text{running}(\epsilon, \bar{S}') \rangle]$
 - (h) $P' = \{\langle L, \bar{s} \rangle^{a(o, \bar{p})} \circ FS\} \cup T \cup Q$
 3. By 1.c), 2.a) and (T-Proc) there is σ' such that
 - (a) $H \vdash GS : \sigma'$
 - (b) $H \vdash T : \star$
 4. By 1.d), 2.a) and (Proc-ok)
 - (a) $H \vdash GS \text{ ok}$
 - (b) $\forall HS \in T. H \vdash HS \text{ ok} \wedge \text{obsIds}(HS) \# \text{obsIds}(GS)$
 - (c) $\forall HS_i, HS_j \in T, i \neq j. \text{obsIds}(HS_i) \# \text{obsIds}(HS_j)$
 5. By 1.a) and 2.g) we have $\vdash H' : \star$.
 6. By 1.b) and (H-ok) we have $\forall o' \in \text{dom}(H). H \vdash H(o') \text{ ok}$.
 7. By (ROHO-ok) we have $H' \vdash \langle \text{Observable} \langle \sigma \rangle, \text{running}(\epsilon, \bar{S}') \rangle \text{ ok}$.
 8. By 2.g), 6. and 7. we have $\forall o' \in \text{dom}(H'). H' \vdash H'(o') \text{ ok}$.
 9. By 2.g) and def. waiters we have $\text{waiters}(H'(o)) = \emptyset$.
 10. By 1.b), 2.g), 9. and (H-ok) we have $\forall o_1 \neq o_2 \in \text{dom}(H'). \text{waiters}(H'(o_1)) \# \text{waiters}(H'(o_2))$.
 11. By 8., 10. and (H-ok) we have $\vdash H' \text{ ok}$.
 12. By 2.b), 3.a), (T-FS1) and (T-FS2) there is τ such that
 - (a) $H \vdash \langle L, \text{yieldNext } z; \bar{s} \rangle^{a(o, \bar{p})} : \tau$

