Half a Year in Macro Paradise

Eugene Burmako

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

12 June 2013
What this talk covers

- New developments in macros after 2.10.0
- Reflection on our experience with macros
- The future of macros in Scala 2.10+
What this talk doesn’t cover

- New developments powered by macros
  - Pickles and spores (Heather’s talk today at 13:30)
  - scala-async (Philipp’s and Jason’s talk today at 14:30)
  - shapeless (Miles’ talk today at 14:30)
  - scala-workflow (Evgeny’s project at GitHub)
  - Akka typed channels (the video of Roland presenting at NEScala)
  - Yin-Yang (Vojin’s paper at infoscience.epfl.ch)
  - Specialization 2.0 (Nicolas’ and Vlad’s project at GitHub)
  - Type-safe JSON (Greg’s talk at Geecon)
  - Improvements for the cake pattern (John Sullivan’s talk today at 11:15)
  - Parallel collections 2.0 (coming this summer)
What this talk doesn’t cover

- New developments powered by macros (see the previous slide)
- Best practices (my upcoming talk at Scalapeño)
- Design details (my upcoming talk at Strange Loop)
Macros in Scala 2.10
Macros

- New experimental feature in Scala 2.10.0
- Macros are functions written in Scala against reflection API
- They are invoked by the compiler during compilation
- A lot of cool things can be done with a compiler API, so there are multiple macro flavors
Def macros

- The only macro flavor in Scala 2.10.0
- Calls to def macros expand into programmatically generated code
- http://docs.scala-lang.org/overviews/macros/overview.html
Example

log(Error, "does not compute")

if (Config.loggingEnabled)
    Config.logger.log(Error, "does not compute")

- We will now write a macro that automates logging
- Without macros this is impossible to achieve at zero performance cost
Example

def log(severity: Severity, msg: String): Unit = ...

- Macro signatures look like signatures of normal methods
Example

def log(severity: Severity, msg: String): Unit = macro impl

def impl(c: Context)
  (severity: c.Expr[Severity],
   msg: c.Expr[String]): c.Expr[Unit] = ...

- Macro signatures look like signatures of normal methods
- Macro bodies are just stubs, implementations are defined outside
Example

def log(severity: Severity, msg: String): Unit = macro impl

def impl(c: Context)
    (severity: c.Expr[Severity],
    msg: c.Expr[String]): c.Expr[Unit] = {
import c.universe._
reify {
    if (Config.loggingEnabled)
        Config.logger.log(severity.splice, msg.splice)
    }
}

- Macro signatures look like signatures of normal methods
- Macro bodies are just stubs, implementations are defined outside
- Implementations use reflection API to analyze and generate code
What are macros good for?

- Code generation
- Language virtualization
- Type computations
- Compile-time checks
Macros vs textual code generation

Highlights:

- Structured (macros work with ASTs)
- Type-aware (macros integrate with the typechecker)
- Reflective (macros can reflect against the program being compiled)
Macros vs textual code generation

Highlights:

- Structured (macros work with ASTs)
- Type-aware (macros integrate with the typechecker)
- Reflective (macros can reflect against the program being compiled)

Limitations:

- Only hardcore (macros 1.0 are really cumbersome)
- Only expressions (macros 1.0 only include def macros)
- Only local (macros 1.0 cannot make global changes to the program)
- Only transient (macros 1.0 cannot generate code for humans)
Why am I highlighting the “1.0” part?

- Because macros are rapidly evolving
- In part thanks to external contributors like you!
- A lot of cool things have been implemented after the 2.10.0 release
- Which makes a lot of problems and restrictions go away
- How? Now we’re going to find out!
Macros in paradise
Macro paradise

- An experimental fork of scalac, available for 2.10.x and 2.11.0: http://docs.scala-lang.org/overviews/macros/paradise.html

- Compatible with the latest releases, i.e. with 2.10.2 and 2.11.0-M3 (this means you can use the libraries published for those releases!)

