To Macros and Beyond!

How macros changed Scala, and what’s coming next

Eugene Burmako

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

19 July 2016
Who am I?

$ git config --get remote.origin.url
https://github.com/scala/scala

$ git shortlog --author=Burmako -sn --no-merges | awk ...
763 commits

$ git log --author=Burmako --numstat | awk ...
146492 lines added
96869 lines removed
In today’s talk

- Battle-tested design that marries macros and types
- The best and the worst things about Scala macros
- What’s coming next in future versions of the language
A whirlwind tour
Scala macros started as my first semester project at EPFL

I just joined the PhD program and knew close to nothing about Scala

But I really liked macros in Nemerle
Use case

- Database query and access library for Scala
- Joint EPFL + Lightbend project, kicked off in Sep 2011
- Needed some form of language-integrated queries
Use case

val users: Table[User] = ...
users.map(u => u.name)

val users: Table[User] = ...
new Table(Map(users, Ref("name", classOf[String])))

- Allow users to write queries in normal Scala
- Somehow transform these queries into a custom representation
- Compile the representation to SQL, etc
Idea #1

class Table[T](val query: Query[T]) {
    def map[U](fn: scala.reflect.Expr[T => U]) = ... 
}

val users: Table[User] = ...
users.map(u => u.name)

- Have the compiler magically convert T to Expr[T]
- Expression trees can then be analyzed at runtime
- This is how it’s done in C# and F#
Idea #2

object Macros {
    def map(fn: compiler.Tree) = ...
}

val users: Table[User] = ...
users.map(u => u.name)

- Have the compiler run user-defined code during compilation
- During compilation, code is represented with trees anyway
- Compile-time processing brings a number of additional benefits
class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro ... 
}

- Macros look like normal Scala methods
class Table[T](val query: Query[T]) {
  def map[U](fn: T => U): Table[U] = macro Macros.map
}

object Macros {
  def map(c: Context)(fn: c.Tree): c.Tree = ...
}

- Macros look like normal Scala methods
- Implementations are written against a dedicated reflection API
Implementation

class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro Macros.map
}

object Macros {
    def map(c: Context)(fn: c.Tree): c.Tree = {
        val subquery: c.Tree = translate(fn)
        q"new Table(Map(${c.prefix}, $subquery))"
    }
}

▶ Macros look like normal Scala methods
▶ Implementations are written against a dedicated reflection API
▶ Reflection API includes quasiquotes to perform AST manipulations
val users: Table[User] = ...
users.map(u => u.name)

val users: Table[User] = ...
new Table(Map(users, Ref("name", classOf[String])))

- When the user writes `Table.map`, the compiler calls `Macros.map`
- `Macros.map` expands in a domain-specific fashion
- Compiler replaces the call to `Table.map` with the macro expansion
Language-integrated queries via macros

class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro Macros.map
}

object Macros {
    def map(c: Context)(fn: c.Tree): c.Tree = {
        val subquery: c.Tree = translate(fn)
        q"new Table(Map(${c.prefix}, $subquery))"
    }
}
The best thing about Scala macros
The best thing about Scala macros

- After the public release, macros have spread like wildfire
- E.g. all libraries in Lightbend stack have used or are using our stuff
- Why did this happen?
The best thing about Scala macros

- Macros piggyback on a familiar concept of a typed method call
- This allows existing code to easily absorb new meanings
- A lot of language features become transparently enriched
Enriched interpolation

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

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

Scala’s type system can’t typecheck format strings
Enriched interpolation

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

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

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

- Scala’s type system can’t typecheck format strings
- With macros, this stops being an issue
- By integrating with other features, macros solve the problem at hand
String interpolation

scala> val world = "world"
world: String = "world"

scala> s"hello, $world!"
res0: String = "hello, world!"
String interpolation

scala> val world = "world"
world: String = "world"

scala> s"hello, $world!"
res0: String = "hello, world!"

scala> desugar(s"hello, $world!")
res1: Tree = StringContext("hello, ", "!").s(world)
String interpolation

scala> val world = "world"
world: String = "world"

scala> s"hello, $world!"
res0: String = "hello, world!"

scala> desugar(s"hello, $world!")
res1: Tree = StringContext("hello, ", ").s(world)

scala> // string interpolation also works in patterns
    // check out our docs for more details
Extension methods

scala> f"hello, $world!"
error: value f is not a member of StringContext
Extension methods

scala> f"hello, $world!"
error: value f is not a member of StringContext

scala> implicit class AnyNameYouWant(ctx: StringContext) {
    |   def f(args: Any*): String = ...
    | }
defined class AnyNameYouWant
Extension methods

scala> f"hello, $world!"
error: value f is not a member of StringContext

scala> implicit class AnyNameYouWant(ctx: StringContext) {
    |   def f(args: Any*): String = ...
    | }
defined class AnyNameYouWant

scala> f"hello, $world!"
res2: String = ...