- (b) $FS = \epsilon \vee H \vdash FS : \sigma'$
13. By 2.c), 12.a) and (T-Frame1) there is Γ such that
- (a) $\Gamma \vdash \text{yieldNext } z; \bar{s} : \sigma$
- (b) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
- (c) $\forall (x : \rho) \in \Gamma. \text{typeof}(L(x), H) = \rho$
14. By 2.c), 2.g) and 13.c) we have $\forall (x : \rho) \in \Gamma. \text{typeof}(L(x), H') = \rho$.
15. By 13.a-b), 14. and (T-Frame1) we have $H' \vdash \langle L, \bar{s} \rangle^{a(o, \bar{p})} : \sigma$.
16. By 2.c), 2.g) and 12.b) we have $FS = \epsilon \vee H' \vdash FS : \sigma'$.
17. By 15., 16., (T-FS1) and (T-FS2) we have $H' \vdash \langle L, \bar{s} \rangle^{a(o, \bar{p})} \circ FS : \sigma''$ for some σ'' .
18. Let $R = \langle K[x \mapsto \text{Some}(L(z))], \bar{t} \rangle^{a(o', \bar{q})}$ where $\langle K, x = \text{await } y; \bar{t} \rangle^{a(o', \bar{q})} \in \bar{F}$
19. By 18. and (T-Frame1)
- (a) $H \vdash \langle K, x = \text{await } y; \bar{t} \rangle^{a(o', \bar{q})} : \tau''$ for some τ''
- (b) $\text{typeof}(K(x), H) = \text{Option}\langle \sigma \rangle$
20. By 19.a) and (T-Frame1) there is Γ' such that
- (a) $\Gamma' \vdash x = \text{await } y; \bar{t} : \tau''$
- (b) $\text{dom}(\Gamma') \subseteq \text{dom}(K)$
- (c) $\forall (w : \rho) \in \Gamma'. \text{typeof}(K(w), H) = \rho$
- (d) $\text{okObs}(H, o', \tau'')$
21. By 20.a-b), (C-Await) and (C-Asn)
- (a) $\Gamma' \vdash \bar{t} : \tau''$
- (b) $\text{dom}(\Gamma') \subseteq \text{dom}(K[x \mapsto \text{Some}(L(z))])$
22. By 13.a) and (C-Yield) we have $z : \sigma \in \Gamma$.
23. By 13.c) and 22. we have $\text{typeof}(L(z), H) = \sigma$.
24. By 2.c), 2.g) and 23. we have $\text{typeof}(L(z), H') = \sigma$.
25. By 24. and def. *typeof* we have $\text{typeof}(\text{Some}(L(z)), H') = \text{Option}\langle \sigma \rangle$.
26. By 19.b) and 20.c) we have $x : \text{Option}\langle \sigma \rangle \in \Gamma'$.
27. By 2.c), 2.g), 20.c), 25., 26. and def. *typeof* we have $\forall (w : \rho) \in \Gamma'. \text{typeof}(K[x \mapsto \text{Some}(L(z))](w), H') = \rho$.
28. By 2.c), 2.g) and 20.d) we have $\text{okObs}(H', o', \tau'')$.
29. By 21.a-b), 27., 28. and (T-Frame1) we have $H' \vdash R : \tau''$.
30. Since R was chosen arbitrarily in 18., by 2.e) and (T-Proc) we have $H' \vdash Q : \star$.
31. By 2.c), 2.g) and 3.b) we have $H' \vdash T : \star$.
32. By 17., 30., 31. and (T-Proc) we have $H' \vdash P' : \star$.
33. By 4.a) and (FS-ok)
- (a) $H \vdash \langle L, \text{yieldNext } z; \bar{s} \rangle^{a(o, \bar{p})} \mathbf{ok}$
- (b) $H \vdash FS \mathbf{ok}$
- (c) $\{o\} \# \text{obsIds}(FS)$
34. By 2.c) and 2.g)
- (a) $\text{Running}(H'(o))$
- (b) $\forall o' \in \text{dom}(H'). o \notin \text{waiters}(H'(o')) \wedge (o \in \text{subscribers}(H'(o'))) \iff o' \in \bar{p}$
35. By 34.a-b) and (AF-ok) we have $H' \vdash \langle L, \bar{s} \rangle^{a(o, \bar{p})} \mathbf{ok}$.
36. By 2.c), 2.g), 33.b) and (FS-ok) we have $H' \vdash FS \mathbf{ok}$.
37. Define $GS' := \langle L, \bar{s} \rangle^{a(o, \bar{p})} \circ FS$
38. By 33.c), 35., 36. and (FS-ok) we have $H' \vdash GS' \mathbf{ok}$.
39. By 2.b) and 37. we have $\text{obsIds}(GS) = \text{obsIds}(GS')$.
40. By 2.c), 2.g), 4.b), 37. and (FS-ok) we have $\forall HS \in T. H' \vdash HS \mathbf{ok} \wedge \text{obsIds}(HS) \# \text{obsIds}(GS')$.
41. By 2.c-g), def. *resume*, (AF-ok), (FS-ok), and (E-Await1-4) we have $\forall IS \in Q. H' \vdash IS \mathbf{ok}$.
42. By 2.c), 6. and (ROHO-ok) we have $\forall F_i, F_j \in \bar{F}, i \neq j. \text{obsIds}(F_i) \# \text{obsIds}(F_j)$.
43. By 2.d-e), 42. and def. *resume* we have $\forall IS_i, IS_j \in Q, i \neq j. \text{obsIds}(IS_i) \# \text{obsIds}(IS_j)$.
44. By 1.d), 2.a-e), 39., (Proc-ok) and (E-Await1)
- (a) $\forall IS \in Q. \text{obsIds}(IS) \# \text{obsIds}(GS')$
- (b) $\forall IS \in Q. \forall HS \in T. \text{obsIds}(IS) \# \text{obsIds}(HS)$
45. By 2.h), 4.c), 37., 38., 40., 41., 43., 44.a-b) and (Proc-ok) we have $H' \vdash P' \mathbf{ok}$.
46. 5., 11., 32. and 45. conclude this case.
- Cases (E-RAsync-Return) and (E-YieldDone) follow analogously. \square

