Macro Paradise

Eugene Burmako

École Polytechnique Fédérale de Lausanne
`http://scalamacros.org/`

18 December 2012

Before we begin

I'd like to heartily thank:

- Early adopters and contributors fearlessly trying out macros and reflection since December 2011
- Reflection group turning impossible problems into great ideas, one meeting at a time

Today's talk is about

- Macros in upcoming Scala 2.10.0-final
- New features implemented since RC1
- Plans for the future

Screencast: http://vimeo.com/user8565009/macro-paradise-talk

Discussion: scala-language/thread/21c0cdce38715771

State of the art

Macros

Macros are functions that are called by the compiler during compilation. Within these functions the programmer has access to compiler APIs.

— http://scalamacros.org

Def macros

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)

  def assert(cond: Boolean, msg: Any) = macro impl
  def impl(c: Context)
          (cond: c.Expr[Boolean], msg: c.Expr[Any]) =
    if (assertionsEnabled)
      c.universe.reify(if (!cond.splice) raise(msg.splice))
    else
      c.universe.reify(())
}
```

- Seamless integration into existing language
- Macros use compiler API to create abstract syntax trees
- `reify` implements the notion of quasiquoting

Macros are powerful

Currently available:

- Creating new expressions

We plan to experiment with:

- Creating new types
- Adding fields and methods to existing types
- Steering type inference and implicit search

Macros are useful

- Slick
- ScalaMock v3
- SBT v0.13
- Play v2.1
- Expecty
- Scalaxy
- Sqltyped
- Declosurify
- More ideas at scalamacros.org

Macros are viable

- Implementation footprint is less than 1kloc
- And we have already simplified the compiler itself using macros
- Scala reflection, which exposes compiler internals for macro writers, works good enough to be released (although in experimental status)
- The SIP committee overseeing additions and changes to Scala is convinced that macros are worth trying out

Macros are magic

- Tree construction is hard, because `reify` has limited usability: excellent explanation by Travis Brown
- Symbol manipulation is even harder: resetAttrs cargo cult, check out an answer at Stack Overflow for details
- Reflection API is at times lacking: befriend `asInstanceOf`
- Error messages and debugging for generated code are tricky

We acknowledge these problems and will do our best to address them. Our latest developments will be covered in a few minutes. Our long-term plans are outlined in the last section of the talk.

Macro paradise

Good news

Since our last report in November, we have made progress

Bad news

- Scala 2.10.0 is feature frozen for, oh my, already four months
- Scala 2.10.x isn't going to welcome new shiny features due to compatibility restrictions
- Scala 2.11.0 is scheduled to happen only in a year

Macro paradise

Will be released together with 2.10.0 (on the first week of January 2013), so far lives in `scalamacros/kepler`:

- `paradise/macros` branch at `scala/scala`
- Nightlies easily available in SBT and Maven
- Code is experimental, but successful features are going be merged into trunk around major releases
- Just like the good old times of 2.10.0-Mx: we hack macros and reflection, you can use new features and fixes immediately

Type macros

Type macros

```
type H2Db(url: String) = macro impl

object Db extends H2Db("coffees")

val brazilian = Db.Coffees.insert("Brazilian", 99, 0)
Db.Coffees.update(brazilian.copy(price = 10))
println(Db.Coffees.all)
```

- Seamless integration into existing language
- Wherever you can write types, you can use type macros
- However you can define with def macros (value parameters, type parameters, overloading, overriding, etc), you can define type macros

Macro implementation

```
type H2Db(url: String) = macro impl

def impl(c: Context)(url: c.Expr[String]) = {
  val name = c.enclosingClass.name + "_Generated"
  val clazz = ClassDef(..., Template(..., generateCode()))
  c.introduceTopLevel(clazz)
  Apply(Ident(name), List(Literal(Constant(c.eval(url)))))
}

object Db extends H2Db("coffees")
```

- An entire class gets generated and inserted into the symbol table
- Macro itself expands into a constructor call, as if the user has written `object Db extends DB_Generated("coffees")`
- Full source code at Github

Features

- Can be used wherever a type is expected
- `c.introduceTopLevel` to generate top-level, i.e. non-nested classes and objects (hint: also available in def macros)
- When used in parent role, have full control over parents, self-types and members of child classes and objects (there'll be an example shortly)

Roadmap

Planned:

- Erasable types
- Caching for invocations and generated types

Out of scope (this will be explored later in annotation macros):

- Addition of inner classes or objects
- Manipulation of existing classes or objects except for type macros parent roles

Quasiquotes

History

- January 2012: prototype based on `Code.lift`
- February 2012: prototype based on runtime parsing with `scala.tools.nsc.Global`, implemented by Natallie Baikevich
- February 2012: `reify`, which quickly became the official quasiquoting facility for Scala macros
- December 2012: milestone based on compile-time parsing and reification, implemented by Denys Shabalin

A motivating example

