

What Are Macros Good For?

Eugene Burmako

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

17 July 2013

This talk has a newer revision that lives at
scalamacros.org/paperstalks/2014-02-04-WhatAreMacrosGoodFor.pdf

What are macros good for?

- ▶ Code generation
- ▶ Static checks
- ▶ Domain-specific languages

Macros 101

What are macros?

- ▶ An experimental feature of 2.10.x and 2.11.0
- ▶ You write functions against the reflection API
- ▶ Compiler invokes them during compilation

Macro flavors

- ▶ Many ways to hook into the compiler → many macro flavors
- ▶ Type macros, annotation macros, untyped macros, etc
- ▶ However in 2.10.x and 2.11.0 there are only def macros

Def macros

```
log(Error, "does not compute")
```



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

- ▶ Def macros replace well-typed terms with other well-typed terms
- ▶ Generated code might contain arbitrary Scala constructs
- ▶ Codegeneration might involve arbitrary computations

Def macros

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

- ▶ Macro signatures look like signatures of normal methods

Def macros

```
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

Def macros

```
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

Summary

```
log(Error, "does not compute")
```



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

- ▶ Local expansion of method calls
- ▶ Well-formed and well-typed arguments
- ▶ Now what is this good for?

Code generation

Code generation

- ▶ Create code on-the-fly
- ▶ More convenient and robust than textual codegen
- ▶ Impossible to create globally visible classes

Example #1 - Performance

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  for (i <- 0 until size) a(i) = el  
  a  
}
```

- ▶ We want to write beautiful generic code, and Scala makes that easy
- ▶ Unfortunately, abstractions oftentimes bring overhead
- ▶ E.g. in this case erasure will cause boxing leading to a slowdown

Example #1 - Performance

```
def createArray[@specialized T: ClassTag](...) = {  
  val a = new Array[T](size)  
  for (i <- 0 until size) a(i) = el  
  a  
}
```

- ▶ Methods can be @specialized, but it's viral and heavyweight
- ▶ Viral = the entire call chain needs to be specialized
- ▶ Heavyweight = specialization leads to duplication of bytecode

Example #1 - Performance

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  def specBody[@specialized T](el: T) {  
    for (i <- 0 until size) a(i) = el  
  }  
  classTag[T] match {  
    case ClassTag.Int => specBody(el.asInstanceOf[Int])  
    ...  
  }  
  a  
}
```

- ▶ We want to specialize just as much as we need
- ▶ As described in the fresh [Bridging Islands of Specialized Code](#) paper
- ▶ But that's tiresome to do by hand, and here's where macros shine

Example #1 - Performance

```
def specialized[T: ClassTag](code: => T) = macro ...
```

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  specialized[T] {  
    for (i <- 0 until size) a(i) = el  
  }  
  a  
}
```

- ▶ specialized macro gets pretty code and transforms it into fast code
- ▶ This is a typical scenario of using macros for performance
- ▶ Can be non-trivial to implement, but things should become better

Example #2 - Materialization

```
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

Example #2 - Materialization

```
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
- ▶ Instances of type classes are provided once
- ▶ And then scalac fills them in automatically

Example #2 - 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

Example #2 - 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
- ▶ Therefore performance remains excellent

Example #2 - Implicit macros (2.10.2+)

`// no code necessary`

- ▶ Implicit values can be generated by a macro
- ▶ Used with great success in [scala-pickling](#)

Example #2 - Implicit macros (2.10.2+)

```
trait Reads[T] { def reads(json: JsValue): JsResult[T] }  
  
object Reads {  
  implicit def materializeReads[T]: Reads[T] = macro ...  
}
```

- ▶ When scalac looks for implicits, it traverses the implicit scope
- ▶ Implicit scope transcends lexical scope
- ▶ Among others it includes members of the targets companion

Example #2 - Implicit macros (2.10.2+)

```
fromJson[Person](json)
```



```
fromJson[Person](json)(materializeReads[Person])
```



```
fromJson[Person](json)(new Reads[Person]{ ... })
```

- ▶ Every time a `Reads[T]` isn't found, the compiler will call our macro
- ▶ More information in [my talk about materialization](#)

Example #2 - Implicit macros (2.10.2+)

```
implicit def listShowable[T](implicit s: Showable[T]) =  
  new Showable[List[T]] {  
    def show(x: List[T]) = {  
      x.map(s.show).mkString("List(", ", ", ", ")")  
    }  
  }
```

```
show(List(42))  
// prints: List(42)
```

- ▶ This is an example of how macros synergize with other features
- ▶ Here a macro is transparently integrated with a vanilla `listShowable`
- ▶ We will see a couple more applications of the same principle today

Example #2 - A #typelevel alternative

```
product    :: C[A] => C[B] => C[(A, B)]  
coproduct  :: C[A] => C[B] => C[Either[A, B]]  
project    :: C[B] => (A => B, B => A) => C[A]
```

- ▶ Lars Hupel [recently introduced](#) an approach inspired by Haskell's GP
- ▶ Uniform representation + type class combinators = materialization
- ▶ Beautiful, but has some problems with performance

Example #3 - Fake type providers

