Adventures in Meta-Programming

Macros vs Shapeless

Dave Gurnell, @davegurnell

underscore

meta-programming

[**mee**-tuh-**proh**-gram-ing]

noun

1. the practice of writing code that writes code;

domain specific languages

scrapping boilerplate

meta-programming

[**mee**-tuh-**proh**-gram-ing]

noun

1. the practice of writing code that writes code;
2. a pretty big time sink;
3. sometimes an uphill struggle.

when do they
work well?

when do they
work less well?

macros

shapeless

simple applications
of each technique

tips to make things
easier

the punch line...

(spoiler warning)

macros are good
for syntaxy stuff

shapeless is good
for typey stuff

we can do most stuff
with ADTs and type classes

meta-programming provides
a thin layer on top

constructing values

```scala
case class IceCream(
  name: String,
  numCherries: Int,
  inCone: Boolean)


create[IceCream]
// IceCream("", 0, false)
```

macros

```
create[IceCream]
```

```
IceCream("", 0, false)
```

```scala
def create[A]: A =
  macro Macros.createMacro[A]
```

```scala
def createMacro[A: WeakTypeTag]: Tree = {
  val targetType  = weakTypeOf[A]

  val applyMethod = findApplyMethod(targetType)

  val applyParams = applyMethod
    .paramLists.map { paramList =>
      paramList.map { param =>
        createApplyParam(param.typeSignature)
      }
    }

  q"$applyMethod(...$applyParams)"
}
```

```scala
def createApplyParam(paramType: Type): Tree =
  if(paramType <:< typeOf[String]) {
    q""" "" """
  } else if(paramType <:< typeOf[Int]) {
    q"0"
  } else if(paramType <:< typeOf[Boolean]) {
    q"false"
  } else {
    c.abort(c.enclosingPosition, "FAIL!")
  }
```

```
create[IceCream]
```

```
IceCream("", 0, false)
```

analysis

only handles three parameter types

not customisable by the user

scala-reflect is deprecated!

macros v2

```scala
trait Pure[A] {
  def value: A
}

implicit val stringPure: Pure[String] =
  new Pure[String] { def value = "" }

implicit val intPure: Pure[Int] =
  new Pure[Int] { def value = 0 }

implicit val booleanPure: Pure[Boolean] =
  new Pure[Boolean] { def value = false }
```

```scala
def createApplyParam(paramType: Type): Tree =
  q"""
  _root_.scala.Predef
    .implicitly[_root_.Pure[$paramType]]
    .value
  """
```

```
def createApplyParam(paramType: Type): Tree =
  q"""
  _root_.scala.Predef
    .implicitly[_root_.Pure[$paramType]]
    .value
  """

// implicitly[Pure[Foo]].value
```

```
create[IceCream]
```

```
create[IceCream](IceCream(
  implicitly[Pure[String]].value,
  implicitly[Pure[Int]].value,
  implicitly[Pure[Boolean]].value
))
```

analysis

user-customisable and extendable

handles any parameter type

brittle in various ways

shapeless

```scala
trait Pure[A] {
  def value: A
}

implicit val stringPure: Pure[String] =
  new Pure[A] { def value = "" }

implicit val intPure: Pure[Int] =
  new Pure[A] { def value = 0 }

implicit val booleanPure: Pure[Boolean] =
  new Pure[A] { def value = false }
```

