

# Easy Metaprogramming For Everyone!

Eugene Burmako (@xeno\_by)  
Denys Shabalin (@den\_sh)

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

17 June 2014

# Metaprogramming is...

*Metaprogramming is the writing of computer programs that write or manipulate other programs or themselves as their data.*

—Wikipedia

# Metaprogramming is useful

- ▶ Code generation
- ▶ Program verification
- ▶ Style checking
- ▶ Refactoring
- ▶ Incremental compilation
- ▶ Documentation generation
- ▶ ...

## Before Scala 2.10



## Baby steps of Scala metaprogramming

- ▶ Text-based introspection with scalap
- ▶ Unstable and undocumented compiler plugins
- ▶ Ad-hoc textual code generation

## After Scala 2.10



## Current state of things

Metaprogramming with `scala.reflect`:

- + Full-fledged model of Scala available in standard distribution
- + Structured code generation with macros and quasiquotes

# Current state of things

Metaprogramming with `scala.reflect`:

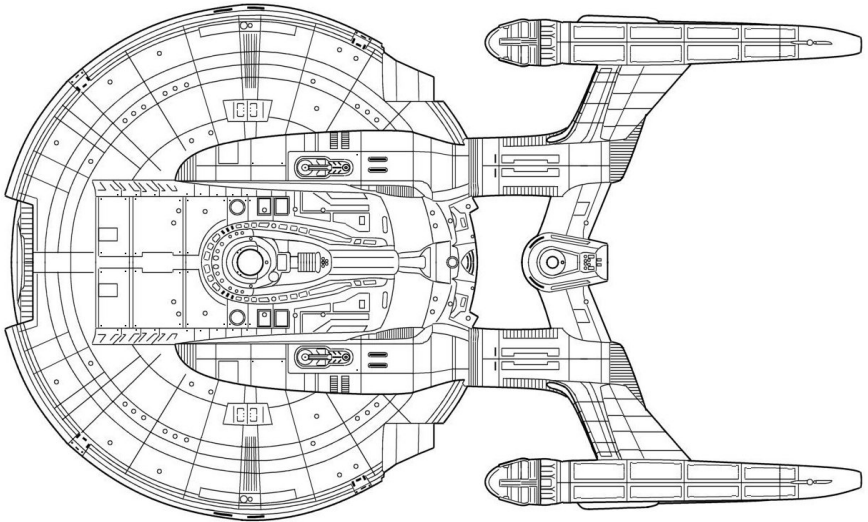
- + Full-fledged model of Scala available in standard distribution
- + Structured code generation with macros and quasiquotes
- Complicated: optimized towards compiler developers, not library users
- Brittle: a lot of unspecified and hard-to-satisfy invariants
- Locked-in: tightly bound to `scalac` internals



## Nevertheless `scala.reflect` has proven to be useful

- ▶ Enables libraries like `async`, `pickling`, `scala-blitz`, etc
- ▶ Empowers existing solutions in `scalatest`, `Play!`, `parboiled`, etc
- ▶ Foundation for high-level abstractions: `shapeless`, `yin-yang`, etc

We are building an even better tech



# Meet our new metaprogramming platform

- ▶ Name: `scala.meta` (formerly known as Project Palladium)
- ▶ Goal: Build a tool to conveniently work with programs as data
- ▶ Status: Alpha version, public preview coming this fall

## Language model

# Scala language model à la `scala.reflect`

# Scala language model à la `scala.reflect`

- ▶ Trees
  - ▶ TermTrees
  - ▶ TypTrees
  - ▶ DefTrees
  - ▶ ...

# Scala language model à la `scala.reflect`

- ▶ Trees
  - ▶ TermTrees
  - ▶ TypTrees
  - ▶ DefTrees
  - ▶ ...
- ▶ Types
  - ▶ Symbols

# Scala language model à la `scala.reflect`

- ▶ Trees
  - ▶ TermTrees
  - ▶ TypTrees
  - ▶ DefTrees
  - ▶ ...
- ▶ Types
  - ▶ Symbols
  - ▶ Scopes
  - ▶ Names
  - ▶ Annotations
  - ▶ Constants
  - ▶ Modifiers
  - ▶ ...



