Applied Materialization

When Macros Meet Implicits

Eugene Burmako

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

13 June 2013

Agenda

My yesterday's talk at ScalaDays was about:

- New developments in macros after 2.10.0
- Reflection on our experience with macros
- The future of macros in Scala 2.10+

Today we will:

- Take on the challenge of generic programming
- Code up a couple of macros using new features from macro paradise
- Discuss materialization, my favorite application of macros

Setting the stage

Our running example

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

def toPickle[T: Pickler](x: T): Pickle = {
  implicitly[Pickler[T]].pickle(x)
}
```

- This is an example of typeclass-based design
- Type classes are an idiomatic way of writing extensible code in Scala
- But let's leave off the type class discussion until Vlad's talk in July

How it works - part 1

```
def toPickle[T: Pickler](x: T) = {
  implicitly[Pickler[T]].pickle(x)
}

def toPickle[T](x: T)(implicit evidence$1: Pickler[T]) = {
  implicitly[Pickler[T]](evidence$1).pickle(x)
}
```

- The context bound is desugared into an implicit parameter
- The call to `implicitly` summons that implicit and calls it to action

How it works - part 2

```
def toPickle[T](x: T)(implicit p: Pickler[T]): Pickle = ...

implicit object IntPickler extends Pickler[Int] { ... }

toPickle(42) // toPickle(42)(IntPickler)
toPickle("42") // compile-time error
```

- To use `toPickle` we declared implicit picklers in scope
- The compiler is then smart enough to figure them out

How it works - part 3

```
def toPickle[T](x: T)(implicit p: Pickler[T]): Pickle = ...

object IntPickler extends Pickler[Int] { ... }
implicit def listPickler[T: Pickler]: List[Pickler[T]] = ...

toPickle(List(42)) // toPickle(...)(listPickler(IntPickler))
```

- It gets even better
- For instance, implicits are composable
- Here a `List[T]` pickler gets built up from a pickler for `T`

How it works - part 4

- The strength of the typeclass-based design lies in its flexibility
- Implicits are composable
- Implicits can be scoped
- Implicits allow for unanticipated evolution
- Though again I'm getting ahead of the upcoming July talk

The problem statement

```
def toPickle[T: Pickler](x: T): Pickle = ...
case class Person(name: String)
toPickle(Person("Eugene"))
```

- How do we make this code snippet compile?
- Is our approach going to be scalable?

A closer look

Straightforward approach

```
case class Person(name: String)

implicit val personPickler = new Pickler[Person] {
  def pickle(x: Person): Pickle = {
    emptyPickle + toPickle(x.name)
  }
}
```

- Straightforward and obvious
- But what if we have some more classes?
- Do we copy/paste the code changing names and adding more fields?
- We need some technique to abstract the tedium away

Reflective approach

```
implicit def genericPickler[T: TypeTag]: Pickler[T] =
  new Pickler[T] { def pickle(x: T) = reflect(x) }

def reflect(x: T): Pickle = {
  val fields = typeOf[T].declarations.collect {
    case sym: MethodSymbol if sym.isGetter => sym }
  val m = currentMirror.reflect(x)

  fields.foldLeft(emptyPickle)((p, f) => {
    p + toPickle(m.reflectMethod(g)())
  })
}
```

- Properly generalizes over case classes, no client code required at all
- But has subpar performance
- And is not type-safe (can you spot the bug?)

Reflective approach

```
implicit def genericPickler[T: TypeTag]: Pickler[T] =
  new Pickler[T] { def pickle(x: T) = reflect(x) }

def reflect(x: T): Pickle = {
  val fields = typeOf[T].declarations.collect {
    case sym: MethodSymbol if sym.isGetter => sym }
  val m = currentMirror.reflect(x)

  fields.foldLeft(emptyPickle)((p, f) => {
    p + toPickle(m.reflectMethod(g)(): Any)
  })
}
```

- Properly generalizes over case classes, no client code required at all
- But has subpar performance
- And is not type-safe, because the reflective invocation returns `Any`

Typelevel approach

- The good thing about reflection is that it can treat data uniformly
- The bad thing is that the representation it uses is dynamically typed
- Luckily there exists a statically typed solution
- Enter `HLists`

Typelevel approach

