Lambda World 1 October 2016

Room to Grow

Evolving functional programming languages

Erik Osheim (@d6)

stripe

who am i?

- **typelevel** member λ
- maintain **spire**, **cats**, and other scala libraries
- interested in expressiveness **and** performance ☯
- support machine learning at **stripe**

code at *http://github.com/non*

what is this talk about?

- growing a functional programming language
- informed by work in scala
- (but hopefully somewhat general)
- trying to explain how we got here
- and to motivate future work

methodology ☼

- focused on surface-level interface and ergonomics
- less concerned with *Programming language theory (PLT)*
- (mostly because I'd be out of my depth!)
- generalizing ~5 years of work across several projects
- ingest salt as necessary

what is it that makes fp great?

Many things.

I'd suggest starting with *referential transparency*, enabling:

- type-driven development
- context independence
- equational reasoning
- parametricity

referential transparency?

Expressions of type A evaluate to *values* of type A.

```
scala> "lambda.world".split('.').size
res0: Int = 2
```

Big idea: replace "pure" expressions with their results.

referential transparency?

```
scala> "lambda.world".split('.').size * 2 + 1
res0: Int = 5

scala> 2 * 2 + 1
res1: Int = 5

scala> 5
res2: Int = 5
```

Given *RT*, these are all equivalent.
(They can be substituted for one another.)

referential transparency?

Given an `IO` type and the following:

```
def launchTheRocket(): IO[Unit]
def bindle(xs: List[Double]): Double
def spindle(xs: List[Double]): IO[Double]
```

We can assume that `bindle` does not call `launchTheRocket()`.

(Some restrictions apply.)

referential transparency?

Restrictions:

- No mutation.
- No `.unsafePerformIO`
- No trickiness with threads, globals, etc.
- No fun :P

Any of these may result in a *breach of contract*.

why is substitution so important?

Productivity gains come from solving many problems once:

- Correctly and efficiently.
- Without unnecessary complexity.
- In a reusable way.

☁ ☁ ☀ ☁ ☁

The dream is solving the "software crisis" (Dijkstra, 1972).

why is substitution so important?

Referentially-transparent substitution supports:

- Refactoring.
- Design patterns that don't leak.
- Abstract common parts of any[1] set of expressions (DRY).
- Reducing context required for changes.
- Makes "risky" changes obvious.

---

[1] Assuming the types line up.

what was that about types?

Without types we'd only care about the shapes of expressions, and abstraction (assuming *RT*) is trivial.

Enter Lisp macros[2]:

```
(defmacro for-loop [[sym init check change :as params] & steps]
  `(loop [~sym ~init value# nil]
     (if ~check
       (let [new-value# (do ~@steps)]
         (recur ~change new-value#))
       value#)))
```

---

[2] Lisp: ♪ *99 problems but a macro ain't one.* ♪

types make this more difficult?

Dynamically-typed Python code:

```
class Dog(object):
  ks = ['name', 'breed', 'age', 'weight', 'wellTrained']
  def __init__(self, **kw):
    for k in ks: self.__setattr__(k, kw[k])

  def encode(self):
    d = {k, self.__getattr__(k) for k in ks}
    return json.dumps(d)
```

types make this more difficult?

Statically-typed Scala code:

```
case class Dog(
    name: String,
    breed: String,
    age: Int,
    weight: Double,
    wellTrained: Boolean) {

  def encode(d: Dog): Json = Json.encodeMap(Map(
    "name"        -> Json.encodeString(d.name),
    "breed"       -> Json.encodeString(d.breed),
    "age"         -> Json.encodeInt(d.age),
    "weight"      -> Json.encodeDouble(d.weight),
    "wellTrained" -> Json.encodeBoolean(d.wellTrained)))
}
```

types make this more difficult?

Statically-typed Haskell code:

```
data Dog = Dog {
  name :: String,
  breed :: String,
  age :: Int,
  weight :: Double,
  wellTrained :: Bool
}

encodeDog :: Dog -> Json
encodeDog d = encodeAssoc [
  ("name",        encodeString(name d)),
  ("breed",       encodeString(breed d)),
  ("age",         encodeInt(age d)),
  ("weight",      encodeDouble(weight d)),
  ("wellTrained", encodeBool(wellTrained d))
]
```

types make this more difficult?