Extension methods

scala> f"hello, $world!"
error: value f is not a member of StringContext

scala> implicit class AnyNameYouWant(ctx: StringContext) {
    |   def f(args: Any*): String = ...
    | } 
defined class AnyNameYouWant

scala> f"hello, $world!"
res2: String = ...
Enriched interpolation

```scala
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
- Macros enrich string interpolation with new powers
Other examples

More examples of enriching existing language features with macros:

- Quasiquotes
- async/await
- Deep embedding of DSLs
- Typeclass derivation
- Fast automatic serialization
- Type providers
The most complicated thing about Scala macros
Implementation effort

- 750-1000 commits
- 150-250kloc accumulated changes
- 50kloc added
Implementation breakdown

In increasing order of time spent:

- Macro engine (2kloc, several key findings)
Implementation breakdown

In increasing order of time spent:

- Macro engine (2kloc, several key findings)
- Discussions with the language committee (hundreds of hours)
Implementation breakdown

In increasing order of time spent:

- Macro engine (2kloc, several key findings)
- Discussions with the language committee (hundreds of hours)
- Community building (dozens of talks, thousands of emails)
Implementation breakdown

In increasing order of time spent:

- Macro engine (2kloc, several key findings)
- Discussions with the language committee (hundreds of hours)
- Community building (dozens of talks, thousands of emails)
- Fleshing out scala.reflect (9kloc, 12kloc docs, several rewrites)
The worst thing about Scala macros
How we implemented our reflection API

Compiler internals exposed behind a thin abstraction layer:

- Can be immediately delivered to the users
- Provide an escape hatch into the world of ultimate power
- Require a little bit of learning
How most macro writers felt about it
Problem #1: Overwhelming surface of the API

- Trees
  - TermTrees
  - TypTrees
  - DefTrees
  - ...

- Types
  - Symbols
  - Scopes
  - Names
  - Annotations
  - Constants
  - Modifiers
  - ...

Problem #2: Irreversible desugarings

for (x <- List(1, 2, 3)) yield x * x

immutable.this.List.apply[Int](1, 2, 3).map[Int, List[Int]]((x: Int) => x.*(x))(immutable.this.List.canBuildFrom[Int])

- Scala compiler often desugars parse trees in complicated ways
- This makes it really hard to write robust macros
- Because you can’t just do WYSIWYG pattern matches on syntax
Problem #3: Lock-in

- Scala macros manipulate compiler internals to work with code
- This means that they are completely opaque to third-party tools
- Also what if we want to refactor/rewrite the current compiler?
- Especially pertinent given the increasing importance of Dotty
Macros reloaded
Our plan

- Design a better reflection API from first principles
- Once the API works well, use it to build better macros
Scala.meta

- A clean-room implementation of the language model
- Nothing is desugared (e.g. for loops or string interpolations)
- Nothing is thrown away (e.g. comments or formatting details)
- Designed to be platform-independent from day one
Benefits of comprehensive trees

In scala.reflect:

```scala
scala> q"for (x <- List(1, 2, 3)) yield x * x"
res0: Tree = List(1, 2, 3).map(((x) => x.$times(x)))
```

In scala.meta:

```scala
scala> q"for (x <- List(1, 2, 3)) yield x * x"
res0: Term.ForYield = for (x <- List(1, 2, 3)) yield x * x
```
Benefits of comprehensive trees

In scala.reflect:

scala> q"/* the answer to the ultimate question */ 42"
res0: Literal = 42

In scala.meta:

scala> q"/* the answer to the ultimate question */ 42"
res0: Lit = /* the answer to the ultimate question */ 42
An unexpected journey

Apparently, a principled reflection API is also useful outside macros:

- Codacy (uses scala.meta in style checking rules)
- Scalafmt (uses scala.meta to obtain token-based view of code)
- ...

46
An unexpected journey

- The tooling story is now at least as important as macros
- What novel tools can we build when we have a better reflection API?
- We have a bunch of ideas and are actively looking into this
Back to macros

- Once scala.meta was ready, we started experimenting with macros
- Let’s see how our design process went
Once scala.meta was ready, we started experimenting with macros

Let's see how our design process went

This design hasn’t shipped yet and may be changed in the future
Step #1: Take current macros

