What Did We Learn In Scala. Meta?

Eugene Burmako

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

11 February 2016

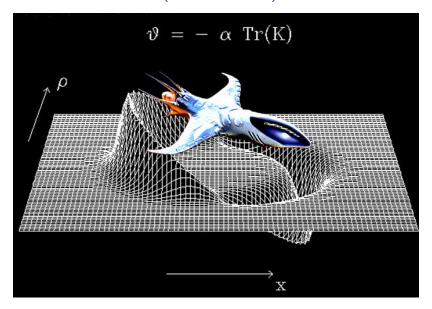
A big experiment (Berlin 2014)



Initial practical results (San Francisco 2015)



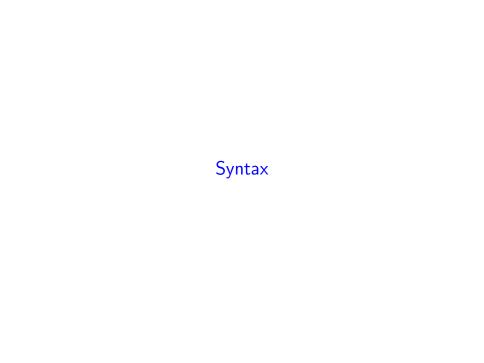
Theoretical discussion (Krakow 2016)



In this talk

Q: What did we learn in scala.meta?

A: How to design better ASTs.



Problem (scala.reflect)

Trees can't precisely represent Scala's syntax

Lossy representation (scala.reflect)

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x" forLoop: Tree = List(1, 2, 3).map(((x) => x.$times(x)))
```

Lossy representation (scala.reflect)

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x"
forLoop: Tree = List(1, 2, 3).map(((x) \Rightarrow x.\$times(x)))
scala> showRaw(forLoop)
res0: String = Apply(
  Select(
    Apply(Ident(TermName("List")), ...),
    TermName("map")),
  List(
    Function(
      List(ValDef(Modifiers(PARAM), TermName("x"), ...)),
      Apply(
        Select(Ident(TermName("x")), TermName("$times")),
        List(Ident(TermName("x"))))))
```

Lossy representation (scala.reflect)

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x"
forLoop: Tree = List(1, 2, 3).map(((x) \Rightarrow x.$times(x)))
scala> showRaw(forLoop)
res0: String = Apply(
  Select(
    Apply(Ident(TermName("List")), ...),
    TermName("map")),
  List(
    Function (
      List(ValDef(Modifiers(PARAM), TermName("x"), ...)),
      Apply(
        Select(Ident(TermName("x")), TermName("$times")),
        List(Ident(TermName("x"))))))
```

Lax representation (scala.reflect)

```
scala> val List = tq"scala.List"
List: Select = scala.List
scala > val list = q"$List(1, 2, 3)"
list: Tree = scala.List(1, 2, 3)
scala> toolbox.eval(list)
s.t.r.ToolBoxError: type scala.List is not a value
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:178)
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:170)
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:148)
```

Lax representation (scala.reflect)

```
scala> val List = tq"scala.List"
List: Select = scala.List
scala > val list = q"$List(1, 2, 3)"
list: Tree = scala.List(1, 2, 3)
scala> toolbox.eval(list)
s.t.r.ToolBoxError: type scala.List is not a value
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:178)
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:170)
  at s.t.r.ToolBoxFactory$...apply(ToolBoxFactory.scala:148)
```

Funny representation (scala.reflect)

```
scala> q"class C(x: Int)"
res0: ClassDef =
class C extends scala.AnyRef {
   <paramaccessor> private[this] val x: Int = _;
   def <init>(x: Int) = {
      super.<init>();
      ()
   }
}
```

Consequences (scala.reflect)

scala.reflect has a lossy, lax and funny representation of Scala syntax:

- ▶ It's impossible to have 100% robust solutions
- Every metaprogram needs to know about funny encodings
- ▶ Complexity estimate: SyntacticXXX extractors and supporting infrastructure in the implementation of quasiquotes (~2kloc)

Solution (scala.meta)

Faithfully model all intricacies of syntax even if it's hard

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x" forLoop: Term = for (x <- List(1, 2, 3)) yield x * x
```

```
scala> val forLoop = q''for (x \leftarrow List(1, 2, 3)) yield x * x''
forLoop: Term = for (x \leftarrow List(1, 2, 3)) yield x * x
scala> forLoop.show[Structure]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")),
    Term.Apply(Term.Name("List"), ...))),
  Term.ApplyInfix(
    Term.Name("x"),
    Term.Name("*").
    Nil, Seq(Term.Name("x"))))
```

```
scala> val forLoop = q''for (x \leftarrow List(1, 2, 3)) yield x * x''
forLoop: Term = for (x \leftarrow List(1, 2, 3)) yield x * x
scala> forLoop.show[Structure]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat.Var.Term(Term.Name("x")),
    Term.Apply(Term.Name("List"), ...))),
  Term.ApplyInfix(
    Term.Name("x"),
    Term.Name("*").
    Nil, Seq(Term.Name("x"))))
```

```
scala> val forLoop = q''for (x \leftarrow List(1, 2, 3)) yield x * x''
forLoop: Term = for (x \leftarrow List(1, 2, 3)) yield x * x
scala> forLoop.show[Structure]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")),
    Term.Apply(Term.Name("List"), ...))),
  Term.ApplyInfix(
    Term.Name("x"),
    Term.Name("*").
    Nil, Seq(Term.Name("x"))))
```

