scala.reflect

Eugene Burmako

EPFL, LAMP

19 May 2012

Agenda

- Intro
- API
- Demo
- Summary

Reflection

A way for a program to learn about itself.

Who should we ask?

- OS
- JVM
- Compiler

Scala reflection

Interface to the Scala compiler:

- Full-fledged (cf. scalap, Code.lift, manifests)
- Cross-stage (available both at compile-time and at runtime)
- Uniform (the same API – universes, mirrors, trees, symbols, types – is used everywhere)

Before we start

- This is a fantasy (yet, a blessed fantasy) API. It will be implemented only in M4 (i.e. released in 1-2 weeks).
- Reflection in M3 is quite similar to what we'll be discussing today, except that runtime reflection is fundamentally broken.
- In the final release (and, possibly, in M4 as well) reflection will be factored out into scala-reflect.jar. It's a new core jar in the Scala distribution (along with scala-library.jar and scala-compiler.jar), estimated size is 2-3 Mb.
- For the brave ones, proto-M4 reflection might be found at scalamacros/topic/reflection and at odersky/topic/reflection. Proceed with care, as virtually anything might blow up in your face. You're welcome to bug me at eugene.burmako@epfl.ch.

Agenda

Intro

API

Demo

Summary

Entry point: runtime

This is something well-known from Java or C#, a runtime reflection API. You know what I mean.

```
> val ru = scala.reflect.runtime.universe
ru: scala.reflect.runtime.Universe = ...

> val mirror = ru.reflectClass(classOf[C])
mirror: scala.reflect.runtime.Universe#ClassMirror = ...

> val c = mirror.symbol
c: scala.reflect.runtime.Universe#Symbol = ...

> val m = c.typeSignature.member("x").suchThat(_.isMethod)
m: scala.reflect.runtime.Universe#Symbol = ...
```

Note the terminology: universes as umbrellas, mirrors as reflectors, symbols as reference points. It is shared across entry points.

Entry point: compile-time

The beauty of our reflection API manifests itself in macros. The environment is different, yet, the interface is the same.

```
class Queryable[T] {
  def map[U](p: T => U): Queryable[U] =
    macro QImpl.map[T, U]
}

object QImpl {
  def map[T: c.TypeTag, U: c.TypeTag]
         (c: scala.reflect.makro.Context)
         (p: c.Expr[T => U])
         : c.Expr[Queryable[U]] = ...
}
```

Context is an entry point to compile-time reflection. It exposes a universe (the compiler) and a mirror (the symbol table).

Entry point: compile-time

The control is inversed here. At runtime you call the compiler. At compile-time the compiler calls you.

Currently we support only one kind of compile-time hooks - macro definitions. You declare a method to be a macro def by writing its body in a special way, and then every invocation of that method will be compiled by calling into the corresponding macro implementation.

We also plan to research macro types (typedefs that expand into generated types) and macro annotations (annotations that modify program elements - expressions, methods, types). Who knows what is going to be next =)

Entry point: cross-stage

Compile-time and runtime parts of your program can play together to the mutual benefit.

For example, you can save artifacts from the compile-time universe and pass them to the runtime. Makes sense - these artifacts are quite useful, but a lot of them are destroyed by compilation.

This is called reification. With the unified reflection API, it is both possible, easy to implement and future-proof.

Reified types

Scala 2.8 introduced manifests, a tool to persist type arguments in the presence of erasure. Manifests encode reified types in a series of factory calls.

```
> implicitly[Manifest[List[String]]]
...
Manifest.classType[List[String]](
  classOf[List],
  Manifest.classType[String](classOf[java.lang.String]))
...
res0: Manifest[List[String]] =
    scala.collection.immutable.List[String]
```

Usefulness of manifests is limited. They are doing fine as array creators, they can also express simple polymorphic types, but they totally bail on complex things such as compounds and existentials. The most reliable part of manifests is the toString method.

Reified types

In Scala 2.10 type tags improve upon manifests. They are thin wrappers over compiler Types exposed in the reflection API.

```
> implicitly[ru.TypeTag[List[String]]]
...
materializing requested TypeTag[List[String]]
using materializeTypeTag[List[String]](ru)
...
$u.ConcreteTypeTag[List[String]]($u.TypeRef(
  $u.ThisType("scala.collection.immutable"),
  $m.staticClass("scala.collection.immutable.List"),
  List($m.staticClass("java.lang.String").asType)))
...
res0: ru.TypeTag[List[String]] =
    ru.ConcreteTypeTag[List[String]]
```

Having a type tag you can do (almost) whatever the compiler can do with a type, e.g. destructure it, create types from it, use it during compilation (yes, I'm talking runtime compilation).

Reified trees

Scala 2.8 sneakily introduced code lifting, a tool to persist compiler trees. Trees produced by Code.lift are similar to compiler ASTs, yet their implementation is disparate.

```
> Code.lift{ val y = x + 2 }
Code[Unit](ValDef(
  LocalValue(reflect.this.NoSymbol, "y", NoType),
  Apply(
    Select(Literal(x), Method("scala.Int.$plus"), ...),
    List(Literal(2))))))
```

As a result, lifted trees only expose a small subset of operations that can be done on compiler trees. For example, they are typed, yet they cannot be typechecked, they have symbols, but those cannot be fully inspected and so on.

Reified trees

In Scala 2.10 we have the reify macro that takes a Scala snippet (an expression, a declaration, a bunch of declarations) and reproduces it at runtime.

```
> ru.reify{ val y = x + 2 }
val free$x1: $u.Symbol =
  $u.build.newFreeTerm("x", $m.staticClass("Int"), x)
$u.Expr[Unit]($u.ValDef(
    $u.NoMods,
    $u.newTermName("y"),
    $u.TypeTree(),
    $u.Apply(
      $u.Select($u.Ident(free$x1), "$plus"),
      List($u.Literal($u.Constant(2))))))
```

Reify uses compiler trees to persist the ASTs. As of such, reified trees can be typechecked, compiled, can serve as templates for other trees, etc.

Agenda

Intro

API

Demo

Summary

Inspect members

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

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

scala> res1.tpe.members
res2: Iterable[ru.Symbol] = List(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, typecheck and invoke members

```
val d = new DynamicProxy{ val dynamicProxyTarget = x }
d.noargs
d.noargs()
d.nullary
d.value
d.-
d.$("x")
d.overloaded("overloaded")
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
```

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.typeTag[T]
foo: [T](x: T)(implicit evidence$1:
    ru.TypeTag[T])ru.TypeTag[T]

scala> foo(List(1, 2, 3))
res1: ru.TypeTag[List[Int]] =
    ru.ConcreteTypeTag[List[Int]]

scala> res1.tpe
res2: ru.Type = List[Int]
```

Compile at runtime

```
import ru._
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.

Compile at compile-time

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: scala.reflect.makro.Context)
    (cond: c.Expr[Boolean], msg: c.Expr[Any])
    : c.Expr[Unit] =
  if (assertionsEnabled)
    c.reify(if (!cond.eval) raise(msg.eval))
  else
    c.reify(())
}
```

Also known as macros. Can be used for a multitude of real-world use-cases: advanced DSLs, language-integrated queries, type providers, integration of externals DSLs with string interpolation, your fancy macro hackery could also be mentioned here.

Agenda

Intro

API

Demo

Summary

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).

Status

Recently released 2.10.0-M3 includes reflection and macros.

Compile-time reflection in M3 works well, and we've already improved the compiler with it.

Runtime reflection in M3 is broken beyond repair, so we have rewritten it. This will show up in M4 (i.e. in 1-2 weeks).

Thanks!

eugene.burmako@epfl.ch