A.4. Proof of Theorem 1

Theorem (Subject Reduction). *If $H \vdash H : \star$ and $H \mathbf{ok}$ then:*

1. *If $H \vdash F : \sigma, H \vdash F \mathbf{ok}$ and $H, F \longrightarrow H', F'$ then $H' \vdash H' : \star, H' \mathbf{ok}, H' \vdash F' : \sigma, H' \vdash F' \mathbf{ok}$, and $\forall B. H \leq_B H'$.*
2. *If $H \vdash FS : \sigma, H \vdash FS \mathbf{ok}$ and $H, FS \longrightarrow H', FS'$ then $H' \vdash H' : \star, H' \mathbf{ok}, H' \vdash FS' : \sigma, H' \vdash FS' \mathbf{ok}$ and $H \leq_{\text{obsIds}(FS)} H'$.*

3. If $H \vdash P : \star$, $H \vdash P$ **ok** and $H, P \rightsquigarrow H', P'$ then $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash P' : \star$ and $H' \vdash P'$ **ok**.

Proof. Corollary of Lemma 1, Lemma 2, and Lemma 3. \square

A.5. Proof of Theorem 2

Theorem (Progress). If $\vdash H : \star$ and $\vdash H$ **ok** then:

If $H \vdash P : \star$ and $H \vdash P$ **ok** then

1. $H, P \rightsquigarrow H', P'$ for some H', P' ; or
2. $\forall FS \in P$, one of the following holds:
 - (a) $FS = \langle L, \text{return}; \bar{t} \rangle^s \circ \epsilon$ or $FS = \langle L, \text{return } x; \bar{t} \rangle^s \circ \epsilon$
 - (b) $FS = \langle L, s; \bar{t} \rangle^l \circ FS'$ where $(s = y=x.m(\bar{z})$ or $s = x.m(\bar{z})$ or $s = y=x.f$ or $s = x.f=y$) and $L(x) = \text{null}$
 - (c) $FS = \langle L, y=x.m(\bar{z}); \bar{t} \rangle^l \circ FS'$ where $H(L(x)) = \langle \rho, FM \rangle$, $\text{mbody}(\rho, m) = \text{mb} : (\bar{\sigma} \bar{x}) \rightarrow^a \psi$, and $\exists p_i \in \{L(z_i) \mid z_i \in \bar{z} \wedge \sigma_i = \psi_i\}$ such that $H(p_i) = \langle \psi_i, \text{done}(\bar{S}) \rangle$
 - (d) $FS = \langle L, y=\text{await } x; \bar{t} \rangle^{a(o, \bar{p})} \circ FS'$ where $L(x) = \text{null}$ or $L(x) \notin \bar{p}$
 - (e) $FS = \langle L, \epsilon \rangle^l \circ FS'$
 - (f) $FS = \langle L, y=\text{get } x; \bar{t} \rangle^l \circ FS'$ where $L(x) = \text{None}$

Proof. If $\epsilon \in P$ the result follows trivially by E-EXIT. Let $\epsilon \neq FS \in P$ such that $FS \neq \langle L, \text{return}; \bar{t} \rangle^s \circ FS'$ and $FS \neq \langle L, \text{return } x; \bar{t} \rangle^s \circ FS'$. Then, $FS = F \circ FS'$ where $F = \langle L, s; \bar{t} \rangle^l$. Furthermore, let $P = \{FS\} \uplus Q$. We proceed with a case analysis of the shape of s .