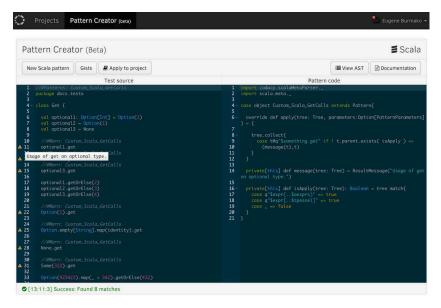
Safety by construction (scala.meta)

```
scala> val forLoop = q''for (x \leftarrow List(1, 2, 3)) yield x * x''
forLoop: Term = for (x \leftarrow List(1, 2, 3)) yield x * x
scala> forLoop.show[Structure]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")),
    Term.Apply(Term.Name("List"), ...))),
  Term.ApplyInfix(
    Term.Name("x"),
    Term.Name("*"),
    Nil, Seq(Term.Name("x"))))
```

Implementation effort

- Several months of iterations on AST definitions
- ▶ Several weeks of adapting the old parser to emit new trees
- ► GSoC project on implementing quasiquotes for new trees
- ▶ The functionality is self-contained (no changes to the compiler!)

Demo time: Scala code patterns at codacy.com



Challenge #1: Scala's syntax is hard

- ▶ Have to model syntactic irregularities
- Unfortunately there's a bunch of them
- ► Constant tension between precise modelling and retaining sanity

Challenge #1: Some examples of irregularities

- "Patterns" in val/var declarations
- ▶ Pattern variables in type(!) patterns
- new and super constructor calls
- "Names" in private[Foo] and Bar.this qualifiers
- **.**..

Challenge #2: Concrete syntax trees are hard

```
scala> val q"$fn(..$args)" = q"2 + 2"
scala.MatchError: 2 + 2 (of class Term$ApplyInfix$Impl)
    ... 33 elided

scala> val q"$arg $op (..$args)" = q"2 + 2"
arg: Term = 2
op: Term.Name = +
args: Seq[Term.Arg] = List(2)
```

- ► CSTs are harder to process uniformly than ASTs
- ▶ Need more experience to better understand the trade-offs
- Maybe there doesn't exist a one-size-fits-all solution

Challenge #3: Need to understand dialects

- ▶ It's not enough to support just Scala 2.11/2.12
- ▶ Some features are about to be deprecated in future versions
- Dotty is gaining traction, and it's going to have new features

Tokens

Problem (scala.reflect)

Trees can't reflect on underlying lexemes

Forgotten trivia (scala.reflect)

```
scala> q"/** doc */ class C(x: Int)"
res0: ClassDef = class C...
```

Can positions be the answer? (scala.reflect)

```
$ parse -Xprint-pos "/** doc */ class C(x: Int)"
[[syntax trees at end of parser]]// Scala source: tmpiIkEYU
[11:26] package [11:11] {
  [11:26]class C extends [18:26][26]scala.AnyRef {
    [19:25] private[this] val x: [22:25]Int = _;
    <19:25>def <init>(<19:25>x: [22]Int) = <19:25>{
      [NoPosition] [NoPosition] super.<init>();
     <19:25>()
```

There are still problems with trivia (scala.reflect)

```
$ parse -Xprint-pos "/** doc */ class C(x: Int)"
[[syntax trees at end of parser]]// Scala source: tmpiIkEYU
[11:26]package [11:11] {
  [11:26]class C extends [18:26][26]scala.AnyRef {
    [19:25] private[this] val x: [22:25]Int = _;
    <19:25>def <init>(<19:25>x: [22]Int) = <19:25>{
      [NoPosition] [NoPosition] super.<init>();
     <19:25>()
```

