Magic at compile time

Metaprogramming in scala

About me

Bartosz Bąbol

Twitter: @BBartosz91

Blog: www.bbartosz.com

About me

Bartosz Bąbol

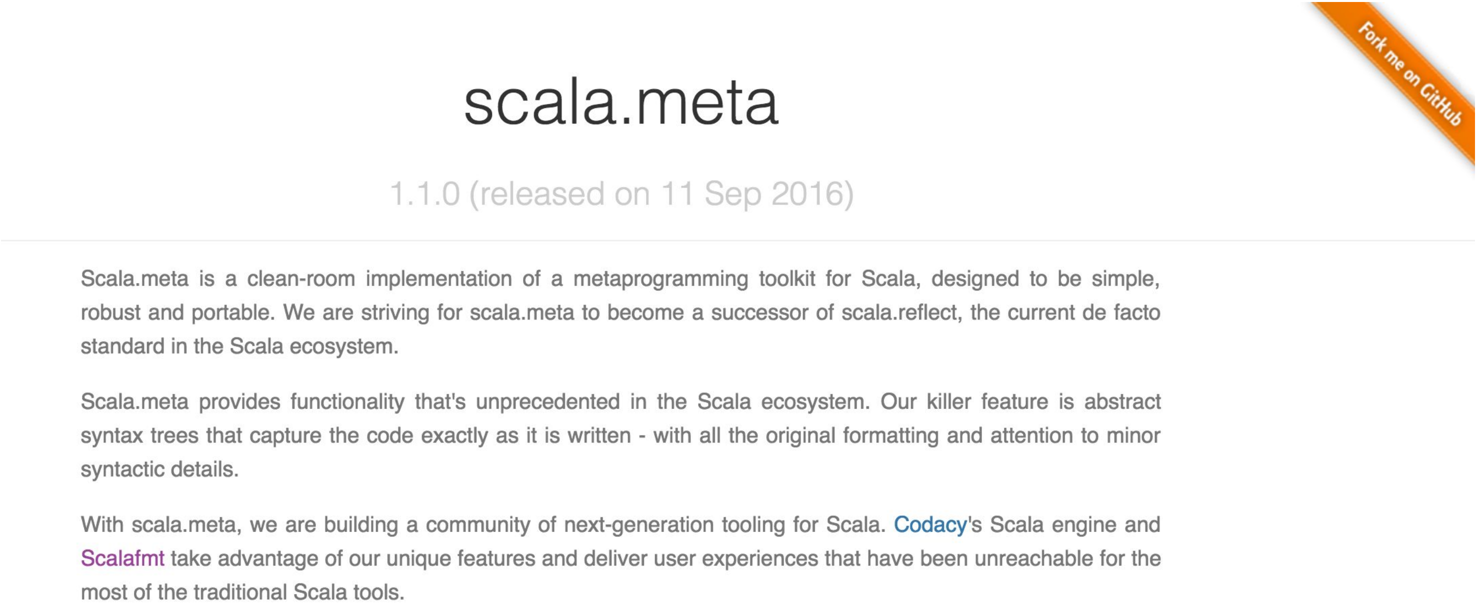
Twitter: @BBartosz91

Blog: www.bbartosz.com

agoda

jobs@agoda.com

Introduction



scala.meta

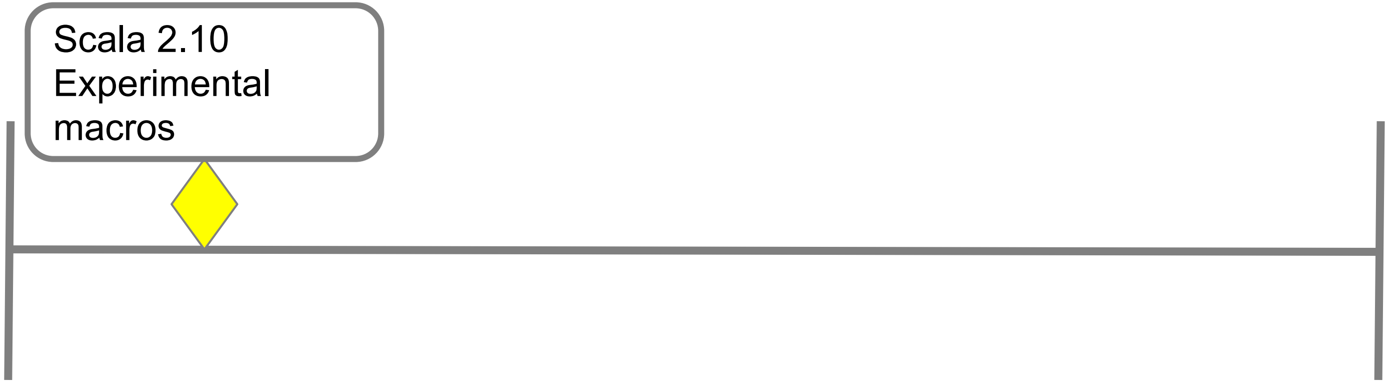
1.1.0 (released on 11 Sep 2016)

Scala.meta is a clean-room implementation of a metaprogramming toolkit for Scala, designed to be simple, robust and portable. We are striving for scala.meta to become a successor of scala.reflect, the current de facto standard in the Scala ecosystem.

Scala.meta provides functionality that's unprecedented in the Scala ecosystem. Our killer feature is abstract syntax trees that capture the code exactly as it is written - with all the original formatting and attention to minor syntactic details.

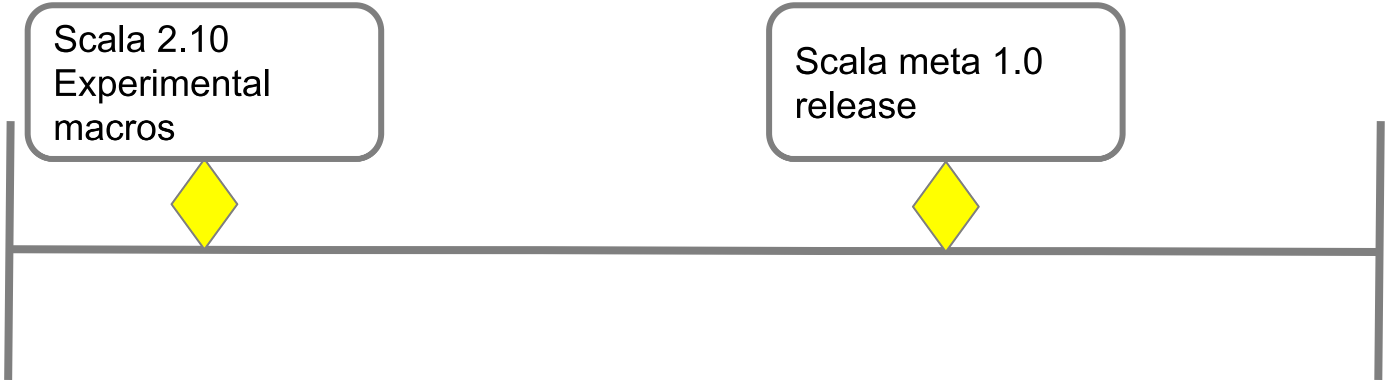
With scala.meta, we are building a community of next-generation tooling for Scala. Codacy's Scala engine and Scalafmt take advantage of our unique features and deliver user experiences that have been unreachable for the most of the traditional Scala tools.

