State of the Meta, Spring 2015

Eugene Burmako (@xeno_by)

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

17 March 2015

scala.meta

The goal of scala.meta is to make metaprogramming easy

— scalameta.org

scala.meta

> *The goal of scala.meta is to make metaprogramming easy, ensuring that it:*
> *1) Doesn't require knowledge of compiler internals*
>
> — scalameta.org

scala.meta

The goal of scala.meta is to make metaprogramming easy, ensuring that it:
1) Doesn't require knowledge of compiler internals
2) Is safe from compiler crashes by construction

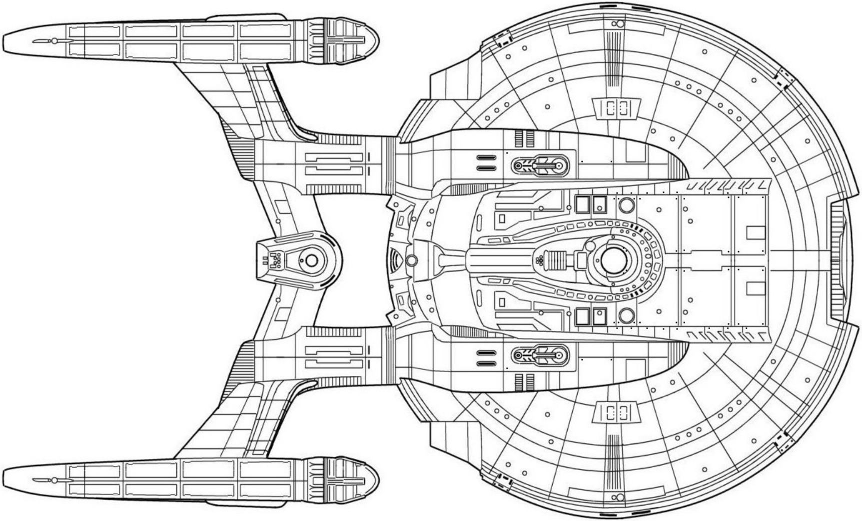
— scalameta.org

scala.meta

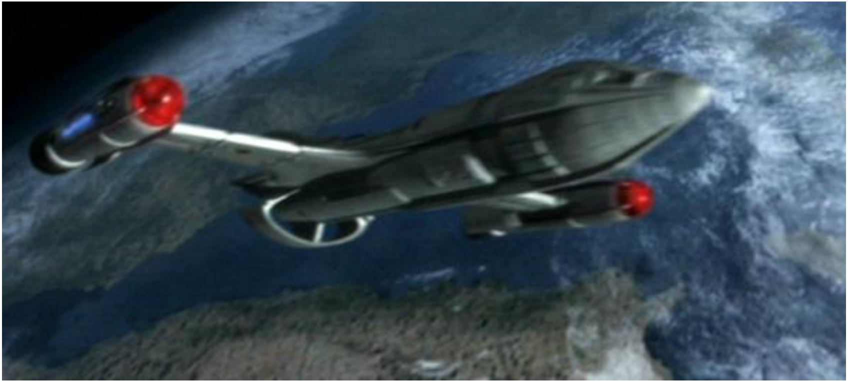
The goal of scala.meta is to make metaprogramming easy, ensuring that it:
1) Doesn't require knowledge of compiler internals
2) Is safe from compiler crashes by construction
3) Is portable across a wide range of implementors

— scalameta.org

ScalaDays Berlin



ScalaDays San Francisco



Outline

- The next-generation metaprogramming ecosystem
- Quick introduction into scala.meta
- Key concepts of the syntactic API
- Key concepts of the semantic API

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
- Alexander Podkhalyuzin
- Jatin Puri
- Dmitry Petrashko
- Denys Shabalin

Part 1: The next-generation metaprogramming ecosystem

High-level view

- scala.reflect has known issues, but it works and is well-understood
- What does scala.meta have to offer, concretely?
- Here's our proposed design of the future metaprogramming ecosystem
- There's work being done towards implementing it as we speak