Some lexemes don't have positions (scala.reflect)

```
$ parse -Xprint-pos "/** doc */ class C(x: Int)"
[[syntax trees at end of parser]]// Scala source: tmpiIkEYU
[11:26]package [11:11] {
  [11:26]class C extends [18:26][26]scala.AnyRef {
    [19:25] private[this] val x: [22:25]Int = _;
    <19:25>def <init>(<19:25>x: [22]Int) = <19:25>{
      [NoPosition] [NoPosition] super.<init>();
     <19:25>()
```

Consequences (scala.reflect)

scala.reflect ignores trivia and doesn't have fully-featured positions:

- ▶ Tools that want to work on lexical level must reinvent the wheel
- ▶ This hampers the evolution of the tool ecosystem

Solution (scala.meta)

Reify tokens

Reified tokens

```
scala> "/** doc */ class C(x: Int)".parse[Stat]
res0: scala.meta.Stat = /** doc */ class C(x: Int)
```

Reified tokens

```
scala> "/** doc */ class C(x: Int)".parse[Stat]
res0: scala.meta.Stat = /** doc */ class C(x: Int)

scala> res0.tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
/** doc */ (0..10), (10..11), class (11..16), (16..17),
C (17..18), ( (18..19), x (19..20), : (20..21), (21..22),
Int (22..25), ) (25..26), EOF (26..26))
```

Reified tokens

```
scala> "/** doc */ class C(x: Int)".parse[Stat]
res0: scala.meta.Stat = /** doc */ class C(x: Int)

scala> res0.tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
/** doc */ (0..10), (10..11), class (11..16), (16..17),
C (17..18), ( (18..19), x (19..20), : (20..21), (21..22),
Int (22..25), ) (25..26), EOF (26..26))
```

Reified tokens

```
scala> "/** doc */ class C(x: Int)".parse[Stat]
res0: scala.meta.Stat = /** doc */ class C(x: Int)

scala> res0.tokens
res1: scala.meta.tokens.Tokens = Tokens(BOF (0..0),
/** doc */ (0..10), (10..11), class (11..16), (16..17),
C (17..18), ( (18..19), x (19..20), : (20..21), (21..22),
Int (22..25), ) (25..26), EOF (26..26))
```

Implementation effort

- Several days to do the first sketch of tokens
- ▶ Need several more weeks to come up with an optimized representation
- Several weeks to change the old tokenizer to emit reified tokens
- Again, the functionality is self-contained

Demo time: Scalafmt (created by @olafurpg)





Code style should not be enforced by review, but by automate rewriting. Evolve the style using PRs against the rewriting config.



9:00 AM - 7 Feb 2016

Demo time: Scalafmt (created by @olafurpg)

```
trait Style {
  def maxColumn: Int = 80
  def indent: Int = 2
  ...
}
object ScalaFmt {
  def format(code: String, style: Style): String = ???
}
```

Challenge #1: Who owns trivia?

```
class C {
  def y = 2

  /** doc */
  def z = 3
}
```

- ▶ Do tokens for y and z include indentation?
- What about documentation comments?
- Roslyn's provides a great practical heuristic

Challenge #2: When to work with tokens?

- Working with trees prevents syntax errors
- ▶ However clang-format shows that sometimes that's too high-level
- ► How should the low-level API look like?
- Need more experience to better understand the trade-offs

Semantics

Problem (scala.reflect)

Attributed trees have platform-dependent representation

Desugaring (scala.reflect)

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x"
forLoop: Tree = List(1, 2, 3).map(((x) => x.$times(x)))

scala> toolbox.typecheck(forLoop)
res0: Tree = immutable.this.List.apply[Int](1, 2, 3)
.map[Int, List[Int]](((x: Int) => x.*(x)))
(immutable.this.List.canBuildFrom[Int])
```

Desugaring (dotty)