Long story short

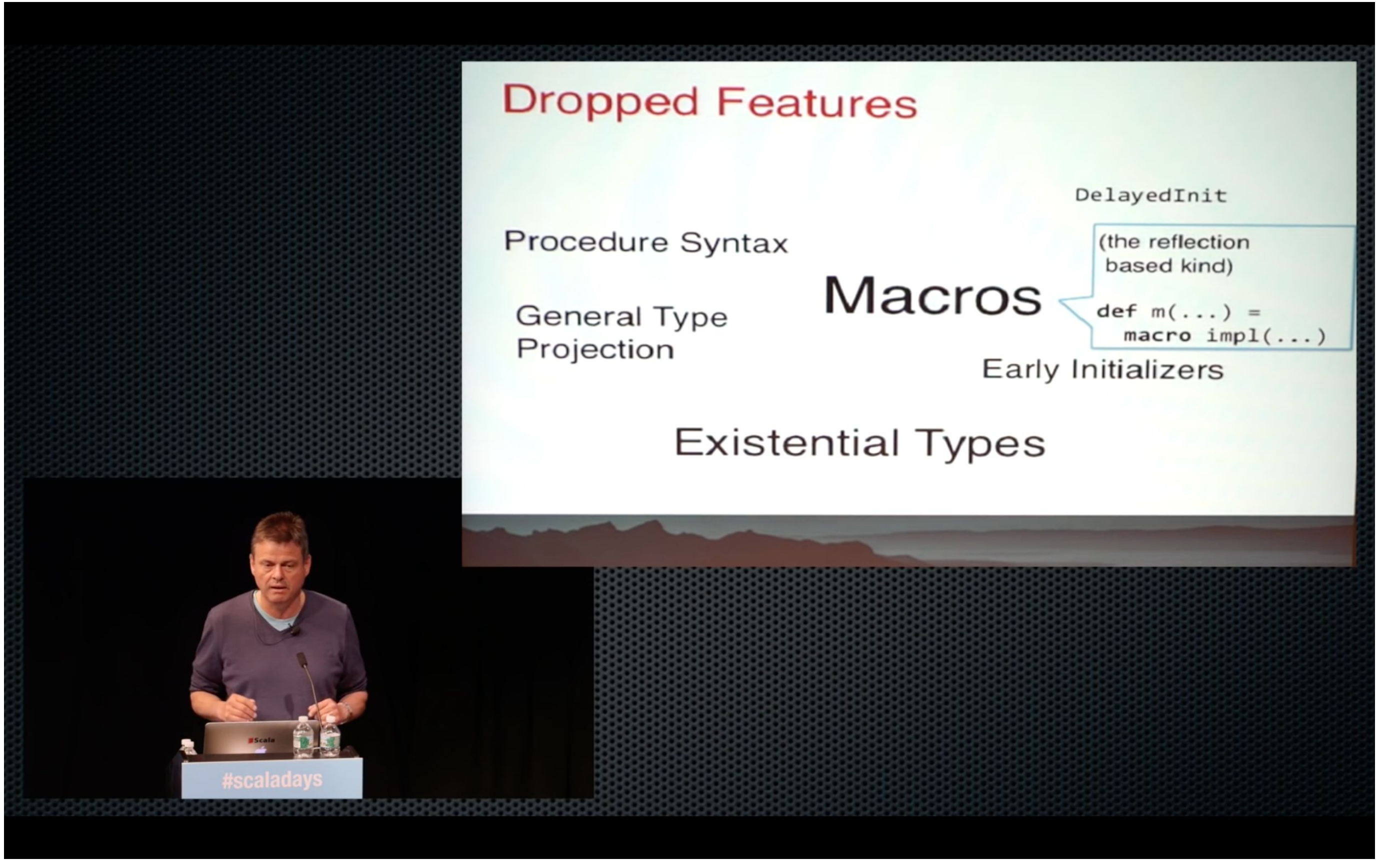


Scala 2.10
Experimental
macros

Long story short



Macros are dead



Dropped Features

DelayedInit

Procedure Syntax

(the reflection based kind)