```
println(Db.Coffees.all)  
Db.Coffees.insert("Brazilian", 99, 0)
```

- ▶ In F# one can generate wrappers over datasources
- ▶ These wrappers can then be used in a strongly-typed manner
- ▶ Can this be implemented with def macros?

Example #3 - Fake type providers

```
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 {}  
}
```

- ▶ Unfortunately for this use case, def macros expand locally
- ▶ Therefore the best we can have is a bunch of local classes

Example #3 - Fake type providers

```
scala> val db = h2db("jdbc:h2:coffees.h2.db")
db: AnyRef {
  type Coffee { val name: String; val price: Int; ... }
  val Coffees: Table[this.Coffee]
} = $anon$1...

scala> db.Coffees.all
res1: List[Db$1.this.Coffee] = List(Coffee(Brazilian,99,0))
```

- ▶ Luckily for us, local classes are erased to structural types
- ▶ This means that we are strongly-typed (static checking, completions)
- ▶ But we have to live with reflective overhead of structural types

Example #3 - Fake type providers

```
class Coffee(row: Row["...".type]) with Dynamic {  
  def selectDynamic = macro ...  
}
```

```
db: AnyRef{type Coffee <: Dynamic; ...}  
coffee.name // rewritten to: coffee.selectDynamic("name")
```



```
coffee.row["name"].asInstanceOf[String]
```

- ▶ It turns out that we don't need to go through structural types
- ▶ Dynamic typing + compile-time macros = static typing
- ▶ But now we lost IDEs, because we no longer expose any members

Example #3 - Fake type providers

```
class Coffee(row: Row["...".type]) {  
  def name: String = macro ...  
}
```

```
db: AnyRef{type Coffee <: AnyRef{def name: String}; ...}  
coffee.name
```



```
coffee.row["name"].asInstanceOf[String]
```

- ▶ Therefore we need to go even deeper
- ▶ Reflective calls + compile-time macros = static calls
- ▶ Meet structural types powered by [vampire methods](#)

Example #3 - Real type providers

```
@H2Db("jdbc:h2:coffees.h2.db") object Db  
println(Db.Coffees.all)  
Db.Coffees.insert("Brazilian", 99, 0)
```

- ▶ This was a fun exercise stretching the boundaries of macrology
- ▶ The technique was discovered just last weekend, so use it with care
- ▶ In the meanwhile keep an eye on [macro annotations](#)

Static checks

Static checks

- ▶ Check your program during compilation
- ▶ Report errors and warnings
- ▶ Impossible to do global checks

Example #4 - Strongly-typed strings

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

```
x: String = 42
```

```
scala> "%d".format(x)
```

```
j.u.IllegalArgumentException: d != java.lang.String  
  at java.util.Formatter$FormatSpecifier.failConversion...
```

- Strings are typically perceived to be unsafe

Example #4 - Strongly-typed strings

```
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 #4 - Strongly-typed strings

```
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
- ▶ Similar techniques can be used for regexen, binary literals, etc

Example #4 - Quasiquotes

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



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

- ▶ Now our strings are both flexible and statically checked
- ▶ This means that we can **robustly embed** entire languages
- ▶ That's how quasiquotes were born (already in 2.11.0-M4)

Example #5 - Akka typed channels

```
trait Request
case class Command(msg: String) extends Request

trait Reply
case object CommandSuccess extends Reply
case class CommandFailure(msg: String) extends Reply

val actor = someActor
actor ! Command
```

- ▶ Akka actors are dynamically typed, i.e. the ! method takes Any
- ▶ This loosens type guarantees provided by Scala

Example #5 - Akka typed channels

```
trait Request
case class Command(msg: String) extends Request

trait Reply
case object CommandSuccess extends Reply
case class CommandFailure(msg: String) extends Reply

type Spec = (Request, Reply) :+: TNil
val actor = new ChannelRef[Spec](someActor)
actor <-!- Command // doesn't compile
```

- ▶ We can implement type specification for actors even in standard Scala
- ▶ But this became practical only when we got macros
- ▶ Akka typed channels are specifically designed to make use of macros

Example #5 - Akka typed channels

```
type Spec = (Request, Reply) :+ TNil  
val actor = new ChannelRef[Spec](someActor)  
actor <-!- Command // doesn't compile
```

- ▶ The <-!- macro takes the type of its prefix and extracts the spec
- ▶ Then it takes the argument type and validates it against the spec
- ▶ This all can be done with implicits and type-level computations
- ▶ But it will be non-trivial both for the programmer and for the users

Example #6 - Spores

```
def future[T](body: => T) = ...
```

```
def receive = {  
  case Request(data) =>  
    future {  
      val result = transform(data)  
      sender ! Response(result)  
    }  
}
```

- ▶ Capturing sender in the above closure is dangerous
- ▶ That's because sender is not a value, but a stateful method
- ▶ To validate captures we can use macros: [SIP-21 – Spores](#)

Example #6 - Spores

