Macros vs Types

Eugene Burmako & Lars Hupel

École Polytechnique Fédérale de Lausanne
Technische Universität München

March 1, 2014

Macros vs Types

- Types have been used to metaprogram Scala for ages
- Macros are the new player on the field
- Debates are hot in the IRC and on Twitter
- Time to figure out who's the best once and for all!

Let the games begin!

Following the “What are macros good for?” talk, we will see how the contenders fare in three disciplines:

- Code generation
- Static checks
- Domain-specific languages

Code generation

Code generation

Every language ecosystem has it

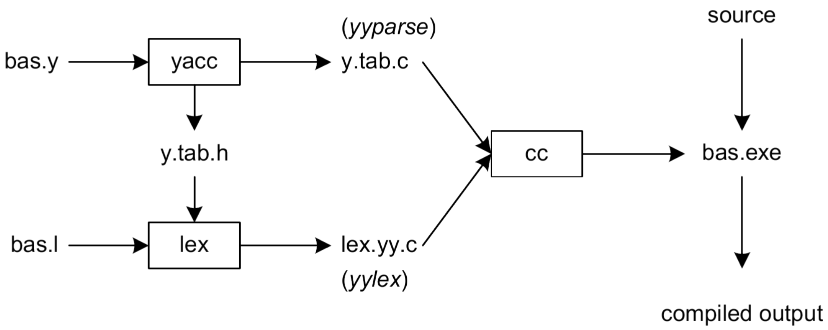
Code generation

Every language ecosystem has it, even Haskell

- `lens`
  derive lenses for fields of a data type
- `yesod`
  templating, routing
- `invertible-syntax`
  constructing partial isomorphisms for constructors

Textual code generation

Example: Parser generators



Textual codegen is too low-tech

- Easy to mess up when concatenating strings
- Little knowledge about the program being compiled
- Needs to be hooked into the build process
- We need a better solution!

Enter types

- Scala's type system is Turing-complete
- This enables some form of code generation
- But it's not particularly straightforward

Enter macros

- Functions that are run at compile time
- Operate on abstract syntax trees not on strings
- Communicate with compiler to learn things about the program
- A lot of popular Scala libraries are already using macros

Use case: Spire Ops

This is a typical situation with high-level abstractions in Scala
There are a lot of ways to write pretty code...

```
import spire.algebra._
import spire.implicits._

def nice[A: Ring](x: A, y: A): A =
  (x + y) * z
```

Use case: Spire Ops

But oftentimes pretty code is going to be slow, because of all the magic flying around, like in this case of typeclass-based design

```
import spire.algebra._
import spire.implicits._

def nice[A: Ring](x: A, y: A): A =
  (x + y) * z

def desugared[A](x: A, y: A)(implicit ev: Ring[A]): A =
  new RingOps(new RingOps(x)(ev).+(y))(ev).*(z) // slow!
```

Use case: Spire Ops

There usually exist alternatives that provide great performance, but often they aren't as good-looking as we'd like them to be

```
import spire.algebra._
import spire.implicits._

def nice[A: Ring](x: A, y: A): A =
  (x + y) * z

def desugared[A](x: A, y: A)(implicit ev: Ring[A]): A =
  new RingOps(new RingOps(x)(ev).+(y))(ev).*(z) // slow!

def fast[A](x: A, y: A)(implicit ev: Ring[A]): A =
  ev.times(ev.plus(x, y), z) // fast, but not pretty!
```

Use case: Spire Ops

However with macros you no longer have to choose – macros can transform pretty solutions into fast code

```
import spire.algebra._
import spire.implicits._

def nice[A: Ring](x: A, y: A): A =
  (x + y) * z

def desugared[A](x: A, y: A)(implicit ev: Ring[A]): A =
  new RingOps(new RingOps(x)(ev).+(y))(ev).*(z) // slow!

def fast[A](x: A, y: A)(implicit ev: Ring[A]): A =
  ev.times(ev.plus(x, y), z) // fast, but not pretty!
```

What are types bringing into the mix?