Macros

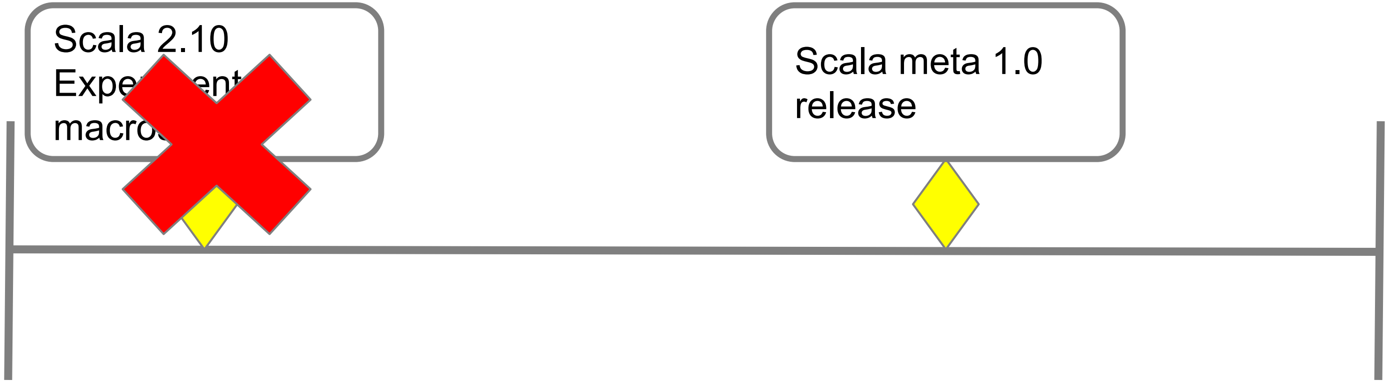
```
def m(...) =
  macro impl(...)
```

General Type Projection

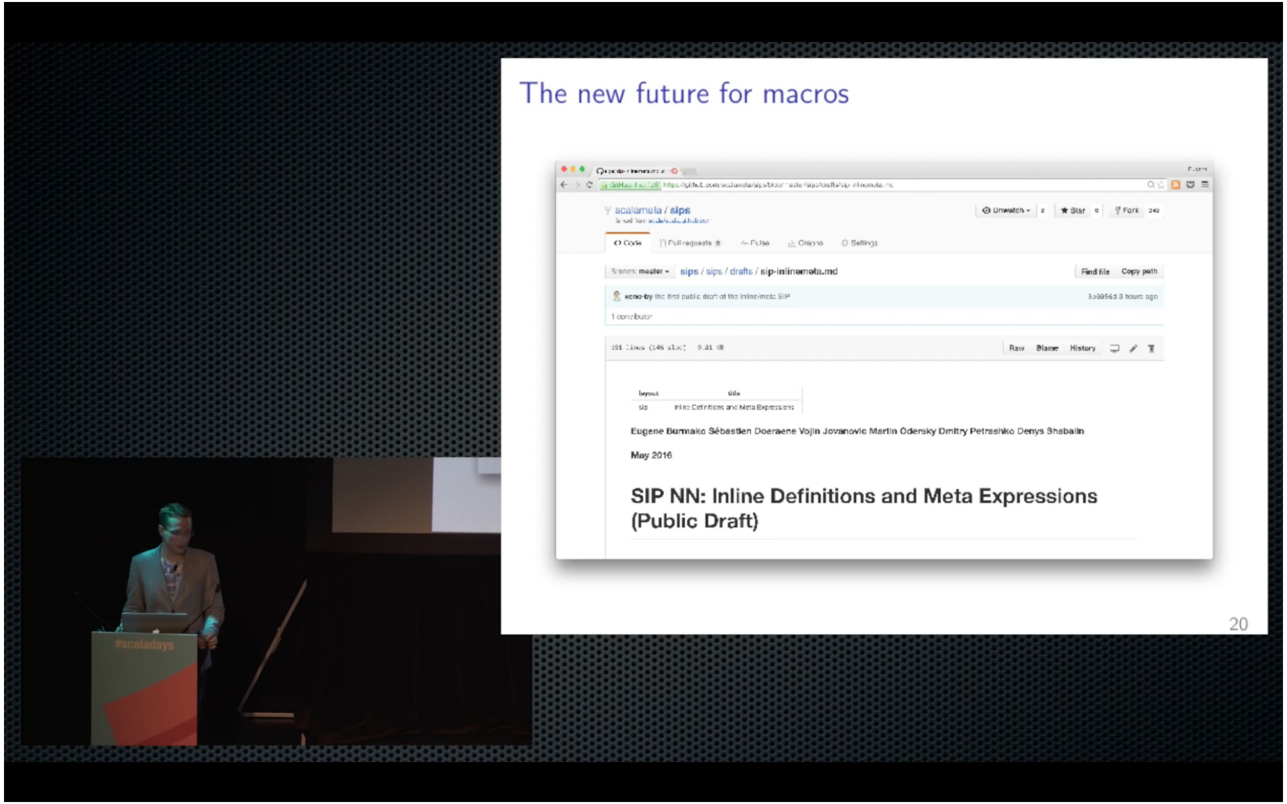
Early Initializers

Existential Types

Long story short



New inline macros



The new future for macros

sips / sips / drafts / sip-inlinemeta.md

xeno-by the first public draft of the inline/meta SIP

1 contributor

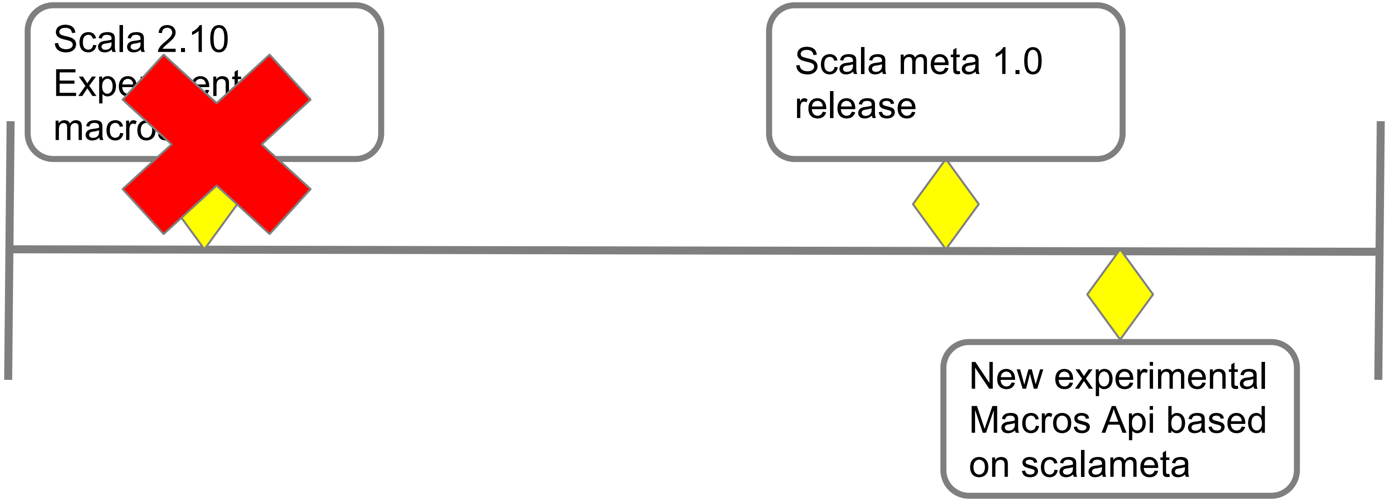
layout	title
sip	Inline Definitions and Meta Expressions

Eugene Burmako Sébastien Doeraene Vojin Jovanovic Martin Odersky Dmitry Petrashko Denys Shabalin

May 2016

SIP NN: Inline Definitions and Meta Expressions (Public Draft)

Long story short

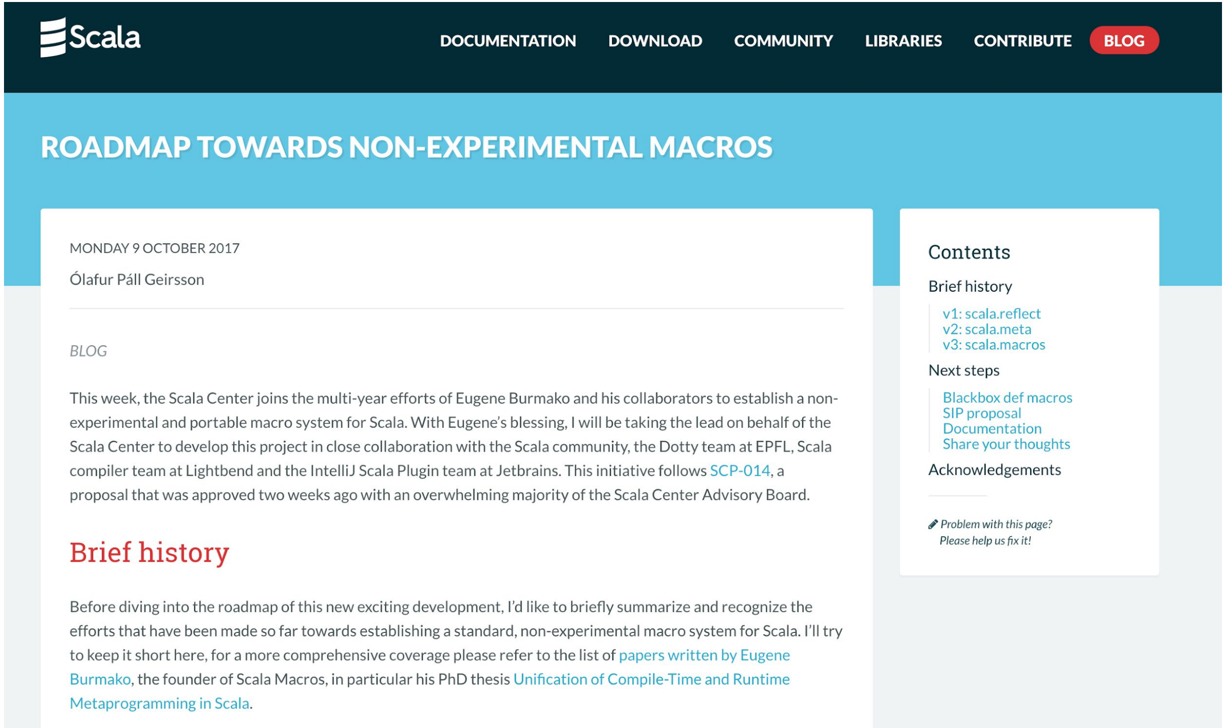


Scala 2.10
Expe ent
macro

Scala meta 1.0
release

New experimental
Macros Api based
on scalameta

Long story short



ROADMAP TOWARDS NON-EXPERIMENTAL MACROS

MONDAY 9 OCTOBER 2017

Ólafur Páll Geirsson

BLOG

This week, the Scala Center joins the multi-year efforts of Eugene Burmako and his collaborators to establish a non-experimental and portable macro system for Scala. With Eugene's blessing, I will be taking the lead on behalf of the Scala Center to develop this project in close collaboration with the Scala community, the Dotty team at EPFL, Scala compiler team at Lightbend and the IntelliJ Scala Plugin team at Jetbrains. This initiative follows SCP-014, a proposal that was approved two weeks ago with an overwhelming majority of the Scala Center Advisory Board.

Brief history

Before diving into the roadmap of this new exciting development, I'd like to briefly summarize and recognize the efforts that have been made so far towards establishing a standard, non-experimental macro system for Scala. I'll try to keep it short here, for a more comprehensive coverage please refer to the list of papers written by Eugene Burmako, the founder of Scala Macros, in particular his PhD thesis Unification of Compile-Time and Runtime Metaprogramming in Scala.

