

# What Are Macros Good For?

Eugene Burmako

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

02 December 2013

This talk has a newer revision that lives at  
[scalamacros.org/paperstalks/2014-02-04-WhatAreMacrosGoodFor.pdf](http://scalamacros.org/paperstalks/2014-02-04-WhatAreMacrosGoodFor.pdf)

## What are macros?

- ▶ An experimental feature of 2.10 and 2.11
- ▶ 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 and 2.11 there are only def macros

## Def macros

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



```
if (Logger.enabled)  
  Logger.log("does not compute")
```

- ▶ Def macros replace well-typed terms with other well-typed terms
- ▶ Generated code can contain arbitrary Scala constructs
- ▶ Code generation can involve arbitrary computations

## Def macros

```
def log(msg: String): Unit = ...
```

- ▶ Macro signatures look like signatures of normal methods

## Def macros

```
def log(msg: String): Unit = macro impl
```

```
def impl(c: Context)(msg: c.Expr[String]): c.Expr[Unit] = ...
```

- ▶ Macro signatures look like signatures of normal methods
- ▶ Macro bodies are just stubs, referring macro impls defined outside

## Def macros

```
def log(msg: String): Unit = macro impl
```

```
def impl(c: Context)(msg: c.Expr[String]): c.Expr[Unit] = {  
  import c.universe._
```

```
}
```

- ▶ Macro signatures look like signatures of normal methods
- ▶ Macro bodies are just stubs, referring macro impls defined outside
- ▶ Implementations use reflection API to analyze and generate code

## Def macros

```
def log(msg: String): Unit = macro impl
```

```
def impl(c: Context)(msg: c.Expr[String]): c.Expr[Unit] = {  
  import c.universe._  
  q"""  
    if (Logger.enabled)  
      Logger.log($msg)  
  """  
}
```

- ▶ Macro signatures look like signatures of normal methods
- ▶ Macro bodies are just stubs, referring macro impls defined outside
- ▶ Implementations use reflection API to analyze and generate code



# Quasiquotes

```
q"""  
  if (Logger.enabled)  
    Logger.log($msg)  
"""
```

- ▶ `q"..."` string interpolators that build code are called quasiquotes
- ▶ They are very convenient to create and pattern match code snippets
- ▶ In 2.10 quasiquotes are available via [the macro paradise plugin](#)
- ▶ In 2.11 quasiquotes are available in the standard Scala distribution

## Summary

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



```
if (Logger.enabled)  
  Logger.log("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 terms and types on-the-fly
- ▶ More convenient and robust than textual codegen

## Example #1 - Term generation

```
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 - Term generation

```
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 - Term generation

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

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

## Example #1 - Term generation

```
def specialized[T: ClassTag](code: => Any) = 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
- ▶ Also see tomorrow's talk on [Macro-Based Scala Parallel Collections](#)



## Example #2 - Type generation

```
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 #2 - Type generation

```
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 expand locally, therefore we get a bunch of local classes
- ▶ Locals are invisible from the outside, so it's a game over? Nope!

## Example #2 - Type generation

```
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))
```

- ▶ Scala can figure out and expose local signatures to the outer world
- ▶ Used by Specs2 to automatically create matchers for custom classes

## Example #2 - Type generation

```
scala> val db = h2db("jdbc:h2:coffees.h2.db")
db: { type Coffee { ... }; val Coffees: List[this.Coffee]; }
```

- ▶ This is a fun technique stretching the boundaries of macrology
- ▶ There are [some caveats](#), so it should be used with caution
- ▶ Alternatively you could use macro annotations available in 2.10 and 2.11 via the macro paradise plugin

## Example #3 - 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 #3 - 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 #3 - 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, e.g. [one presented at the Scala'13 workshop](#)
- ▶ But each of them has its downsides

## Example #3 - 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 #3 - Implicit macros (2.10.2+)

```
// no code necessary
```

- ▶ Implicit values can be transparently generated by implicit macros
- ▶ Used with success in pickling and shapeless
- ▶ Details on how this works can be found in [our documentation](#)

## Static checks

## Static checks

- ▶ Check your program during compilation
- ▶ Report errors and warnings as you go

## Example #4 - Advanced type signatures

```
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
- ▶ E.g. here we have a sneaky type error that leads to a runtime crash

## Example #4 - Advanced type signatures

```
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 #4 - Advanced type signatures

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

- ▶ The <-!- macro takes the type of its target and extracts the spec
- ▶ Then it takes the argument type and validates it against the spec
- ▶ If necessary, the macro produces precise and clear compilation errors

## Example #4 - Advanced type signatures

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

- ▶ This all can be done with implicits and type-level computations
- ▶ But that's non-trivial both for the library authors and for the users
- ▶ Macros aren't ideal either, and we plan to further research this

## Example #5 - Advanced static checks

```
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 #5 - Advanced static checks

```
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 #5 - Advanced static checks

```
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 #6 - Language virtualization

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

- ▶ Database queries can be written in SQL

## Example #6 - Language virtualization

```
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 #6 - Language virtualization

```
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 #6 - Language virtualization

```
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 takes an AST corresponding to the predicate
- ▶ This AST is then analyzed and transformed into a query fragment
- ▶ Now we have a deeply embedded DSL, just like in LINQ and Slick



## Example #7 - Internal DSLs

```
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 #7 - Internal DSLs

```
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](#)
- ▶ Similar to C#'s `async/await` and parts of Clojure's `core/async`

## Example #7 - Internal DSLs

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

- ▶ At the heart of macro-based DSLs is the ability to analyze code
- ▶ The `async` macro sees detailed inner structure of code representing its argument and can transform that structure to its liking
- ▶ Also see today's talk [JScala - Write Your JavaScript In Scala](#)

## Example #8 - External DSLs

```
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 #8 - External DSLs

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

```
x: String = 42
```

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

```
j.u.IllegalArgumentException: 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

## Example #8 - External DSLs

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

```
val x = "42"
```

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

- ▶ String interpolation desugars custom string literals into method calls
- ▶ These methods can be macros that validate strings at compile-time

## Example #8 - External DSLs

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



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

- ▶ Here the `f` macro just inserts type ascriptions in strategic places
- ▶ But this approach can be used to embed much more complex DSLs
- ▶ This means static validation, typechecking and maybe even interop

# Summary



# What are macros good for?

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