Architecture in a nutshell

- Library of platform-independent metaprogramming APIs
  `https://github.com/scalameta/scalameta`

Architecture in a nutshell

- Library of platform-independent metaprogramming APIs
  `https://github.com/scalameta/scalameta`

- Collection of platform-dependent implementations
  `https://github.com/scalameta/scalahost`,
  ... (intellij, dotty, etc)

Architecture in a nutshell

- Ecosystem of platform-independent metaprograms
  macros, style checkers, code formatters, refactorings,
  ... (other tools, utilities, libraries)

- Library of platform-independent metaprogramming APIs
  `https://github.com/scalameta/scalameta`

- Collection of platform-dependent implementations
  `https://github.com/scalameta/scalahost`,
  ... (intellij, dotty, etc)

Architecture in more detail

scala.meta

TASTY binaries

The frontend/backend separation

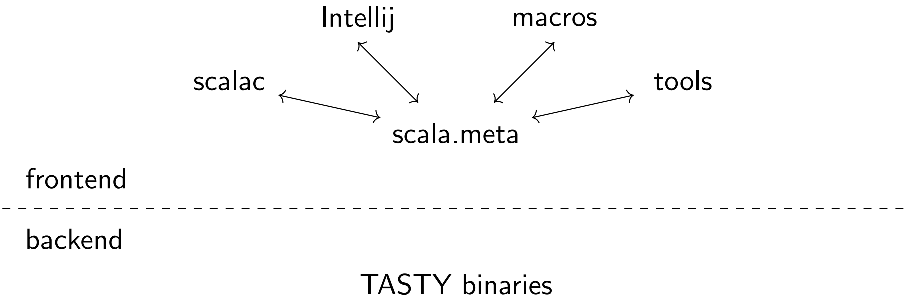
scala.meta

frontend

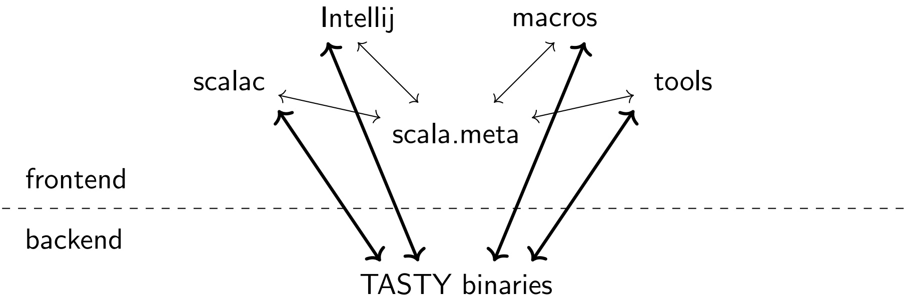
backend

TASTY binaries

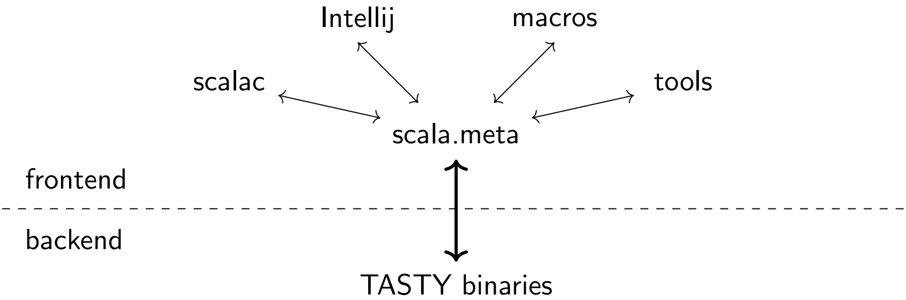
Notable frontend metaprograms



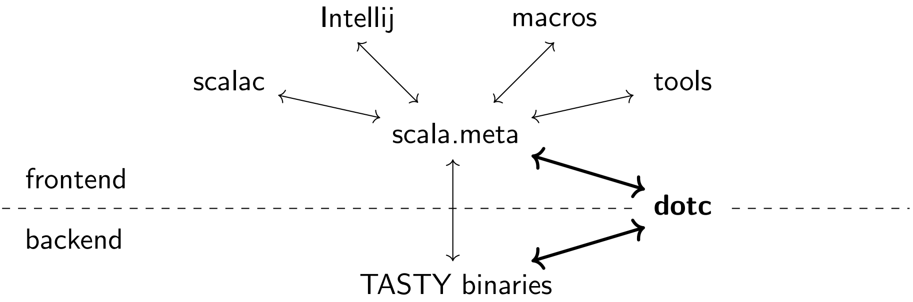
How do metaprograms read TASTY? (Design #1)



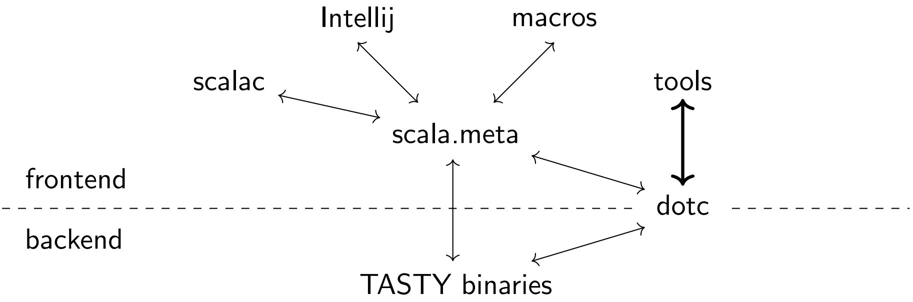
How do metaprograms read TASTY? (Design #2)



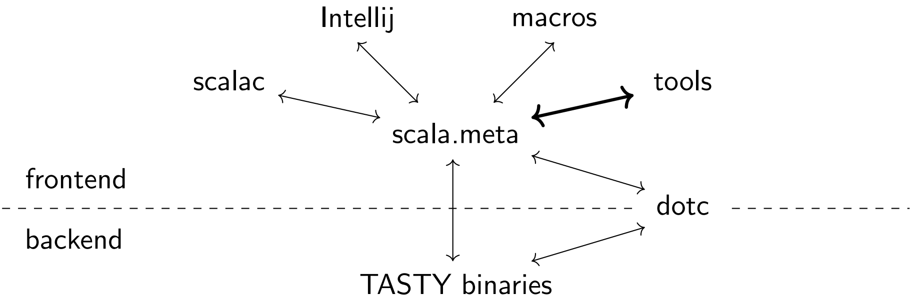
What about Dotty?



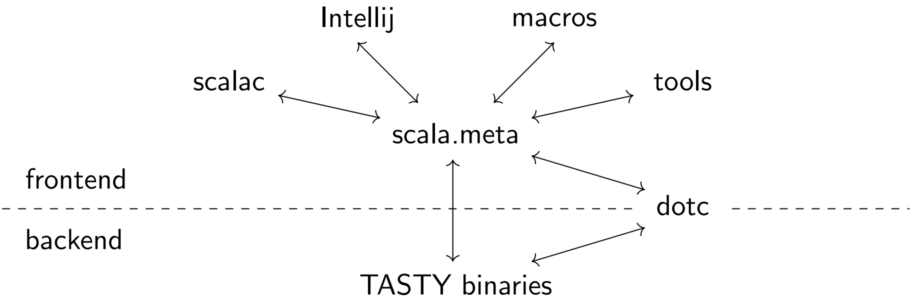
Revisiting tools (Design #1)



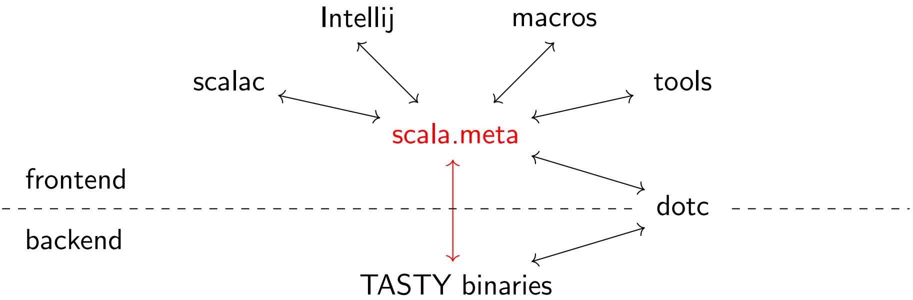
Revisiting tools (Design #2)