```
case class Apple() extends Fruit
case class Pear() extend Fruit

val a: Apple = Apple()
val p: Pear = Pear()

val hlist = a :: p :: a :: p :: HNil
val list = a :: p :: a :: p :: Nil
```

- On the surface `HList` is quite similar to `List`

Typelevel approach

```
case class Apple() extends Fruit
case class Pear() extend Fruit

val a: Apple = Apple()
val p: Pear = Pear()

type APAP = Apple :: Pear :: Apple :: Pear :: HNil
val hlist: APAP = a :: p :: a :: p :: HNil
val list: List[Fruit] = a :: p :: a :: p :: Nil
```

- On the surface `HList` is quite similar to `List`
- But it is much more precise type-wise
- Yesterday at ScalaDays Miles worked magic enabled by `HLists`
- And Alois brought `HLists` even further by adding labels

Typelevel approach

```
trait Generic[T, R] {
  def to(t: T): R
  def from(r: R): T
}

implicit val persIso = new Generic[Person, String :: HNil] {
  def to(t: Person) = t.name :: HNil
  def from(r: String :: HNil) = Person(r.head)
}
```

- After the uniform representation for data is picked
- For every data type we define an isomorphism to the repr

Typelevel approach

```
implicit val hnilPickler: Pickler[HNil] =
  new Pickler[HNil] { def pickle(x: HNil) = emptyPickle }

implicit def hlistPickler[H, T <: HList]
  (implicit ph: Pickler[H],
            pt: Pickler[T]): Pickler[H :: T] = {
  new Pickler[H :: T] {
    def pickle(x: H :: T) =
      ph.pickle(x.head) + pt.pickle(x.tail)
  }
}
```

- Now the compiler knows how to treat our data types uniformly
- Therefore a single serializer for repr will make all our data serializable
- This is type-safe and overall cool, but still not very performant

A detour: the bootstrapping challenge

```
implicit val persIso = new Generic[Person, String :: HNil] {
  def to(t: Person) = t.name :: HNil
  def from(r: String :: HNil) = Person(r.head)
}
```

- To serialize data types generically, we need to isomorphize them
- But isomophization itself is a generic programming task!
- Which comes first, the chicken or the egg?

A detour: the bootstrapping challenge

```
implicit val persIso = new Generic[Person, String :: HNil] {
  def to(t: Person) = t.name :: HNil
  def from(r: String :: HNil) = Person(r.head)
}
```

- To serialize data types generically, we need to isomorphize them
- But isomophization itself is a generic programming task!
- Which comes first, the chicken or the egg?
  - The chicken
  - Isomophization can be treated differently from the other GP problems
  - Once somehow solved, it will take care of everything else

Summary of generic programming techniques

Technique	Client-side convenience[1]	Performance[2]	Library-side convenience[3]
Manual*	Bad	Excellent	Excellent
Reflection	Excellent	Bad	Decent
Typelevel	Good	Good	Good

* Not a generic programming technique, is here just for comparison
[1] How much effort is required to add support for a new data type?
[2] How does the performance fare against manually written code?
[3] How much effort is required from a library author?

Level 1: def macros

Macro-based approach

```
implicit val personPickler = new Pickler[Person] {
  def pickle(x: Person): Pickle = {
    emptyPickle + toPickle(x.name)
  }
}
```

- Temporarily back to square one
- We will start with the simplest thing possible
- And will make it enjoyable to use

Macro-based approach

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

- Def macros expand method calls into code blocks
- Expansion happens at compile-time when compiler sees a macro call
- When invoked, macros programmatically construct their expansion
- Arguments of a macro call are available via the reflection API
- Therefore we can write a macro to generate the body of the implicit

Let's write a macro

- Compile-time reflection has the same API as runtime reflection
- With the newly introduced quasiquotes code generation is a breeze
- Therefore we can easily turn our reflective pickler into a macro!
- For details tag along or follow our documentation

Before we begin

- Some of the features we are going to use aren't yet in Scala 2.10
- Those features come from macro paradise, an experimental fork of `scalac`, available for 2.10.x and 2.11.0 (details at docs.scala-lang.org)
- A lot of new developments from paradise are going to end up in 2.11.0
- When introducing features, I will be mentioning their Scala versions

Step 1: Start with a reflective pickler

