Let Our Powers Combine!

Eugene Burmako

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

02 July 2013

Agenda

- Compile-time metaprogramming with macros
- Integration with rich syntax and static types
- Hammers: a number of macro flavors for Scala
- Nails: case studies for the macro flavors
- No details of the underlying macro system

Hammers: The Macro Flavors

Macros

- Realize textual abstraction in Scala 2.10.0+
- Written in Scala against the Scala reflection API
- Invoked by the compiler during compilation
- Influence compilation: change code, affect types, etc

Macro flavors

- Scala has rich syntax
- It distinguishes terms and types, expressions and definitions
- To abstract over syntax we take that into account:
  - Terms => def macros
  - Types => type macros
  - Definitions => macro annotations

Def macros

```
printf("hello %s", "world!")
```

```
def tmp$1: String = "world!"
print("hello ")
print(tmp$1)
```

- Def macros syntactically replace terms with other terms
- Generated code might contain arbitrary Scala constructs
- Code generation might involve arbitrary computations

Def macros

```
def printf(format: String, args: Any*): Unit = macro impl

def impl(c: Context)(
      format: c.Expr[String],
      args: c.Expr[Any]*): c.Expr[Unit] = {






}
```

- Def macros = definitions + implementations
- Definitions look and feel like normal Scala methods (almost!)
- Implementations are code-transforming metaprograms

Def macros

```
def printf(format: String, args: Any*): Unit = macro impl

def impl(c: Context)(
      format: c.Expr[String],
      args: c.Expr[Any]*): c.Expr[Unit] = {
  import c.universe._
  val q"${sFormat: String}" = format
  val (defs, parts) = parse(sFormat, args)



}
```

- Def macros = definitions + implementations
- Definitions look and feel like normal Scala methods (almost!)
- Implementations are code-transforming metaprograms

Def macros

```
def printf(format: String, args: Any*): Unit = macro impl

def impl(c: Context)(
      format: c.Expr[String],
      args: c.Expr[Any]*): c.Expr[Unit] = {
  import c.universe._
  val q"${sFormat: String}" = format
  val (defs, parts) = parse(sFormat, args)
  val stmts = defs ++ parts.map(part => q"print($part)")
  q"..$stmts"
}
```

- Def macros = definitions + implementations
- Definitions look and feel like normal Scala methods (almost!)
- Implementations are code-transforming metaprograms

Type macros

```
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 syntactically replace types with other types
- Can generate auxiliary classes or objects in the process

Type macros

```
type H2Db(connString: String) = macro impl

def impl(c: Context)(connString: c.Expr[String]) = {





}
```

- Type macros = definitions + implementations
- Definitions look and feel like normal Scala types (almost!)
- Implementations are code-transforming metaprograms

Type macros

```
type H2Db(connString: String) = macro impl

def impl(c: Context)(connString: c.Expr[String]) = {
  val schema = loadSchema(connString)
  val name = schema.dbName + "H2Db"
  val members = generateCode(schema)



}
```

- Type macros = definitions + implementations
- Definitions look and feel like normal Scala types (almost!)
- Implementations are code-transforming metaprograms

Type macros

```
type H2Db(connString: String) = macro impl

def impl(c: Context)(connString: c.Expr[String]) = {
  val schema = loadSchema(connString)
  val name = schema.dbName + "H2Db"
  val members = generateCode(schema)
  c.introduce(q"@synthetic trait $name { $members }")
  q"$name"
}
```

- Type macros = definitions + implementations
- Definitions look and feel like normal Scala types (almost!)
- Implementations are code-transforming metaprograms

Macro annotations

```
@serializable class C(x: Int)
```

```
class C(x: Int) {
  def serialize = ...
}
```

- Macro annotations syntactically replace definitions with other definitions

Macro annotations

```
class serializable extends MacroAnnotation {
  def transform = macro impl
}

def impl(c: Context) = {





}
```

- Macro annotations = definitions + implementations
- Definitions look and feel like normal Scala annotations (almost!)
- Implementations are code-transforming metaprograms

Macro annotations