Summary

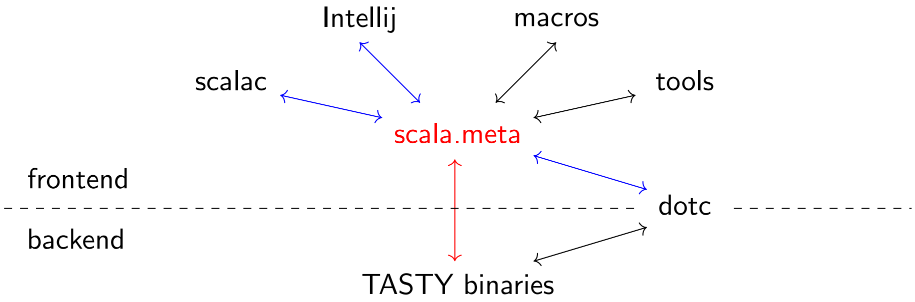

Intellij
macros
scalac
tools
scala.meta
frontend
dotc
backend
TASTY binaries

Summary



⟷ scalameta/scalameta (platform-independent)

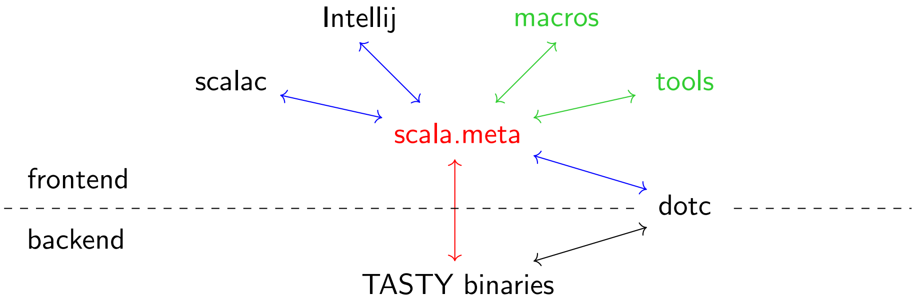
Summary



⟷ scalameta/scalameta (platform-independent)

⟷ organization/platformhost (platform-dependent)

Summary



⟷ scalameta/scalameta (platform-independent)
⟷ organization/platformhost (platform-dependent)
⟷ your/project (platform-independent)

Part 2: Quick introduction into scala.meta

Getting all definitions in a source file

Getting all definitions in a source file

```
import scala.tools.meta.Metaprogram
import scala.tools.meta.Settings


class Analysis(val settings: Settings) extends Metaprogram {







}

val metaprogram = new Analysis(new Settings())
```

Getting all definitions in a source file

```
import scala.tools.meta.Metaprogram
import scala.tools.meta.Settings
import scala.meta.internal.util.SourceFile

class Analysis(val settings: Settings) extends Metaprogram {







}

val metaprogram = new Analysis(new Settings())
val sourceFile = new SourceFile(new java.io.File(args(1)))
```