- Case $s = y=x$.
 1. By (E-Var), $H, \langle L, y=x; \bar{t} \rangle^l \longrightarrow H, F'$ for some F'
 2. By (E-Frame), $H, F \circ FS' \longrightarrow H, F' \circ FS'$
 3. By (E-Schedule), $H, \{FS\} \cup Q \rightsquigarrow H, \{F' \circ FS'\} \cup Q$
- Case $s = y=x.f$.
 1. By $H \vdash P : \star$ and (T-Proc)
 - (a) $H \vdash FS : \sigma$ for some σ
 - (b) $H \vdash Q : \star$
 2. By 1.a) and (T-FS1-6)
 - (a) $FS' = \epsilon \implies H \vdash F : \sigma$
 - (b) $FS' \neq \epsilon \implies (H \vdash F : \tau \text{ for some } \tau \wedge H \vdash FS' : \sigma)$
 3. By 2.a) and 2.b), $H \vdash F : \tau$ for some τ
 4. By 3. and (T-Frame1)
 - (a) $\Gamma \vdash y=x.f : \tau$
 - (b) $\Gamma \vdash \bar{t} : \tau$
 - (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
 - (d) $\forall (z : \rho) \in \Gamma. \text{typeof}(L(z), H) = \rho$
 5. By 4.a) and (C-Asn), $\Gamma \vdash x.f : \tau'$ such that $(y : \tau') \in \Gamma$.
 6. By 5. and (C-Field)
 - (a) $(x : \rho') \in \Gamma$
 - (b) $\text{ftype}(\rho', f) = \tau'$
 7. By 4.d), 6.a-b), $H(L(x)) = (\rho', FM)$
 8. By 7., $\vdash H : \star$ and Definition 3
 - (a) $\text{dom}(FM) = \text{fields}(\rho')$
 - (b) $\forall f \in \text{dom}(FM). \text{typeof}(FM(f), H) = \text{ftype}(\rho', f)$
 9. By 6.b) and 8.a), $f \in \text{dom}(FM)$
 10. By 7., 9. and (E-Field), $H, F \longrightarrow H, \langle L[y \mapsto FM(f)], \bar{t} \rangle^l$
 11. 10., (E-Frame), and (E-Schedule) conclude this case.
- Case $s = y=\text{await } x$.
 1. By the assumptions and (T-Proc)
 - (a) $H \vdash FS : \sigma$ for some σ
 - (b) $H \vdash Q : \star$
 2. We show that $H \vdash F : \tau$ for some τ .
 - (a) Case $FS' = \epsilon$: by 1.a) and (T-FS1), $H \vdash F : \sigma$
 - (b) Case $FS' \neq \epsilon \wedge l = s$: by 1.a) and (T-FS5), $H \vdash F : \tau$ for some τ
 - (c) Case $FS' \neq \epsilon \wedge l = a(o, \bar{p})$: by 1.a) and (T-FS2), $H \vdash F : \tau$ for some τ
 3. By 2., (T-Frame1), (Meth-OK), and (AsyncMeth-OK)

- (a) $\Gamma \vdash y = \text{await } x : \tau$
- (b) $\Gamma \vdash \bar{t} : \tau$
- (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
- (d) $\forall (z : \rho) \in \Gamma. \text{typeof}(L(z), H) = \rho$
- (e) $l = a(o, \bar{p})$
- 4. By 3.a), (C-Asn) and (T-Await)
 - (a) $\Gamma \vdash \text{await } x : \text{Option}\langle \rho \rangle$ for some ρ
 - (b) $(x : \text{Observable}\langle \rho \rangle) \in \Gamma$
- 5. By $H \vdash P$ **ok** and (Proc-ok), $H \vdash FS$ **ok**
- 6. By 5. and (FS-ok), $H \vdash F$ **ok**
- 7. By 3.e), 6. and (AF-ok)
 - (a) $\text{Running}(H(o))$
 - (b) $\forall o' \in \text{dom}(H). o \notin \text{waiters}(H(o')) \wedge (o \in \text{subscribers}(H(o')) \Leftrightarrow o' \in \bar{p})$
- 8. By 3.d) and 4.b), $\text{typeof}(L(x), H) = \text{Observable}\langle \rho \rangle$
- 9. By 7.b) and 8.
 - (a) $o \notin \text{waiters}(H(L(x)))$
 - (b) $o \in \text{subscribers}(H(L(x))) \Leftrightarrow L(x) \in \bar{p}$
- 10. By 9.b), if $L(x) \in \bar{p}$ one of (E-Await1-4) applicable, so that $H, FS \rightarrow H', FS''$
- 11. By 9.b), if $L(x) \notin \bar{p}$ none of (E-Await1-4) applicable (nor any other reduction rule)
- 12. 10., 11., and (E-Schedule) conclude this case.
- Cases $s = x.f = y$, $s = y = \text{new } C()$, and others: analogous to case $s = y = x.f$.

