The State of the Meta

Eugene Burmako (@xeno_by)

École Polytechnique Fédérale de Lausanne
http://scalameta.org/

22 November 2014
Outline

- Why metaprogramming matters
- The present (scala.reflect)
- The future (scala.meta)
Why metaprogramming matters
Metaprogramming is the writing of computer programs that write or manipulate other programs or themselves as their data.

—Wikipedia
Use case #1: Code generation

Sometimes you have to reach for a sledgehammer:

- High performance
- Integration with external systems
- Abstraction over heterogeneous datatypes
Use case #2: Advanced static checks

Types are useful, but every now and then they fall short:

- Restrict the types of variables that are captured in closures
- Ensure that communication between actors is sound
- Prevent overflows or underflows in computation-heavy code
Use case #3: Advanced domain-specific languages

Even flexible syntax and powerful types have their limits:

- LINQ
- async/await
- External DSLs
Use case #4: Tooling

- Style checking
- Refactoring
- Incremental compilation
- And much-much more
The present (scala.reflect)
The present (scala.reflect)

- Decent support for compile-time metaprogramming (macros)
- Some support for runtime metaprogramming (reflection, codegen)
- Hardcore compiler internals or reinvented wheels for everything else
Macros by example

```scala
def createArray[T: ClassTag](size: Int, el: T) = {
  val a = new Array[T](size)
  for (i <- 0 until size) a(i) = el
  a
}
```

- We want to write beautiful generic code, and Scala makes that easy
- Unfortunately, abstractions oftentimes bring overhead
Macros by example

Object createArray(int size, Object el, ClassTag tag) {
    Object a = tag.newArray(size);
    for (int i=0; i<size; i++) ScalaRunTime.array_update(...);
    return a;
}

- Scala runs on the JVM, so polymorphic signatures have to be erased
- Erasure leads to boxing and boxing creates significant slowdowns
- The snippet above shows a sketch of what createArray compiles to
# Macros by example

```scala
def createArray[@specialized T: ClassTag](...) = {
  val a = new Array[T](size)
  for (i <- 0 until size) a(i) = el
  a
}
```

- Methods can be specialized, but it’s viral and heavyweight
- **Viral** = the entire call chain needs to be specialized
- **Heavyweight** = specialization leads to duplication of bytecode
Macros by example

```scala
def createArray[T: ClassTag](size: Int, el: T) = {
  val a = new Array[T](size)
  def specBody[@specialized T](a: Array[T], el: T) = {
    for (i <- 0 until size) a(i) = el
  }
  classTag[T] match {
    case ClassTag.Int => specBody(
      a.asInstanceOf[Array[Int]], el.asInstanceOf[Int])
    ...
  }
  a
}
```

- We want to specialize just as much as we need
- But that's tiresome to do by hand, and this is where macros shine
Macros by example

```scala
def specialized[T](x: => Any) = macro impl[T]

def createArray[T: ClassTag](size: Int, el: T) = {
  val a = new Array[T](size)
  specialized[T] {
    for (i <- 0 until size) a(i) = el
  }
  a
}
```

- specialized macro gets pretty code and transforms it into fast code
- The transformation is carried out by an associated metaprogram `impl`
- For more details, see *Bridging Islands of Specialized Code*
How macros work

- We publish the language model as part of the standard distribution
- `scala-reflect.jar` contains definitions of trees, symbols, types, etc
- Macros are written against data structures from `scala.reflect`
- And executed inside `scalac` which implements the `scala.reflect` API
How macros work

```scala
def specialized[T](x: => Any) = macro impl[T]

def impl[T](c: scala.reflect.macros.Context)(x: c.Tree)(implicit T: c.WeakTypeTag[T]) = {
import c.universe._
val (specVars, specBody) = Helpers.specialize(x, T)
q"""
  def specBody[@specialized T]($specVars) = $specBody
  classTag[$T] match { case ..$cases }
"""
}
```

- The entry point to compile-time metaprogramming is a context
- From there we can import a universe of trees, symbols, types, etc
How macros work

```scala
def specialized[T](x: => Any) = macro impl[T]

def impl[T](c: scala.reflect.macros.Context)(x: c.Tree)(implicit T: c.WeakTypeTag[T]) = {
  import c.universe._
  val (specVars, specBody) = Helpers.specialize(x, T)
  q""
    def specBody[@specialized T](..$specVars) = $specBody
    classTag[$T] match { case ..$cases }
  ""
}
```