Contents

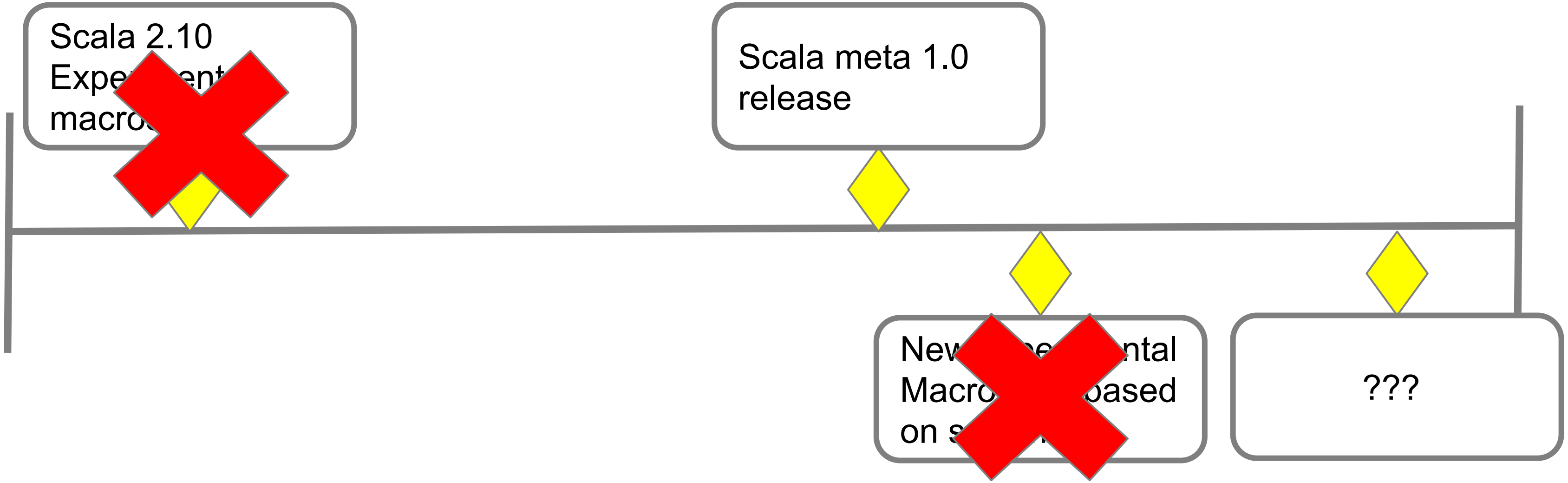
Brief history
- v1: scala.reflect
- v2: scala.meta
- v3: scala.macros

Next steps
- Blackbox def macros
- SIP proposal
- Documentation
- Share your thoughts

Acknowledgements

Problem with this page?
Please help us fix it!

Long story short



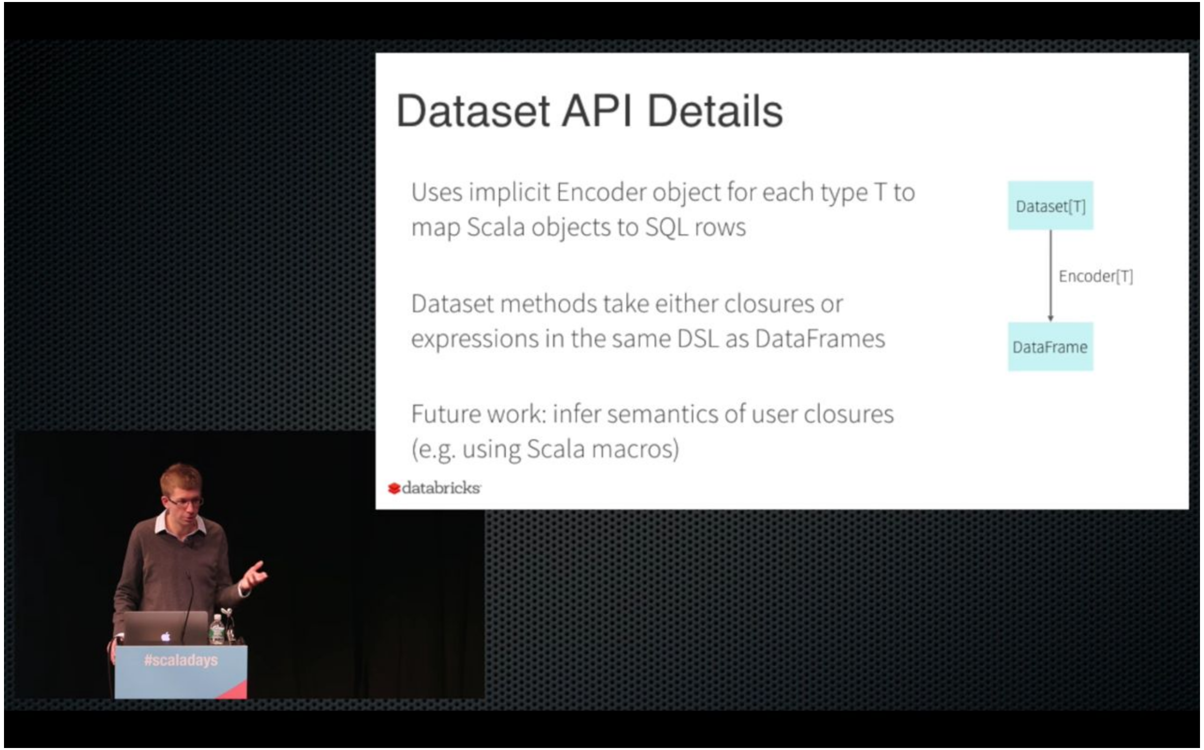
Scala 2.10
Expe ent
macro

Scala meta 1.0
release

New ntal
Macro based
on s

???

Spark 2.0 roadmap



Dataset API Details

Uses implicit Encoder object for each type T to map Scala objects to SQL rows

Dataset methods take either closures or expressions in the same DSL as DataFrames

Future work: infer semantics of user closures (e.g. using Scala macros)

Dataset[T]

Encoder[T]

DataFrame

Who uses metaprogramming?

- Play Framework
- Spray
- Shapeless
- Cats
- Slick
- ReactiveMongo
- Rapture

What is metaprogramming?

The prefix *meta-* is used to mean *about (its own category)*

What is metaprogramming?

The prefix *meta-* is used to mean *about (its own category)*

- *Meta-joke*

What is metaprogramming?

The prefix *meta-* is used to mean *about (its own category)*

- *Meta-joke*
- *Meta-data*

What is metaprogramming?