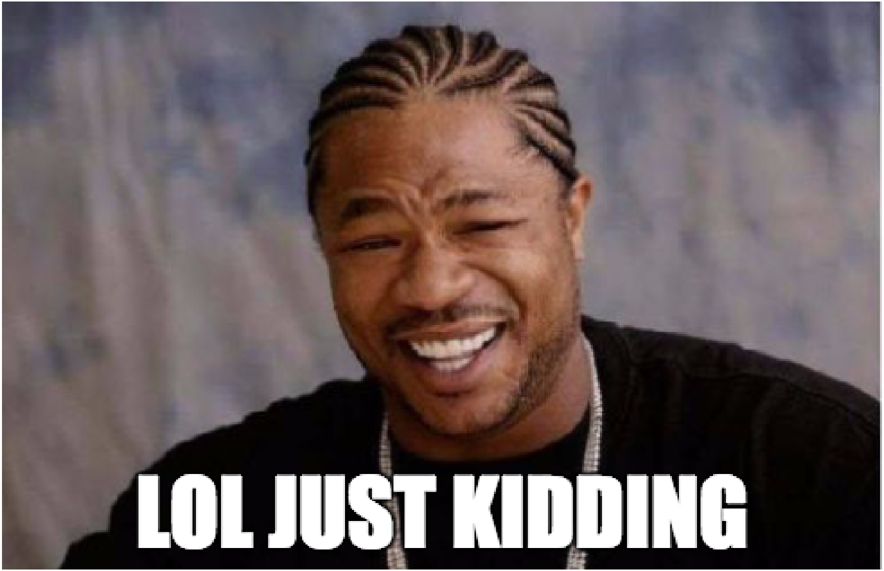
Getting all definitions in a source file

```
import scala.tools.meta.Metaprogram
import scala.tools.meta.Settings
import scala.meta.internal.util.SourceFile

class Analysis(val settings: Settings) extends Metaprogram {
  def apply(sourceFile: SourceFile) = {
    import multiverse._
    val unit = newCompilationUnit(sourceFile)
    val parser = newUnitParser(unit)

  }
}

val metaprogram = new Analysis(new Settings())
val sourceFile = new SourceFile(new java.io.File(args(1)))
metaprogram.apply(sourceFile)
```


LOL JUST KIDDING

On a serious note

- Just `import scala.meta._`
- +0-2 additional lines of code depending on the context

Syntactic APIs

```
import scala.meta._
import scala.meta.dialects.Scala211
```

- Tokenization
- Parsing
- Positions
- Quasiquotes

Semantic APIs

```
import scala.meta._
implicit val c: scala.meta.semantic.Context = ...
```

- Name resolution
- Typechecking
- Members
- Subtyping
- ...

What's a context?

```
trait Context {
  def dialect: Dialect

  def tpe(term: Term): Type
  def tpe(param: Term.Param): Type.Arg
  def defns(ref: Ref): Seq[Member]
  def members(tpe: Type): Seq[Member]

  def isSubType(tpe1: Type, tpe2: Type): Boolean
  def lub(tpes: Seq[Type]): Type
  ... // 7 more methods
}
```

Where do contexts come from?

- Scalac: https://github.com/scalameta/scalahost
- Intellij: in the works
- Dotty: planned
- Anywhere else: anyone can implement a context

Part 3: Key concepts of the 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 (ScalaDays Berlin)

- 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
res0: Vector[meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), (7..7), { (8..8), (9..9),
def (10..12), (13..13), x (14..14), (15..15), = (16..16),
(17..17), 2 (18..18), (19..19), } (20..20), EOF (21..20))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res0: Vector[meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), (7..7), { (8..8), (9..9),
def (10..12), (13..13), x (14..14), (15..15), = (16..16),
(17..17), 2 (18..18), (19..19), } (20..20), EOF (21..20))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res0: Vector[meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), (7..7), { (8..8), (9..9),
def (10..12), (13..13), x (14..14), (15..15), = (16..16),
(17..17), 2 (18..18), (19..19), } (20..20), EOF (21..20))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res0: Vector[meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), (7..7), { (8..8), (9..9),
def (10..12), (13..13), x (14..14), (15..15), = (16..16),
(17..17), 2 (18..18), (19..19), } (20..20), EOF (21..20))
```

Tokens

```
scala> "class C { def x = 2 }".tokens
res0: Vector[meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), (7..7), { (8..8), (9..9),
def (10..12), (13..13), x (14..14), (15..15), = (16..16),
(17..17), 2 (18..18), (19..19), } (20..20), EOF (21..20))
```

High-fidelity parsers