```
import scala.reflect.runtime.universe._
object Pickler {
  implicit def genericPickler[T: TypeTag]: Pickler[T] = {
    val T = typeOf[T]
    val fields = T.declarations.collect { ... }
    def reflect(x: T): Pickle = ...
    new Pickler[T] { def pickle(x: T) = reflect(x) }
  }
}
```

Step 2: Rebrand it as a macro

```
def genericPickler[T]: Pickler[T] =
  macro PicklerMacro.genericPickler[T]

trait PicklerMacro extends scala.reflect.macros.Macro {
  def genericPickler[T: TypeTag]: Pickler[T] = {
    val T = typeOf[T]
    val fields = T.declarations.collect { ... }
    def reflect(x: T): Pickle = ...
    new Pickler[T] { def pickle(x: T) = reflect(x) }
  }
}
```

Step 3: Make it produce trees

```
def genericPickler[T]: Pickler[T] =
  macro PicklerMacro.genericPickler[T]

trait PicklerMacro extends scala.reflect.macros.Macro {
  def genericPickler[T: WeakTypeTag]: Tree = {
    val T = weakTypeOf[T]
    val fields = T.declarations.collect { ... }
    def reflect: Tree = ...
    q"new Pickler[$T] { def pickle(x: $T) = $reflect }"
  }
}
```

- This macro can be written in Scala 2.10.0, yet in a very verbose way
- However here we use quasiquotes planned for 2.11.0-M4 (8 Jul 2013)
- Those who wrote macros in 2.10, note how easy it is to do it now!

Step 4: Enjoy!

In the source file you write:

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

Under the covers it becomes:

```
implicit val personPickler = new Pickler[Person] {
  def pickle(x: Person): Pickle = {
    emptyPickle + toPickle(x.name)
  }
}
```

Summary of generic programming techniques

Technique	Client-side convenience	Performance	Library-side convenience
Manual	Bad	Excellent	Excellent
Reflection	Excellent	Bad	Decent
Typelevel	Good	Good	Good
Macros†	Good	Excellent	Decent

† Requires def macros (Scala 2.10.0+)

Level 2: materialization

A detour: synergy

In Scala, macros work in harmony with rich syntax and static types:

- String interpolation + macros
- Implicits + macros
- Types + macros
- Annotations + macros

More on that in my recent paper:
"Scala Macros: Let Our Powers Combine!"

Revisiting our current solution

```
def genericPickler[T]: Pickler[T] = macro ...
implicit val personPickler = Pickler.genericPickler[Person]
implicit val repoPickler = Pickler.genericPickler[Repo]
implicit val commitPickler = Pickler.genericPickler[Commit]
...
```

- One generic implementation
- One line of code per data type

Implicit macros

```
implicit def genericPickler[T]: Pickler[T] = macro ...
```

- One generic implementation
- `Zero` lines of code per data type
- When a `Pickler` is missing, one is generated on the fly
- This is a new feature in Scala 2.10.2+

How it works - part 1

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

object Pickler {
  implicit def genericPickler[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

How it works - part 2

```
implicit def genericPickler[T]: Pickler[T] = macro ...

trait PicklerMacro extends Macro {
  def genericPickler[T: WeakTypeTag]: Tree = {
    ...
    q"new Pickler[$T] { def pickle(x: $T) = $reflect }"
  }
}
```

- Here's our def macro from before
- We have just made it implicit
- Are we done yet? No!

How it works - part 2

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle(list)
```

- To illustrate the caveat let's take a recursive data type
- And see how its materializer is going to expand

How it works - part 2

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle(list)({
  new Pickler[List] {
    def pickle(x: List) = {
      emptyPickle +
      toPickle(x.head) +
      toPickle(x.tail)
    }
  }
})
```

- After the first expansion we get two recursive calls to `toPickle`
- The first one will be resolved to `IntPickler`, that's easy
- But what about the second one?

How it works - part 2

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle(list)({
  new Pickler[List] {
    def pickle(x: List) = {
      emptyPickle +
      toPickle(x.head)(IntPickler) +
      toPickle(x.tail)(new Pickler[List] { ... })
    }
  }
})
```

- After the first expansion we get two recursive calls to `toPickle`
- The first one will be resolved to `IntPickler`, that's easy
- But what about the second one? Uh-oh!

