State of the Meta, Summer 2015

Eugene Burmako (@xeno_by)

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

Presented on 09 June 2015
Last updated on 21 Sep 2015
Video available in the ScalaDays collection

scala.meta

Simple, robust and portable metaprogramming foundation for Scala

— github.com/scalameta

Main goal

- Support all kinds of frontend metaprogramming tasks
- Especially novel tooling
- But also def macros and macro annotations
- More on that today in the live demo!

Presentation outline

- Syntactic API
- Semantic API
- Live demo
- Roadmap

Credits

Big thanks to everyone who helped turning scala.meta into reality!

- Uladzimir Abramchuk
- Eric Beguet
- Igor Bogomolov
- Eugene Burmako
- Mathieu Demarne
- Martin Duhem
- Adrien Ghosn
- Vojin Jovanovic
- Guillaume Massé
- Mikhail Mutcianko
- Dmitry Naydanov
- Artem Nikiforov
- Vladimir Nikolaev
- Martin Odersky
- Oleksandr Olgashko
- Alexander Podkhalyuzin
- Jatin Puri
- Dmitry Petrashko
- Denys Shabalin

Part 1: Syntactic API

Getting started

```
$ scala

scala> import scala.meta._
import scala.meta._

scala> import scala.meta.dialects.Scala211
import scala.meta.dialects.Scala211
```

Design goals

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

Implementation vehicle

First-class tokens

Tokens

```
scala> "class C { def x = 2 }".tokens
...
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5), (5..6), C (6..7), (7..8), { (8..9), (9..10),
def (10..13), (13..14), x (14..15), (15..16), = (16..17),
(17..18), 2 (18..19), (19..20), } (20..21), EOF (21..21))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5), (5..6), C (6..7), (7..8), { (8..9), (9..10),
def (10..13), (13..14), x (14..15), (15..16), = (16..17),
(17..18), 2 (18..19), (19..20), } (20..21), EOF (21..21))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5), (5..6), C (6..7), (7..8), { (8..9), (9..10),
def (10..13), (13..14), x (14..15), (15..16), = (16..17),
(17..18), 2 (18..19), (19..20), } (20..21), EOF (21..21))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5), (5..6), C (6..7), (7..8), { (8..9), (9..10),
def (10..13), (13..14), x (14..15), (15..16), = (16..17),
(17..18), 2 (18..19), (19..20), } (20..21), EOF (21..21))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5), (5..6), C (6..7), (7..8), { (8..9), (9..10),
def (10..13), (13..14), x (14..15), (15..16), = (16..17),
(17..18), 2 (18..19), (19..20), } (20..21), EOF (21..21))
```

High-fidelity parsers

```
scala> "class C".parse[Stat]
res2: scala.meta.Stat = class C

scala> "class C {}".parse[Stat]
res3: scala.meta.Stat = class C {}
```

High-fidelity parsers

```
scala> "class C".parse[Stat]
res2: scala.meta.Stat = class C

scala> "class C {}".parse[Stat]
res3: scala.meta.Stat = class C {}

scala> res2.tokens
res4: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5),  (5..6), C (6..7), EOF(7..7))

scala> res3.tokens
res5: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
class (0..5),  (5..6), C (6..7),
 (7..8), { (8..9), } (9..10), EOF (10..10))
```

Automatic and precise range positions

```
scala> "class C { def x = 2 }".parse[Stat]
res6: scala.meta.Stat = class C { def x = 2 }

scala> val q"class C { $method }" = res6
method: scala.meta.Stat = def x = 2
```

Automatic and precise range positions

```
scala> "class C { def x = 2 }".parse[Stat]
res6: scala.meta.Stat = class C { def x = 2 }

scala> val q"class C { $method }" = res6
method: scala.meta.Stat = def x = 2

scala> method.tokens
res6: scala.meta.tokens.Tokens = Tokens(
def (10..13), (13..14), x (14..15), (15..16),
= (16..17), (17..18), 2 (18..19))
```