```
def future[T](body: Spore[T]) = ...
```

```
def spore[T](body: => T): Spore[T] = macro ...
```

```
def receive = {  
  case Request(data) =>  
    future(spore {  
      val result = transform(data)  
      sender ! Response(result) // doesn't compile  
    })  
}
```

- ▶ The spore macro takes its body and figures out all free variables
- ▶ If any of the free variables are deemed dangerous, an error is reported

Example #6 - Spores

```
def future[T](body: Spore[T]) = ...
```

```
implicit def anyToSpore[T](body: => T): Spore[T] = macro ...
```

```
def receive = {  
  case Request(data) =>  
    future {  
      val result = transform(data)  
      sender ! Response(result) // doesn't compile  
    }  
}
```

- ▶ The conversion to Spore can be made implicit
- ▶ That will verify closures without bothering the user

Domain-specific languages

Domain-specific languages

- ▶ Take a program written in an internal or external DSL
- ▶ Work with it as with a domain-specific data structure

Example #7 - Language-INtegrated Query

```
val usersMatching = query[String, (Int, String)](  
    "select id, name from users where name = ?")  
usersMatching("John")
```

- ▶ Database queries can be written in SQL

Example #7 - Language-INtegrated Query

```
val usersMatching = query[String, (Int, String)](  
  "select id, name from users where name = ?")  
usersMatching("John")
```

```
case class User(id: Column[Int], name: Column[String])  
users.filter(_.name === "John")
```

- ▶ Database queries can be written in SQL
- ▶ They can also be written in a DSL, at times slightly awkward

Example #7 - Language-INtegrated Query

```
val usersMatching = query[String, (Int, String)](  
  "select id, name from users where name = ?")  
usersMatching("John")
```

```
case class User(id: Column[Int], name: Column[String])  
users.filter(_.name === "John")
```

```
case class User(id: Int, name: String)  
users.filter(_.name == "John")
```

- ▶ Database queries can be written in SQL
- ▶ They can also be written in a DSL, at times slightly awkward
- ▶ Or they can be written in Scala and virtualized by a macro

Example #7 - Language-INtegrated Query

```
trait Query[T] {  
  def filter(p: T => Boolean): Query[T] = macro ...  
}
```

```
val users: Query[User] = ...  
users.filter(_.name == "John")
```



```
Query(Filter(users, Equals(Ref("name"), Literal("John"))))
```

- ▶ The filter macro turns a method call on a query into another query
- ▶ The derived query captures the previous one and the predicate
- ▶ We've built a deeply-embedded query DSL

Example #7 - Comparison with staging

```
implicit def queryOps[T](q: Rep[Query[T]]) = new {  
  def filter(p: Rep[T] => Rep[Boolean]): Rep[Query[T]] = ...  
}
```

```
val users: Rep[Query[User]] = ...  
users.filter(_.name == "John")
```

- ▶ In order to deeply embed your DSL, work with `Rep[T]` instead of `T`
- ▶ Powered by implicits and `scala-virtualized`, a fork of `scalac`
- ▶ Staged execution: compile your program, stage it, run the result
- ▶ This technique is called [lightweight modular staging](#)

Example #7 - Comparison with staging

- ▶ Macros allow for earlier error detection
- ▶ Macros don't need stage annotations
- ▶ Staging composes better

Example #7 - Comparison with staging

```
case class Coffee(name: String, price: Double)
val coffees: Queryable[Coffee] = Db.coffees
```

```
// closed world
coffees.filter(c => c.price < 10)
```

```
// open world
def isAffordable(c: Coffee) = c.price < 10
scoffees.filter(isAffordable)
```

- ▶ Code is only processed specially within macro arguments
- ▶ Therefore separate compilation doesn't work out of the box
- ▶ We experiment with ways of dealing with that

Example #8 - Async

```
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 #8 - Async

```
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
- ▶ Macros can do the transformation automatically: [SIP-22 – Async](#)
- ▶ C# `yield` and Python generators can also be emulated by macros

Example #9 - Datomic

```
scala> Query("""
```

```
|      [ :find ?e ?n
```

```
|        :in $ ?char
```

```
|        :where [ ?e :person/name ?n ]
```

```
|                [ ?e person/character ?char ]
```

```
|      ]
```

```
|      """)
```

```
res0: TypedQueryAuto2[DatomicData, DatomicData, (DatomicData,  
DatomicData)] = [ :find ?e ?n :in $ ?char :where ... ]
```

- ▶ Macros can also deeply embed external DSLs
- ▶ In this example a query to Datomic becomes [first-class Scala citizen](#)
- ▶ This means static validation, typechecking and maybe even interop

Summary

What are macros good for?

- ▶ Code generation
- ▶ Static checks
- ▶ Domain-specific languages