How it works - part 2

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle({
  implicit object ListPickler extends Pickler[List] {
    def pickle(x: List) = {
      emptyPickle +
      toPickle(x.head)(IntPickler) +
      toPickle(x.tail)(ListPickler)
    }
  }
  ListPickler
})
```

- We also need to deal with possible recursion
- And we do that by tying the knot using implicits themselves!

Nitpicking time!

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle(list)
```

- Our design has just stood up to a serious test
- But, in fact, this very example is spectacularly incomplete
- It hints at design issues we haven't yet discussed. What are they?

Nitpicking time!

```
case class List(head: Int, tail: List)

val list: List = ...
toPickle(list)
```

- Our design has just stood up to a serious test
- But, in fact, this very example is spectacularly incomplete
- It hints at design issues we haven't yet discussed. What are they?
  - Are algebraic data types supported?
  - Can we declare `head` to be polymorphic?
  - If not polymorphic, can it be `Any`?

Heather's recent paper answers all these questions, and her cool new scala-pickling project makes the dreams come true!

Summary of generic programming techniques

Technique	Client-side convenience	Performance	Library-side convenience
Manual	Bad	Excellent	Excellent
Reflection	Excellent	Bad	Decent
Typelevel	Good	Good	Good
Macros†	Good / Excellent‡	Excellent	Decent

† Requires def macros (Scala 2.10.0+)

‡ Requires implicit macros (Scala 2.10.2+)

Level 3: fundep materialization

Isomorphisms

```
trait Generic[T, R] {
  def to(t: T): R
  def from(r: R): T
}
```

- Let's use our newly acquired proficiency in materialization
- By writing a materializer for the isomorphisms mentioned earlier

Materializing isomorphisms

```
implicit def materialize[T]: Generic[T] =
  macro GenericMacro.materialize[T]

trait GenericMacro extends Macro {
  def materialize[T: WeakTypeTag]: Tree = {
    val T = weakTypeOf[T]
    val R = calculateRepr(T)
    q"new Generic[$T, $R] { ... }"
  }
}
```

- Copy/pasting, adjusting - okay, so far so good
- There is a mistake here. Where is it?

Materializing isomorphisms

```
implicit def materialize[T, R]: Generic[T, R] =
  macro GenericMacro.materialize[T]

trait GenericMacro extends Macro {
  def materialize[T: WeakTypeTag]: Tree = {
    val T = weakTypeOf[T]
    val R = calculateRepr(T)
    q"new Generic[$T, $R] { ... }"
  }
}
```

- Copy/pasting, adjusting - okay, so far so good
- There is a mistake here. Where is it?
  - We forgot the second type parameter of `Generic`

Using the materializer

```
implicit def materialize[T, R]: Generic[T, R] =
  macro GenericMacro.materialize[T]

implicit def genericPickler[T, R]
  (implicit iso: Generic[T, R],
            p: Pickler[R]): Pickler[T] = {
  new Pickler[T] {
    def pickle(x: T) = p.pickle(iso.to(x))
  }
}
```

- Double materialization!
- Requests for a pickler for `T` will be processed by `genericPickler`
- That will materialize `Generic[T, R]` which will figure out `R`
- And that will materialize `Pickler[R]` that does the heavylifting

Problem #1

```scala
implicit val hnilPickler: Pickler[HNil] = ...

implicit def hlistPickler[H, T <: HList]
  (implicit ph: Pickler[H],
            pt: Pickler[T]): Pickler[H :: T] = ...

implicit def genericPickler[T, R]
  (implicit iso: Generic[T, R],
            p: Pickler[R]): Pickler[T] = ...
```

- We have overlapping implicits
- Is that going to be a problem?

