Scala Macros for Mortals,
or: How I Learned To Stop Worrying and Mumbling “WTF?!?!?”

Brendan McAdams <brendan@boldradius.com>

@rit

1 The “WTF” of Macros - NEScala ’16
What Are Macros?
(There's some really good documentation)
Summarize it in one word
"metaprogramming"
But Seriously, What Are Macros?

- ‘metaprogramming’, from the Latin: ‘WTF?’.
- I mean, “code that writes code”.
- Write ‘extensions’ to Scala which are evaluated/expanded at compile time.
- Macros may generate new code or simply evaluate existing code.
Examples of Macros

Def Macros

- Def Macros are used to write, essentially, new methods.
- Facility for us to write powerful new syntax that feels ‘built-in’, such as Shapeless’ “This Shouldn’t Compile” `illTyped` macro...

```scala
scala> illTyped { """1+1 : Int"""" }
<console>:19: error: Type-checking succeeded unexpectedly.
Expected some error.
    illTyped { """1+1 : Int"""" }
    ^
```

6 The "WTF" of Macros - NEScala '16
Examples of Macros

Annotation Macros

- Annotations Macros let us write annotations which can be then rewritten or expanded at compile time:

```scala
@hello
object Test extends App {
  println(this.hello)
}
```

- ... And a lot more.
I'm Hoping To Make This Easy For You

- I'm pretty new to this Macro thing, and hoping to share knowledge from a beginner's standpoint.
- Without naming names, many Macros talks are given by Deeply Scary Sorcerers and Demigods who sometimes forget how hard this stuff is for newbies.
- Let's take a look at this through really fresh, profusely bleeding eyeballs.
Once Upon A Time...

- The only way to add compile time functionality to Scala was by writing compiler plugins.
- Esoteric, harder to ship (i.e. user must include a compiler plugin), not a lot of docs or examples.
- Required deep knowledge of the AST: Essentially generating new Scala by hand-coding ASTs.†
- I've done a little bit of compiler plugin work: the AST can be tough to deal with.§

† Abstract Syntax Tree. A simple “tree” of case-class like objects to be converted to bytecode... or JavaScript.
§ Some of the cool stuff in Macros like Quasiquotes can be used in Compiler Plugins now, too.
An AST Amuse Bouche

Given a small piece of Scala code, what might the AST look like?

class StringInterp {
    val int = 42
    val dbl = Math.PI
    val str = "My hovercraft is full of eels"

    println(s"String: $str Double: $dbl Int: $int Int Expr: ${int * 1.0}")
}

My God... It's Full of ... Uhm

Block(
  List(
    ClassDef(Modifiers(), TypeName("StringInterp"), List(), Template(
      List(Ident(TypeName("AnyRef"))), noSelfType, List(DefDef(Modifiers(), termNames.CONSTRUCTOR,
        List(),
        List(List()),
        TypeTree(), Block(List(Apply(Select(Super(This(typeNames.EMPTY), termNames.EMPTY), termNames.CONSTRUCTOR), List())), Literal(Constant(())))), ValDef(Modifiers(), TermName("int"),
        TypeTree(), Literal(Constant(42))), ValDef(Modifiers(), TermName("dbl"), TypeTree(),
        Literal(Constant(3.141592653589793))), ValDef(Modifiers(), TermName("str"), TypeTree(),
        Literal(Constant("My hovercraft is full of eels"))), Apply(Select(Ident(scala.Predef),
        TermName("println")), List(Apply(Select(Select(This(scala.StringContext), TermName("apply")),
        List(Literal(Constant("String: "))), Literal(Constant(" Double: "))), Literal(Constant(" Int: "))),
        Literal(Constant(" Int Expr: ")), Literal(Constant("Int"))),
    )))
))

11 The "WTF" of Macros - NEScala '16
Enter The Macro

- Since Scala 2.10, Macros have shipped as an experimental feature.
- Seem to have been adopted fairly quickly: I see them all over the place.
- AST Knowledge can be somewhat avoided, with some really cool tools to generate it for you.
- Macros make enhancing Scala much easier than writing compiler plugins.
- **NOTE:** You need to define your macros in a separate project / library from anywhere you call it.
HAIL SCIENCE!
**Macro Paradise**

