Simplifying Asynchronous Code with SCALA ASYNC

PHILIPP HALLER

Typesafe

THE PROBLEM

- Asynchronous code
- Using an API that requires you to register callbacks?
  - Then you're writing asynchronous code!
- Problem: callbacks don't scale to large programs
- Scalable systems require non-blocking concurrency
- Our approach: take inspiration from F# and C# 5

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

PRELIMINARIES

- Even though the talk is about Scala, you won't need to know a lot about it
- For most of the non-async-related code you can assume it's like Java

WHAT IS SCALA?

- A statically-typed OO and functional language for the JVM
- Interoperates seamlessly with Java
    - Scala code can extend/invoke Java code without glue code and vice versa
- Who is using Scala?
    - Amazon, Autodesk, BBC, Foursquare, LinkedIn, Siemens, Twitter, ...

GENTLE INTRO TO ASYNC

Async provides two constructs: async and await

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

- Declares block to be asynchronous
- Executes block asynchronously
- Returns future for the result of the block

USING ASYNC

```scala
async {
  // some expensive computation without result
}

val future = async {
  // some expensive computation with result
}

def findAll[T](what: T => Boolean): Future[List[T]] =
  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): Future[List[T]] =
  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 expression returns a future
- Suspends execution of the current `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)
}
```

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"It is $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"It is $dayOfYear: $daysLeft days left!")
}
```

ILLEGAL USES OF AWAIT

- Cannot use `await` within closures
- Cannot use `await` within local functions, local classes, or local objects
- Cannot use `await` within an argument to a by-name parameter

```
def fut(x: Int) = future { x * 2 }
```

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

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

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

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ADVANCED USES

ITERATORS

```
trait Tree[+T]
case class Fork[T](left: Tree[T], el: T, right: Tree[T])
  extends Tree[T]
case object Empty extends Tree[Nothing]
```

ITERATORS

```scala
trait Tree[+T]
case class Fork[T](left: Tree[T], el: T, right: Tree[T])
  extends Tree[T]
case object Empty extends Tree[Nothing]
```

```scala
val t = Fork(Fork(Empty, "a", Empty), "b", Fork(Empty, "d", Empty))
val iter = new TreeIterator(t)

if (iter.hasNext) print(iter.next)
while (iter.hasNext) {
  val el = iter.next
  print(s", $el")
}
```

ITERATORS

```scala
trait Tree[+T]
case class Fork[T](left: Tree[T], el: T, right: Tree[T])
  extends Tree[T]
case object Empty extends Tree[Nothing]
```

```scala
val t = Fork(Fork(Empty, "a", Empty), "b", Fork(Empty, "d", Empty))
val iter = new TreeIterator(t)

if (iter.hasNext) print(iter.next)
while (iter.hasNext) {
  val el = iter.next
  print(s", $el")
}
```

a, b, d

A TREE ITERATOR

```
private[this] abstract class TreeIterator[A, B, R](tree: Tree[A, B]) extends Iterator[R] {
  protected[this] def nextResult(tree: Tree[A, B]): R

  override def hasNext: Boolean = next ne null
  override def next: R = next match {
    case null =>
      throw new NoSuchElementException("next on empty iterator")
    case tree =>
      next = findNext(tree.right)
      nextResult(tree)
  }

  @tailrec private[this] def findNext(tree: Tree[A, B]): Tree[A, B] = {
    if (tree eq null) popPath()
    else if (tree.left eq null) tree
    else {
      pushPath(tree)
      findNext(tree.left)
    }
  }

  private[this] def pushPath(tree: Tree[A, B]) {
    path(index) = tree
    index += 1
  }
  private[this] def popPath(): Tree[A, B] = if (index == 0) null else {
    index -= 1
    path(index)
  }

  private[this] var path = if (tree eq null) null else {
    val maximumHeight = 2 * (32 - Integer.numberOfLeadingZeros(tree.count + 2 - 1)) - 2 - 1
    new Array[Tree[A, B]](maximumHeight)
  }
  private[this] var index = 0
  private[this] var next: Tree[A, B] = findNext(tree)
}
```

A TREE ITERATOR