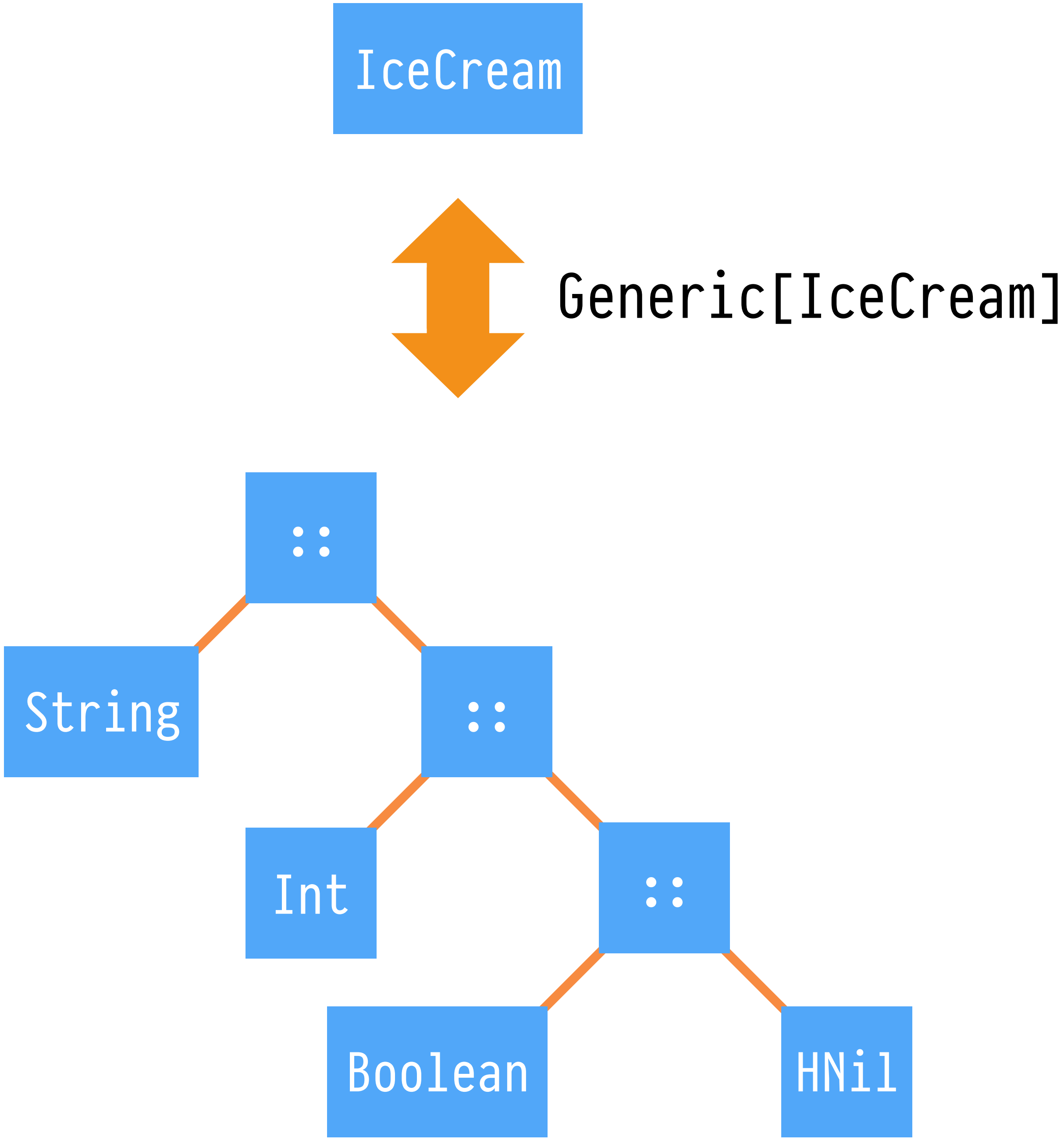
```scala
trait Pure[A] {
  def value: A
}

implicit val stringPure: Pure[String] =
  new Pure[A] { def value = "" }

implicit val intPure: Pure[Int] =
  new Pure[A] { def value = 0 }

implicit val booleanPure: Pure[Boolean] =
  new Pure[A] { def value = false }

implicit def genericPure[A]: Pure[A] =
  ???
```