Let $FS \in P$ such that $FS = \langle L, \text{return } y; \bar{t} \rangle^s \circ FS'$, $FS' \neq \epsilon$. Furthermore, let $P = \{FS\} \uplus Q$.

- 1. By $H \vdash P : \star$ and (T-Proc)
 - (a) $H \vdash \langle L, \text{return } y; \bar{t} \rangle^s \circ FS' : \sigma$ for some σ
 - (b) $H \vdash Q : \star$
- 2. By 1.a) and (T-FS5)
 - (a) $FS' = G_x \circ GS'$
 - (b) $H \vdash \langle L, \text{return } y; \bar{t} \rangle^s : \tau$ for some τ
 - (c) $H \vdash_x^{\tau} G_x \circ GS' : \sigma$
- 3. By 2.b) and (T-Frame1)
 - (a) $\Gamma \vdash \text{return } y; : \tau$
 - (b) $\Gamma \vdash \bar{t} : \tau$
 - (c) $\text{dom}(\Gamma) \subseteq \text{dom}(L)$
- 4. By 3.a) and (C-ReturnExp), $y : \tau \in \Gamma$
- 5. By 3.c) and 4., $y \in \text{dom}(L)$
- 6. By 2.a), 5. and (E-Return-Val), $H, FS \rightarrow H, \langle L'[x \mapsto L(y)], \bar{t} \rangle^l \circ GS'$ where $G_x = \langle L', \bar{t} \rangle_x^l$
- 7. 6. and (E-Schedule) conclude this case.

Case $FS = \langle L, \text{return}; \bar{t} \rangle^s \circ FS'$ is analogous. \square

References

- [1] Gul A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, Artif. Intell., The MIT Press, Cambridge, Massachusetts, 1986.
- [2] Gul A. Agha, Ian A. Mason, Scott F. Smith, Carolyn L. Talcott, A foundation for actor computation, *J. Funct. Program.* 7 (1) (January 1997) 1–72.
- [3] J. Armstrong, Erlang – a survey of the language and its industrial applications, in: *INAP'96 – The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, Hino, Tokyo, Japan, October 1996, pp. 16–18.
- [4] Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams, *Concurrent Programming in Erlang*, Prentice Hall, 1996.
- [5] Kevin Bierhoff, Nels E. Beckman, Jonathan Aldrich, Practical API protocol checking with access permissions, in: *Sophia Drossopoulou (Ed.), ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 5653, Springer, 2009, pp. 195–219.
- [6] Gavin Bierman, Matthew Parkinson, Andrew Pitts, MJ: *An Imperative Core Calculus for Java and Java with Effects*, Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003.
- [7] Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, Mads Torgersen, Pause 'n' play: formalizing asynchronous C#, in: *ECOOP*, Springer, 2012, pp. 233–257.
- [8] Gilad Bracha, *The Dart Programming Language*, Addison-Wesley, Boston, Massachusetts, 2015.
- [9] Eugene Burmako, Scala macros: let our powers combine!, in: *Proceedings of the 4th Workshop on Scala*, ACM, 2013, pp. 3:1–3:10.
- [10] Frank S. de Boer, Dave Clarke, Einar Broch Johnsen, A complete guide to the future, in: *Rocco De Nicola (Ed.), ESOP*, in: *Lect. Notes Comput. Sci.*, vol. 4421, Springer, 2007, pp. 316–330.
- [11] Robert DeLine, Manuel Fähndrich, Typestates for objects, in: *Martin Odersky (Ed.), ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 3086, Springer, 2004, pp. 465–490.
- [12] Torbjörn Ekman, Görel Hedin, Pluggable checking and inferencing of nonnull types for Java, *J. Object Technol.* 6 (9) (2007) 455–475.
- [13] Marius Eriksen, Your server as a function, *Oper. Syst. Rev.* 48 (1) (2014) 51–57.
- [14] Doug Lea, et al., JSR166: concurrency utilities, <http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/>.