```
private[this] abstract class TreeIterator[A, B, R](tree: Tree[A, B]) extends Iterator[R] {
  protected[this] def nextResult(tree: Tree[A, B]): R

  override def hasNext: Boolean = next ne null
  override def next: R = next match {
    case null =>
      throw new NoSuchElementException("next
    case tree =>
      next = findNext(tree.right)
      nextResult(tree)
  }

  @tailrec private[this] def findNext(tree: Tree[A, B]): Tree[A, B] = {
    if (tree eq null) popPath()
    else if (tree.left eq null) tree
    else {
      pushPath(tree)
      findNext(tree.left)
    }
  }

  private[this] def pushPath(tree: Tree[A, B]) {
    path(index) = tree
    index += 1
  }
  private[this] def popPath(): Tree[A, B] = if (index == 0) null else {
    index -= 1
    path(index)
  }

  private[this] var path = if (tree eq null) null else {
    val maximumHeight = 2 * (32 - Integer.numberOfLeadingZeros(tree.count + 2 - 1)) - 2 - 1
    new Array[Tree[A, B]](maximumHeight)
  }
  private[this] var index = 0
  private[this] var next: Tree[A, B] = findNext(tree)
}
```

Iterator for
scala.collection.immutable.RedBlackTree

TREE ITERATOR W/ ASYNC

TREE ITERATOR W/ ASYNC

```
class TreeIterator[T](val container: Tree[T])
  extends AsyncIterator[T, Tree[T]] {

  def size(t: Tree[T]): Int = t match {
    case Fork(l, _, r) => size(l) + size(r) + 1
    case Empty => 0
  }

  def iterate(t: Tree[T]): Future[Unit] = async {
    t match {
      case Fork(left, el, right) =>
        await { iterate(left)  }
        await { yieldAsync(el) }
        await { iterate(right) }
      case Empty => /* do nothing */
    }
  }
}
```

Tree Iterator w/ Async

```
class TreeIterator[T](val container: Tree[T])
  extends AsyncIterator[T, Tree[T]] {

  def size(t: Tree[T]): Int = t match {
    case Fork(l, _, r) =>
    case Empty => 0
  }

  def iterate(t: Tree[T]): Future[Unit] = async {
    t match {
      case Fork(left, el, right) =>
        await { iterate(left)  }
        await { yieldAsync(el) }
        await { iterate(right) }
      case Empty => /* do nothing */
    }
  }
}
```

Asynchronous Method

Tree Iterator w/ Async

```
class TreeIterator[T](val container: Tree[T])
  extends AsyncIterator[T, Tree[T]] {

  def size(t: Tree[T]): Int = t match {
    case Fork(l, _, r) =>
    case Empty => 0
  }

  def iterate(t: Tree[T]): Future[Unit] = async {
    t match {
      case Fork(left, el, right) =>
        await { iterate(left)  }
        await { yieldAsync(el) }
        await { iterate(right) }
      case Empty => /* do nothing */
    }
  }
}
```

Asynchronous Method

ASYNCITERATOR

An implementation using futures/promises

`next()`

- creates a “next element” promise and a “continue” promise for suspending the iteration at `yieldAsync`
- waits for the completion of the “next element” promise after starting/resuming the iteration

`yieldAsync(elem)`

- completes the “next element” promise with `elem`
- returns the “continue” promise’s future

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ASYNC AND OTHER APIS

- By default, Async uses Scala's futures/promises
- It's possible to connect Async to other APIs using the reflection API of Scala 2.10

API INTERFACE

API INTERFACE

