Philosophy of Scala Macros

Eugene Burmako

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

19 September 2013

In this talk

- History: how Scala got macros and how people liked them
- Philosophy: what is the essence of Scala macros
- Relativity: how traditional macros map onto this

How did Scala get macros?

Inception

me 9/17/11

Hi folks! My name is Eugene Burmako. I'm a first-year PhD student from Martin Odersky's lab at EPFL starting a semester project.

The idea, which I like the most at the moment, is to implement a full-fledged metaprogramming facility for Scala with quasiquotations and hygienic macros.

Inception

- First it was just a fun project inspired by Nemerle
- Quickly escalated to a language feature for the upcoming 2.10 release
- Why? Industrial demand for the power of metaprogramming

Slick

- Scala Language-Integrated Connection Kit
- Seamless data access for your Scala application
- Joint EPFL + Typesafe project, initiated in Oct 2011
- Needed some form of language-integrated queries

LINQ

```
val users: Queryable[User] = ...
users.filter(u => u.name == "John")
```

```
val users: Queryable[User] = ...
Queryable(Filter(users, Equals(Ref("name"), Literal("John"))))
```

- Allow users to write queries in normal Scala
- Automatically lift these queries to data structures
- Achieving deep embedding of domain-specific languages

Macros: a sketch

```
class Queryable[T](val query: Query[T]) {
  macro def filter(p: T => Boolean): Queryable[T] = <[
    val liftedp = ${lift(p)}
    Queryable(Filter($this.query, liftedp))
  ]>
}

val users: Queryable[User] = ...
users.filter(u => u.name == "John")
```

- Macros realize textual abstraction
- Compiler expands snippets in the program being compiled
- Programmer defines expanders as normal Scala functions

Related language features

```
class Queryable[T](val query: Query[T]) {
  macro def filter(p: T => Boolean): Queryable[T] = <[
    val liftedp = ${lift(p)}
    Queryable(Filter($this.query, liftedp))
  ]>
}

val users: Queryable[User] = ...
users.filter(u => u.name == "John")
```

- Compile-time function execution
- Quasiquotes
- Hygiene
- Macro flavors

Towards macros in Scala 2.10

- As we have just seen, macros don't like to party alone
- However we didn't really have budget for huge parties
- Therefore we set out to find the most minimalistic design possible
- Even at the cost of leaving some features out
- This is a very empowering experience

Towards macros in Scala 2.10

martin
- show quoted text -
We'd need to be convinced that it is beautifully simple, or it won't go into Scala.

Macros: as implemented in 2.10

Def macros:

- Just a single feature: unhygienic expansion of typed method calls
- CTFE = macro defs + precompiled macro impls invoked by reflection
- Quasiquotes and hygiene bootstrapped over the unhygienic core
- Other macro flavors got deferred to future releases

Macros: as implemented in 2.10

```
class Queryable[T](val query: Query[T]) {
  def filter(p: T => Boolean) = macro ...
}
```

Macros: as implemented in 2.10

```
class Queryable[T](val query: Query[T]) {
  def filter(p: T => Boolean) = macro Macros.filter[T]
}

object Macros {
  def filter[T: c.WeakTypeTag]
      (c: Context { type PrefixType = Queryable[T] })
      (p: c.Expr[T => Boolean]) = ...
}
```

Macros: as implemented in 2.10

```
class Queryable[T](val query: Query[T]) {
  def filter(p: T => Boolean) = macro Macros.filter[T]
}

object Macros {
  def filter[T: c.WeakTypeTag]
      (c: Context { type PrefixType = Queryable[T] })
      (p: c.Expr[T => Boolean]) =
    c.universe.reify {
      val liftedp = lift(p).splice
      new Queryable(Filter(c.prefix.splice.query, liftedp))
    }
}
```

On the verge of the release

odersky
Why macros? I was a long-time skeptic. I now tend to think about them differently because I believe we hit on a brilliantly simple scheme that can express a lot of different use-cases.

How did people like macros?

Adoption

- Even though in 2.10 we only support expansion of typed method calls
- And there are rough edges that we are fixing for 2.11 and 2.12
- Macros ended up being much more useful than we anticipated
- Widely used in popular libraries, industry and research
- All Scala talks at this conference are powered by macros

Recognition

- Jan 2013: released in experimental capacity
- Apr 2013: deemed worthy of becoming part of language standard

The main question of today's talk

Why did macros work?