Yes. Types do make this kind of abstraction harder.

- need to transcend type-casing/pattern-matching.
- requires *type parameters*
- motivates things such as:
  - *type classes*
  - *type members*
  - *path-dependent types*
  - (ultimately *shapeless* and beyond ⛄)

but types are great!

$$\vdash \Lambda\alpha.\lambda x^{\alpha}.x : \forall\, \alpha.\alpha \to \alpha$$

strategies against boilerplate

We'll look at two strategies for dealing with this kind of boilerplate:

1. *Extensions*: "creating" new language features.
    - Language pragmas (à la GHC)
    - Compiler plugins
    - Macros
    - ~~Reflection~~ ☠☠☠
2. *Encodings*: constructing new abstractions using existing language.
    - Everything else (more or less)

The code snippets you are about to see are true.

Only the names have been changed to protect the innocent… and the unsound.

let's take a whirlwind tour through syntax

⚠️ Don't panic! ⚠️

type lambdas

Before (type lambda *encoding*):

```
new Monad[({type λ[α] = WriterT[F, L, α]})#λ] { ... }

new TraverseFilter[({type λ[α] = Map[K, α]})#λ]
  with FlatMap[({type λ[α] = Map[K, α]})#λ] { ... }

trait KleisliSemigroupK[F[_]]
  extends SemigroupK[({ type λ[α] = Kleisli[F, α, α] })#λ]
  { ... }
```

type lambdas

After (*kind-projector* supplies type lambda syntax):

```
new Monad[WriterT[F, L, ?]] { ... }

new TraverseFilter[Map[K, ?]]
  with FlatMap[Map[K, ?]] { ... }

trait KleisliSemigroupK[F[_]]
  extends SemigroupK[λ[α => Kleisli[F, α, α]]]
  { ... }
```

natural transformations

Scala lacks anonymous polymorphic functions.

But we can encode them using traits:

```
trait ~>[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}

val natTrans: Vector ~> List = ...
```

natural transformations

Before (raw polymorphic lambda encoding):

```
def injectFC[F[_], G[_]](implicit I: Inject[F, G]) =
  new (FreeC[F, ?] ~> FreeC[G, ?]) {
    def apply[A](fa: FreeC[F, A]): FreeC[G, A] =
      fa.mapSuspension[Coyoneda[G, ?]](
        new (Coyoneda[F, ?] ~> Coyoneda[G, ?]) {
          def apply[B](fb: Coyoneda[F, B]): Coyoneda[G, B] = fb.trans(I)
        }
      )
  }
```

natural transformations

After (*kind-projector* supplies polymorphic lambda syntax):

```
def injectFC[F[_], G[_]](implicit I: Inject[F, G]) =
  λ[FreeC[F, ?] ~> FreeC[G, ?]](
    _.mapSuspension(λ[Coyoneda[F, ?] ~> Coyoneda[G, ?]](_.trans(I)))
  )
```

whew, ok.

type classes

These methods read identically but the types are unrelated:

```
def minDoubles(xs: List[Double]): Option[Double] =
  x match {
    case Nil => None
    case h :: t => Some(t.foldLeft(h)(_ min _))
  }

def minDecimals(xs: List[BigDecimal]): Option[BigDecimal] =
  x match {
    case Nil => None
    case h :: t => Some(t.foldLeft(h)(_ min _))
  }

minDoubles(1.0 :: -0.0 :: 0.0 :: 3.0 :: Nil)
minDecimals(BigDecimal("3.33") :: BigDecimal("4.33") :: Nil)
```

type classes

We encode a type class pattern using implicit parameters:

```
def minGeneric[A](xs: List[A])(implicit o: Order[A]): Option[A] =
  x match {
    case Nil => None
    case h :: t => Some(t.foldLeft(h)(o.min))
  }

minGeneric(1.0 :: -0.0 :: 0.0 :: 3.0 :: Nil)
minGeneric(BigDecimal("3.33") :: BigDecimal("4.33") :: Nil)

// Order[Double] and Order[BigDecimal] instances not shown
```

does scala have type classes?