```
trait FutureSystem {
  type Prom[A]
  type Fut[A]

  trait Ops {
    val context: reflect.macros.Context
    import context.universe._

    def createProm[A: WeakTypeTag]: Expr[Prom[A]]

    def future[A: WeakTypeTag](a: Expr[A]): Expr[Fut[A]]

    def onComplete[A, U](fut: Expr[Fut[A]],
                         fun: Expr[Try[A] => U]): Expr[Unit]
  }

  def mkOps(c: Context): Ops { val context: c.type }
}
```

Simplified

API Interface Example

```
object ScalaConcurrentFutureSystem extends FutureSystem {
  import scala.concurrent._

  type Prom[A] = Promise[A]
  type Fut[A]  = Future[A]

  def mkOps(c: Context): Ops { val context: c.type } = new Ops {
    val context: c.type = c
    import context.universe._

    def createProm[A: WeakTypeTag]: Expr[Prom[A]] =
      reify { Promise[A]() }

    def future[A: WeakTypeTag](a: Expr[A]): Expr[Fut[A]] =
      reify { future(a.splice) }

    def onComplete[A, U](fut: Expr[Fut[A]], fun: Expr[Try[A] => U]) =
      reify {
        future.splice.onComplete(fun.splice)
      }
```

API INTERFACE SUMMARY

- Straight-forward to connect to other future-like APIs
- Not limited to future-like APIs!
  - Non-blocking I/O
  - ...

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

INTERNALS OVERVIEW

- Macro-based implementation
- Requires Scala 2.10
- Translation in two steps
    - Step 1: ANF transform
    - Step 2: State machine transform

TRANSLATION OF ASYNC

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

TRANSLATION OF ASYNC

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

ANF transform introduces "intermediate results":

```
val sum = async {
  val await$1: Int = await[Int](fut1)
  val await$2: Int = await[Int](fut2)
  await$1.+(await$2)
}
```

STATE MACHINE, PART 1

```
          fut1.onComplete(this)
        case 1 =>
          fut2.onComplete(this)
        case 2 =>
          result
$async.complete(Success(await
$1.$plus(await$2)))
      }
    } catch {
      case NonFatal((tr @ _)) =>
        result
$async.complete(Failure(tr))
    }

    def apply(tr: Try[Any]): Unit = //
see next slide
```

STATE MACHINE, PART 2

```
class stateMachine$3 extends StateMachine[...] {
  // see previous slide

    def apply(tr: Try[Any]): Unit = state
$async match {
      case 0 =>
        if (tr.isFailure)
          result
$async.complete(tr.asInstanceOf[Try[Int]])
        else {
          await$1 = tr.get.asInstanceOf[Int]
          state$async = 1
          resume$async()
        }
      case 1 =>
        if (tr.isFailure)
          result
$async.complete(tr.asInstanceOf[Try[Int]])
```

SUMMARY

- 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)

"macro authors are scalac hackers in my opinion"
-- Adriaan Moors, Scala Compiler Lead at Typesafe

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

ROADMAP

- Introduce a new feature for Scala designed to simplify working with asynchronous code
- How to use Async with SIP-14 (futures)
- Advanced uses
- How to use Async with other asynchronous APIs
- Async Internals
- Conclusion

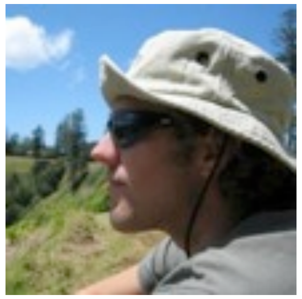
WHO IS THIS FOR?

- Play Framework
  - Pervasive use of futures (SIP-14)
  - async perfect fit, out-of-the-box support
- Akka actors/futures integration (SIP-14)
- Non-blocking I/O (“node.js without callback hell”)
- Some users of delimited continuations

OPEN-SOURCE GIT REPOSITORY

- https://github.com/scala/async
- Open-source under Scala license

CREDITS:



- Jason Zaugg, Typesafe

- Philipp Haller, Typesafe