

Simplifying Asynchronous Code with SCALA **ASYNC**

strangeloop
• SEPT 18-20 2013 •

PHILIPP HALLER

Joint work with JASON ZAUGG



THE PROBLEM

- ▶ Asynchronous code ubiquitous
 - ▶ Intrinsic to programming models used for reactive programming: futures, actors, ...
 - ▶ Required for performance and scalability
- ▶ Problem: usually enforces an unnatural code style
- ▶ Async enables direct-style code while using efficient non-blocking APIs under the hood

INSPIRATION

- ▶ Yes, we're avoiding NIH!
- ▶ Popular additions to C# and F#
- ▶ Our twist:
 - ▶ Don't change the Scala language
 - ▶ Async is “just” a macro

THIS TALK

- ▶ Overview
- ▶ Future
- ▶ Conclusion

GENTLE INTRO TO ASYNC

Async provides two constructs: `async` and `await`

```
async { <expr> }
```

- ▶ Declares block to be asynchronous
- ▶ By default, executes block asynchronously
- ▶ Returns future for the result of the block

USING ASYNC

```
async {  
    // some expensive computation without result  
}  
  
val future = async {  
    // some expensive computation with result  
}  
  
def findAll[T](what: T => Boolean) = async {  
    // find it all  
}
```

USING ASYNC

```
async {  
    // some expensive computation without result  
}  
  
val future = async {  
    // some expensive computation with result  
}  
  
def findAll[T](what: T => Boolean) = async {  
    // find it all  
}
```

“Asynchronous Method”

AWAIT

Within an `async { }` block, `await` provides a *non-blocking* way to await the completion of a future

```
await(<expr>)
```

- ▶ Only valid within an `async { }` block
- ▶ Argument `<expr>` returns a future
- ▶ Suspends execution of the enclosing `async { }` block until argument future is completed

USING AWAIT

```
val fut1 = future { 42 }
val fut2 = future { 84 }

async {
    println("computing...")
    val answer = await(fut1)
    println(s"found the answer: $answer")
}

val sum = async {
    await(fut1) + await(fut2)
}
```

IN SHORT

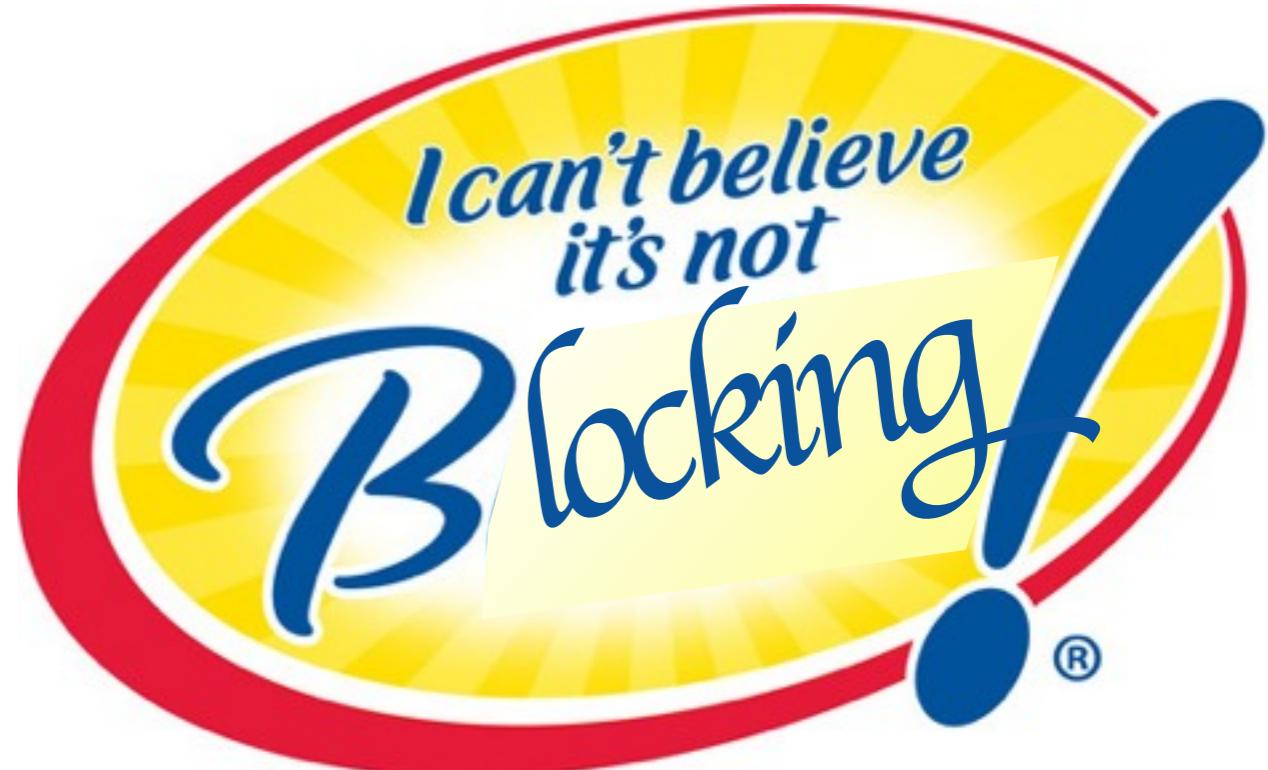
```
def async[T](body: => T): Future[T]
```

```
def await[T](future: Future[T]): T
```