## A lot of concepts that interact in inconsistent ways

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)  
  
scala> toolbox.typecheck(list).tpe  
res1: toolbox.u.Type = List[Int]
```

## A lot of concepts that interact in inconsistent ways

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> toolbox.typecheck(list).tpe  
res1: toolbox.u.Type = List[Int]
```

```
scala> tq"List[Int]"  
res2: universe.Tree = List[Int]
```

## A lot of concepts that interact in inconsistent ways

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> toolbox.typecheck(list).tpe  
res1: toolbox.u.Type = List[Int]
```

```
scala> tq"List[Int]"  
res2: universe.Tree = List[Int]
```

```
scala> res1 == res2  
res3: Boolean = false
```

## A lot of concepts that interact in inconsistent ways

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> toolbox.typecheck(list).tpe  
res1: toolbox.u.Type = List[Int]
```

```
scala> tq"List[Int]"  
res2: universe.Tree = List[Int]
```

```
scala> res1 == res2  
res3: Boolean = false
```

```
scala> toolbox.typecheck(res2, toolbox.TYPEmode).tpe  
res4: toolbox.u.Type = List[Int]
```

```
scala> res1 == res4  
res5: Boolean = true
```

# Scala language model à la `scala.meta`

Trees

## Scala language model à la `scala.meta`

Trees.

# Scala language model à la `scala.meta`

## Trees.

- ▶ In `scala.meta`, we model everything just with its abstract syntax
- ▶ Types, members, names, modifiers: all represented with trees
- ▶ There's only one data structure, so there's only one way to do it

## Terms are trees

```
scala> q"List(1, 2, 3)"  
res0: meta.Term = List(1, 2, 3)
```

```
scala> q"List(1, 2, 3)".tpe  
res1: meta.Type = List[Int]
```



## Types are trees

```
scala> t"List[Int]"  
res2: meta.Type = List[Int]
```

```
scala> t"List[Int]" == q"List(1, 2, 3)".tpe  
res3: Boolean = true
```

```
scala> t"List[Int]" <:< t"List[_]"  
res4: Boolean = true
```

```
scala> t"List[Int]".subtypes  
res5: Seq[meta.Type] = List(::[Int], Nil.type)
```

## Symbols are trees

```
scala> val head1 = t"List[Int]".defs("head")
head1: meta.Def = def head: Int
```

```
scala> val head2 = q"List(1, 2, 3).head".defn
head2: meta.Def = def head: Int
```

```
scala> head1 == head2
res8: Boolean = true
```

```
scala> head1.owner
res9: meta.Scope = class List { ... }
```

```
scala> head1.show[Raw]
res10: String = Decl.Def( Nil, Term.Name(head), Nil, Nil, Int)
```

Something seems fishy



## You might be thinking

- ▶ If there's just trees and nothing more
- ▶ How do we know that `List` in `List(1, 2, 3)` is `scala.List`?

## In scala.reflect

### Trees only

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> tb.eval(q"val List = 42; $list")  
compilation has failed: Int does not take parameters
```

## In scala.reflect

### Trees only

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> tb.eval(q"val List = 42; $list")  
compilation has failed: Int does not take parameters
```

### Trees and symbols

```
scala> val List = mirror.staticModule("s.c.i.List")  
res0: universe.ModuleSymbol = object List
```

```
scala> val list = q"$List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> tb.eval(q"val List = 42; $list")  
res2: Any = List(1, 2, 3)
```

# The missing technology

## **TECHNICAL REPORT NO. 194**

### **Hygienic Macro Expansion**

**by**

**E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba**

**Indiana University**

**May, 1986**

## Hygiene in action

### In `scala.reflect`

```
scala> val list = q"List(1, 2, 3)"  
list: universe.Tree = List(1, 2, 3)
```

```
scala> tb.eval(q"val List = 42; $list")  
compilation has failed: Int does not take parameters
```

### In `scala.meta`

```
scala> val list = q"List(1, 2, 3)"  
list: meta.Tree = List(1, 2, 3)
```

```
scala> q"val List = 42; $list".eval  
res1: Any = List(1, 2, 3)
```



## Tree design

## Trees are now comprehensive

In `scala.reflect`

```
scala> q"for (i <- List(1, 2, 3)) println(i)"  
res0: universe.Tree = List(1, 2, 3).foreach(i => println(i))
```

## Trees are now comprehensive

In `scala.reflect`

```
scala> q"for (i <- List(1, 2, 3)) println(i)"  
res0: universe.Tree = List(1, 2, 3).foreach(i => println(i))
```

In `scala.meta`

```
scala> q"for (i <- List(1, 2, 3)) println(i)"  
res1: meta.Term = for (i <- List(1, 2, 3)) println(i)
```

## Trees are now comprehensive

- ▶ In `scala.meta`, we keep all the information about the program
- ▶ Nothing is desugared (e.g. `for` loops or string interpolations)
- ▶ Nothing is thrown away (e.g. comments or formatting details)