- [15] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, Anne-Marie Kermarrec, The many faces of publish/subscribe, *ACM Comput. Surv.* 35 (2) (2003) 114–131.
- [16] Manuel Fähndrich, K. Rustan M. Leino, Declaring and checking non-null types in an object-oriented language, in: Ron Crocker, Guy L. Steele Jr. (Eds.), *OOPSLA*, ACM, 2003, pp. 302–312.
- [17] Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen, A programmer's reduction semantics for classes and mixins, in: Jim Alves-Foss (Ed.), *Formal Syntax and Semantics of Java*, in: *Lect. Notes Comput. Sci.*, vol. 1523, Springer, 1999, pp. 241–269.
- [18] Steffen Forkmann, Tomas Petricek, Ryan Riley, Mauricio Scheffer, Jack Fox, The FSharp.Control.AsyncSeq library, <http://fsprojects.github.io/FSharp.Control.AsyncSeq/>.
- [19] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, Alan Schmitt, JoCaml: a language for concurrent distributed and mobile programming, in: Johan Jeuring, Simon L. Peyton Jones (Eds.), *Advanced Functional Programming*, in: *Lect. Notes Comput. Sci.*, vol. 2638, Springer, 2002, pp. 129–158.
- [20] Cédric Fournet, Georges Gonthier, The reflexive CHAM and the join-calculus, in: Hans-Juergen Boehm, Guy L. Steele Jr. (Eds.), *POPL*, ACM, 1996, pp. 372–385.
- [21] Reactive Streams Working Group. *Reactive Streams 1.0*, <http://www.reactive-streams.org/>, April 2015.
- [22] Philipp Haller, On the integration of the actor model in mainstream technologies: the Scala perspective, in: Gul A. Agha, Rafael H. Bordini, Assaf Marron, Alessandro Ricci (Eds.), *AGERE!@SPLASH*, ACM, 2012, pp. 1–6.
- [23] Philipp Haller, Tom Van Cutsem, Implementing joins using extensible pattern matching, in: *COORDINATION*, Springer, 2008, pp. 135–152.
- [24] Philipp Haller, Martin Odersky, Scala actors: unifying thread-based and event-based programming, *Theor. Comput. Sci.* 410 (2–3) (2009) 202–220.
- [25] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, Vojin Jovanovic, SIP-14: futures and promises, 2012, <http://docs.scala-lang.org/sips/completed/futures-promises.html>.
- [26] Philipp Haller, Jason Zaugg, SIP-22: Async, <http://docs.scala-lang.org/sips/pending/async.html>, 2013.
- [27] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde (Eds.), *The C# Programming Language*, fourth edition, Addison-Wesley, 2011.
- [28] Hewitt Carl, Viewing control structures as patterns of passing messages, *Artif. Intell.* 8 (3) (1977) 323–364.
- [29] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
- [30] Kohei Honda, Mario Tokoro, An object calculus for asynchronous communication, in: Pierre America (Ed.), *ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 512, Springer, 1991, pp. 133–147.
- [31] Kohei Honda, Nobuko Yoshida, Marco Carbone, Multiparty asynchronous session types, *J. ACM* 63 (1) (2016) 9:1–9:67.
- [32] Atsushi Igarashi, Benjamin C. Pierce, Philip Wadler, Featherweight Java: a minimal core calculus for Java and GJ, *ACM Trans. Program. Lang. Syst.* 23 (3) (2001) 396–450.
- [33] Ecma International, *Dart Programming Language Specification*, 4th edition (ECMA-408), <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-408.pdf>, 2015.
- [34] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, Yanling Wang, Cyclone: a safe dialect of C, in: Carla Schlatte Ellis (Ed.), *USENIX Annual Technical Conference, General Track*, USENIX, 2002, pp. 275–288.