IceCream
Generic[IceCream]
::
String
::
Int
::
Boolean
HNil

```scala
implicit val hnilPure: Pure[HNil] =
  instance(HNil)

implicit def hconsPure[H, T <: HList](
  implicit
  hPure: Lazy[Pure[H]],
  tPure: Pure[T]
): Pure[H :: T] =
  instance(hPure.value.value :: tPure.value)

implicit def genericPure[A, R](
  implicit
  gen: Generic.Aux[A, R],
  pure: Lazy[Pure[R]]
): Pure[A] =
  instance(gen.from(pure.value.value))
```

analysis

user-customisable

macros are syntactic,
shapeless is structural

compile times!
(see “inductive implicits” in TLS)

what wins?

in this case, IMO shapeless

shapeless’ `Generic` isolates
the meta-programming

50% of the code,
no deprecated APIs

data validation

https://github.com/davegurnell/checklist

```
case class IceCream(
  name: String,
  cherries: Int,
  cone: Boolean)
```

```scala
case class IceCream(
  name: String,
  cherries: Int,
  cone: Boolean)

val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))
```

```scala
case class IceCream(
  name: String,
  cherries: Int,
  cone: Boolean)

val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))

rule(IceCream("", -1, false))
// List(
//   Error("Must be non-empty", List("name")),
//   Error("Must be >= 0",       List("cherries"))
// ))
```

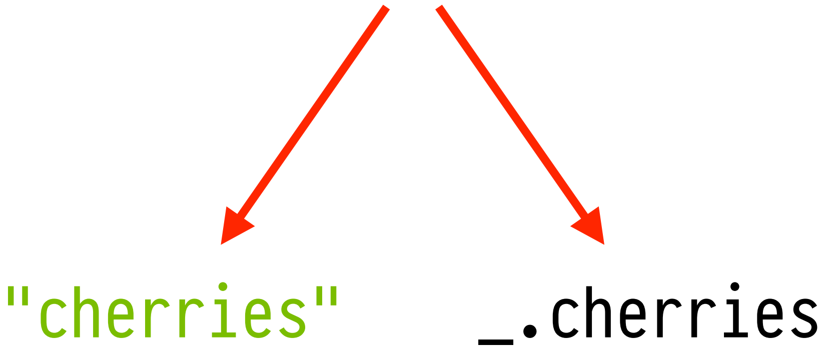
```scala
case class IceCream(
  name: String,
  cherries: Int,
  cone: Boolean)

val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))

rule(IceCream("", -1, false))
// List(
//   Error("Must be non-empty", List("name")),
//   Error("Must be >= 0",      List("cherries"))
// ))
```

macros

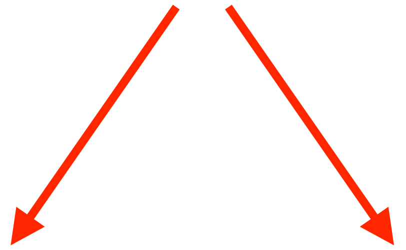
```
val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))
```



"cherries"

_.cherries

```
val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))
```



"cherries"   _.cherries

```scala
val rule = gte(0)

rule.apply(-1)
// List(Error("Must be >= 0", Nil))
```

```
val rule = gte(0)
  .prefixed("cherries")

rule.apply(-1)
// List(Error("Must be >= 0", List("cherries")))
```

```scala
val rule = gte(0)
  .prefixed("cherries")
  .contramap[IceCream](_.cherries)

rule.apply(IceCream("Sundae", -1, false))
// List(Error("Must be >= 0", List("cherries")))
```

```scala
val rule = Rule[IceCream]
  .field(_.name)(nonEmpty)
  .field(_.cherries)(gte(0))
```

```scala
val rule = Rule[IceCream]
  .and(nonEmpty
    .prefixed("name")
    .contramap[IceCream](_.name))
  .and(gte(0)
    .prefixed("cherries")
    .contramap[IceCream](_.cherries))
```

```scala
trait Rule[A] {
  def apply(value: A): List[Error]

  def field(func: A => B)(rule: Rule[B]): Rule[A] =
    macro Macros.fieldMacro(func)(rule)
}
```

```scala
def fieldMacro(func: Tree)(rule: Tree): Tree = {
  val name = func match {
    case q"($param) => $obj.$name" =>
      q"${name.toString}"
    case other =>
      c.abort(c.enclosingPosition, "FAIL!")
  }

  q"""
  ${c.prefix}.and(
    $rule
      .prefixed($name)
      .contramap($func)
  )
  """
}
```

```
Rule[IceCream]
  .field(_.cherries)(gte(0))
```