Our hypothesis

- Macros piggyback on a familiar concept of a typed method call
- Macros transparently empower features represented with method calls

Features represented with method calls

- Fields: `foo` and `foo_=`
- Application: `apply`
- Pattern matching: `unapply`, `isEmpty`, `get` and `_N`
- Implicits: `implicit` modifier on methods
- For comprehensions: `flatMap`, `map`, `withFilter` and `foreach`
- String interpolation: extension methods on `StringContext`
- Dynamic: `selectDynamic`, `updateDynamic` and `applyDynamic`

Our experience

- Scala'13: Let Our Powers Combine!
- Scalapeño 2013: What Are Macros Good For?
- Today we'll see some excerpts

Example #1 - Empowered method calls

```
val futureDOY: Future[Response] =
  WS.url("http://api.day-of-year/today").get

val futureDaysLeft: Future[Response] =
  WS.url("http://api.days-left/today").get

futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok(s"$dayOfYear: $daysLeft days left!")
  }
}
```

- Turning a synchronous program into an async one isn't easy
- One has to manually manage callbacks, introduce temps, etc

Example #1 - Empowered method calls

```
def async[T](body: => T): Future[T] = macro ...
def await[T](future: Future[T]): T = macro ...

async {
  val dayOfYear = await(futureDOY).body
  val daysLeft = await(futureDaysLeft).body
  Ok(s"$dayOfYear: $daysLeft days left!")
}
```

- Turning a synchronous program into an async one isn't easy
- But `scala/async` macros can do the transformation automatically
- To the user it looks and feels like a call to a vanilla method

Example #2 - Empowered interpolation

```
scala> val x = "42"
x: String = 42

scala> "%d".format(x)
j.u.IllegalFormatConversionException: d != java.lang.String
  at java.util.Formatter$FormatSpecifier.failConversion...
```

- Strings are typically perceived to be unsafe

Example #2 - Empowered interpolation

```
scala> val x = "42"
x: String = 42

scala> "%d".format(x)
j.u.IllegalFormatConversionException: d != java.lang.String
  at java.util.Formatter$FormatSpecifier.failConversion...

scala> f"$x%d"
<console>:31: error: type mismatch;
 found   : String
 required: Int
```

- Strings are typically perceived to be unsafe
- Though with macros they don't have to be
- Even more so with the new string interpolation

Example #2 - Empowered interpolation

```
implicit class Formatter(c: StringContext) {
  def f(args: Any*): String = macro ...
}

val x = "42"
f"$x%d" // rewritten into: StringContext("", "%d").f(x)
```

```
{
  val arg$1: Int = x // doesn't compile
  "%d".format(arg$1)
}
```

- `f` is a macro that inserts type ascriptions in strategic places
- With macros, interpolation and many other features gain new powers

Example #2 - Empowered interpolation

```
reify(List[T](element.splice))
```

```
q"List[$T]($element)"
```

- Now our strings are both flexible and statically checked
- This means that we can deeply embed entire languages
- That's exactly how we bootstrapped quasiquotes

Example #3 - Empowered implicits

```
trait Pickler[T] {
  def pickle(picklee: T): Pickle
}

def pickle[T](picklee: T)(implicit p: Pickler[T]): Pickle
```

- Type classes are an idiomatic way of writing extensible code in Scala
- This is an example of typeclass-based design of a pickling library

Example #3 - Empowered implicits

```
def pickle[T](picklee: T)(implicit p: Pickler[T]): Pickle

implicit val IntPickler = new Pickler[Int] {
  def pickle(picklee: Int): Pickle = ...
}
pickle(42) // you write
pickle(42)(IntPickler) // you get

pickle("42") // compilation error
```

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

Example #3 - Before macros

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

implicit val personPickler = new Pickler[Person] {
  def pickle(picklee: Person): Pickle = {
    val p = new Pickle
    p += ("name" -> p.name.pickle)
    p += ("age" -> p.age.pickle)
    p
  }
}
```

- Everything is done manually, hence boilerplate
- There are alternatives, but they have downsides

Example #3 - Vanilla macros

```
implicit val personPickler = Pickler.generate[Person]
```

- Boilerplate can be generated by a macro
- The code ends up being the same as if it were written manually
- Therefore performance remains excellent

Example #3 - Implicit macros

```
// no code necessary
```

- Implicit values can be generated by a macro
- Much like `deriving` in Haskell

Example #3 - Implicit macros