- The Macro project for Scala is evolving *quickly*.
  - They release and add new features *far more frequently* than Scala does.
- “Macro Paradise” is a compiler plugin meant to bring Macro improvements into Scala⁷ as they become available.
  - One of the features currently existing purely in Macro Paradise is Macro Annotations.
- You can learn more about Macro Paradise at http://docs.scala-lang.org/overviews/macros/paradise.html

⁷ Focused on reliability with the current production release of Scala.
Macro Annotations

ADT Validation

- Macro Annotations let us build annotations that expand via Macros.
- I've written a Macro that verifies the "Root" type of an ADT is valid. The rules:
  - The root type must be either a trait or an abstract class.
  - The root type must be sealed.
- I've done this with AST manipulation to demo what that looks like.
Macro Annotations

ADT Validation

- You can find this code at https://github.com/bwmcadams/supreme-macro-adventure
  - I was feeling whimsical, and used part of a suggested random repo name from Github...

- Let's look at some chunks of ScalaTest “should compile” / “should not compile” code I use to validate my ADT Macro
Macro Annotations

ADT Validation

"A test of annotating stuff with the ADT Compiler Annotation" should "Reject an unsealed trait" in {
  ""
  | @ADT trait Foo
  """.stripMargin mustNot compile
}

it should "Reject a Singleton Object" in {
  ""
  | @ADT object Bar
  """.stripMargin mustNot compile
}
Macro Annotations

ADT Validation

it should "Approve a sealed trait" in {
  ""
  | @ADT sealed trait Spam {
  |   def x: Int
  | }
  """.stripMargin must compile
}

it should "Approve a sealed, abstract class" in {
  ""
  | @ADT sealed abstract class Eggs
  """.stripMargin must compile
}
Macro Annotations

ADT Validation

it should "Approve a sealed trait with type parameters" in {
  ""
  | @ADT sealed trait Klang[T] {
  |    def x: Int
  |  }
  """.stripMargin must compile

it should "Approve a sealed, abstract class with type parameters" in {
  ""
  | @ADT sealed abstract class Odersky[T]
  """.stripMargin must compile
}
ADT Validation

- First, we need to define an annotation:

  ```scala
  @compileTimeOnly("Enable Macro Paradise for Expansion of Annotations via Macros.")
  final class ADT extends StaticAnnotation {
    def macroTransform(annottees: Any*): Any = macro ADTMacros.annotation_impl
  }
  ```

- `@compileTimeOnly` makes sure we've enabled Macro Paradise: otherwise, our annotation fails to expand at compile time.

- `macroTransform` delegates to an actual Macro implementation which validates our ‘annottees’. 
ADT Validation

A quick note on the ‘annottees’ variable...

- This annotation macro is called *once per annotated class*. The fact that it has to take varargs can be confusing.

- If you annotate a class with a companion object, *both* are passed in.
  - If you annotate an object with a companion class, only the object is passed in.

- You must return *both* from your macro, or you get an error: *top-level class with companion can only expand into a block consisting in eponymous companions*
We *could* do this with the AST...

```scala
  import c.universe._
  import Flag._

  val p = c.enclosingPosition

  val inputs = annottees.map(_.tree).toList

  val result: Tree = {
    // Tree manipulation code
  }

  // if no errors, return the original syntax tree
  c.Expr[Any](result)
}
```
Matching Our Tree

inputs match {
  // both classes & traits
  case (cD @ ClassDef(mods, name, tparams, impl)) :: Nil ⇒
    validateClassDef(cD, mods, name, tparams, impl, companion = None)
  // annotated class with companion object.
  case (cD @ ClassDef(mods, name, tparams, impl)) :: (mD: ModuleDef) :: Nil ⇒
    validateClassDef(cD, mods, name, tparams, impl, companion = Some(mD))
  case (o @ ModuleDef(_, name, _)) :: Nil ⇒
    c.error(p, s"ADT Roots (object $name) may not be Objects.")
  // ... corner cases such as vals, vars, defs
}
case x :: Nil ⇒
  c.error(p, s"Invalid ADT Root ($x) [${x.getClass}].")
  x

