Scala 2.10 Reflection and Macros

compile-time and run-time metaprogramming

or how I learned to love the reification of the compiler API

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

some of the good and the bad:

a significant bit of the compiler API is now exposed

Part of the compiler was refactored using the "multiple cake pattern" to provide a unified API at runtime as reflection and compile time as macros.

some background:

Martin Odersky: Reflection and Compilers

Documentation is still a bit scarce. I imagine this will be improving soon with the final release of 2.10.0.

This presentation heavily quotes **scalamacros.org** and the Scaladocs.

learn by exploration

- compiler flags
  - -Xprint:parser (for naked trees)
  - -Xprint:typer (for typechecked trees)
  - -Yshow-trees and its cousins
  - -Xprint-types
  - -explaintypes
  - check out **ScalaSettings.scala**
- at the REPL

```
scala> import reflect.runtime.universe._
import reflect.runtime.universe._

scala> showRaw(reify { 42 })
res0: String = Expr(Literal(Constant(42)))
```

get to know scala's AST

- **Trees.scala** in compiler source
- **Scala Refactoring, Mirko Stocker's master's thesis**, section describing the scala AST, p95, old but still likely relevant, I've only skimmed this sofar
- **Compiler Internals**, walk-through sessions by Martin Odersky, older but still useful I imagine, haven't had the chance to check these out yet

universes and mirrors

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

“ Mirrors abstract population of symbol tables. Each universe can have multiple mirrors, which can share symbols with each other within their parent universe. ”

a universe provides interface for the following:

- Types → types
- Symbols → definitions
- Trees → abstract syntax trees
- Names → term and type names
- Annotations → annotations
- Positions → source positions of tree nodes
- FlagSet → sets of flags that apply to symbols and definition trees
- Constants → compile-time constants

reflection examples

inspect members

```
scala> import reflect.runtime.universe._
import reflect.runtime.universe._

scala> class Z { def theAnswer = 42 }
defined class Z

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

scala> res0.members
res1: reflect.runtime.universe.MemberScope = Scopes(method theAnswer, con
structor Z, ...method $asInstanceOf, method $isInstanceOf, method synchro
nized, method ...
```

type signature

```
scala> import reflect.runtime.universe._
import reflect.runtime.universe._

scala> class Z { def theAnswer = 42 }
defined class Z

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

scala> res0.member(newTermName("theAnswer")).asMethod.typeSignature
res1: reflect.runtime.universe.Type = => scala.Int
```

invoke method

```
scala> import reflect.runtime.universe._
import reflect.runtime.universe._

scala> runtimeMirror(getClass.getClassLoader)
res0: reflect.runtime.universe.Mirror = JavaMirror with scala.tools.nsc.
interpre...

scala> class Z { def theAnswer = 42 }
defined class Z

scala> typeOf[Z]
res1: reflect.runtime.universe.Type = Z

scala> res1.declaration(newTermName("theAnswer")).asMethod
res2: reflect.runtime.universe.MethodSymbol = method theAnswer

scala> res0.reflect(new Z)
res4: reflect.runtime.universe.InstanceMirror = instance mirror for Z@27
833133

scala> res4.reflectMethod(res2)
res5: reflect.runtime.universe.MethodMirror = method mirror for Z.theAns
wer: scala.Int (bound to Z@27833133)

scala> res5()
res7: Any = 42
```

handle erasure

RIP Manifest

```
scala> import reflect.runtime.universe._
import reflect.runtime.universe._

scala> runtimeMirror(getClass.getClassLoader)
res0: reflect.runtime.universe.Mirror = JavaMirror with scala.tools.nsc.i
nterpre...

scala> def i[T : TypeTag](x: T) = typeOf[T] match {
     |    case i if i =:= typeOf[List[String]] => "List of String"
     |    case i if i =:= typeOf[List[Int]] => "List of Int"
     |    case i => "other"
     | }
i: [T](x: T)(implicit evidence$1: reflect.runtime.universe.TypeTag[T])Str
ing

scala> i(List(42))
res0: String = List of Int

scala> i(List("boz"))
res1: String = List of String

scala> i("zed")
res2: String = other
```

reflection API is rich and complex
much more than what you've seen here
happy hunting

macros

reminiscent of Lisp macros

written in full-fledged Scala

works on expression trees

adapted to incorporate type safety and rich syntax

cannot change syntax of Scala

“ Macros are functions that are called by the compiler during compilation. Within these functions the programmer has access to compiler APIs. For example, it is possible to generate, analyze and typecheck code. ”

possible uses

- Advanced domain-specific languages
- Language integrated queries
- Type providers
- Integration of external DSLs
- Aspect-oriented programming
- Type-safe bindings
- Generation of boilerplate
- Data types "a la carte"
- Units of measure

macro examples

didn't have time to make slides, we can go to some **exploratory code** if we want

keep tabs on, **thread safety issue**

future

- explore projects using macros
  - **slick** (prolly deserves a presentation on its own)
  - **Play 2.1 JSON API - Inception**
  - **lift-json for Scala 2.10**
  - ...
- compare and contrast with scala-virtualized
- dream up and write some examples motivated by pragmatic use cases