```
trait Pickler[T] { def pickle(picklee: T): Pickle }

object Pickler {
  implicit def materializePickler[T]: Pickler[T] = macro ...
}
```

- When `scalac` looks for implicits, it traverses the implicit scope
- Implicit scope transcends lexical scope
- Among others it includes members of the target's companion

Example #3 - Implicit macros

```
pickle(person)
```

```
pickle(person)(Pickler.materializePickler[Person])
```

```
pickle(person)(new Pickler[Person]{ ... })
```

- Every time a `Pickler[T]` isn't found, the compiler will call our macro
- More information in my Applied Materialization talk from this June

Implicits and macros: a match made in heaven

- Implicits are a thing of conceptual beauty
- Macros make them even better
- Don't miss out today's talk by Heather Miller:
  Pickles & Spores: Improving Distributed Programming in Scala
- Tomorrow's talk by Miles Sabin and Edwin Brady:
  Scala vs Idris: Dependent types, now and in the future

Implicits and macros: a retrospective

me 18/10/2011 (from an early design document)

Okay, we can have macro defs, macro types and macro packages. Lets explore the design space a bit more to look for other sensible applications of macros.

The gist of all these macro XXX thingies is defining something that can be used in place of those XXXs. For example, macro types can be used wherever you use regular types (e.g. in generic type arguments).

*To put it in a nutshell, macros can virtualize definitions, i.e. XXXs that are covered in Chapter 4 Basic Declarations and Definitions. Namely: vals, vars, defs, types, classes, traits (***but not implicits: that would be too much, lol***).*

Summary

Def macros worked because:

- They piggyback on a familiar concept of a typed method call
- They transparently empower features desugared into method calls

The phenomenon of whitebox macros

Limitations of def macros

- Def macros are neat, but they are missing an important feature
- The ability to affect bindings

Limitation #1 - Can't affect local bindings

```
def lambda(params: ???)(body: ???): ??? = macro ...
lamdba(x, y){ x + y } // does not compute
```

- Def macros typecheck arguments prior to expansion
- Therefore no `lambda` macro in 2.10

Limitation #2 - Can't affect global bindings

```
def h2db(connString: String): Any = macro ...
val db = h2db("jdbc:h2:coffees.h2.db")

val db = {
  trait Db {
    case class Coffee(...)
    val Coffees: Table[Coffee] = ...
  }
  new Db {}
}
```

- Def macros typecheck expansions as expression
- Therefore no definitions can be visible to the outer world in 2.10
- (Actually they can, in a way, but that's a story for another talk)

Macro paradise

- In Dec 2012, after 2.10 ship has sailed, I established macro paradise
- Focus on features that extend the notion of def macros
- Implemented untyped macros, type macros and macro annotations

Untyped macros

```
def lambda(params: _)(body: _) = macro ...
lamdba(x, y){ x + y }
```

- Untyped macros suppress typechecking of arguments before expansion
- Type safety isn't subverted, as expansions are typechecked as usual

Type macros

```
type H2Db(url: String) = macro ...
object Db extends H2Db("jdbc:h2:coffees.h2.db")
```

```
@synthetic trait CoffeesH2Db$1 {
  case class Coffee(...)
  val Coffees: Table[Coffee] = ...
}
object Db extends CoffeesH2Db$1
```

- Type macros are to types what def macros are to terms
- By expanding into synthetic types they can introduce global bindings

Macro annotations

```
@h2db("jdbc:h2:coffees.h2.db")
object Db
```

```
object Db {
  case class Coffee(...)
  val Coffees: Table[Coffee] = ...
}
```

- Just as def macros expand calls, annotation macros expand definitions
- This can naturally introduce definitions at local and global scale
- These definitions can be scoped thanks to Scala's first-class modules

Two faces of Scala macros

- Blackbox macros: faithfully conform to their type signatures
- Whitebox macros: can't have signatures in Scala's type system

Note that:
(1) Blackbox macros != generative macros
(2) Blackbox macros != statically typed quasiquotes

The phenomenon of whitebox macros

- Manipulation of bindings is fundamental to Lisp macros
- However in Scala land it doesn't seem to be very popular
- Adoption of the whitebox flavors from paradise isn't quite stellar
- Why didn't it work?

Explanation #1 - Scala is not very whitebox

- In Scala people are much more used to reasoning about types
- Even if the types are very advanced
- It is considered to be very stylish to hack up typelevel solutions
- A lot of features to work with terms and types, not so much for defs
- Therefore there are no powerful synergies to exploit (yet!)

Explanation #2 - Whitebox is not very Scala