- Nightlies are published to Sonatype and are easily accessible in SBT:

  scalaVersion := "2.11.0-SNAPSHOT" or "2.10.2-SNAPSHOT"
  scalaOrganization := "org.scala-lang.macro-paradise"
  resolvers += Resolver.sonatypeRepo("snapshots")
Cool new features

- Quasiquotes (Denys Shabalin)
- Implicit macros
- Type macros
- Macro annotations
- Untyped macros
- JIT compilation (Oleg Biruk)
- Relaxed macros
Quasiquotes

// tree manipulation 1.0
reify(List[T](element.splice))

// tree manipulation 2.0
q"List[$T]($element)"
Untyped snippets

val fieldMemberType: Type = ...
reify {
  new TypeBuilder {
    type FieldType = fieldMemberType.splice // error!
  }
}

q"new TypeBuilder { type FieldType = $fieldMemberType }"

- Unlike reify, quasiquotes don’t require their snippets to be typed
- From experience, this is a vital feature for a metaprogramming system
Better splicing

```scala
def foo(xs: Any*) = ...
val args: List[Expr[Any]] = ...
reify { foo(args.splice) } // error!
```

- reify supports splicing single strongly-typed trees and types
- Quasiquotes allow splicing virtually anything anywhere it makes sense
Pattern matching

expr match {
  case reify(foo.splice(x.splice)) => x // error!
}

expr match {
  case q"$foo($x)" => x
}

- Being strongly-typed, reify is hard to marry with destructuring
- Quasiquotes can pattern match in arbitrary positions in snippets
Implicit macros

trait Reads[T] {
  def reads(json: JsValue): JsResult[T]
}

object Json {
  def fromJson[T](json: JsValue) (implicit fjs: Reads[T]): JsResult[T]
}

▶ Type classes are an idiomatic way of writing extensible code in Scala
▶ This is an example of typeclass-based design in Play
Implicit macros

def fromJson[T](json: JsValue)
  (implicit fjs: Reads[T]): JsResult[T]

implicit val IntReads = new Reads[Int] {
  def reads(json: JsValue): JsResult[T] = ...
}

fromJson[Int](json) // you write
fromJson[Int](json)(IntReads) // you get

- With type classes we externalize the moving parts
- And then specify them elsewhere
- Instances of type classes are provided once
- And then scalac fills them in automatically
Before macros

case class Person(name: String, age: Int)

implicit val personReads = (
  (__ \ 'name).reads[String] and
  (__ \ 'age).reads[Int]
)(Person)

- Everything is done manually, hence boilerplate
- There are alternatives, but they have downsides
Vanilla macros (2.10.0)

implicit val personReads = Json.reads[Person]

> Boilerplate can be generated by a macro
> The code ends up being the same as if it were written manually
Implicit macros (2.10.2+)

// no code necessary

- Implicit values can be synthesized on-the-fly by a macro
- Used with great success in scala-pickling
- More information in my tomorrow’s talk in San Francisco
Type macros

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

- Term macros can generate terms, type macros generate types
- Imagine we need to create a strongly-typed wrapper for a database
- Type macros are a great solution for that!
Type macros

object Db extends H2Db("Coffees")

- The H2Db macro takes a connection string
- ...

29
Type macros

```scala
object Db extends H2Db("Coffees")

trait H2Db_Coffees {
  class Coffee { ... }
  val Coffees: Table[Coffee] = ...
}
object Db extends H2Db_Coffees
```

- The H2Db macro takes a connection string
- Then connects to the database and generates the wrapper
- Similar to type providers in F#
Type macros

type H2Db(url: String) = macro impl

- Definition and usage of type macros are the same as for def macros
- We start with a macro def and write its signature
Type macros

type H2Db(url: String) = macro impl

def impl(c: Context)(url: c.Tree) = {
  val wrapper = q"trait Wrapper { ${generateCode(url)} }"}
  ...
}

- Now we proceed with the implementation
- The implementation creates a trait that encapsulates a database
Type macros

type H2Db(url: String) = macro impl

def impl(c: Context)(url: c.Tree) = {
  val wrapper = q"trait Wrapper { $\{generateCode(url)\} }"
  val wrapperRef = c.introduceTopLevel(wrappersPkg, wrapper)
  ...
}

