Metaprogramming in Scala 2.10

Eugene Burmako

EPFL, LAMP

28 April 2012
(updated on 6 January 2013 for 2.10.0-final)

Agenda

Metaprogramming

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

—Wikipedia

Compiler

Q: How to enable metaprogramming?

A: Who has more data about a program than a compiler?

Let's expose the compiler to the programmer.

Reflection

In 2.10 we expose data about programs via reflection API.

The API is spread between scala-reflect.jar (interfaces and implementations) and scala-compiler.jar (runtime compilation).

Hands on

Today we will learn the fundamentals of reflection API and learn to learn more about it via a series of hands-on examples.

Macros

Q: Hey! What about macros?

A: Reflection is at the core of macros, reflection provides macros with an API, reflection enables macros. Our focus today is understanding reflection, macros are just a tiny bolt-on.

For more information about macros, their philosophy and applications, take a look at my other talks: http://scalamacros.org/talks.html.

Agenda

Intro

Reflection

Universes

Demo

Macros

Summary

Core data structures

- Trees
- Symbols
- Types

```
C:\Projects\Kepler>scalac -Xshow-phases
phase name id  description
---------- --  -----------
    parser  1  parse source into ASTs, simple desugaring
     namer  2  resolve names, attach symbols to named trees
     typer  4  the meat and potatoes: type the trees
   pickler  8  serialize symbol tables
```

I'll do my best to explain these concepts, but it's barely possible to do it better than Paul Phillips. Be absolutely sure to watch the Inside the Sausage Factory talk.

Trees

Short-lived, mostly immutable, mostly plain case classes.

```
Apply(Ident("println"), List(Literal(Constant("hi!"))))
```

A list of all trees can be found in docs and in sources.

Learn to learn

- -Xprint:parser (for naked trees)
- -Xprint:typer (for typechecked trees)
- -Yshow-trees and its cousins
- ru.showRaw(ru.reify(...)) // where ru stands for scala.reflect.runtime.universe
- also check out the optional parameters of showRaw!

Q: Where do I pull these compiler flags from?

A: scala/tools/nsc/settings/ScalaSettings.scala

-Yshow-trees

```
// also try -Yshow-trees-stringified
// and -Yshow-trees-compact (or both simultaneously)
>scalac -Xprint:parser -Yshow-trees HelloWorld.scala
[[syntax trees at end of parser]]// Scala source:
    HelloWorld.scala
PackageDef(
  "<empty>"
  ModuleDef(
    0
    "Test"
    Template(
      "App" // parents
      ValDef(
        private
        "_"
        <tpt>
        <empty>
      )
      ...
```

showRaw

```
// ru stands for scala.reflect.runtime.universe
scala> ru.reify{ object Test { println("Hello World!") } }
res0: reflect.runtime.universe.Expr[Unit] = ...

scala> ru.showRaw(res0.tree)
res1: String = Block(List(ModuleDef(
  Modifiers(),
  newTermName("Test"),
  Template(List(Ident(newTypeName("AnyRef"))), List(
    DefDef(Modifiers(), nme.CONSTRUCTOR, List(),
        List(List()), TypeTree(),
        Block(List(Apply(Select(Super(This(tpnme.EMPTY),
        tpnme.EMPTY), nme.CONSTRUCTOR), List())),
        Literal(Constant(())))),
    Apply(Select(Select(This(newTypeName("scala")),
        newTermName("Predef")), newTermName("println")),
        List(Literal(Constant("Hello World!")))))))),
Literal(Constant(())))
```

Symbols

Link definitions and references to definitions. Long-lived, mutable. Declared in scala/reflect/api/Symbols.scala, documented somewhere nearby.

```
def foo[T: TypeTag](x: Any) = x.asInstanceOf[T]
foo[Long](42)
```

foo, T, x introduce symbols (T actually produces two different symbols, but that's a different story). DefDef, TypeDef, ValDef - all of those subtype DefTree.

TypeTag, x, T, foo, Long refer to symbols. They are all represented by Idents, which subtype RefTree.

Symbols are long-lived. This means that any reference to Int (from a tree or from a type) will point to the same symbol instance.

Learn to learn

- -Xprint:namer or -Xprint:typer
- -uniqid
- symbol.kind and -Yshow-symkinds
- :type -v
- showRaw(tree, printIds = true, printKinds = true)
- Don't create them by yourself. Just don't, leave it to Namer. In macros most of the time you create naked trees, and Typer will take care of the rest. Sometimes it inevitable, though: http://stackoverflow.com/questions/11208790.