```scala
class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro Macros.map
}

object Macros {
    def map(c: Context)(fn: c.Tree): c.Tree = {
        val subquery: c.Tree = translate(fn)
        q"new Table(Map(${c.prefix}, $subquery))"
    }
}
```
Step #2: Trim the boilerplate

class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro Macros.map
}

object Macros {
    def map(c: Context)(fn: c.Tree): c.Tree = {
        val subquery: c.Tree = translate(fn)
        q"new Table(Map(${{c.prefix}}, $subquery))"
    }
}

► Highlighted code is pure boilerplate
► It was deemed necessary when macros were a novel, unknown feature
► Now everyone knows what macros are, so it became redundant
Step #3: Try to extract smaller orthogonal parts

class Table[T](val query: Query[T]) {
  def map[U](fn: T => U): Table[U] = macro {
    val subquery: scala.meta.Tree = translate(fn)
    q"new Table(Map($this, $subquery))"
  }
}

- Hardcodes are not cool, especially in language design
- Macro def/impl separation was one big hardcode
- Now we have a second chance, so let’s use it
Step #3: Try to extract smaller orthogonal parts

```scala
class Table[T](val query: Query[T]) {
  def map[U](fn: T => U): Table[U] = macro {
    val subquery: scala.meta.Tree = translate(fn)
    q"new Table(Map($this, $subquery))"
  }
}
```

- Left-hand side of new macro looks good - like a normal Scala method
- Nothing to do here, let’s move along
class Table[T](val query: Query[T]) {
    def map[U](fn: T => U): Table[U] = macro {
        val subquery: scala.meta.Tree = translate(fn)
        q"new Table(Map($this, $subquery))"
    }
}

- Right-hand side of new macro looks pretty hardcoded
- Let's try to think whether it makes sense in isolation
Step #3: Try to extract smaller orthogonal parts

```scala
macro {
  val subquery: scala.meta.Tree = translate(...)
  q"new Table(Map(..., ...))"
}
```

- Surprisingly, a standalone `macro` block does make sense
- We can view it as “run code at compile time and inline its result”
Step #3: Try to extract smaller orthogonal parts

```scala
val table = macro {
  val subquery: scala.meta.Tree = translate(...)
  q"new Table(Map(..., ...))"
}
```

- Surprisingly, a standalone `macro` block does make sense
- We can view it as “run code at compile time and inline its result”
- Putting the `macro` block in different contexts actually looks okay
Step #3: Try to extract smaller orthogonal parts

```
runQuery(macro { 
    val subquery: scala.meta.Tree = translate(...)
    q"new Table(Map(..., ...))"
})
```

- Surprisingly, a standalone `macro` block does make sense
- We can view it as “run code at compile time and inline its result”
- Putting the `macro` block in different contexts actually looks okay
Step #3: Try to extract smaller orthogonal parts

class Table[T](val query: Query[T]) {
    inline def map[U](fn: T => U): Table[U] = macro {
        val subquery: scala.meta.Tree = translate(fn)
        q"new Table(Map($this, $subquery))"
    }
}

- macro blocks can now expand on their own
- So we only need a vehicle to deliver them to macro callsites
- We can do that by introducing inline methods
Sketch of macro expansion

```scala
val users: Table[User] = ...
users.map(u => u.name)
```

▶ When the user writes `Table.map`, the compiler inlines its rhs
▶ Resulting macro block expands in a domain-specific fashion
▶ Macro expansion is inlined again
Macros 2.0

class Table[T](val query: Query[T]) {
  inline def map[U](fn: T => U): Table[U] = meta {
    val subquery: scala.meta.Tree = translate(fn)
    q"new Table(Map($this, $subquery))"
  }
}

- Pretty much good old macros 1.0
- But without definition-site boilerplate
- And without hardcoded definition syntax
Live demo
Wrapping up
Credits

Martin Odersky  Denys Shabalin  Scala community
Summary

- Scala macros are a powerful and popular language feature.
- Their best part is the combination of metaprogramming and types that makes it possible to enrich existing language features.
- Their worst part is the ad hoc metaprogramming API that significantly raises the barrier to entry and complicates tool support.
- A better metaprogramming API that initially targeted macros 2.0 ended up being useful on its own for development of novel tools.