The prefix *meta-* is used to mean *about (its own category)*

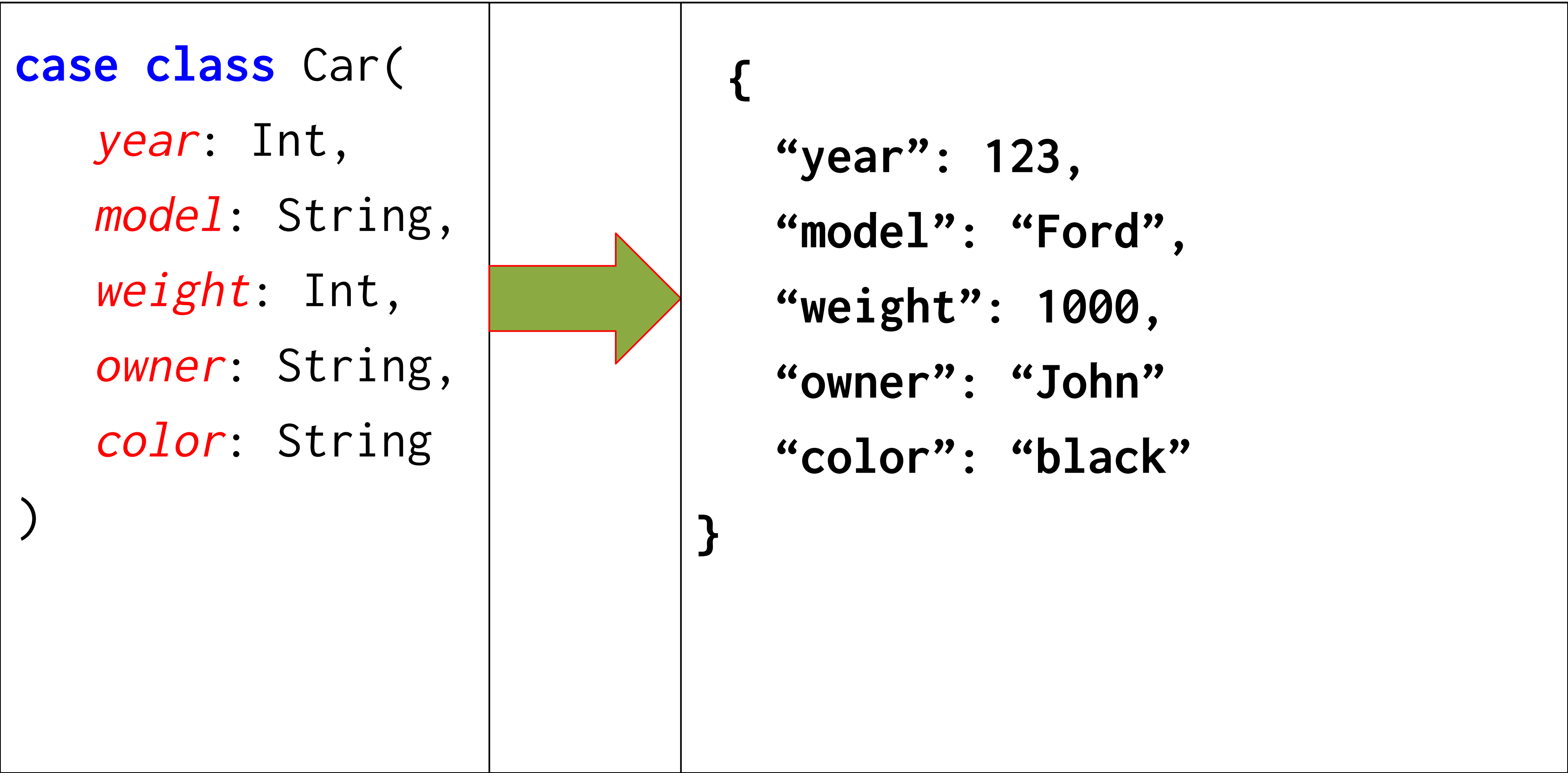
- *Meta-joke*
- *Meta-data*
- *Meta-programming*

What is metaprogramming?

The prefix *meta-* is used to mean *about (its own category)*

- *Meta-joke*
- *Meta-data*
- *Meta-programming*

Case



```
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```

```
 {
   “year”: 123,
   “model”: “Ford”,
   “weight”: 1000,
   “owner”: “John”
   “color”: “black”
}
```

Case

```scala
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```

```scala
implicit val carWrites = (
  (__ \ "year").write[Int] and
  (__ \ "model").write[String] and
  (__ \ "weight").write[Int] and
  (__ \ "owner").write[String] and
  (__ \ "color").write[String]
)(unlift(Car.unapply))
```

Case

```
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```

```
implicit val carWrites = Json.writes[Car]
```

RetryOnFailure

```scala
def failMethod[String](): Unit = {
   val random = Random.nextInt(10)
   println(s"evaluating random= $random")
   utils.methodThrowingException(random)
}
```

RetryOnFailure

```
def failMethod[String](): Unit = {
   import scala.util.Try
   for( a <- 1 to 20){
     val res = Try(//Body of our function)
     }
     if(res.isSuccess) return res.get
   }
   throw new Exception("failMethod fails after 20 repeats")
 }
```

RetryOnFailure

```
@RetryOnFailure(20)
def failMethod[String](): Unit = {
   //Body of our function
}
```

RetryOnFailure

```scala
import scala.meta._

class RetryOnFailure(repeat: Int) extends scala.annotation.StaticAnnotation {
 inline def apply(defn: Any): Any = meta {
   defn match {
     case q"..$mods def $name[..$tparams](...$paramss): $tpeopt = $expr" => {
     //body of annotation
     }
     case _ => abort("@RetryOnFailure can be annotation of method only")
   }
 }
}
```

RetryOnFailure

```
case q"..$mods def $name[..$tparams](...$paramss): $tpeopt = $expr"
```

RetryOnFailure

```
case q"..$mods def $name[..$tparams](...$paramss): $tpeopt = $expr"
```

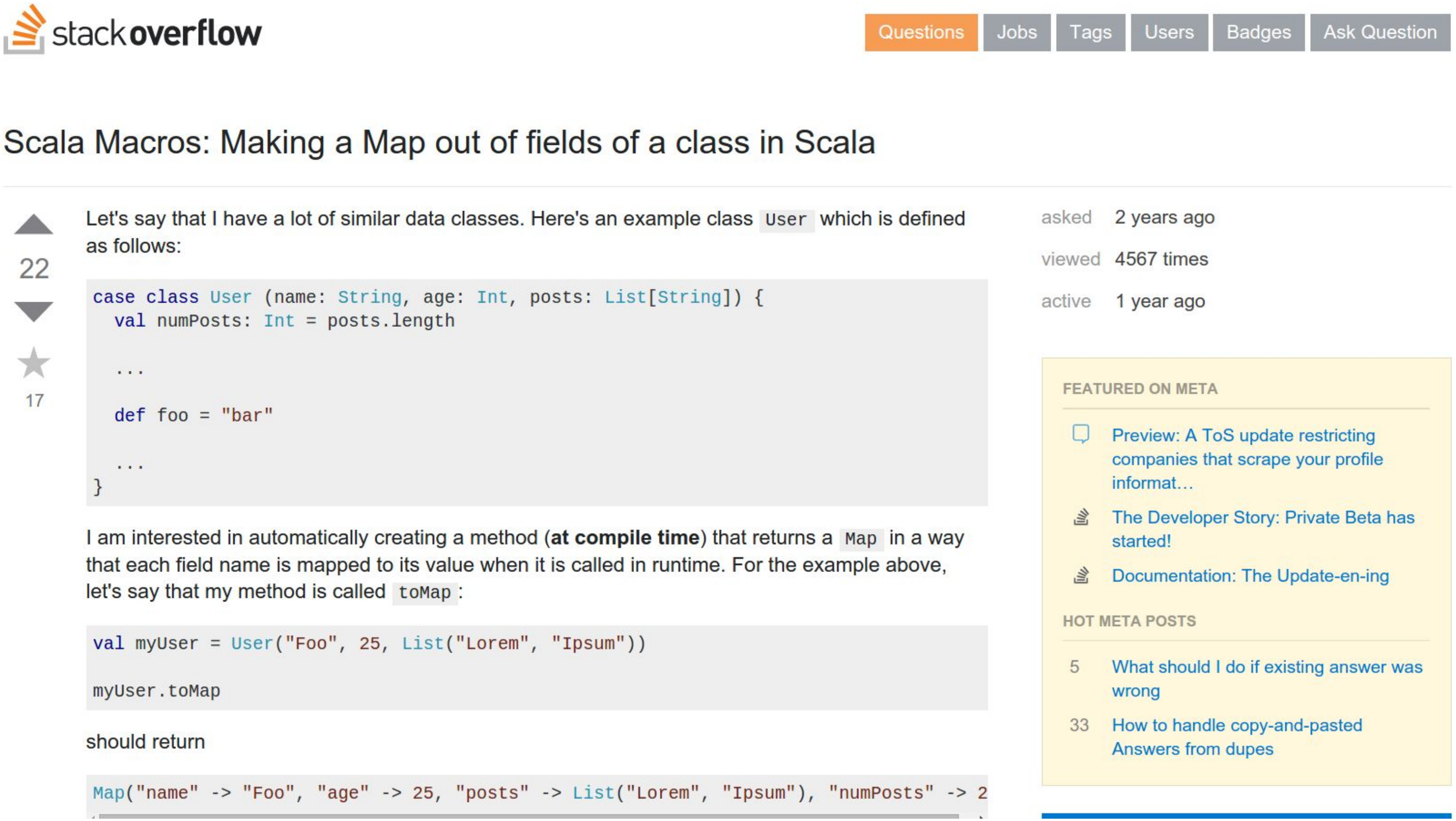
```
..$tparams => List[meta.Type]