```
scala> "class C".parse[Stat]
res1: scala.meta.Stat = class C

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

High-fidelity parsers

```
scala> "class C".parse[Stat]
res1: scala.meta.Stat = class C

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

scala> res1.tokens
res3: Seq[scala.meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6), EOF(7..6))

scala> res2.tokens
res4: Seq[scala.meta.Token] = Vector(BOF (0..-1),
class (0..4), (5..5), C (6..6),
(7..7), { (8..8), } (9..9), EOF (10..9))
```

Automatic and precise range positions

```
scala> "class C { def x = 2 }".parse[Stat]
res5: scala.meta.Stat = class C

scala> val q"class C { $method }" = res5
method: scala.meta.Stat = def x = 2
```

Automatic and precise range positions

```
scala> "class C { def x = 2 }".parse[Stat]
res5: scala.meta.Stat = class C

scala> val q"class C { $method }" = res5
method: scala.meta.Stat = def x = 2

scala> method.tokens
res6: Seq[scala.meta.Token] = Vector(
def (10..12), (13..13), x (14..14), (15..15),
= (16..16), (17..17), 2 (18..18))
```

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"
...
tokens: Vector(BOF (0..-1), class (0..4), (5..5),
$name (6..10), EOF (11..10))
...
```

Derived technologies

First-class tokens enable:

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

Part 4: Key concepts of the semantic API

Getting started

```
$ scala

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

scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...
```

Getting started

```
$ scala

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

scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...
```

Contexts can come from:

- Scalahost.mkStandaloneContext("-cp ..."): useful for experimentation

Getting started

```
$ scala

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

scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...
```

Contexts can come from:

- Scalahost.mkStandaloneContext("-cp ..."): useful for experimentation
- Scalahost.mkGlobalContext(global): useful for compiler plugins

Getting started

```
$ scala

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

scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...
```

Contexts can come from:

- Scalahost.mkStandaloneContext("-cp ..."): useful for experimentation
- Scalahost.mkGlobalContext(global): useful for compiler plugins
- Implicit scope when writing macros: very convenient!

Getting started

```
$ scala

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

scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...
```

Contexts can come from:

- Scalahost.mkStandaloneContext("-cp ..."): useful for experimentation
- Scalahost.mkGlobalContext(global): useful for compiler plugins
- Implicit scope when writing macros: very convenient!
- Anywhere else: anyone can implement a context

Design goals (ScalaDays Berlin)

- 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[Raw]
res0: 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")))))))
```

Cute trees

Scary trees

Key example

`List[Int]`

Key example

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

Key example

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

scala> t"List[Int]".show[Semantics]
res2: String =
Type.Apply(Type.Name("List")[1], List(Type.Name("Int")[2]))
[1] Type.Singleton(Term.Name("package")[4])::scala.package#List
[2] Type.Singleton(Term.Name("scala")[3])::scala#Int
...
```

Name resolution

```
scala> implicit val c = Scalahost.mkStandaloneContext()
c: scala.meta.macros.Context = ...

scala> q"scala.collection.immutable.List".defn
res3: scala.meta.Member.Term = object List extends
SeqFactory[List] with Serializable { ... }

scala> res3.name
res4: scala.meta.Term.Name = List
```

Other semantic APIs

```
scala> q"scala.collection.immutable.List".defs("apply")
res5: scala.meta.Member.Term =
override def apply[A](xs: A*): List[A] = ???

scala> q"scala.collection.immutable.List".parents
res6: 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

Wrapping up

Summary

- scala.meta is a one-stop solution to frontend metaprogramming
- Our key innovations include first-class support for tokens and names
- This gives rise to a powerful, yet simple model of Scala programs
- We are working hard to provide an alpha release soon

Summary

- scala.meta is a one-stop solution to frontend metaprogramming
- Our key innovations include first-class support for tokens and names
- This gives rise to a powerful, yet simple model of Scala programs
- We are working hard to provide an alpha release soon

Upcoming presentations:
Live demo at a Scala Bay meetup in Mountain View (19 March)
Hands-on workshop at flatMap in Oslo (27-28 April)
Next status update at ScalaDays in Amsterdam (8-10 June)