- Thanks to macros code generation becomes accessible and fun
- But: Macros are essentially opaque to humans
- We can and should try to alleviate this with types

Use case: Materialization

We want to have: default implementations for

- `Semigroup` (pointwise addition)
- `Ordering` (lexicographic order)
- `Binary` (pickling/unpickling)

We do not want to: write boilerplate

- Repetitive & error-prone

Use case: Materialization

`scalac` already synthesizes `equals`, `toString` ...

Use case: Materialization

`scalac` already synthesizes `equals`, `toString` ...

Problem

Not extensible

Use case: Materialization

`scalac` already synthesizes `equals`, `toString` ...

Problem

Not extensible

Solution

Materialization based on type classes and implicit macros

Type classes à la Scala

- Type classes are (first-class) traits
- Instances are (first-class) values

Type classes à la Scala

- Type classes are (first-class) traits
- Instances are (first-class) values
- Both can use arbitrary language features

Use case: Materialization

```
implicit def derive[C[_] : TypeClass, T]: C[T] =
  macro TypeClass.derive_impl[C, T]
```

The power of materialization

- First introduced in Shapeless
- Similar to `deriving Eq` in Haskell
- Extensible without modifying the macro(s) itself

The dangers of materialization

Bad

```
implicit def derive[C[_], T]: C[T] =
  macro TypeClass.derive_impl[C, T]
```

Good

```
implicit def derive[C[_] : TypeClass, T]: C[T] =
  macro TypeClass.derive_impl[C, T]
```

Our advice

- Macros are great, but are essentially opaque to humans
- Try to document the codegen surface using types
  (type classes and other advanced techniques really help here!)
- Try to limit the codegen surface to just the "moving parts"
  (maybe more boilerplate, but more predictable)
- We need best practices for documentation & testing

Static checks

Types à la Pierce

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

– Benjamin Pierce, in: Types and Programming Languages

Types à la Pierce

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

– Benjamin Pierce, in: Types and Programming Languages

Types à la Scala

Scala has a sophisticated type system

- Path-dependent types
- Type projections
- Higher-kinded types
- Implicit parameters

Type computations

Implicits allow computations in the type system

- Higher-order unification (SI-2712)
- Generic operations on tuples
- Extensible records
- Statically size-checked collections

Shapeless

The library that makes advanced types accessible!

Type computations

Example: Sized collections

```
// typed as Sized[_2, List[String]]
val hdrs = Sized("Title", "Author")

// typed as List[Sized[_2, List[String]]]
val rows = List(
  Sized("TAPL", "B. Pierce"),
  Sized("Implementation of FP Languages", "SPJ")
)
```

The power of type computation

Computing with implicits is sometimes called "Poor Man's Prolog"

But: Despite the "Poor Man's" part, almost anything can be done

What are macros bringing into the mix?

- Complex type computations are hard to debug (sometimes, `-Xlog-implicits` is not enough)
- Complex type computations often slow down the compiler
- Types don't cover everything, sometimes we need more power

Let's overthrow the tyranny of types!

Macros can do anything, including validation of arguments, so we shouldn't bother with all those complex types anymore

Let's overthrow the tyranny of types!

Macros can do anything, including validation of arguments,
so we shouldn't bother with all those complex types anymore

Bad

```
trait GenTraversableLike[+A, +Repr] {
  def map[B, R](f: A => B)
    (implicit bf: CanBuildFrom[Repr, B, R]): R
}
```

Let's overthrow the tyranny of types!