## Trees are now strongly-typed

In `scala.reflect`

```
case class Apply(fun: Tree, args: List[Tree])
```

In `scala.meta`

```
@ast class Apply(fun: Term, args: Seq[Arg])
```

## In scala.reflect

```
scala> val List = tq"_root_.scala.List"  
List: universe.Select = _root_.scala.List
```

```
scala> val list = q"$List(1, 2, 3)"  
list: universe.Tree = _root_.scala.List(1, 2, 3)
```

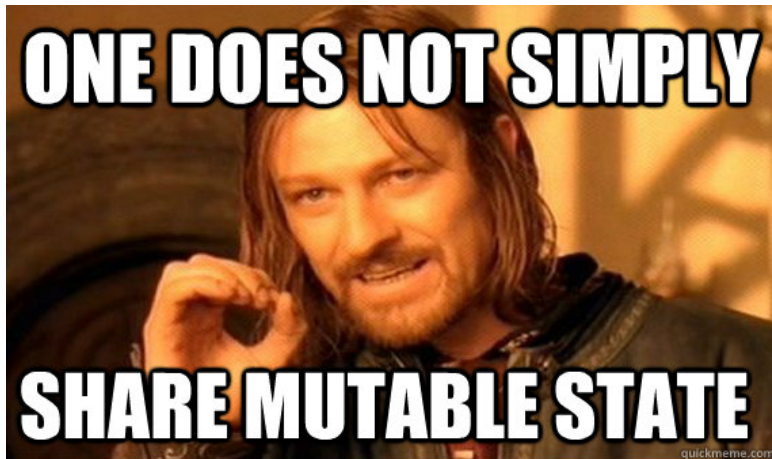
```
scala> toolbox.typecheck(list)  
s.t.r.ToolBoxError: type scala.List is not a value  
  at s.t.r.ToolBoxFactory$.apply(ToolBoxFactory.scala:178)  
  at s.t.r.ToolBoxFactory$.apply(ToolBoxFactory.scala:170)  
  at s.t.r.ToolBoxFactory$.apply(ToolBoxFactory.scala:148)  
  ...
```

## In scala.meta

```
scala> val List = t"_root_.scala.List"
List: meta.Type = _root_.scala.List

scala> val list = q"$List(1, 2, 3)"
<console>:27: error: type mismatch;
 found   : Type
 required: Term
           val list = q"$List(1, 2, 3)"
                        ^
```

Boromir has certain doubts about `scala.reflect`





## Trees are now immutable

In `scala.reflect`

```
abstract class Tree {  
  private[this] var rawtpe: Type = _  
  def tpe: Type = rawtpe  
  def setType(tp: Type): this.type = { rawtpe = tp; this }  
  ...  
}
```

## Trees are now immutable

In `scala.reflect`

```
abstract class Tree {  
  private[this] var rawtpe: Type = _  
  def tpe: Type = rawtpe  
  def setType(tp: Type): this.type = { rawtpe = tp; this }  
  ...  
}
```

In `scala.meta`

```
implicit class SemanticTermOps(val term: Term) extends AnyVal  
  @hosted def tpe: Type = ...  
}
```

## Anytime metaprogramming

# Flavors of metaprogramming

- ▶ Compile-time metaprogramming (macros)
- ▶ Runtime metaprogramming (runtime reflection, toolboxes)
- ▶ Some-time metaprogramming (IDEs, incremental compilation, linters)

## Flavors of environments

- ▶ Scala compiler (`scala-reflect.jar` and `scala-compiler.jar`)
- ▶ IntelliJ IDEA (a very own implementation of Scala's typechecker)
- ▶ Scala IDE (Scala compiler running in a funny mode)
- ▶ SBT (A tiny compiler plugin + a very own analysis infrastructure)
- ▶ DIY (Just have some Scala sources and need to find something out)

## Too complicated again!

- ▶ Typechecking and evaluation kind of work at runtime, but not quite
- ▶ Macros work at compile time, but only in a separate project
- ▶ Macros seem to work in IntelliJ, but they actually don't

With `scala.meta` it's no longer a problem

Demo!

But how is this supposed to work?!





## But how is this supposed to work?!

- ▶ Macros are known to be just bytecode invoked by reflection
- ▶ But when macros aren't compiled yet, there's no bytecode
- ▶ Therefore having macros work in the same file would be most unusual
- ▶ This merits an explanation