IN SHORT

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

```
def await[T](future: Future[T]): T
```



Do I really need this?

PLAY FRAMEWORK EXAMPLE

```
val futureDOY: Future[Response] =  
  WS.url("http://api.day-of-year/today").get  
val futureDaysLeft: Future[Response] =  
  WS.url("http://api.days-left/today").get  
  
futureDOY.flatMap { doyResponse =>  
  val dayOfYear = doyResponse.body  
  futureDaysLeft.map { daysLeftResponse =>  
    val daysLeft = daysLeftResponse.body  
    Ok(s"$dayOfYear: $daysLeft days left!")  
  }  
}  
}
```

PLAY FRAMEWORK EXAMPLE

```
val futureDOY: Future[Response] =  
  WS.url("http://api.day-of-year/today").get  
val futureDaysLeft: Future[Response] =  
  WS.url("http://api.days-left/today").get  
  
async {  
  val dayOfYear = await(futureDOY).body  
  val daysLeft = await(futureDaysLeft).body  
  Ok(s"$dayOfYear: $daysLeft days left!")  
}
```

PLAY FRAMEWORK EXAMPLE

```
val futureDOY: Future[Response] =  
  WS.url("http://api.day-of-year/today").get  
val futureDaysLeft: Future[Response] =  
  WS.url("http://api.days-left/today").get  
  
for { doyResponse <- futureDOY  
      dayOfYear = doyResponse.body  
      daysLeftResponse <- futureDaysLeft  
      daysLeft = daysLeftResponse.body  
} yield Ok(s"$dayOfYear: $daysLeft days left!")
```

ANOTHER EXAMPLE

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

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

BACK TO USING FOR

```
def nameOfMonth(num: Int): Future[String] = ...  
val date = """(\d+)/(\d+)""".r  
  
for { doyResponse <- futureDOY  
      dayOfYear = doyResponse.body  
      response <- dayOfYear match {  
        case date(month, day) =>  
          for (name <- nameOfMonth(month.toInt))  
            yield Ok(s"It's $name!")  
        case _ =>  
          Future.successful(NotFound("Not a..."))  
      }  
} yield response
```

DIRECT STYLE

- ▶ Not forced to introduce names for intermediate results
- ▶ Control flow can be expressed naturally
 - ▶ Suspend within `if-else`, `while`, `match`, `try-catch`, ...

USING AWAIT

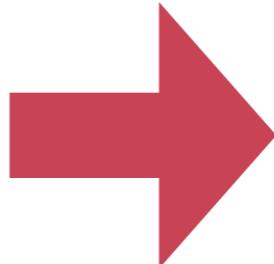
- ▶ Requires a directly-enclosing `async { }`
- ▶ Cannot use `await`
 - ▶ within closures
 - ▶ within local functions/classes/objects
 - ▶ within an argument to a by-name parameter

REMEDY

- ▶ Existing combinator in Futures API can help!

```
def f(x: A): Future[B]
```

```
async {  
    list.map(x =>  
        await(f(x)).toString  
    )  
}
```



```
Future.sequence(  
    list.map(x => async {  
        await(f(x)).toString  
    }))
```

THIS TALK

- ▶ Overview
- ▶ Future
- ▶ Conclusion



NEW!!