- The implementation creates a trait that encapsulates a database
- And then makes the newly created trait visible to the entire program
Type macros

type H2Db(url: String) = macro impl

def impl(c: Context)(url: c.Tree) = {
    val wrapper = q"trait Wrapper { ${generateCode(url)} }"
    val wrapperRef = c.introduceTopLevel(wrappersPkg, wrapper)
    q"$wrapperRef($url)"
}

- The implementation creates a trait that encapsulates a database
- And then makes the newly created trait visible to the entire program
- Afterwards it expands into a reference to the wrapper
Macro paradise hosts a lot of cool new features

Immediately available from Sonatype

Macro paradise is not a thing in itself, it targets upstream Scala

The most successful paradise features have already made it into Scala

Which ones? We’ll see in a few minutes!
The future of macros
Macros 1.0 are great

- Things that were previously impossible are now within reach
  - People are using macros to bring their ideas to life
  - Typesafe employs macros in a number of projects
  - At LAMP we are using macros to power our research
Macros 1.0 are complicated

- Annoying
  - Hard to grasp
  - Hard to use

- Volatile
  - A lot of freedom type-wise
  - A lot of freedom execution-wise
The macro conundrum

- Macros 1.0 are annoying
- Macros 1.0 are volatile
- But we still want macros, because they are so great!
Macros 2.0
Macros 2.0

- Simplify
  - Quasiquotes!
  - The rest of reflection API
  - Better IDE support (debugging, inline expansion, Intellij)
Macros 2.0

- Simplify
  - Quasiquotes!
  - The rest of reflection API
  - Better IDE support (debugging, inline expansion, IntelliJ)

- Stratify
  - Codify the conservative ones (stable subset)
  - Let the powerful ones evolve (experimental subset)
How does one stratify macros?

- By answering a simple question
  - Do we have to expand this macro to typecheck the program?

- This is quite equivalent to the questions
  - Does a human have to expand this macro to understand the program?
  - Does an IDE have to expand this macro to analyze the program?
  - Does this macro really taste like a method?
Blackbox macros

- The conservative ones
- Don’t affect typechecking
- One can say they are opaque to the typer, hence the name
- BlackboxContext = quasiquotes + just a bit more
Whitebox macros

- The powerful ones
- For them everything stays as it is now and will continue evolving
- WhiteboxContext = Context of macros 1.0 + later developments
Our primary goal for now is to make macros easy to use
Then we plan to bring blackbox macros into the language
Are blackbox macros good enough? Time will tell
In the meanwhile we will still be experimenting with whitebox macros
The roadmap for macros in Scala 2.10+
Experimental:

- Reflection (2.10.0+, not going anywhere)
- Macros 1.0 (2.10.0+, not going anywhere)
- Implicit macros (2.10.2+, single-parametric type classes only)
- Quasiquotes (2.10.0+, quasi-supported via paradise 2.10.x)
2.11.0

Experimental (looking good for becoming stable in 2.12):

▶ Blackbox macros
▶ Quasiquotes
▶ Macro bundles

Experimental (needing more time for evaluation):

▶ Reflection
▶ Whitebox macros
▶ Implicit macros (single-parametric type classes only)
▶ asInstanceOf[scala.reflect.internal.SymbolTable]
Paradise

Look good for promotion to 2.11.0, but need time that we might not have before the release:

▶ Implicit macros (multi-parametric type classes)
▶ Macro annotations

Won’t be promoted to 2.11.0, ordered by descending likelihood of making it into any Scala at all:

▶ introduceTopLevel
▶ Untyped macros
▶ Type macros
Summary

- Macros are here to stay
- Blackbox macros are going to be stabilized in 2.12
- But whitebox macros will still stick around as experimental
- So your macros will continue working in 2.11 and probably in 2.12
- Type macros didn’t make it, macro annotations will take their place
Wrapping up
Summary

- Macros 1.0 are popular among production and research users of Scala
- We created a fork of scalac called macro paradise
- In paradise we have been experimenting with our design
- And we came up with a bunch of improvements for macros 1.0
- This will make macros easy to use and accessible for everyone
Or in other words

- Macros were created by man
- They rebelled
- They evolved
- There are many flavors
- And they have a plan