Metaprogramming in Scala 2.10

Eugene Burmako

EPFL, LAMP

28 April 2012

Agenda

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

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-library.jar (interfaces), scala-reflect.jar (implementations) and scala-compiler.jar (runtime compilation).

Let Martin speak

Design of the reflection API is explained in great detail in a recent talk by Martin Odersky:

channel9.msdn.com/Lang-NEXT-2012/Reflection-and-Compilers

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 Scala Days talk:
http://scalamacros.org/talks/2012-04-18-ScalaDays2012.pdf.

Agenda

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  7 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 (when the video is up).

Trees

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

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

Comprehensive list of public trees can be found here:
scala/reflect/api/Trees.scala (be sure to take a look at the "standard pattern match" section in the end of the file).

Learn to learn

- -Xprint:parser
- -Yshow-trees and its cousins
- scala.reflect.mirror.showRaw(scala.reflect.mirror.reify(...))

Q: Where do I pull these compiler flags from?

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

Symbols

Link definitions and references to definitions. Long-lived, mutable. Declared in scala/reflect/api/Symbols.scala (very much in flux!)

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

foo, T, x introduce symbols (T actually produces two different symbols). 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
- -Yshow-trees and its cousins
- -Yshow-syms and much more
- :type -v (see next slides)
- -uniqid
- kind and -Yshow-symkinds
- Don't create them by yourself. Just don't, leave it to Namer.

Types

Immutable, long-lived, sometimes cached case classes declared in scala/reflect/api/Types.scala.

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
- -explaintypes
- :type -v
- asSeenFrom

Learn to learn

```
scala> :type -v def impl[T: c.TypeTag](c: Context) = ???
// Type signature
[T](c: scala.reflect.makro.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)
      )
    )
  )
)
```

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

Universes

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

- Compiler (scala.tools.nsc.Global) is a universe.
- Reflective mirror (scala.reflect.mirror) is a universe too.
- Macro context (scala.reflect.makro.Context) holds a reference to a universe.

Symbol tables

Universes abstract population of symbol tables.

- Compiler loads symbols from pickles using its own *.class parser.
- Reflective mirror uses Java reflection to load and parse ScalaSignatures.
- 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 mirrors you go through the Mirror interface, e.g. symbolOfInstance or classToType.
- In a macro context, you import c.mirror._ 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.

```
def reify[T](expr: T): Expr[T] = macro Universe.reify[T]
def reify[T]
  (cc: Context{ type PrefixType = Universe })
  (expr: cc.Expr[T])
  : cc.Expr[cc.prefix.value.Expr[T]] = ???
```

Hence to pass artifacts around outside the cake, you need to also carry the universe with you. Or create your own cake using the technique outlined in scala-internals discussion.

Agenda

Reflection

```
scala> import scala.reflect.mirror
import scala.reflect.mirror

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

scala> typeTag[X]
res1: TypeTag[X] = ConcreteTypeTag[X]

scala> res1.tpe.members
res2: Iterable[reflect.mirror.Symbol] = List(method
    $asInstanceOf, method $isInstanceOf, method sync
hronized, method ##, method !=, method ==, method ne,
    method eq, constructor Object, method notifyAl
l, method notify, method clone, method getClass, method
    hashCode, method toString, method equals, me
thod wait, method wait, method wait, method finalize,
    method asInstanceOf, method isInstanceOf, meth
od !=, method ==, method foo)
```

Dynamic

```
val d = new DynamicProxy{ val dynamicProxyTarget = x }
d.noargs
d.noargs()
d.nullary
d.value
d.-
d.$("x")
d.overloaded("overloaded with object")
d.overloaded(1)
val car = new Car
d.typeArgs(car) // inferred
d.default(1, 3)
d.default(1)
d.named(1, c=3, b=2) // works, wow
```

More examples at test/files/run/dynamic-proxy.scala.
Implementation is at scala/reflect/DynamicProxy.scala.

Runtime compilation

```
import scala.reflect.mirror._
val tree = Apply(Select(Ident("Macros"),
    newTermName("foo")), List(Literal(Constant(42))))
println(Expr(tree).eval)
```

Eval creates a toolbox under the covers (universe.mkToolBox), which is a full-fledged compiler.

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

Macros

In our hackings above, we used the runtime mirror (scala.reflect.mirror) 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.mirror) or at compile-time (macros).

Status

We're already there. Nightlies already feature reflection and macros. In a few days we will release 2.10.0 M3.

Thanks!

eugene.burmako@epfl.ch