- [35] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, Martin Steffen, ABS: a core language for abstract behavioral specification, in: *FMCO*, Springer, 2010, pp. 142–164.
- [36] Einar Broch Johnsen, Olaf Owe, An asynchronous communication model for distributed concurrent objects, *Softw. Syst. Model.* 6 (1) (2007) 39–58.
- [37] R.E. Johnson, B. Foote, Designing reusable classes, *J. Object-Oriented Program.* 1 (2) (1988) 22–35.
- [38] Robert H. Halstead Jr., Multilisp: a language for concurrent symbolic computation, *ACM Trans. Program. Lang. Syst.* 7 (4) (1985) 501–538.
- [39] Rajesh K. Karmani, Amin Shali, Gul Agha, Actor frameworks for the JVM platform: a comparative analysis, in: Ben Stephenson, Christian W. Probst (Eds.), *PPPJ*, ACM, 2009, pp. 11–20.
- [40] Barbara Liskov, Liuba Shrira, Promises: linguistic support for efficient asynchronous procedure calls in distributed systems, in: Richard L. Wexelblat (Ed.), *PLDI*, ACM, 1988, pp. 260–267.
- [41] Erik Meijer, Your mouse is a database, *Commun. ACM* 55 (5) (2012) 66–73.
- [42] Erik Meijer, Kevin Millikin, Gilad Bracha, Spicing up Dart with side effects, *ACM Queue* 13 (3) (2015) 40.
- [43] Heather Miller, Philipp Haller, Martin Odersky, Spores: a type-based foundation for closures in the age of concurrency and distribution, in: Richard Jones (Ed.), *ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 8586, Springer, 2014, pp. 308–333.
- [44] Robin Milner, *Communicating and Mobile Systems – The Pi-Calculus*, Cambridge University Press, 1999.
- [45] J. Gregory Morrisett, David Walker, Karl Cray, Neal Glew, From system F to typed assembly language, *ACM Trans. Program. Lang. Syst.* 21 (3) (1999) 527–568.
- [46] Martin Odersky, Lex Spoon, Bill Venner, *Programming in Scala*, Artima, second edition, 2010.
- [47] Semih Okur, Cansu Erdogan, Danny Dig, Converting parallel code from low-level abstractions to higher-level abstractions, in: Richard Jones (Ed.), *ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 8586, Springer, 2014, pp. 515–540.
- [48] Johan Östlund, Tobias Wrigstad, *Welterweight Java*, in: *TOOLS* (48), Springer, 2010, pp. 97–116.
- [49] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, Ryan Stutsman, The case for RAMCloud, *Commun. ACM* 54 (7) (2011) 121–130.
- [50] Tomas Petricek, Don Syme, The F# computation expression zoo, in: Matthew Flatt, Hai-Feng Guo (Eds.), *PADL*, in: *Lect. Notes Comput. Sci.*, vol. 8324, Springer, 2014, pp. 33–48.
- [51] Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
- [52] Jan Schäfer, Arnd Poetzsch-Heffter, JCobox: generalizing active objects to concurrent components, in: Theo D'Hondt (Ed.), *ECOOP*, in: *Lect. Notes Comput. Sci.*, vol. 6183, Springer, 2010, pp. 275–299.
- [53] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.
- [54] Don Syme, Tomas Petricek, Dmitry Lomov, The F# asynchronous programming model, in: Ricardo Rocha, John Launchbury (Eds.), *PADL*, Springer, 2011, pp. 175–189.
- [55] J. Robert von Behren, Jeremy Condit, Eric A. Brewer, Why events are a bad idea (for high-concurrency servers), in: Michael B. Jones (Ed.), *HotOS, USENIX*, 2003, pp. 19–24.
- [56] Andrew K. Wright, Matthias Felleisen, A syntactic approach to type soundness, *Inf. Comput.* 115 (1) (November 1994) 38–94.