NEW FEATURE

- ▶ Sequential debugging
 - ▶ Runs async blocks sequentially
 - ▶ Calls to await are **blocking**: use only in debug mode!
 - ▶ Maintains the stack of the current thread throughout each async block

EXPERIMENTAL EXTENSION

- ▶ Extend `async` for *consuming and producing observables*

EXPERIMENTAL EXTENSION

- ▶ Extend `async` for *consuming and producing observables*

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

EXPERIMENTAL EXTENSION

- ▶ Extend `async` for *consuming and producing observables*

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

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

RAY: RASYNC, AWAIT*, YIELD*

- ▶ Extend and generalize async/await
- ▶ *Direct-style reactive streams*

```
val forwarder = rasync[Int] {  
    var next: Option[Int] =  
        awaitNextOrDone(stream)  
  
    while (next.nonEmpty) {  
        yieldNext(next.get)  
        next = awaitNextOrDone(stream)  
    }  
    None  
}
```

RAY: RASYNC, AWAIT*, YIELD*

- ▶ Extend and generalize async/await
- ▶ *Direct-style reactive streams*

returns an Observable[Int]

```
val forwarder = rasync[Int] {  
    var next: Option[Int] =  
        awaitNextOrDone(stream)  
  
    while (next.nonEmpty) {  
        yieldNext(next.get)  
        next = awaitNextOrDone(stream)  
    }  
  
    None  
}
```

RAY: RASYNC, AWAIT*, YIELD*

- ▶ Extend and generalize async/await
- ▶ *Direct-style reactive streams*

returns an Observable[Int]

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

awaits “next” or “done”
event of observable

None

RAY: RASYNC, AWAIT*, YIELD*

- ▶ Extend and generalize async/await
- ▶ *Direct-style reactive streams*

returns an Observable[Int]

```
val forwarder = rasync[Int] {  
    var next: Option[Int] =  
        awaitNextOrDone(stream)  
  
    while (next.nonEmpty) {  
        yieldNext(next.get)  
        awaitNextOrDone(stream)  
    }  
    None  
}
```

awaits “next” or “done”
event of observable

yields next event of current
observable

THIS TALK

- ▶ Overview
 - ▶ Future
 - ▶ Conclusion
- 

CONCLUSION

- ▶ `async` macro does a lot of hard work for you
- ▶ Generated code...
 - ▶ is non-blocking
 - ▶ spends a single class per `async` block
 - ▶ avoids boxing of intermediate results (which is more difficult with continuation closures)

WHAT IS IT FOR?