…$paramss => List[List[meta.Term.Param]]
```

RetryOnFailure

```
q"""..$mods def $name[..$tparams](...$paramss): $tpeopt = {
     import scala.util.Try
     for( a <- 1 to $repeats){
       val res = Try($expr)
       if(res.isSuccess){
         return res.get
       }
     }
     throw new Exception("Method fails after "+$repeats + " repeats")}"""
```

Quasiquotes	Tree
q“”” “functional conf” “””	Lit("functional conf")
q”x+y”	Term.ApplyInfix(Term.Name("x"), Term.Name("+"), Nil, Seq(Term.Name("y")))



Scala Macros: Making a Map out of fields of a class in Scala

Let's say that I have a lot of similar data classes. Here's an example class `User` which is defined as follows:

```
case class User (name: String, age: Int, posts: List[String]) {
  val numPosts: Int = posts.length

  ...

  def foo = "bar"

  ...
}
```

I am interested in automatically creating a method (**at compile time**) that returns a `Map` in a way that each field name is mapped to its value when it is called in runtime. For the example above, let's say that my method is called `toMap`:

```
val myUser = User("Foo", 25, List("Lorem", "Ipsum"))

myUser.toMap
```

should return

```
Map("name" -> "Foo", "age" -> 25, "posts" -> List("Lorem", "Ipsum"), "numPosts" -> 2
```

asked 2 years ago

viewed 4567 times

active 1 year ago

FEATURED ON META

- Preview: A ToS update restricting companies that scrape your profile informat…
- The Developer Story: Private Beta has started!
- Documentation: The Update-en-ing

HOT META POSTS

5 What should I do if existing answer was wrong

33 How to handle copy-and-pasted Answers from dupes

```
@Mappable
case class User(
id: Long, name: String, email: String, age: Int)

val user = User(1, "John", "a@a.pl", 24)

user.toMap
       ↓
Map(“id” -> 1, “name” -> “John”, “email” -> “a@a.pl”, “age” -> 24)
```

@Mappable

```
import scala.annotation.StaticAnnotation
import scala.meta._

class Mappable extends StaticAnnotation {
 inline def apply(defn: Any): Any = meta {
   defn match {
     case q"..$mods class $tname[..$tparams] (...$paramss) extends $template" =>
       template match {
         case template"{ ..$stats } with ..$ctorcalls { $param => ..$body }" => {
         //body}}
     case _ => throw new Exception("@Mappable can be annotation of class only")
}}}
```

@Mappable

```
case q"..$mods class $tname[..$tparams] (...$paramss) extends $template" =>
    template match {
        case template"{ ..$stats } with ..$ctorcalls { $param => ..$body }" => {
         //body
     }
}
```

@Mappable

```
val expr = paramss.flatten.map(p => q"${p.name.toString}").zip(paramss.flatten.map{
   case param"..$mods $paramname: $atpeopt = $expropt" => paramname
 }).map{case (q"$paramName", paramTree) => {
   q"${Term.Name(paramName.toString)} -> ${Term.Name(paramTree.toString)}"
}}

val resultMap = q"Map(..$expr)"

val newBody = body :+ q"""def toMap: Map[String, Any] = $resultMap"""
val newTemplate = template"{ ..$stats } with ..$ctorcalls { $param => ..$newBody }"

q"..$mods class $tname[..$tparams] (...$paramss) extends $newTemplate"
```

@Mappable

```
val expr = paramss.flatten.map(p => q"${p.name.toString}").zip(paramss.flatten.map{
   case param"..$mods $paramname: $atpeopt = $expropt" => paramname
 }).map{case (q"$paramName", paramTree) => {
   q"${Term.Name(paramName.toString)} -> ${Term.Name(paramTree.toString)}"
}}

val resultMap = q"Map(..$expr)"


val newBody = body :+ q"""def toMap: Map[String, Any] = $resultMap"""
val newTemplate = template"{ ..$stats } with ..$ctorcalls { $param => ..$newBody }"


q"..$mods class $tname[..$tparams] (...$paramss) extends $newTemplate"
```

@Mappable

```
val expr = paramss.flatten.map(p => q"${p.name.toString}").zip(paramss.flatten.map{
   case param"..$mods $paramname: $atpeopt = $expropt" => paramname
 }).map{case (q"$paramName", paramTree) => {
   q"${Term.Name(paramName.toString)} -> ${Term.Name(paramTree.toString)}"
}}

val resultMap = q"Map(..$expr)"

val newBody = body :+ q"""def toMap: Map[String, Any] = $resultMap"""
val newTemplate = template"{ ..$stats } with ..$ctorcalls { $param => ..$newBody }"

q"..$mods class $tname[..$tparams] (...$paramss) extends $newTemplate"
```

@Mappable

```
val expr = paramss.flatten.map(p => q"${p.name.toString}").zip(paramss.flatten.map{
   case param"..$mods $paramname: $atpeopt = $expropt" => paramname
 }).map{case (q"$paramName", paramTree) => {
   q"${Term.Name(paramName.toString)} -> ${Term.Name(paramTree.toString)}"
}}

val resultMap = q"Map(..$expr)"

val newBody = body :+ q"""def toMap: Map[String, Any] = $resultMap"""
val newTemplate = template"{ ..$stats } with ..$ctorcalls { $param => ..$newBody }"

q"..$mods class $tname[..$tparams] (...$paramss) extends $newTemplate"
```

@Mappable

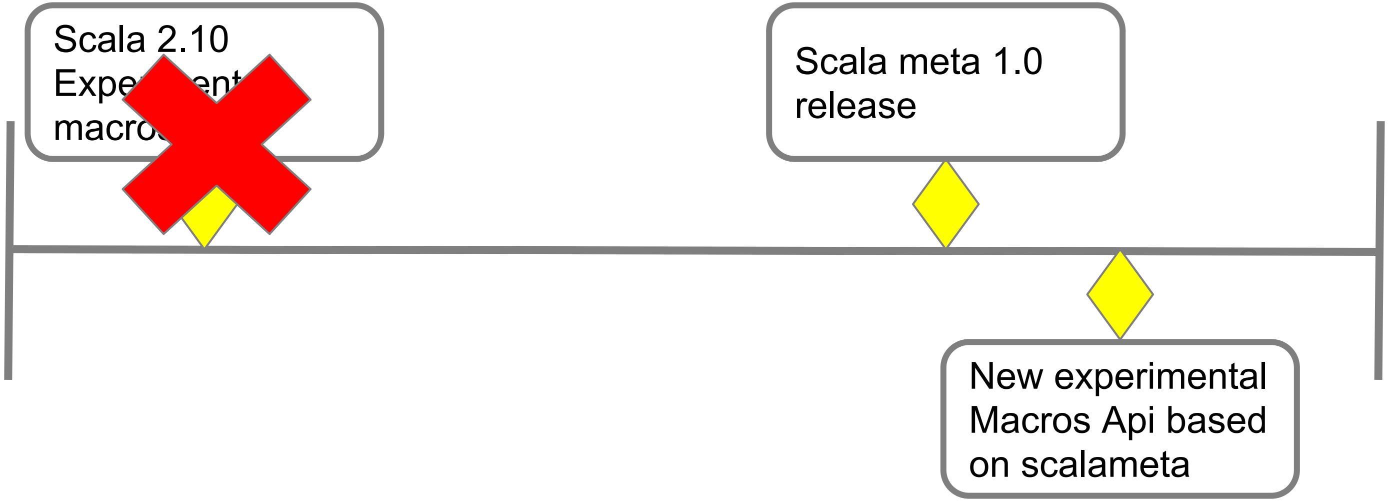
```
val expr = paramss.flatten.map(p => q"${p.name.toString}").zip(paramss.flatten.map{
   case param"..$mods $paramname: $atpeopt = $expropt" => paramname
 }).map{case (q"$paramName", paramTree) => {
   q"${Term.Name(paramName.toString)} -> ${Term.Name(paramTree.toString)}"
}}