Macros can do anything, including validation of arguments,
so we shouldn't bother with all those complex types anymore

Bad

```
trait GenTraversableLike[+A, +Repr] {
  def map[B, R](f: A => B)
    (implicit bf: CanBuildFrom[Repr, B, R]): R
}
```

Good

```
trait GenTraversableLike {
  def map(f: Any): Any = macro ...
}
```

Completely replacing types with macros: not a good idea

Ma
so

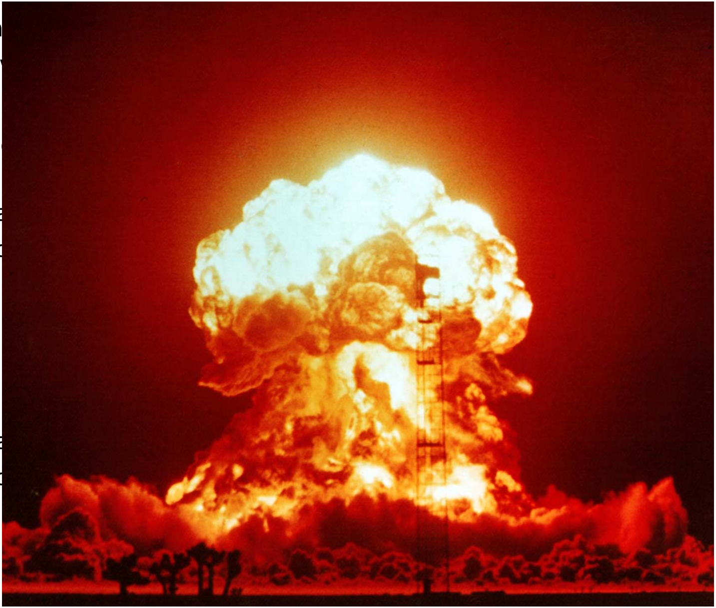
Ba

tra
c

}

Go

tra
c
}



Reasonable use case: Checked arithmetics

Spire provides a `checked` macro to detect arithmetic overflows
Types can't capture this, so it's okay to use a macro here

```
// returns None when x + y overflows
Checked.option {
  x + y < z
}
```

Reasonable use case: WartRemover

@puffnfresh has written a flexible Scala code linting tool that can alert one about questionable coding practices

```
scala> def safe(expr: Any) = macro Unsafe.asMacro
safe: (expr: Any)Any

scala> safe { null }
<console>:10: error: null is disabled
              safe { null }
                     ^
```

Our advice

- For static checks use types whenever practical
- Macros if impossible or heavyweight
- Try to document and encapsulate the magic using types (type classes are particularly nice for this purpose)

Domain-specific languages

Domain-specific languages

As per "DSLs in Action":

- Embedded *aka* internal
- Standalone *aka* external
- Non-textual

Domain-specific languages

As per “DSLs in Action”:

- Embedded *aka* internal $\leftarrow$ in this talk
- Standalone *aka* external
- Non-textual

Use case: Slick

An embedded DSL for data access

Instead of writing database code in SQL

```
select c.NAME from COFFEES c where c.ID = 10
```

Use case: Slick

An embedded DSL for data access

Instead of writing database code in SQL

```
select c.NAME from COFFEES c where c.ID = 10
```

Write database code in Scala

```
for (c <- coffees if c.id == 10) yield c.name
```

Three approaches

- Lifted embedding (types)
- Direct embedding (macros)
- Shadow embedding (macros + types)

Lifted embedding (types)

Types can do domain-specific validation and virtualization

Domain rules are encoded in an extra layer of types

```
object Coffees extends Table[(Int, String, ...)] {
  def id = column[Int]("ID", O.PrimaryKey)
  def name = column[String]("NAME")
  ...
}
```

Lifted embedding (types)

Types are quite heavyweight under the covers

What you write in a Slick DSL

```
Query(Coffees) filter
  (c => c.id === 10) map
  (c => c.name)
)
```

What actually happens under the covers

```
Query(Coffees) filter
  (c => c.id: Column[Int] === 10: Column[Int]) map
  (c => c.name: Column[String])
```

Lifted embedding (types)

Types can be really bad at error messages

Trying to compile

```
Query(Coffees) map (c =>
  if (c.origin === "Iran") "Good"
  else c.quality
)
```

Produces the following error

```
Don’t know how to unpack Any to T and pack to G
not enough arguments for method map: (implicit shape:
slick.lifted.Shape[Any,T,G]) slick.lifted.Query[G,T].
Unspecified value parameter
```

Direct embedding (macros)