-uniqid and -Yshow-symkinds

```
>cat Foo.scala
def foo[T: TypeTag](x: Any) = x.asInstanceOf[T]
foo[Long](42)

// there is a mysterious factoid hidden in this printout!
>scalac -Xprint:typer -uniqid -Yshow-symkinds Foo.scala
[[syntax trees at end of typer]]// Scala source: Foo.scala
def foo#8339#METH
  [T#8340#TPE >: Nothing#4658#CLS <: Any#4657#CLS]
  (x#9529#VAL: Any#4657#CLS)
  (implicit evidence$1#9530#VAL:
      TypeTag#7861#TPE[T#8341#TPE#SKO])
  : T#8340#TPE =
  x#9529#VAL.asInstanceOf#6023#METH[T#8341#TPE#SKO];

Test#14#MODC.this.foo#8339#METH[Long#1641#CLS](42)
(scala#29#PK.reflect#2514#PK.`package`#3414#PKO
.mirror#3463#GET.TypeTag#10351#MOD.Long#10361#GET)
```

:type -v

We have just seen how to discover symbols used in trees.

However, symbols are also used in types.

Thanks to Paul (who hacked this during one of Scala Nights) there's an easy way to inspect types as well. Corresponding REPL incantation is shown on one of the next slides.

Starting from 2.10.0-M5 you can also use showRaw (defined in all universes: the scala.reflect.runtime.universe, all macro context universes) to print out raw structure of types.

Types

Immutable, long-lived, sometimes cached case classes declared in scala/reflect/api/Types.scala (also see docs).

Store the information about the full wealth of the Scala type system: members, type arguments, higher kinds, path dependencies, erasures, etc.

Learn to learn

- -Xprint:typer
- -Xprint-types
- :type -v
- showRaw(type, printIds = true, printKinds = true)
- -explaintypes

-Xprint-types

-Xprint-types is yet another option that modifies tree printing.
Nothing very fancy, let's move on to something really cool.

:type -v

```
scala> :type -v def impl[T: c.TypeTag](c: Context) = ???
// Type signature
[T](c: scala.reflect.macros.Context)(implicit evidence$1:
    c.TypeTag[T])Nothing

// Internal Type structure
PolyType(
  typeParams = List(TypeParam(T))
  resultType = MethodType(
    params = List(TermSymbol(c: ...))
    resultType = MethodType(
      params = List(TermSymbol(implicit evidence$1: ...))
      resultType = TypeRef(
        TypeSymbol(final abstract class Nothing)
      )
    )
  )
)
```

showRaw (available since 2.10.0-M5)

```
scala> object O {
  def impl[T: c.TypeTag](c: Context) = ???
}
defined module O

scala> val meth = ru.reify(O).staticType.typeSymbol.
  typeSignature.member(newTermName("impl"))
meth: reflect.runtime.universe.Symbol = method impl

scala> println(showRaw(meth.typeSignature))
PolyType(
  List(newTypeName("T")),
  MethodType(List(newTermName("c")),
    MethodType(List(newTermName("evidence$1")),
      TypeRef(ThisType(scala), scala.Nothing, List()))))
```

-explaintypes

```
>cat Test.scala
class Foo { class Bar; def bar(x: Bar) = ??? }

object Test extends App {
  val foo1 = new Foo
  val foo2 = new Foo
  foo2.bar(new foo1.Bar)
}

// prints explanations of type mismatches
>scalac -explaintypes Test.scala
Test.foo1.Bar <: Test.foo2.Bar?
  Test.foo1.type <: Test.foo2.type?
    Test.foo1.type = Test.foo2.type?
    false
  false
false
Test.scala:6: error: type mismatch;
...
```

Big picture

- Trees are created naked by Parser.
- Both definitions and references (expressed as ASTs) get their symbols filled in by Namer (tree.symbol).
- When creating symbols, Namer also creates their completers, lazy thunks that know how to populate symbol types (symbol.info).
- Typer inspects trees, uses their symbols to transform trees and assign types to them (tree.tpe).
- Shortly afterwards Pickler kicks in and serializes reachable symbols along with their types into ScalaSignature annotations.

Agenda

- Intro
- Reflection
- Universes
- Demo
- Macros
- Summary

Universes

Universes are environments that pack together trees, symbols and their types.

- Compiler (scala.tools.nsc.Global) is a universe.
- Reflection runtime (scala.reflect.runtime.universe) is a universe too.
- Macro context (scala.reflect.macros.Context) holds a reference to a universe.

Mirrors

Mirrors abstract population of symbol tables.

Each universe can have multiple mirrors, which can share symbols with each other within their parent universe.