val resultMap = q"Map(..$expr)"

val newBody = body :+ q"""def toMap: Map[String, Any] = $resultMap"""
val newTemplate = template"{ ..$stats } with ..$ctorcalls { $param => ..$newBody }"

q"..$mods class $tname[..$tparams] (...$paramss) extends $newTemplate"
```

Long story short

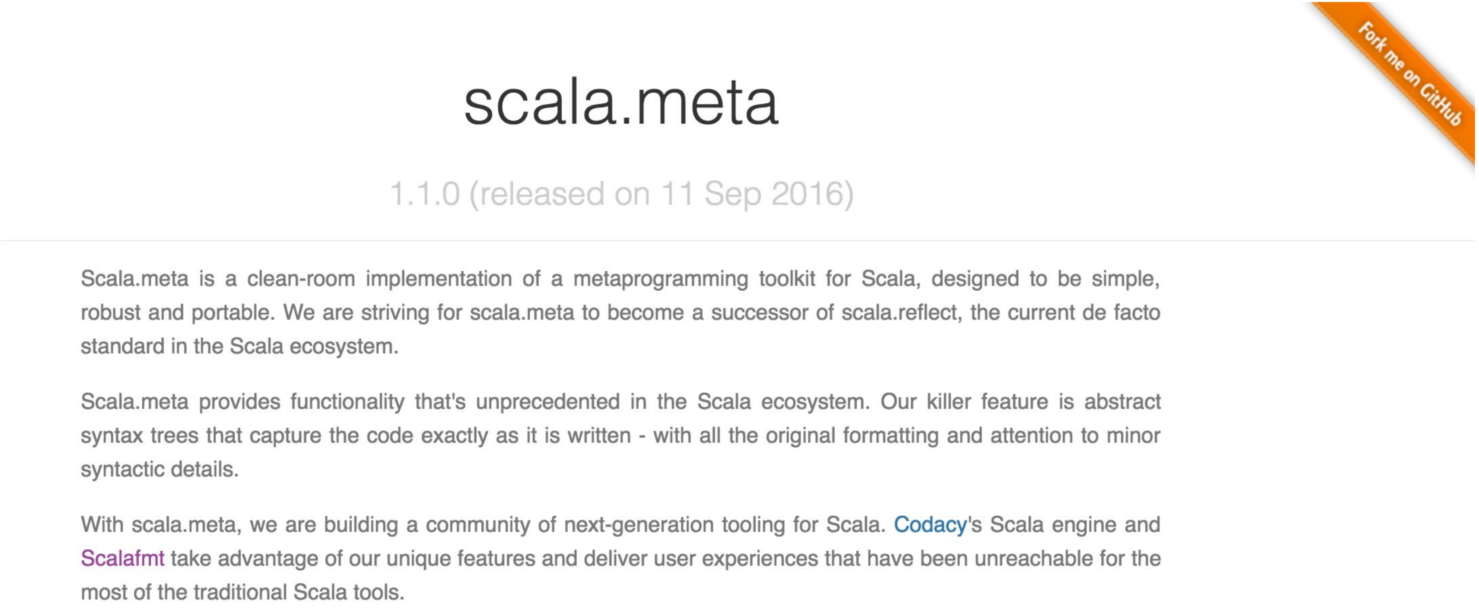


Scala 2.10
Expe ent
macro

Scala meta 1.0 release

New experimental Macros Api based on scalameta

ScalaMeta 1.0



scala.meta

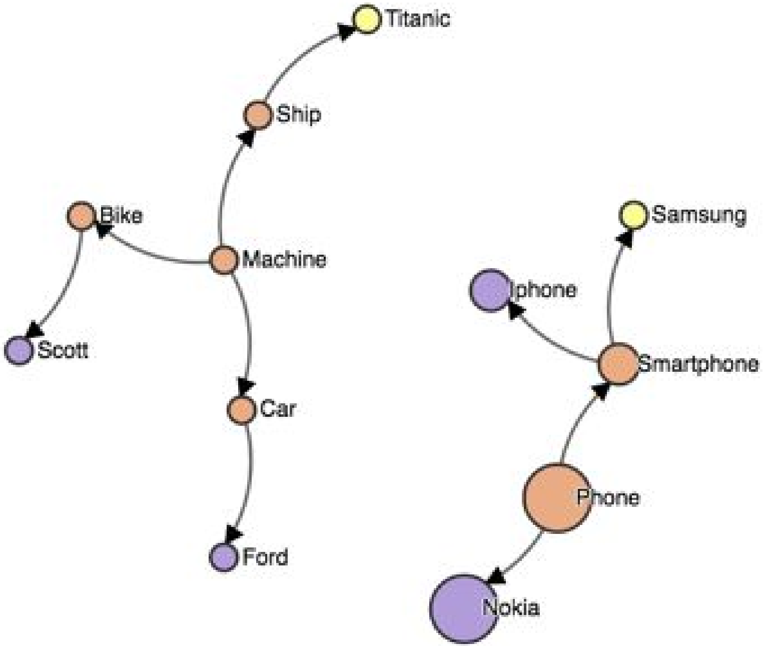
1.1.0 (released on 11 Sep 2016)

Scala.meta is a clean-room implementation of a metaprogramming toolkit for Scala, designed to be simple, robust and portable. We are striving for scala.meta to become a successor of scala.reflect, the current de facto standard in the Scala ecosystem.

Scala.meta provides functionality that's unprecedented in the Scala ecosystem. Our killer feature is abstract syntax trees that capture the code exactly as it is written - with all the original formatting and attention to minor syntactic details.

With scala.meta, we are building a community of next-generation tooling for Scala. Codacy's Scala engine and Scalafmt take advantage of our unique features and deliver user experiences that have been unreachable for the most of the traditional Scala tools.

Motivation


Titanic
Ship
Bike
Machine
Samsung
Iphone
Scott
Smartphone
Car
Phone
Ford
Nokia

RAILS

play

Scalameta 1.0

1. Lexical analysis
2. Parsing
3. Semantic analysis
4. Optimization
5. Code Generation

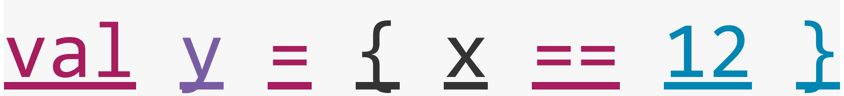
Scalameta 1.0

1. Lexical analysis
2. Parsing
3. Semantic analysis
4. Optimization
5. Code Generation

Lexical analysis

```
val y = { x == 12 }
```

Lexical analysis



```
val y = { x == 12 }
```

Lexical analysis

```scala
import scala.meta._


val code = "y = x == 12".tokenize


val t: Tokens = code match {
  case Tokenized.Success(tokens) => tokens
  case Tokenized.Error(_, _, details)=>throw new Exception(details)
}
```

Lexical analysis

```
val t: Tokens = ...
```

```
t.tokens.toList
res0: List(, val,  , y,  , =,  , x,  , ==,  , 12, )
```

```
t.structure
res1: Tokens(BOF [0..0), val [0..3), [3..4), y [4..5), [5..6), = [6..7), [7..8), x [8..9),
[9..10), == [10..12), [12..13), 12 [13..15), EOF [15..15))
```