```
class serializable extends MacroAnnotation {
  def transform = macro impl
}

def impl(c: Context) = {
  val logic = generateCode(c.annottee)
  val serialize = q"def serialize = $logic"



}
```

- Macro annotations = definitions + implementations
- Definitions look and feel like normal Scala annotations (almost!)
- Implementations are code-transforming metaprograms

Macro annotations

```
class serializable extends MacroAnnotation {
  def transform = macro impl
}

def impl(c: Context) = {
  val logic = generateCode(c.annottee)
  val serialize = q"def serialize = $logic"
  val q"class $name($params) { $members }" = c.annottee
  q"class $name($params) { $members :+ serialize }"
}
```

- Macro annotations = definitions + implementations
- Definitions look and feel like normal Scala annotations (almost!)
- Implementations are code-transforming metaprograms

Overarching theme

- Metaprograms are hidden behind vanilla Scala features
- This integrates macros into the language in a very natural way
- Therefore allowing to enrich existing features with new meanings

Synergy with rich syntax and static types

- Scala builds a lot of features on top of others
  - Application: `apply`
  - Getters and setters: `foo` and `foo_=`
  - For comprehensions: `flatMap`, `map`, `withFilter`, `foreach`
  - Dynamic: `selectDynamic`, `updateDynamic`, `applyDynamic`
  - String interpolation: extension methods on `StringContext`
  - Implicits: `implicit` modifier on methods
- All those can be defined as macros, gaining
  - Compile-time programmability
  - Code generation powers

Nails: The Macro Applications

Language virtualization

Language virtualization

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

```
coffees.filter(LessThan(Ref("price"), Literal(10)))
```

- Take a code snippet written in Scala and represent it as data
- Then interpret this data, potentially with different semantics
- Can be used for deep embedding of internal DSLs

Language virtualization

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

  def toList: List[T] = {
    val translatedQuery = query.translate
    translatedQuery.execute.asInstanceOf[List[T]]
  }
  ...
}
```

- Query operators can be implemented as macros
- These macros have their arguments lifted automatically
- Lifted arguments can then be remembered and accumulated
- No additional language features are necessary

Language virtualization

```
object QueryableMacros {
  def filter[T: c.TypeTag](c: Context)(p: c.Tree) = {
    import c.universe._
    val lifted: c.Tree = QueryableMacros.lift(p)




  }
  ...
}
```

- Query operators can be implemented as macros
- These macros have their arguments lifted automatically
- Lifted arguments can then be remembered and accumulated
- No additional language features are necessary

Language virtualization

```
object QueryableMacros {
  def filter[T: c.TypeTag](c: Context)(p: c.Tree) = {
    import c.universe._
    val lifted: c.Tree = QueryableMacros.lift(p)
    val T: c.Type = typeOf[T]
    val callee: c.Tree = c.prefix
    q"Queryable[$T]($callee.query.filter($lifted))"
  }
  ...
}
```

- Query operators can be implemented as macros
- These macros have their arguments lifted automatically
- Lifted arguments can then be remembered and accumulated
- No additional language features are necessary

Comparison with staging

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

Composability

```
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 lifted within macro arguments
- Therefore separate compilation doesn't work out of the box
  - Decompilation to recreate abstract syntax trees
  - `@lifted` macro annotation to retain abstract syntax trees

Type providers

Type providers

```
type NetFlix = ODataService<"...">
let netflix = NetFlix.GetDataContext()
let avatarTitles = query {
  for t in netflix.Titles do
  where (t.Name.Contains "Avatar") sortBy t.Name take 100
}
```

- F# features type providers, a compile-time facility for code generation
- Type providers can be mostly reimplemented with macros

Picking a macro flavor

```
object Db extends H2Db("jdbc:h2:coffees.h2.db")
```

```
@H2Db("jdbc:h2:coffees.h2.db")
object Db
```

- Type macros provide very similar codegen capabilities to type providers
- But thanks to first-class modules macro annotations also fit the bill

Erased type providers

```
object Netflix {
  case class Title(name: String)
  def Titles: List[Title] = ...

  case class Director(name: String)
  def Directors: List[Director] = ...