Macros can also validate and virtualize Scala code

Type signatures are simple and error messages are to the point

```
case class Coffee(id: Int, name: String, ...)

Query[Coffee] filter
  (c => c.id: Int == 10: Int) map
  (c => c.name: String)
```

Direct embedding (macros)

Macros can do static checks, but sometimes that's non-trivial to get right

Trying to use an unsupported feature

```
Query[Coffee] map (c => c.id.toDouble)
```

Crashes at runtime

This is what we get when we try to reinvent types

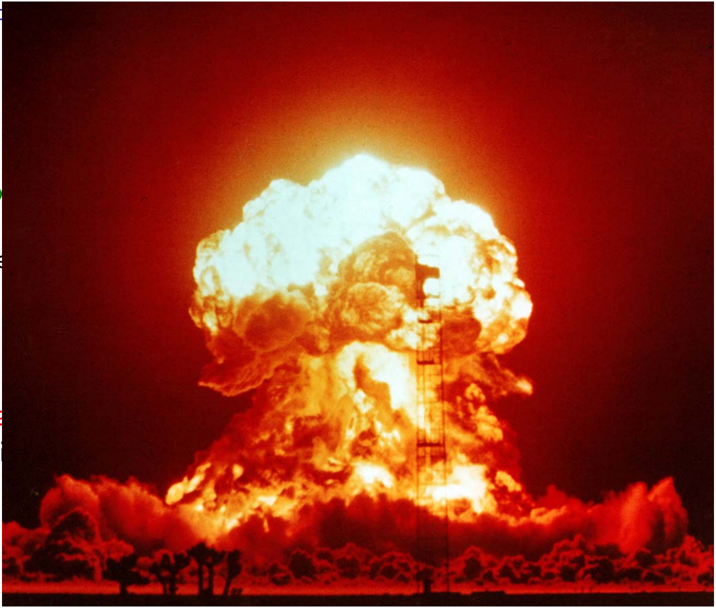
Direct embedding (macros)

Macros c

Try

Que

Cra
Thi



Shadow embedding (macros + types)

Based on YinYang, which uses macros and therefore enjoys all benefits of macros

Type signatures are simple and error messages are to the point

```
case class Coffee(id: Int, name: String, ...)

slick {
  Query[Coffee] filter
    (c => c.id: Int == 10: Int) map
    (c => c.name: String)
  }
}
```

Shadow embedding (macros + types)

Uses types to moderate APIs available inside DSL blocks

DSL author specifies the set of available APIs using types

```
// In Scala's standard library (front-end)
final abstract class Int private extends AnyVal {
  ...
  def toDouble: Double
  ...
}

// In Slick's lifted embedding (back-end)
value toDouble is not a member of Column[Int]
```

Shadow embedding (macros + types)

The best of two worlds

Trying to do something unsupported

```
slick {
  Query[Coffee] map
    (c => c.id.toDouble)
}
```

Produces comprehensible and comprehensive errors

```
in Slick method toDouble is not a member of Int
```

Shadow embedding (macros + types)

An important limitation of the current macro system

Macros can't see ASTs of everything in the program

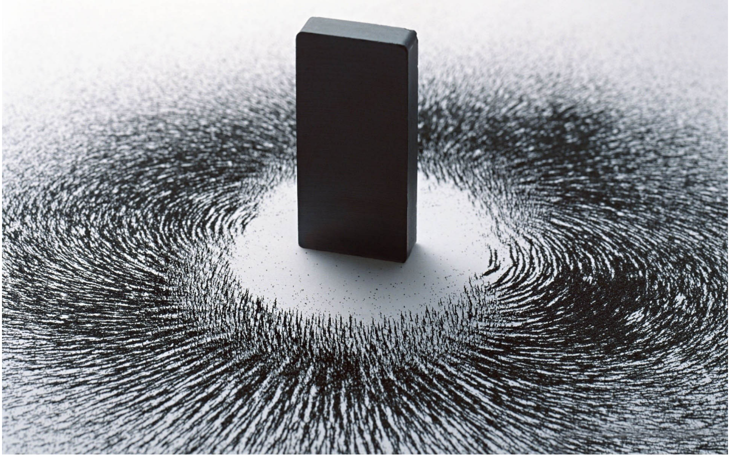
```
def idIsTen(c: Coffee) = c.id == 10

slick {
  Query[Coffee] filter idIsTen
}
```