case Nil ⇒
  c.error(p, "Cannot validate ADT Root of empty Tree."")
  // the errors should cause us to stop before this but needed to match up our match type
  reify {}.tree
Validating "Valid" Possibilities

```scala
def validateClassDef(cD: c.universe.ClassDef, mods: c.universe.Modifiers,
                     name: c.universe.TypeName, tparams: List[c.universe.TypeDef],
                     impl: c.universe.Template, companion: Option[ModuleDef]): c.universe.Tree = {

  if (mods.hasFlag(TRAIT)) {
    if (!mods.hasFlag(SEALED)) {
      c.error(p, s"ADT Root traits (trait $name) must be sealed.")
    }
  } else {
    c.info(p, s"ADT Root trait $name sanity checks OK.", force = true)
  }

  companion match {
    case Some(mD) ⇒ q"$cD; $mD"
    case None ⇒ cD
  }
```

26 The "WTF" of Macros - NEScala '16
Validating "Valid" Possibilities

```scala
} else if (!mods.hasFlag(ABSTRACT)) {
    c.error(p, s"ADT Root classes (class $name) must be abstract.")
    cD
} else if (!mods.hasFlag(SEALED)) {
    // class that's abstract
    c.error(p, s"ADT Root classes (abstract class $name) must be sealed.")
    cD
} else {
    c.info(p, s"ADT Root class $name sanity checks OK.", force = true)
    companion match {
        // Using ClassDef match, Scala requires tree includes all annottees (companions) sent in.
        case Some(mD) ⇒ q"$cD; $mD"
        case None ⇒ cD
    }
}
```
Macros & The AST

- Macros are still really built with the AST, but lately Macros provide tools to generate ASTs from code (which is what I use, mostly).

- The first, and simplest, is `reify`, which we can use to generate an AST for us.
Peeking at AST Examples for “Inspiration”

Remember my first example of the AST? I actually printed it out using `reify`:

```scala
println(showRaw(reify {
  class StringInterp {
    val int = 42
    val dbl = Math.PI
    val str = "My hovercraft is full of eels"

    println(s"String: $str Double: $dbl Int: $int Int Expr: ${int * 1.0}"
  }
}).tree))
```

`.tree` will replace the `reify` ‘expansion’ code with the AST associated. `showRaw` converts it to a printable format for us.
The "WTF" of Macros - NEScala '16
Quasiquotes for More Sanity

- There's really no way – yet – to avoid the AST Completely. But the Macro system continues to improve to give us ways to use it less and less.
- Quasiquotes, added in Scala 2.11, lets us write the equivalent of String Interpolation code that ‘evals’ to a Syntax Tree.
- We'll introduce Quasiquotes, and, time permitting, we're going to also look at a Quasiquotes version of the ADT Macro.
Quasiquotes in Action

Setting Up Our Imports

There are some implicits we need in scope for Quasiquotes Ah, the joy of imports...

```scala
import language.experimental.macros
import reflect.macros.Context
import scala.annotation.StaticAnnotation
import scala.reflect.runtime.{universe => ru}
import ru._
```

Now we're ready to generate some Syntax Trees!
Quasiquotes in Action

Writing Some Trees

Quasiquotes look like String Interpolation, but we place a `q` in front of our string instead of `s`... and generate code!

```
scala> q"def echo(str: String): String = str"
```

```
res4: reflect.runtime.universe.DefDef =
    def echo(str: String): String = str
```
Quasiquotes in Action