- Reflection artifacts that model Scala code are first-class
- We can pass them around to/from helpers, can create new ones
How macros work

```scala
def specialized[T](x: => Any) = macro impl[T]

def impl[T](c: scala.reflect.macros.Context)(x: c.Tree)(implicit T: c.WeakTypeTag[T]) = {
  import c.universe._
  val (specVars, specBody) = Helpers.specialize(x, T)
  q"""
    def specBody[@specialized T](..$specVars) = $specBody
    classTag[$T] match { case ..$cases }
  """
}
```

- Upon completion, a macro returns a tree representing new code
- This tree will replace the macro invocation at the call site
There’s more

- Type providers
- Automatic generation of typeclass instances
- Language virtualization
- Deep integration with external DSLs
- For details see our “What Are Macros Good For?” talk
The future (scala.meta)
The future (scala.meta)

- This year we set out on a journey to create an ultimate metaprogramming API
- Here we’ll tell you about our design decisions and their impact
scala.meta principles

- Language model should be independent of language implementations
- Interface for syntax trees should be based on hygienic quasiquotes
- Binaries should provide access to their attributed abstract syntax trees
Principle #1: Independent language model

- `scala.reflect` grew organically from scalac internals
- That was a nice shortcut to get things going easily
- But in the long run dependence on scalac makes everything harder
Principle #1: Independent language model

scalameta
Simple, robust and portable metaprogramming foundation for Scala
http://scalameta.org

scalameta
Simple, robust and portable metaprogramming toolkit for Scala
Updated a day ago

scalahost
Scala host for scala.meta
Updated 10 days ago
Principle #1: Independent language model

Diversity is real:

- ...Intellij (a reimplemention of Scala’s typechecker)
- ...Dotty (the next generation Scala compiler)
- ...Typelevel (an experimental community fork)
- ...Policy (a compiler/language technology fork)
Principle #2: Hygienic quasiquotes

ClassDef(
    NoMods,
    newTypeName("D"),
    Nil,
    Template(
        List(Ident(newTypeName("C"))),
        emptyValDef,
        List(DefDef(NoMods, nme.CONSTRUCTOR, ...)))

- Once, the most reliable way to do ASTs was via vanilla datatypes
- Above you can see what it took to say class D extends C
- Needless to say, only brave souls cared to use such an API
Principle #2: Hygienic quasiquotes

```scala
val tree = q"class D extends C"
val q"class $_ extends ..$parents" = tree
```

- Then we introduced quasiquotes, a template-based approach to ASTs
- Quasiquotes became the highlight of the Scala 2.11 release
- At this point our metaprogramming API started to look friendly
Principle #2: Hygienic quasiquotes

import foo.bar.C
val tree = q"class D extends C"
q"class C; $tree"

- The final puzzle to solve is prevention of name clashes
- This is what is called hygiene of a metaprogramming API
- Probably the most common pitfall in Scala macros at the moment
Principle #3: Persistent syntax trees

- Every compilation produces attributed syntax trees
- These trees contain a wealth of semantic information about programs
- These trees get thrown away every time
Principle #3: Persistent syntax trees

What if we saved attributed trees instead of throwing them away?

- Easy interpretation
- Easy code analysis
- And other neat metaprogramming tricks
Principle #3: Persistent syntax trees

scala-internals ›

Attacking binary compatibility at the root with typed trees

37 posts by 17 authors ◊  g+1

martin

Oct 9

Scala has suffered from binary incompatibility problems ever since it has become moderately popular. Sure, we have made progress. There's now a tool (MiMa) to check for compatibility violations and a policy to maintain strict binary compatibility for minor versions of Scala. At the same time, the burden of maintaining binary compatibility makes bug fixes and library and compiler evolution much more cumbersome than before. It can now take 2 years or more to get a bugfix in which would break binary compatibility. And an increasing proportion of our effort going into a fix or improvement is spent on questions of maintaining binary compatibility. So, we are between a rock and a hard place: not binary compatible enough for the public at large yet already greatly slowed down by our own efforts to maintain the level of bc that we have achieved so far.
Principle #3: Persistent syntax trees

It turns out that tree persistence is so much more:

- Separate the frontend and the backend of the compiler
- Typed syntax trees become the new bytecode
- Linker produces binaries native to target platforms
- Solves bincompat problems and enables amazing optimizations!
Our status

- We have an initial design of the metaprogramming API
- And are currently validating it on a number of use cases
- Something publicly usable should appear next year
- Most likely as a compiler plugin for Scala 2.11
Summary
Summary

- Metaprogramming is useful, and Scala programmers use it
- You don’t just grow a metaprogramming API from compiler internals
- High tech is paramount for usability (quasiquotes, hygiene, etc)
- Tree persistence can enable unexpected technological advances