```
class D extends Lifter {
  def x = 2
  // def asyncX = future { 2 }
}

val d = new D
d.asyncX onComplete {
  case Success(x) => println(x)
  case Failure(_) => println("failed")
}
```

- `Lifter` is a type macro
- It takes the body of the host class (`Template` in scalac parlance) and for each method adds its async version
- Full source code at Github

Macro implementation is "simple"

```
case ClassDef(_, _, _, Template(_, _, ctor :: defs)) =>
  val defs1 = defs collect {
    case DefDef(mods, name, tparams, vparamss, tpt, body) =>
      val tpt1 = if (tpt.isEmpty) tpt else AppliedTypeTree(
        Ident(newTermName("Future")), List(tpt))
      val body1 = Apply(
        Ident(newTermName("future")), List(body))
      val name1 = newTermName("async" + name.capitalize)
      DefDef(mods, name1, tparams, vparamss, tpt1, body1)
  }
  Template(Nil, emptyValDef, ctor +: defs ::: defs1)
```

- When `reify` fails, one has to assemble trees manually
- The code here is a bit simplified, but it still shows how cumbersome and verbose manual tree construction is.

Quasiquotes

Before:

- Apply(Ident(newTermName("future")), List(body))
- AppliedTypeTree(Ident(newTermName("Future")), List(tpt))
- case ClassDef(_, name, _, Template(_, _, _ :: defs)) ⇒ ...

After:

- q"future { $body }"
- tq"Future[$tpt]"
- case q"class $name { ..${_ :: defs} }" ⇒ ...

Roadmap

Available:

- Construction and deconstruction of abstract syntax trees
- Splicing and matching of lists and lists of lists

Planned:

- Error reporting
- Extensive support for language constructs
- Hygiene and referential transparency

Tentative plans for the future

Robust tree manipulation

- Hygiene and referential transparency resistant to `resetAttrs`
- Fixes for non-idempotencies in typer: SI-5464

Better infrastructure

- IDE support for debugging expanded code: SI-5922
- Sane error messages about malformed expansions: SI-6822
- Lifecycle management for macro-produced artifacts: SI-6752

This area is being investigated by Dmitry Naydanov, who's upgraded the macro engine and built a prototype of a macro debugger for Intellij. Scala IDE is also going to eventually support debugging of macro expansions. We're looking forward to incorporating this functionality into macro paradise once it's ready.

Implicit macros

```
trait Serializer[T] {
  def write(pickle: Pickle, x: T): Unit
}

def serialize[T](x: T)(implicit s: Serializer[T]): Pickle

implicit def generator[T]: Serializer[T] = macro impl[T]
def impl[T](c: Context): c.Expr[Serializer[T]] = ...
```

- Sort of work right now, except for SI-5923
- A fix would entail a principled redesign of how macros and type inference interact SI-6755

Macro annotations

```
class atomic extends MacroAnnotation {
  def complete(defn: _) = macro("generate a backing field")
  def typeCheck(defn: _) = macro("return defn itself")
}

@atomic var fld: Int
```

- Statically-typed analogue of Python's decorators
- Operates on arbitrary definitions
- Two-step expansion: macro-level + micro-level

Untyped macros

```
val s = "foo=bar"
s.forAllMatches("""^(?<key>.*?)=(?<value>.*)$""",
    println("key = %s, value = %s".format(key, value)))

def forAllMatches(pattern: String, f: _): Unit = macro impl
```

- Macro arguments are typechecked before macros are called
- However sometimes this is inconvenient, especially when one wants to adjust with lexical scope in a macro
- Type safety isn't subverted, because macro expansions are typechecked as usual
- SI-5405 tracks progress in this direction

Summary

Summary

- Macro paradise, scheduled to be released during the first week of January 2013, will encapsulate development of new macro features
- Type macros (beta quality) and quasiquotes (milestone quality) are waiting for the new year to be included in `paradise/macros` at `scala/scala`
- These features will be available shortly after release as SNAPSHOT builds of `org.scala-lang.macro-paradise`
- To play with the new functionality before the release, build `paradise/macros at scalamacros/kepler`
- Future development might include erasable type macros, robust tree manipulation, IDE support, implicit macros, macro annotations, untyped macros