```
println(t.syntax)
res0: val y = x == 12
```

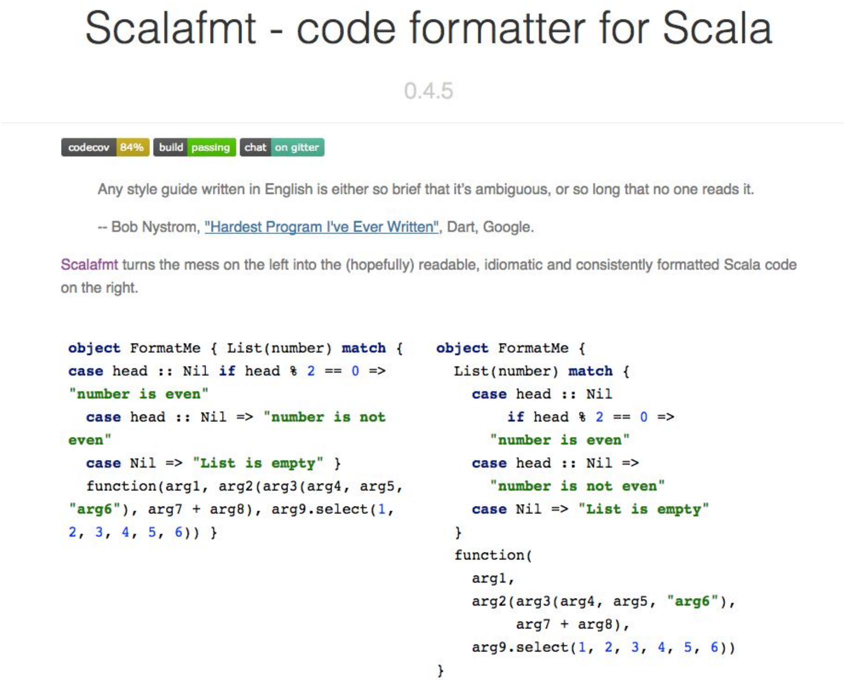
Lexical Analysis

```
sth.getOrElse(null)
sth.orNull
```

```
sth.filter(condition).headOption
find(condition)
```

```
“${saveRateSettingParam}”
“$saveRateSettingParam”
```

ScalaFmt



Scalafmt - code formatter for Scala

0.4.5

codecov 84% build passing chat on gitter

> Any style guide written in English is either so brief that it's ambiguous, or so long that no one reads it.
>
> -- Bob Nystrom, "Hardest Program I've Ever Written", Dart, Google.

Scalafmt turns the mess on the left into the (hopefully) readable, idiomatic and consistently formatted Scala code on the right.

```scala
object FormatMe { List(number) match {
case head :: Nil if head % 2 == 0 =>
"number is even"
  case head :: Nil => "number is not
even"
  case Nil => "List is empty" }
  function(arg1, arg2(arg3(arg4, arg5,
"arg6"), arg7 + arg8), arg9.select(1,
2, 3, 4, 5, 6)) }
```

```scala
object FormatMe {
  List(number) match {
    case head :: Nil
        if head % 2 == 0 =>
      "number is even"
    case head :: Nil =>
      "number is not even"
    case Nil => "List is empty"
  }
  function(
    arg1,
    arg2(arg3(arg4, arg5, "arg6"),
         arg7 + arg8),
    arg9.select(1, 2, 3, 4, 5, 6))
}
```

Scalameta 1.0

1. Lexical analysis
2. **Parsing**
3. Semantic analysis
4. Optimization
5. Code Generation

def parse[U]

```
val code = "List[String]".parse[Type]

val extracted = extract(code)

def extract[T](code: Parsed[T]): T = {
  code match {
    case Parsed.Success(tree) => tree
    case Parsed.Error(pos, msg, details)  => throw new Exception(msg)
  }
}
```

def parse[U]

```
val code     = "val a: List[String]= List()".parse[Stat]
val caseExpr = "case true => sth".parse[Case]
val term        = "x + y".parse[Term]
val arg      = "a: List[String]".parse[Term.Arg]
```

```scala
val caseExpr = "case true => sth".parse[Stat]
```

Back to quasiquotes

```
val q"..$mods val ..$patsnel: $tpeopt = $expr" =
                  "val a: List[String]= Nil".parse[Stat].get


 val q"..$mods trait $tname[..$tparams] extends $template" =
                      "trait Conf { val name: String }".parse[Stat].get
```

object Constants

```scala
object Constants {
  val java = "java"
  val scala = "scala"
  val ruby1 = "ruby"
  val ruby2 = "ruby"
}
```

object Constants

```
object Constants {
  val java = "java"
  val scala = "scala"
  val ruby1 = "ruby"
  val ruby2 = "ruby"
}
```

object Constants

```
validate(
    new java.io.File("Constants.scala").parse[Source].get
)
```

object Constants

```
def validate(source: Source): Any
```

object Constants

```
def validate(source: Source) = source match {
  case source"..$stats" => stats.collect(_ match {

  })
}
```

object Constants

```
def validate(source: Source): Any = source match {
  case source"..$stats" => stats.collect(_ match {
    case q"..$mods object ${Term.Name(name)} extends $template" if name ==
"Constants" => template match {

    }
  )
}
```

object Constants

```
  template match {
    case template"{ ..$stats2 } with ..$ctorcalls { $param => ..$stats3 }" =>{
     val vals: List[Val] = stats3.foldLeft(List[Val]()) {
      (acc, elem) => elem match {
       case q"..$mods2 val ..$patsnel: $tpeopt = $expr" => acc :+ Val(patsnel.head, expr.toString)
       case _ => acc
      }
     }
     vals.groupBy(_.valValue).foreach{ case
      (valueKey, listOfVals) => if (listOfVals.length > 1 ) throw new Exception(s"$valueKey is assigned
more than once to different vals: ${listOfVals.map(_.valName)}")
     }
    }
   }
```

Example no.2- Code metrics

```
trait Phone{
 def makeCall = "calling"
 def writeMsg = "msg"
 def charge   = "charge"
}
case class Nokia() extends Phone{
 val contacts = List()
 val dateOfProduction = 2005
 def takePhoto = "photo"
}
trait Smartphone extends Phone {
 def openBrowser = "opening"
}
```

```
case class Iphone() extends Smartphone {
 def removeJack = {
   "removing"
 }
}
object Samsung extends Smartphone
```

Example no.2- Code metrics

```scala
case class Counts(classNo: Int, objectNo: Int, traitNo: Int, packageObjNo: Int) {
  ...
}

object Counts {
  val initial = Counts(0, 0, 0 ,0)
}
```

```
object CodeMetrics {
  val counts = allScalaFiles.foldLeft(Counts.initial)((acc, file) => {
    file match {
      case source"..$whateverItIsInFile" =>
whateverItIsInFile.foldLeft(acc)((accInFile: Counts, elem) => elem match {
        //Increment statistics
      })
    }
  })
}
```

```
        case q"..$mods object $name extends $template" =>
          accInFile.incObjectNo
        case q"..$mods class $tname[..$tparams] (...$paramss) extends $template" =>
          accInFile.incClassNo
        case q"..$mods trait $tname[..$tparams] extends $template" =>
          accInFile.incTraitNo
        case q"package object $name extends $template" =>
          accInFile.incPackageObjNo
        case _ => accInFile
```

```scala
object Main extends App {
  implicit val system = ActorSystem("my-actor-system")
  implicit val materializer = ActorMaterializer()
  implicit val executionContext = system.dispatcher

  MyApplicationBoot.run
}

object MyApplicationBoot {

// XYZ WILL KILL YOU IF YOU WILL START THOSE IMPLICITS HERE
  def run(implicit actorSystem: ActorSystem, actorMaterializer: ActorMaterializer,
executionContext: ExecutionContext) = {
    //run application
  }
}
```

Thank you for listening

Links:

- http://scalameta.org/
- https://goo.gl/3ayKmA
- https://goo.gl/2FVSCf
- https://goo.gl/lncq2N
- https://goo.gl/Y5WRbo
- https://goo.gl/SNxO2L
- www.bbartosz.com