Non-problem #1

```scala
implicit val hnilPickler: Pickler[HNil] = ...

implicit def hlistPickler[H, T <: HList]
  (implicit ph: Pickler[H],
            pt: Pickler[T]): Pickler[H :: T] = ...

implicit def genericPickler[T, R]
  (implicit iso: Generic[T, R],
            p: Pickler[R]): Pickler[T] = ...
```

- We have overlapping implicits
- Is that going to be a problem?
  - Not really
  - For any given `T`, `scalac` can figure out the most specific one

Problem #2

```
09:09 ~/Projects/210x $ scalac Test.scala -Xlog-implicits
Test.scala:6: error: could not find implicit value
for parameter iso: Generic[Test.Foo,R]
  toPickle(Person("Eugene"))
          ^
```

- The ever so helpful missing implicit message!
- Let's figure out what went wrong with the help of `-Xlog-implicits`

Problem #2

```
09:09 ~/Projects/210x $ scalac Test.scala -Xlog-implicits
Test.scala:13: materialize is not a valid implicit value
for Generic[Person, R] because:
hasMatchingSymbol reported error: type mismatch;
 found   : Generic[Person, String :: HNil]
 required: Generic[Person, Nothing]
  toPickle(Person("Eugene"))
          ^
```

- What's going on? Where did the `Nothing` come from?

Problem #2

```
toPickle(Person("Eugene"))
```

```
toPickle(Person("Eugene"))(materialize[?, ?])
```

- When inferring the implicit argument for the call to `toPickle`
- `scalac` finds `Generic.materialize` as a candidate
- And then it needs to check whether the type parameters work out
- The first is replacing all of them with unknowns

Problem #2

```
toPickle(Person("Eugene"))(materialize[?, ?])
```

```
toPickle(Person("Eugene"))(materialize[Person, ?])
```

- Using the information provided in the call to `toPickle`
- `scalac` is able to infer `T` to `Person`
- However `R` remains unknown, because nothing hints `scalac` about it

Problem #2

```
toPickle(Person("Eugene"))(materialize[Person, ?])
```

```
toPickle(Person("Eugene"))(materialize[Person, Nothing])
```

- Macros cannot expand with uninferred type arguments
- Therefore `scalac` has to go to extreme measures
- Inferring `R` to a default, which is in this case `Nothing`

Problem #2

```
toPickle(...)(materialize[Person, Nothing])
```

```
toPickle(...)(new Generic[Person, String :: HNil] { ... })
```

- Now the macro gets to finally expand
- But unfortunately typer now expects `Generic[Person, Nothing]`
- And that leads to the compilation error we observed

Let's take a step back

```
implicit val personIso:
  Generic[Person, String :: HNil] = ...
implicit val repoIso:
  Generic[Repo, Url :: String :: HNil] = ...
implicit val commitIso:
  Generic[Commit, Id :: Array[Byte] :: HNil] = ...

implicit def genericPickler[T, R]
  (implicit iso: Generic[T, R],
            p: Pickler[R]): Pickler[T] = ...
toPickle(Person("Eugene"))
```

- No macros for now. Can the compiler figure out this one?
- Yes, it can, because we don't have conflicting implicit instances
- Therefore all that we need here is just a little nudge from the macro

Fundep materialization

- My first try was the `onInfer` callback (New Year's Eve 2013)
- But later we found out a beautifully simple solution (May 2013)
- Let macros expand even if they contain undertermined type params
- The type of the expansion will help the compiler infer the undets!

Summary of generic programming techniques

Technique	Client-side convenience	Performance	Library-side convenience
Manual	Bad	Excellent	Excellent
Reflection	Excellent	Bad	Decent
Typelevel	Good / Excellent§	Good	Good
Macros†	Good / Excellent‡	Excellent	Decent

† Requires def macros (Scala 2.10.0+)

‡ Requires implicit macros (Scala 2.10.2+)

§ Requires fundep materialization (Paradise, maybe Scala 2.11.0+)

Summary

Summary

- Macros can effectively abstract away boilerplate
- In Scala 2.10 macros are useful
- In Scala 2.11 macros will become enjoyable
- Typelevel programming can sometimes be a viable alternative

Summary of generic programming techniques

Technique	Client-side convenience	Performance	Library-side convenience
Manual	Bad	Excellent	Excellent
Reflection	Excellent	Bad	Decent
Typelevel	Good / Excellent§	Good	Good
Macros†	Good / Excellent‡	Excellent	Decent

† Requires def macros (Scala 2.10.0+)
‡ Requires implicit macros (Scala 2.10.2+)
§ Requires fundep materialization (Paradise, maybe Scala 2.11.0+)