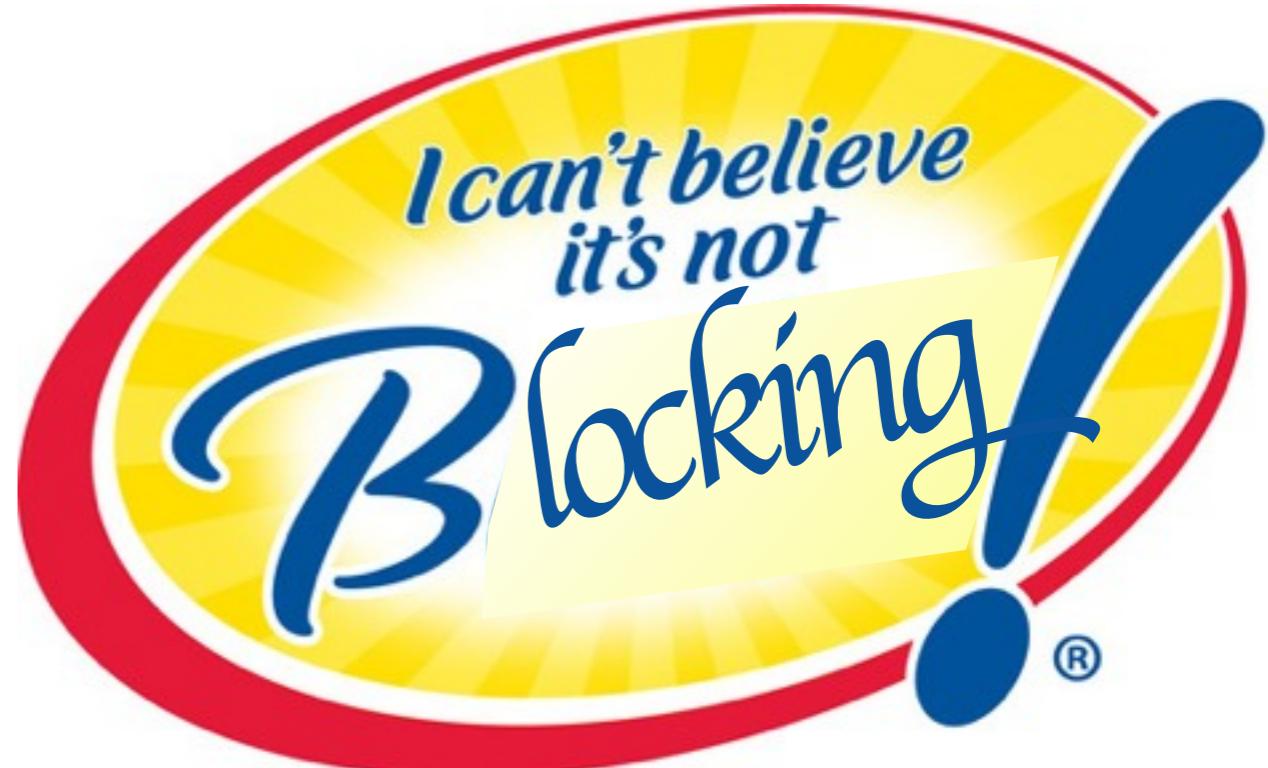
- ▶ Play Framework
 - ▶ Pervasive use of futures (SIP-14)
 - ▶ Async perfect fit, out-of-the-box support
- ▶ Akka actors/futures integration
- ▶ Non-blocking I/O
- ▶ Some uses of delimited continuations
- ▶ Connect to other asynchronous APIs



TAKEAWAY

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

```
def await[T](future: Future[T]): T
```



ROADMAP

- ▶ *Present*
 - ▶ Stable implementation for Scala 2.10 and Scala 2.11
 - ▶ <https://github.com/scala/async>
 - ▶ SIP: <http://docs.scala-lang.org/sips/pending/async.html>
- ▶ *Near future*
 - ▶ New module of Scala 2.11 distribution

ROADMAP



Try it today!

- ▶ *Present*
 - ▶ Stable implementation for Scala 2.10 and Scala 2.11
 - ▶ <https://github.com/scala/async>
 - ▶ SIP: <http://docs.scala-lang.org/sips/pending/async.html>
- ▶ *Near future*
 - ▶ New module of Scala 2.11 distribution

PLUG:

New Coursera course on:

Principles of Reactive Programming

Featuring:

Martin Odersky

Erik Meijer

Roland Kuhn

...and: Scala Async!

FREE!!

Q&A

???

CREDITS:



- ▶ Jason Zaugg, Typesafe
- ▶ Philipp Haller, Typesafe

@retronym

@philippkhaller

ASYNC VS. CPS PLUGIN

- ▶ Delimited continuations provided by CPS plugin can be used to implement `async/await`
- ▶ CPS plugin could support `await` within closures
- ▶ CPS-transformed code creates more closures (a closure is created at each suspension point)
- ▶ CPS plugin requires type annotations like `cpsParam[Int, String]`
- ▶ Error messages contain type annotations

INTERNAL OVERVIEW

- ▶ `async` is a *macro*
- ▶ `await` is a stub method
- ▶ Translation/macro expansion in two steps
 - ▶ Step 1: ANF transform (“introduce temporaries”)
 - ▶ Step 2: State machine transform

DEBUGGING

- ▶ Stepping, setting breakpoints supported
- ▶ Similar trade-off as in for-comprehensions
 - ▶ Artifacts of expanded program visible
- ▶ More IDE support planned (e.g., show expanded code)