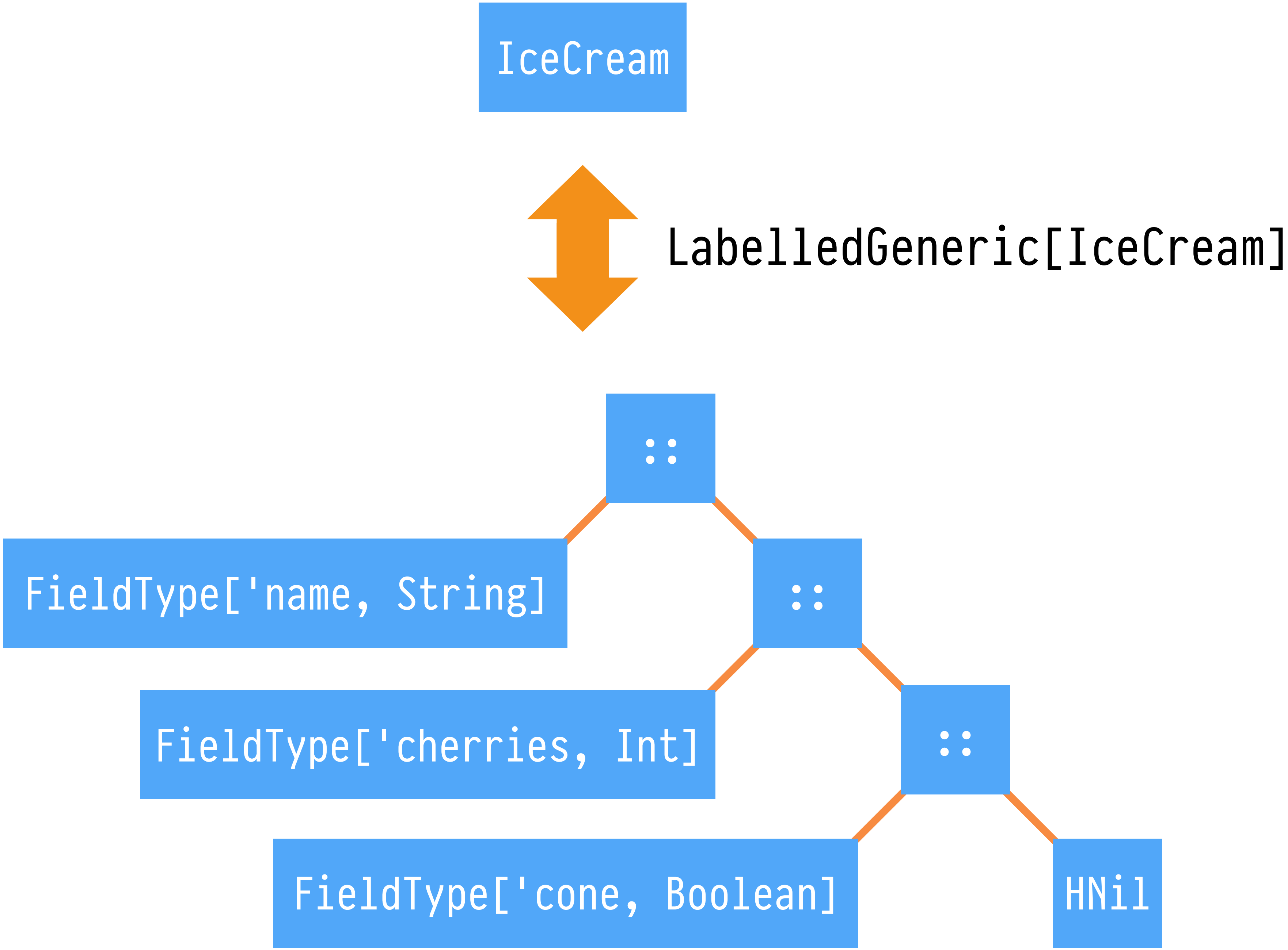
```
Rule[IceCream].and(
  get(0)
    .prefixed("cherries")
    .contramap[IceCream](_.cherries)
)
```

analysis

syntactic solution
to a syntactic problem

code is short and simple
(easy to maintain/replace)

shapeless



IceCream

LabelledGeneric[IceCream]