```
class Day(name: String) {
  override def equals(other: Any): Boolean = ...
  override def hashCode(): Int = ...
  override def toString() = ...
}

object Day {
  val Monday = new Day("Monday")
  val Tuesday = new Day("Tuesday")
  ...
}
```

- Suppose we want to come up with an alternative to Scala enums
- To do that we need to generate the aforementioned boilerplate

Explanation #2 - Whitebox is not very Scala

```
@Enum
object Day {
  Monday
  Tuesday
  ...
}
```

- We can use the ablity of macro annotations to reshape their annottees
- The `Enum` annotation infers enum names from the constructor and generates the necessary code
- For more information see the discussion of Simon Ochsenreither's work at scala-internals

Explanation #2 - Whitebox is not very Scala

```
def Monday = ...

@Enum
object Day {
  Monday
  Tuesday
  ...
}
```

- Unfortunately closer scrutiny exposes disturbing syntactic irregularities
- What are the "hanging" names supposed to mean to a newcomer?
- How do these names work with definitions in enclosing scopes?

Explanation #2 - Whitebox is not very Scala

```
@Enum
object Day {
  val Monday, Tuesday, ...: Day
}
```

- This slightly less succinct proposal looks much nicer
- Enum values remain values, they just get enriched
- But now again, where does the `Day` class come from?

Summary

Whitebox macros didn't quite work because:

- They are too malleable
- They can easily transcend intuition about Scala code

Nowadays in macro paradise we're experimenting with metaphors for whitebox macros. Finding the best way to tap into the power of Lisp while remaining idiomatic is a goal worth pursuing!

The bottom line

The bottom line

- Started as a fun project, Scala macros got quickly promoted to a language feature in Scala 2.10 thanks to industrial demand.
- Scala macros are a success. Their "just a method" look and feel allows code to absorb new meanings without losing comprehensibility.
- A lot of language features in Scala desugar into method calls, which makes these features transparently empowered by macros.
- When applied to Scala, the concept of traditional Lisp macros transforms into blackbox macros and whitebox macros.
- Finding a way to naturally integrate whitebox macros is an exciting challenge that we're solving, aiming for the Scala 2.12 release

Additional slides

Modern macro paradise

- Nowadays in paradise we look for a metaphor for whitebox macros
- Let's see some of the use cases we have explored

Use case #1 - Enrich your definition

Applying an orthogonal concept to a definition:

- Case classes
- Lazy values
- Enumeration modules
- Inheritance by delegation

Use case #1 - Enrich your definition

```
class Wrapper(@delegate wrapped: Api) extends Api {
  // the macro delegates Api methods
  // to the annotated parameter
}
```

- Annotations are a nice metaphor for this scenario
- They have a clear scope of effect
- People are used to annotations doing magic in Java and .NET
- However there's an open problem of controlling their power

Use case #2 - Generation of boilerplate

- Some boilerplate can be avoided with advanced typelevel techniques
- But there are always boiler plates to scrap

Use case #2 - Generation of boilerplate

```
type db = SqliteDb<"Data Source=coffees.sqlite" >
printfn "%A" db.Coffees.all
```

```
@h2db("jdbc:h2:coffees.h2.db") object Db {}
println(Db.Coffees.all)
```

- Annotations can handle this scenario, but it becomes a stretch
- Virtual modules generated by F# look natural
- But public classes summoned out of thin air - not so much

Use case #2 - Generation of boilerplate

```
mixin h2db("jdbc:h2:coffees.h2.db")
println(Coffees.all)
```

- A straightforward HULK GENERATE approach might be an answer
- Similar to splicing in TH, `mixin` in D and `@eval` in Julia

Use case #3 - Context injection

Abstraction over contextual operations:

- Database manipulations
- STM transactions
- Domain-specific languages

Use case #3 - Context injection

```
Vectors {
  val v = Vector.rand(100)
}
```

```
abstract class DSLprog extends VectorsApp {
  def apply = {
    val v = Vector.rand(100)
  }
}
(new DSLprog with VectorsCompiler).result
```

- Inspired by scope injection from the world of LMS and Delite
- Recent paper presented at ECOOP'13 provides a nice example