Hacky quasiquotes in scala.reflect

```
$ scala -Yquasiquote-debug

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> val name = TypeName("C")
name: reflect.runtime.universe.TypeName = C

scala> q"class $name"
...
```

Hacky quasiquotes in scala.reflect

```
$ scala -Yquasiquote-debug

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> val name = TypeName("C")
name: reflect.runtime.universe.TypeName = C

scala> q"class $name"
...
code to parse:
class qq$a2912896$macro$1
parsed:
Block(List(), ClassDef(Modifiers(),
TypeName("qq$a2912896$macro$1"), List(), Template(...))
...
```

Principled quasiquotes in scala.meta

```
$ scala -Dquasiquote.debug

scala> import scala.meta._, dialects.Scala211
import scala.meta._
import dialects.Scala211

scala> val name = t"C"
...
name: scala.meta.Type.Name = C

scala> q"class $name"
...
```

Principled quasiquotes in scala.meta

```
$ scala -Dquasiquote.debug

scala> import scala.meta._, dialects.Scala211
import scala.meta._
import dialects.Scala211

scala> val name = t"C"
...
name: scala.meta.Type.Name = C

scala> q"class $name"
...
Adhoc(List(BOF (0..0), class (0..5), (5..6),
$name (0..5), EOF (0..0)))
...
```

Derived technologies

First-class tokens enable:

- High-fidelity parsers
- Automatic and precise range positions
- Principled quasiquotes

Part 2: Semantic API

Getting started

```
$ scala

scala> import scala.meta._
import scala.meta._

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...
```

Getting started

```
$ scala

scala> import scala.meta._
import scala.meta._

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...
```

Contexts can come from:

- Mirror(...): useful for standalone apps

Getting started

```
$ scala

scala> import scala.meta._
import scala.meta._

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...
```

Contexts can come from:

- Mirror(...): useful for standalone apps
- Proxy(global): useful for compiler plugins

Getting started

```
$ scala

scala> import scala.meta._
import scala.meta._

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...
```

Contexts can come from:

- Mirror(...): useful for standalone apps
- Proxy(global): useful for compiler plugins
- Anywhere else: anyone can implement a context

Design goals

- 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

Implementation vehicle

First-class names

Bindings in scala.reflect

```
$ scala

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> showRaw(q"class C { def x = 2; def y = x }")
...
```

Bindings in scala.reflect

```
$ scala

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> showRaw(q"class C { def x = 2; def y = x }")
res1: String = ClassDef(
  Modifiers(), TypeName("C"), List(),
  Template(
    List(Select(Ident(scala), TypeName("AnyRef"))),
    noSelfType,
    List(
      DefDef(NoMods, termNames.CONSTRUCTOR, ...),
      DefDef(NoMods, TermName("x"), ..., Literal(Constant(2))),
      DefDef(NoMods, TermName("y"), ..., Ident(TermName("x"))))))
```

Bindings in scala.reflect

```
$ scala

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> showRaw(q"class C { def x = 2; def y = x }")
res1: String = ClassDef(
  Modifiers(), TypeName("C"), List(),
  Template(
    List(Select(Ident(scala), TypeName("AnyRef"))),
    noSelfType,
    List(
      DefDef(NoMods, termNames.CONSTRUCTOR, ...),
      DefDef(NoMods, TermName("x"), ..., Literal(Constant(2))),
      DefDef(NoMods, TermName("y"), ..., Ident(TermName("x"))))))
```

Bindings in scala.reflect

```
$ scala

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> showRaw(q"class C { def x = 2; def y = x }")
res1: String = ClassDef(
  Modifiers(), "C", List(),
  Template(
    List(Select(Ident(scala), "AnyRef")),
    noSelfType,
    List(
      DefDef(NoMods, termNames.CONSTRUCTOR, ...),
      DefDef(NoMods, "x", ..., Literal(Constant(2))),
      DefDef(NoMods, "y", ..., Ident("x")))))
```