  ...
}
```

- Sometimes being strongly-typed is too wasteful
- If data comes in untyped anyway, it might be unnecessary to wrap it
- In F# one can avoid unnecessary classes using erased type providers
- In Scala we can't reimplement them, but we can emulate

Emulating erasure

```
object Netflix {
  type Title = XmlEntity
  def Titles: List[Title] = ...

  type Director = XmlEntity
  def Directors: List[Director] = ...

  ...
}
```

- We want to replace strongly-typed wrappers with underlying classes
- And not to lose static typing like in the code of the example
- This is possible due to synergies of macros and vanilla Scala features!

Emulating erasure

```
object Netflix {
  type Title = XmlEntity["http://.../Title".type]
  def Titles: List[Title] = ...

  type Director = XmlEntity["http://.../Director".type]
  def Directors: List[Director] = ...

  ...
}
```

- Wrappers are replaced with type aliases parameterized by identities
- (The `"..."`.type cannot be written, but can be generated!)
- All selections and calls are then statically intercepted with macros

Emulating erasure

```
class XmlEntity[Url] extends Dynamic {
  def selectDynamic(field: String) = macro XmlEntity.impl
}

object XmlEntity {
  def impl(c: Context)(field: c.Tree) = {









  }
}
```

Emulating erasure

```
class XmlEntity[Url] extends Dynamic {
  def selectDynamic(field: String) = macro XmlEntity.impl
}

object XmlEntity {
  def impl(c: Context)(field: c.Tree) = {
    import c.universe._
    val TypeRef(_, _, tUrl) = c.prefix.tpe
    val ConstantType(Constant(sUrl: String)) = tUrl
    val schema = loadSchema(sUrl)




  }
}
```

Emulating erasure

```
class XmlEntity[Url] extends Dynamic {
  def selectDynamic(field: String) = macro XmlEntity.impl
}

object XmlEntity {
  def impl(c: Context)(field: c.Tree) = {
    import c.universe._
    val TypeRef(_, _, tUrl) = c.prefix.tpe
    val ConstantType(Constant(sUrl: String)) = tUrl
    val schema = loadSchema(sUrl)
    val Literal(Constant(sField: String)) = field
    if (schema.contains(sField)) q"${c.prefix}($sField)"
    else c.abort(s"value $sField is not a member of $sUrl")
  }
}
```

Materialization of type class instances

Type classes as objects and implicits

```
trait Showable[T] { def show(x: T): String }
def show[T](x: T)(implicit s: Showable[T]) = s show x

implicit object IntShowable {
  def show(x: Int) = x.toString
}

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

- In Scala type classes can be modelled with traits
- Type class instances are modelled with implicit values
- Implicit search automates type-driven selection of instances

Boilerplate

```
class C(x: Int)
implicit def cShowable = new Showable[C] {
  def show(c: C) = "C(" + c.x + ")"
}

class D(x: Int)
implicit def dShowable = new Showable[D] {
  def show(d: D) = "D(" + d.x + ")"
}
```

- Instance definitions for similar types are frequently very similar
- Leads to proliferation of boilerplate code
- There are techniques to abstract boilerplate, but they have downsides

Materialization

```
trait Showable[T] { def show(x: T): String }

object Showable {
  implicit def materialize[T]: Showable[T] = macro ...
}
```

- Implicit scope of `Showable` includes the members of its companion
- Failed searches for `Showable[T]` fall back to `materialize[T]`
- `materialize[T]` analyzes `T` and generates an appropriate instance

Synergy

```
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)
```

- Materialization seamlessly melds into vanilla implicit search
- In the example above, `Showable[List[Int]]` will be provided by `listShowable`
- Building up on a materialized instance of `Showable[Int]`

Comparison with generic programming

- Materialization provides performance of manually written code
- Generic programming is arguably easier to use (no tree manipulation)
- Is it possible to combine the strong points of both approaches?

Summary

Summary

- Scala Macros, macros that enjoy rich syntax and static types
- Seamlessly integrate with vanilla Scala features
- Empower existing features with code generation and compile-time programmability capabilities
- Work well in a production version of Scala for a number of case studies

Future work

- Formalize the design of macros in a flexible and tractable framework
- Explore compile-time techniques that don't fit into textual abstraction