Writing Some Trees

```scala
val wtfException = q""

case class OMGWTFBBQ(message: String = null)
  extends Exception
  with scala.util.control.NoStackTrace
""

wtfException: reflect.runtime.universe.ClassDef =

  case class OMGWTFBBQ extends Exception with scala.util.control.NoStackTrace
    with scala.Product with scala.Serializable {
    <caseaccessor> <paramaccessor> val message: String = _;
    def <init> (message: String = null) = {
      super.<init>()
    }
  }
```

The "WTF" of Macros - NEScala ’16
Extracting with Quasiquotes

It turns out Quasiquotes can do extraction too, which I find sort of fun.

```
scala> val q""""case class $cname[..$tparams](..$params)
   extends $parent with ..$traits { ..$body }"""" = wtfException

cname: reflect.runtime.universe.TypeName = OMGWTFBBQ_
tparams: List[reflect.runtime.universe.TypeDef] = List()
params: List[reflect.runtime.universe.ValDef] =
   List(<caseaccessor> <paramaccessor> val message: String = null)
parent: reflect.runtime.universe.Tree = Exception
traits: List[reflect.runtime.universe.Tree] = List(scala.util.control.NoStackTrace)
body: List[reflect.runtime.universe.Tree] = List()
```
BRILLIANT!
ADT Macro with Quasiquotes

- With Quasiquotes, we can implement our ADT in a pure match with pattern guards.
- It is nearly half the # of lines.
val result: Tree = inputs match {
  case (t @ q"$mods trait $name[..$tparams] extends ..$parents { ..$body }") :: Nil
    if mods.hasFlag(SEALED) ⇒
      c.info(p, s"ADT Root trait $name sanity checks OK.", force = true)
    t
  case (t @ q"$mods trait $name[..$tparams] extends ..$parents { ..$body }") :: Nil ⇒
    c.error(p, s"ADT Root traits (trait $name) must be sealed.")
    t
Classes Validation

// there's no bitwise AND (just OR) on Flags

```scala
  case (cls @ q"$mods class $name[..$tparams] extends ..$parents { ..$body }") :: Nil
    if mods.hasFlag(ABSTRACT) && mods.hasFlag(SEALED) ⇒
      c.info(p, s"ADT Root class $name sanity checks OK.", force = true)
    cls
  case (cls @ q"$mods class $name[..$tparams] extends ..$parents { ..$body }") :: Nil ⇒
    c.error(p, s"ADT Root classes (class $name) must be abstract and sealed.")
    cls
```

Singletons & Trait Companions Validation

```scala
case (o @ q"$mods object $name") :: Nil ⇒
  c.error(p, s"ADT Roots (object $name) may not be Objects.")
o
// companions
case (t @ q"$mods trait $name[..$tparams] extends ..$parents { ..$body }") ::
  (mD: ModuleDef):: Nil
  if mods.hasFlag(SEALED) ⇒
    c.info(p, s"ADT Root trait $name sanity checks OK.", force = true)
q"$t; $mD"
```

```scala
case (t @ q"$mods trait $name[..$tparams] extends ..$parents { ..$body }") ::
  (mD: ModuleDef) :: Nil ⇒
  c.error(p, s"ADT Root traits (trait $name) must be sealed.")
q"$t; $mD"
```
// there's no bitwise AND (just OR) on Flags

case (cls @ q"$mods class $name[..$tparams] extends ..$parents { ..$body }") ::
  (mD: ModuleDef) :: Nil ⇒
  c.info(p, s"ADT Root class $name sanity checks OK.", force = true)
  q"$cls; $mD"

case (cls @ q"$mods class $name[..$tparams] extends ..$parents { ..$body }") ::
  (mD: ModuleDef) :: Nil ⇒
  c.error(p, s"ADT Root classes (class $name) must be abstract and sealed.")
  q"$cls; $mD"
AWESOME.

AWESOME TO THE MAX.
Closing Thoughts

- Macros are undoubtedly cool, and rapidly evolving. But be cautious.
- “When all you have is a hammer, everything starts to look like a thumb…”
  — me
- Macros can enable great development, but also hinder it if overused. Think carefully about their introduction, and their impact on your codebase.
Questions?