Our advice

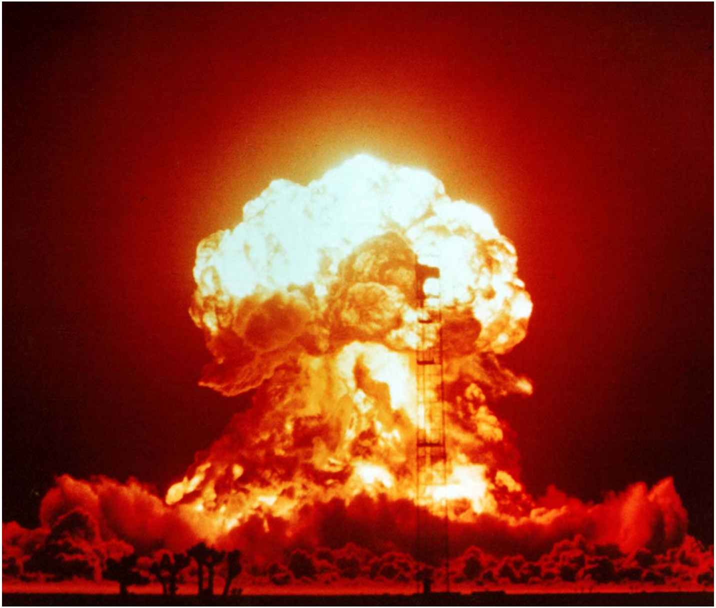
- Types work, but sometimes become too heavyweight both for the DSL author and for the users
- With macros a lot of traditional ceremony is unnecessary, and that makes DSL development faster and more productive
- But: Macros currently have inherent problems with modularity (we're working on this)
- If you decide to go with macros, always try to document and encapsulate macro magic with types as much as possible

Summary

Types are more declarative, but less powerful

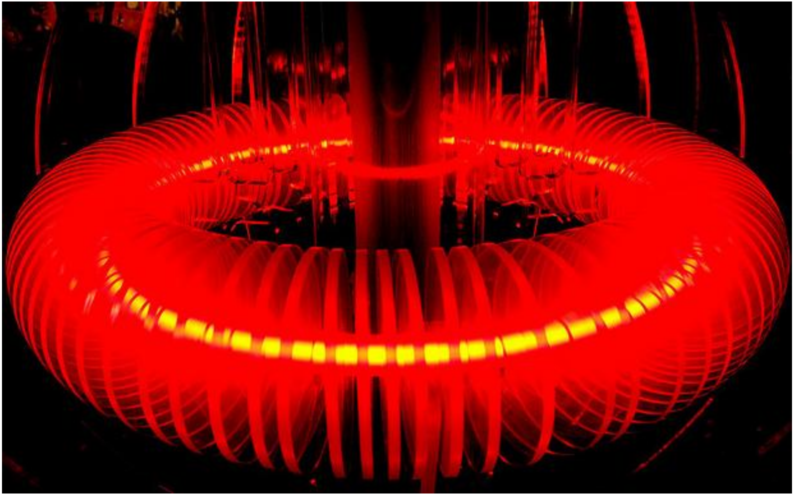


Macros are more powerful, but less declarative



Embrace reason, use whatever's simpler

Also try combining strong points of both



Credits

- Erik Osheim for the Spire article at typelevel
- Amir Shaikhha for the shadow embedding thesis
- Vojin Jovanovic and Stefan Zeiger for DSL help
- Denys Shabalin and others for their comments
- Tom Niemann for the parser generators diagram
- Flickr for the Hanoi towers picture
- wallpapersus.com for the magnet picture
- Wikimedia Commons for the nuclear explosion picture
- Flickr for the fusion reactor picture
- Star Trek for the picture of Spock