::

FieldType['name, String]

::

FieldType['cherries, Int]

::

FieldType['cone, Boolean]

HNil

```scala
val rule = Rule[IceCream]
  .field('name)(nonEmpty)
  .field('cherries)(gte(0))
```

```
trait Rule[A] {
  def apply(value: A): List[Error]

  def field[B]
      (field: Witness)
      (rule: Rule[B])
      (implicit wrap: HasField[A, field.T, B]): Rule[A] =
    this.and(wrap(rule))
}
```

```scala
/**
 * Proof that an object of type A
 * has a field of type B named K.
 */
trait HasField[A, K, B] {
  def name: String
  def zoom(value: A): B

  def apply(rule: Rule[B]): Rule[A] =
    rule.prefixed(name).contramap(zoom)
}
```

```scala
implicit def hlistHasField[L <: HList, K, F](
  implicit
  ev: K <:< Symbol,
  witness: Witness.Aux[K],
  selector: Selector.Aux[L, K, F]
): HasField[L, K, F] =
  new HasField[L, K, F] {
    val name = witness.value.name
    def zoom(value: L): F = selector(value)
  }

implicit def genericHasField[A, L, K, F](
  implicit
  ev: K <:< Symbol,
  gen: LabelledGeneric.Aux[A, L],
  hf: HasField[L, K, F]
): HasField[A, K, F] =
  new HasField[A, K, F] {
    val name = hf.name
    def zoom(value: A): F = hf.zoom(gen.to(value))
  }
```

what wins?

in this case, IMO macros

short code,
easy to maintain/replace

syntactic problem,
syntactic solution

summary

macros are good
for syntactic stuff

shapeless is good
for structural stuff

both of our solutions
were 90% regular scala
and 10% meta-programming

meta-programming is
a convenience
not a solution

macros

Dave Gurnell - Macros for the Rest of Us
https://www.youtube.com/watch?v=ZVYdiAudr-I

Tomer Gabel - Leveraging Scala Macros for Better Validation
https://www.youtube.com/watch?v=Li19Cif7uS8

Chris Birchall - Meta-Program and/or Shapeless all the Things!
https://skillsmatter.com/skillscasts/9294

shapeless

Dave Gurnell - The Type Astronaut’s Guide to Shapeless
http://underscore.io/books/shapeless-guide

Dave Gurnell - Establishing Orbit with Shapeless
https://skillsmatter.com/skillscasts/9136

Sam Halliday - Shapeless for Mortals
https://skillsmatter.com/skillscasts/6875

functional design

Noel Welsh - Six Core Principles for Learning Scala
https://www.youtube.com/watch?v=J8wUy1XxL5o

Dave Gurnell - Functional Data Validation (Part 1)
https://skillsmatter.com/skillscasts/5837

Dave Gurnell - Functional Data Validation (Part 2)
https://www.youtube.com/watch?v=0DPGpyt6joE

thank you

https://github.com/davegurnell/macros-vs-shapeless

underscore

bonus macro!

unindent

https://github.com/davegurnell/unindent

```
val lorem =
  s"""
  |Lorem ipsum
  |dolor sit amet
  |consectetur
  """.trim.stripMargin

println(lorem)
// "Lorem ipsum\ndolor sit\nconsectetur"
```

```
import unindent._

val lorem =
  i"""
  Lorem ipsum
  dolor sit amet
  consectetur
  """

println(lorem)
// "Lorem ipsum\ndolor sit\nconsectetur"
```