- Compiler loads symbols from pickles using its own *.class parser. It has only one mirror, the rootMirror.
- Reflective mirror uses Java reflection to load and parse ScalaSignatures. Every classloader corresponds to its own mirror created with ru.runtimeMirror(classloader).
- Macro context refers to the compiler's symbol table.

Entry points

Using a universe depends on your scenario.

- You can play with compiler's universe (aka global) in REPL's :power mode.
- With runtime reflection you typically go through the Mirror interface, e.g. scala.reflect.runtime.currentMirror, then cm.reflect and then you can get/set fields, invoke methods, etc. Read up more in our docs.
- In a macro context, you import c.universe._ and can use imported factories to create trees and types (don't create symbols manually, remember?).

Path dependency

An important quirk is that all universe artifacts are path-dependent on their universe. Note the reflect.runtime.universe prefix in the type of the result printed below.

```
scala> ru.reify(2.toString)
res0: reflect.runtime.universe.Expr[String] =
    Expr[String](2.toString())
```

When you deal with runtime reflection, you simply import scala.reflect.runtime.universe._, and enjoy, because typically there is only one runtime universe.

However with macros it's more complicated. To pass artifacts around (e.g. into helper functions), you need to also carry the universe with you. Or you can employ the technique outlined in our docs.

Thread safety

Unfortunately, in its current state released in Scala 2.10.0, reflection is not thread safe.

Check out the documentation at http://docs.scala-lang.org/overviews/reflection/thread-safety.html for a detailed explanation.

Agenda

- Intro
- Reflection
- Universes
- Demo
- Macros
- Summary

Inspect members

```
scala> import scala.reflect.runtime.{universe => ru}
import scala.reflect.runtime.{universe=>ru}

scala> trait X { def foo: String }
defined trait X

scala> ru.typeOf[X]
res0: reflect.runtime.universe.Type = X

scala> res0.members
res1: reflect.runtime.universe.MemberScope = Scopes(
    method $asInstanceOf, method $isInstanceOf, method
    synchronized, method ##, method !=, method ==, method
    ne, method eq, constructor Object, method notifyAll,
    method notify, method clone, method getClass, method
    hashCode, method toString, method equals, method
    wait, method wait, method wait, method finalize,
    method asInstanceOf, method isInstanceOf, method !=,
    method ==, method foo)
```

Analyze and invoke members

This thing is quite involved for a single slide.
Check out our reflection guide: analysis, invocation.

Defeat erasure

```
scala> def foo[T](x: T) = x.getClass
foo: [T](x: T)Class[_ <: T]

scala> foo(List(1, 2, 3))
res0: Class[_ <: List[Int]] = class
    scala.collection.immutable.$colon$colon

scala> def foo[T: ru.TypeTag](x: T) = ru.typeOf[T]
foo: [T](x: T)(implicit evidence$1: ru.TypeTag[T])ru.Type

scala> foo(List(1, 2, 3))
res1: reflect.runtime.universe.Type = List[Int]

scala> ru.showRaw(res1)
res2: String =
    TypeRef(ThisType(scala.collection.immutable),
    scala.collection.immutable.List,
    List(TypeRef(ThisType(scala), scala.Int, List())))
```

Compile at runtime

```scala
import scala.reflect.runtime.universe._
import scala.tools.reflect.ToolBox
val tree = Apply(Select(Literal(Constant(40)),
    newTermName("$plus")), List(Literal(Constant(2))))
val cm = ru.runtimeMirror(getClass.getClassLoader)
println(cm.mkToolBox().eval(tree))
```

Toolbox is a full-fledged compiler (the scala.tools.reflect.ToolBox import requires scala-compiler.jar on the classpath). Unlike the regular compiler, it uses Java reflection encapsulated in the provided mirror to populate its symbol table.

Toolbox wraps the input AST, sets its phase to Namer (skipping Parser) and performs the compilation into an in-memory directory.

After the compilation is finished, toolbox fires up a classloader that loads and lauches the code.

Agenda

Intro

Reflection

Universes

Demo

Macros

Summary

Macros

In our hackings above, we used the runtime universe (scala.reflect.runtime.universe) to reflect against program structure.

We can do absolutely the same during the compile time. The universe is already there (the compiler itself), the API is there as well (scala.reflect.api.Universe inside the macro context).

We only need to ask the compiler to call ourselves during the compilation (currently, our trigger is macro application and the hook is the macro keyword).

The end.

No, really

That's it.

Agenda

Summary

- In 2.10 you can have all the information about your program that the compiler has (well, almost).
- This information includes trees, symbols and types. And annotations. And positions. And more.
- You can reflect at runtime (scala.reflect.runtime.universe) or at compile-time (macros).

Thanks!

eugene.burmako@epfl.ch