```
$ typecheck 'for (x <- List(1, 2, 3)) yield x * x'
[[syntax trees at end of frontend]]// Scala source: tmpwUlD8P
List.apply[Int']([1,2,3]: Int*).map[Int',
    scala.collection.immutable.List[Int']'
]({
    def $anonfun(x: Int): Int' = x.*(x)
    closure($anonfun)
})(scala.collection.immutable.List.canBuildFrom[Int'])</pre>
```

Consequences (scala.reflect)

scala.reflect represents attributed trees in platform-dependent way:

- Further impairs WYSIWYG metaprogramming
- Makes it very hard to write portable metaprograms
- Macros are probably hit the most

Solution (scala.meta)

Platform-independent semantic model

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x"
forLoop: Term = for (x \leftarrow List(1, 2, 3)) yield x * x
scala> forLoop.show[Semantics]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")[1]{1}<>),
    Term.Apply(Term.Name("List")[2]{2}<1>, ...))),
  Term.ApplyInfix(
    Term. Name ("x") [1] {1} <>,
    Term.Name("*")[3]{4}<>.
    Nil, Seq(Term.Name("x")[1]{1}<>)){1}<>){3}<2>
```

```
scala > forLoop.show[Semantics]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")[1]{1}<>),
    Term.Apply(Term.Name("List")[2]{2}<1>, ...))),
  Term.ApplyInfix(
    Term. Name ("x") [1] {1} <>,
    Term.Name("*")[3]\{4\}<>,
    Nil, Seq(Term.Name("x")[1]{1}<>)){1}<>){3}<2>
[1] {0}::local#4efdc590-bcf6-4980-8ad3-06932cb59446
[2] {5}::scala.collection.immutable.List
[3] {6}::scala#Int.*(I)I
```

```
scala > forLoop.show[Semantics]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")[1]{1}<>),
    Term.Apply(Term.Name("List")[2]{2}<1>, ...))),
  Term.ApplyInfix(
    Term.Name("x")[1]{1}<>,
    Term.Name("*")[3]\{4\}<>,
    Nil, Seq(Term.Name("x")[1]{1}<>)){1}<>){3}<2>
{1} Type.Name("Int")[4]
{2} Type.Singleton(Term.Name("List")[2]{2}<1>)
```

```
scala> forLoop.show[Semantics]
res0: String = Term.ForYield(
  Seq(Enumerator.Generator(
    Pat. Var. Term(Term. Name("x")[1]{1}<>),
    Term.Apply(Term.Name("List")[2]{2}<1>, ...))),
  Term.ApplyInfix(
    Term.Name("x")[1]{1}<>,
    Term.Name("*")[3]\{4\}<>,
    Nil, Seq(Term.Name("x")[1]{1}<>)){1}<>){3}<2>
<1> Term.ApplyType(Term.Name("List")[2]{7}<3>, ...Int...)
<2> Term.Apply(Term.Apply(...), ...)
```

Implementation effort

- Several months working on the v1 of the converter
- Start from scratch
- Several months working on the v2 of the converter
- Still can handle only fairly basic programs
- Maintainability??

Challenge #1: Implementation strategy

	Difficulty	Reliability	Overhead
Compiler plugin	Very hard	Brittle	None
Compiler module	Very hard	Moderate	Moderate
Integrated into typer	Hard	Robust	High

Challenge #2: When to desugar?

```
scala> val forLoop = q"for (x <- List(1, 2, 3)) yield x * x"
forLoop: Tree = List(1, 2, 3).map(((x) => x.$times(x)))

scala> toolbox.typecheck(forLoop)
res0: Tree = immutable.this.List.apply[Int](1, 2, 3)
.map[Int, List[Int]](((x: Int) => x.*(x)))
(immutable.this.List.canBuildFrom[Int])
```

- Sometimes need original trees
- Sometimes need desugared trees
- On-demand desugaring seems reasonable

Challenge #3: How to be platform-independent?

- ▶ Even if we desugar on demand, that's still platform-dependent
- ▶ Unless we have a detailed spec of the typechecker
- (Probably not going to happen)

What did we learn in scala.meta?

▶ Almost all scala.reflect problems can be solved by better ASTs

- ▶ Almost all scala.reflect problems can be solved by better ASTs
- ▶ The rest can be reduced to AST problems

- ▶ Almost all scala.reflect problems can be solved by better ASTs
- ► The rest can be reduced to AST problems
- ▶ And then solved

It's definitely worth it to have richer parse trees:

- ► Concrete syntax trees are essential for writing robust metaprograms
- Reified tokens enable new kinds of useful tools
- Implementation complexity is moderate and isolated

Attributing parse trees is very useful but very hard:

- ► Finally provides a way to write macros in WYSIWYG style
- ▶ And makes it possible to develop robust frontend tooling
- Implementation complexity goes through the roof
- ▶ This is the biggest open question in scala.meta right now

Credits

- Uladzimir Abramchuk
- Eric Beguet
- ▶ Igor Bogomolov
- Eugene Burmako
- Mathieu Demarne
- ► Martin Duhem
- Ólafur Páll Geirsson
- Adrien Ghosn
- Zhivka Gucevska
- Vojin Jovanovic
- Guillaume Massé

- Guillaume Martres
- Mikhail Mutcianko
- Dmitry Naydanov
- Artem Nikiforov
- Vladimir Nikolaev
- Martin Odersky
- Oleksandr Olgashko
- Alexander Podkhalyuzin
- Jatin Puri
- Dmitry Petrashko
- ▶ Valentin Rutz
- Denys Shabalin