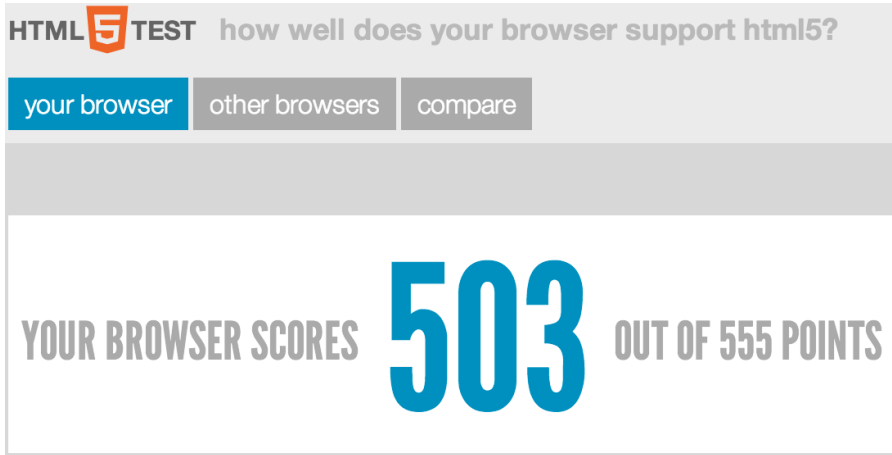
## Explanation, in a nutshell

- ▶ With `scala.meta`, we have principled tools for metaprogramming
- ▶ This makes it possible to make macros principled
- ▶ When macros become a proper abstraction, all becomes much better

## Explanation, part 1: Clear requirements for all hosts

- ▶ A standalone, independent metaprogramming interface
- ▶ Built on simple, strongly-typed and immutable foundation
- ▶ With clear and concise host API
- ▶ Prototype implementations for `scalac` and IntelliJ

## Explanation, part 1: Clear requirements for all hosts



## Explanation, part 2: Equal opportunities for all hosts

- ▶ With our plugin enabled, `scalac` saves all ASTs after typer
- ▶ This means that full info about any program is available at any time
- ▶ No information is lost anymore (e.g. signatures of local definitions)
- ▶ The overhead is actually quite reasonable (e.g. compressed *untyped* ASTs of `scala-library.jar` are less than 15% of the bytecodes)

## Explanation, part 3: Brave new world!

- ▶ Easy access to all ASTs enables a lot of interesting things
- ▶ One of them is host-independent AST interpretation
- ▶ That's exactly what we're using to lift the precompilation restriction

Anytime metaprogramming



## Automatic code rewriting tool

- ▶ As of late, there have been talks about deprecating procedure syntax
- ▶ But it's so common that warnings are only emitted under `-Xfuture`
- ▶ It would be nice to have an automatic migration tool for this!



## Automatic code rewriting tool

- ▶ Unfortunately existing `scalac` functionality isn't quite fit for that
- ▶ Wheels are reinvented in order to do robust parsing and prettyprinting
- ▶ With `scala.meta`, this become very simple!

## Automatic code rewriting tool

```
some/project$ sbt meta  
[info] Loading project definition...  
[info] Starting scala interpreter...
```

## Automatic code rewriting tool

```
some/project$ sbt meta
[info] Loading project definition...
[info] Starting scala interpreter...

> project
res0: Project = Project(List("scalameta/package.scala",
"scalameta/Trees.scala", "scalameta/semantic/Hosts.scala"..))
```

## Automatic code rewriting tool

```
some/project$ sbt meta
[info] Loading project definition...
[info] Starting scala interpreter...

> project
res0: Project = Project(List("scalameta/package.scala",
"scalameta/Trees.scala", "scalameta/semantic/Hosts.scala"..))

> project.rewrite {
|   case q"..$mods def $nme[..$tps](...$pss) { ..$body }" =>
|   q"..$mods def $nme[..$tps](...$pss): Unit = { ..$body }"
| }.persist
res1: Project = Project(List("scalameta/package.scala",
"scalameta/Trees.scala", "scalameta/semantic/Hosts.scala"..))
```

Wrapping up

## Brought to you by the Palladium team

This project is brought to you by the Palladium team.

Thank you very much, folks, for making this presentation possible!

- ▶ Uladzimir Abramchuk
- ▶ Igor Bogomolov
- ▶ Eugene Burmako
- ▶ Mathieu Demarne
- ▶ Martin Duhem
- ▶ Adrien Ghosn
- ▶ Mikhail Mutcianko
- ▶ Dmitry Naydanov
- ▶ Artem Nikiforov
- ▶ Vladimir Nikolaev
- ▶ Alexander Podkhalyuzin
- ▶ Jatin Puri
- ▶ Denys Shabalin

## What we've seen today

- ▶ Built on a simple principle that everything is a tree
- ▶ And designed in strongly-typed and fully immutable style
- ▶ `scala.meta` is a clean and portable metaprogramming toolkit
- ▶ Public preview is coming this fall
- ▶ But some results are available right now (e.g. sbt improvements)

Public preview coming this fall at [scalameta.org](http://scalameta.org)!

