A formal model for direct-style asynchronous observables

Philipp Haller[1] and Heather Miller[2]

[1] KTH Royal Institute of Technology, Sweden
phaller@kth.se
[2] EPFL, Switzerland
heather.miller@epfl.ch

1 Introduction

Asynchronous programming has been a challenge for a long time. A multitude of programming models have been proposed that aim to simplify the task. Interestingly, there are elements of a convergence arising, at least with respect to the basic building blocks: futures and promises have begun to play an increasingly important role in a number of languages like Java, C++, ECMAScript, and Scala. The Async extensions of F# [9], C# [1], and Scala [4] provide language support for programming with futures in *direct style*, by avoiding an inversion of control that is inherent in designs based on callbacks.

In this paper we present an integration of the Async model with a richer underlying abstraction, the *asynchronous observables* of the Reactive Extensions model [8]. An asynchronous observable is a stream of *observable events* which an arbitrary number of *observers* can subscribe to. The set of possible event patterns of asynchronous observables is strictly greater than those of futures. An observable (or stream) can (a) produce zero or more regular events, (b) complete normally, or (c) complete with an error (it is even possible for a stream to never complete.) Given the richer substrate of observables, the Async model has to be generalized in several dimensions.

We call our model RAY, inspired by its main constructs, *reactive async*, *await*, and *yield*. This paper makes the following contributions:

- The design of a new programming model, RAY, which integrates the Async model and the Reactive Extensions model;
- Structural operational semantics of the proposed programming model. Our operational semantics generalizes the formal model presented in [1] for C#'s async/await to asynchronous observables.

2 Background

Scala Async. Scala Async provides constructs that aim to facilitate programming with asynchronous events in Scala. The introduced constructs are inspired by extensions that have been introduced in C# version 5 [5]. 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. Example:

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

The `await` on line 2 causes the execution of the `async` block to suspend until `futureDOY` is completed (with a successful result or with an exception). When the future is completed successfully, its result is bound to the `dayOfYear` local variable, and the execution of the `async` block is resumed. When the future is completed with an exception (*e.g.,* 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 principle methods, `async` and `await`, have the following type signatures:

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

Notably, `async` and `await` "cancel each other out:" `await(async { <expr> }) = <expr>`

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

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

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

3 Direct-style asynchronous observables

The following example demonstrates our programming model:

```
val filter = async*[Int] {
  var next: Option[Int] = await(input)
  while (next.nonEmpty) {
    val evt = next.get
    if (p(evt)) yield(evt)
    next = await(input)
  }
}
```

Here, we create a simple filter observable which publishes an `Int` event for each event observed on the `input` observable that satisfies predicate `p`.

We provide a complete formalization in the context of an object-based core language reminiscent of Creol [7] and ABS [6]. Figure 1 shows a subset of expressions of the core language.

t ::=		terms
	| ...	(omitted)
	| `yield`(x)	yield event
e ::=		expressions
	| ...	(omitted)
	| `async*[`σ`]`($\bar{y}$) {e}	observable creation (reactive async)
	| `await`(x)	await event
	| t	term

Figure 1: RAY expressions and terms.

Operational semantics. The core concepts of our operational semantics are heaps, frames, and frame stacks (threads). Frames have the form $\langle L, e\rangle^l$ where L maps local variables to their values, e is an expression, and l is a label. A label is either s denoting a regular, synchronous frame, or $a(o, \bar{p})$ denoting an asynchronous frame; in this case, o is the heap address of a corresponding observable object, and $\bar{p}$ is a sequence of object identifiers of observables that observable o has itself subscribed to.

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 which generalizes an invariant given in [1].

2. *Subject reduction.* Reduction of well-typed programs preserves types.

The proofs of these properties are based on a typing relation, as well as invariants preserved by reduction. A forthcoming technical report [3] provides details of the formal model and proofs.

References

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

[2] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv*, 35(2):114–131, 2003.

[3] Philipp Haller and Heather Miller. A formal model for direct-style asynchronous observables. Technical report, forthcoming, KTH Royal Institute of Technology, Sweden, 2015.

[4] Philipp Haller and Jason Zaugg. SIP-22: Async. `http://docs.scala-lang.org/sips/pending/async.html`, 2013.

[5] Anders Hejlsberg, Mads Togersen, Scott Wiltamuth, and Peter Golde, editors. *The C# Programming Language.* Addison-Wesley, fourth edition, 2011.

[6] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957, pages 142–164. Springer, 2010.

[7] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.

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

[9] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *PADL*, pages 175–189. Springer, 2011.