Bindings in scala.reflect

```
$ scala

scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> showRaw(q"class C { def x = 2; def y = x }")
res1: String = ClassDef(
  Modifiers(), "C", List(),
  Template(
    List(Select(Ident(scala), "AnyRef")),
    noSelfType,
    List(
      DefDef(NoMods, termNames.CONSTRUCTOR, ...),
      DefDef(NoMods, "x", ..., Literal(Constant(2))),
      DefDef(NoMods, "y", ..., Ident("x")))))
```

Bindings in scala.meta

```
$ scala

scala> import scala.meta._, dialects.Scala211
import scala.meta._
import dialects.Scala211

scala> q"class C { def x = 2; def y = x }".show[Structure]
res1: String = Defn.Class(
  Nil, Type.Name("C"), Nil,
  Ctor.Primary(Nil, Ctor.Name("this"), Nil),
  Template(
    Nil, Nil,
    Term.Param(Nil, Name.Anonymous(), None, None),
    Some(List(
      Defn.Def(Nil, Term.Name("x"), ..., Lit.Int(2)),
      Defn.Def(Nil, Term.Name("y"), ..., Term.Name("x"))))))
```

Key example

```
List[Int]
```

Key example

```
scala> t"List[Int]".show[Structure]
res1: String =
Type.Apply(Type.Name("List"), List(Type.Name("Int")))

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...
```

Key example

```
scala> t"List[Int]".show[Structure]
res1: String =
Type.Apply(Type.Name("List"), List(Type.Name("Int")))

scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...

scala> t"List[Int]".show[Semantics]
res3: String =
Type.Apply(Type.Name("List")[1], List(Type.Name("Int")[2]))
[1] {1}::scala.package#List
[2] {2}::scala#Int
...
```

Name resolution

```
scala> implicit val mirror = Mirror(...)
mirror: scala.meta.Mirror = ...

scala> q"scala.collection.immutable.List".defn
res2: scala.meta.Member.Term = object List extends
SeqFactory[List] with Serializable { ... }

scala> res2.name
res3: scala.meta.Term.Name = List
```

Other semantic APIs

```
scala> q"scala.collection.immutable.List".defs("apply")
res4: scala.meta.Member.Term =
override def apply[A](xs: A*): List[A] = ???

scala> q"scala.collection.immutable.List".supermembers
res5: Seq[scala.meta.Member.Term] =
List(abstract class SeqFactory...)
```

Derived technologies

First-class names enable:

- Unification of trees, types and symbols
- Referential transparency and hygiene (under development!)
- Simpler mental model of metaprogramming

Part 3: Live demo

Part 4: Roadmap

Where we've been before

- With scala.meta, we started from complete scratch

Lots of experimentation

- Safe by construction trees
- High-fidelity parsing
- Automatic and precise range positions
- Principled quasiquotes
- Unification of trees, symbols and types
- AST persistence
- AST interpretation
- Simple syntax and compilation for macros
- IDE support for macros
- SBT support for macros
- ...

Where we are now

- Tokens provide an elegant and powerful foundation for syntactic APIs
- Names enable a simple mental model for semantic APIs
- People are already successfully using these new concepts!

Where we will be soon

- Experimentation's temporarily on hold, we're now pushing for 0.1
- Main focus of 0.1 is making scala.meta trees publicly available
- https://github.com/scalameta/scalameta/milestones/0.1

Where we will be soon

- Experimentation's temporarily on hold, we're now pushing for 0.1
- Main focus of 0.1 is making scala.meta trees publicly available
- https://github.com/scalameta/scalameta/milestones/0.1

Contributor alert!
https://github.com/scalameta/scalameta/issues

Wrapping up

Summary

- scala.meta is a one-stop solution to frontend metaprogramming
- Our key innovations include first-class support for tokens and names
- We're now pushing for the 0.1 preview release
- Join us at https://gitter.im/scalameta/scalameta!