- Ed Kmett might say not really.
- I would say sure it does.
- Or at least something analogous (interfaces à la Idris?)
- Is an encoding of a type class a type class? ♲
- Either way you won't find them in the *SLS*[3].

---

[3] Scala Language Specification

how is Order[A] encoded?

Here's the type class encoding for `Order`:

```
trait Order[A] {
  def min(x: A, y: A): A
}

object Order {
  def apply[A](implicit ev: Order[A]): Order[A] = ev

  object ops {
    implicit class OrderOps[A](x: A)(implicit o: Order[A]) {
      def min(y: A): A = o.min(x, y)
    }
  }
}
```

wow, pretty ugly, huh?

The encoding definition is a bit intense:

- Reader needs to recognize encoding.
- A fair bit of machinery to remember.
- Usually results in net reduction in boilerplate.
- But undeniably somewhat ugly.
- Write enough of these and it feels like Java.

can we improve this?

Sure!

To do this we will need to "extend" the language.

(Using macros.)

what does simulacrum do?

```
import simulacrum._

@typeclass trait Semigroup[A] {
  @op("|+|") def append(x: A, y: A): A
}
```

That's it! Better?

a likeness or imitation

Indeed, *simulacrum* adds pseudo-syntax for type classes.

- Removes boilerplate and repitition
- Improves readability
- ...if you're familiar with the encoding!
- If not, the meaning can be obscure.

even more about type classes

Before (a more realistic implicit operator class):

```
final class PartialOrderOps[A](lhs: A)(implicit ev: PartialOrder[A]) {
  def >(rhs: A): Boolean = ev.gt(lhs, rhs)
  def >=(rhs: A): Boolean = ev.gt(lhs, rhs)
  def <(rhs: A): Boolean = ev.gt(lhs, rhs)
  def <=(rhs: A): Boolean = ev.gt(lhs, rhs)

  def compare(rhs: A): Int = ev.compare(lhs, rhs)
  def min(rhs: A): A = ev.min(lhs, rhs)
  def max(rhs: A): A = ev.max(lhs, rhs)
}
```

(*simulacrum* could generate all of this.)

even more about type classes

After (using *machinist* to optimize implicit enrichment):

```
final class PartialOrderOps[A](lhs: A)(implicit ev: PartialOrder[A]) {
  def >(rhs: A): Boolean = macro Ops.binop[A, Boolean]
  def >=(rhs: A): Boolean = macro Ops.binop[A, Boolean]
  def <(rhs: A): Boolean = macro Ops.binop[A, Boolean]
  def <=(rhs: A): Boolean = macro Ops.binop[A, Boolean]

  def partialCompare(rhs: A): Double = macro Ops.binop[A, Double]
  def tryCompare(rhs: A): Option[Int] = macro Ops.binop[A, Option[Int]]
  def pmin(rhs: A): Option[A] = macro Ops.binop[A, A]
  def pmax(rhs: A): Option[A] = macro Ops.binop[A, A]
}
```

Wait, isn't that worse?

yes, but…

- *machinist* is not about increasing expressiveness.
- macro re-routes method calls from implicit classes
- decreases the cost of the implicit operator encoding.
- with fast/primitive operators: ~5x speed ups are possible.

(Future: *machinist* and *simulacrum* team up to fight crime.)

fixing bugs

See also the SI-2712 compiler plugin.

- Fixes longstanding Scala bug
- Enables inference of type constructors (à la *kind-projector*)
- Hopefully only necessary for a short time.
- (Arguably this isn't so much an extension as a "hotfix")

so extensions are great?

Extensions are a clear way to evolve the language.

- Support new abstractions
- Remove ugly boilerplate
- Improve compiler performance
- Great, right?

Well…

the other side

Downsides:

- Use up syntactic/semantic "space"
- Somewhat opaque / hard to google
- Often apply globally (esp. compiler plugins)
- Usually orthogonal (or even incompatible)
- Analogous to "jargon"

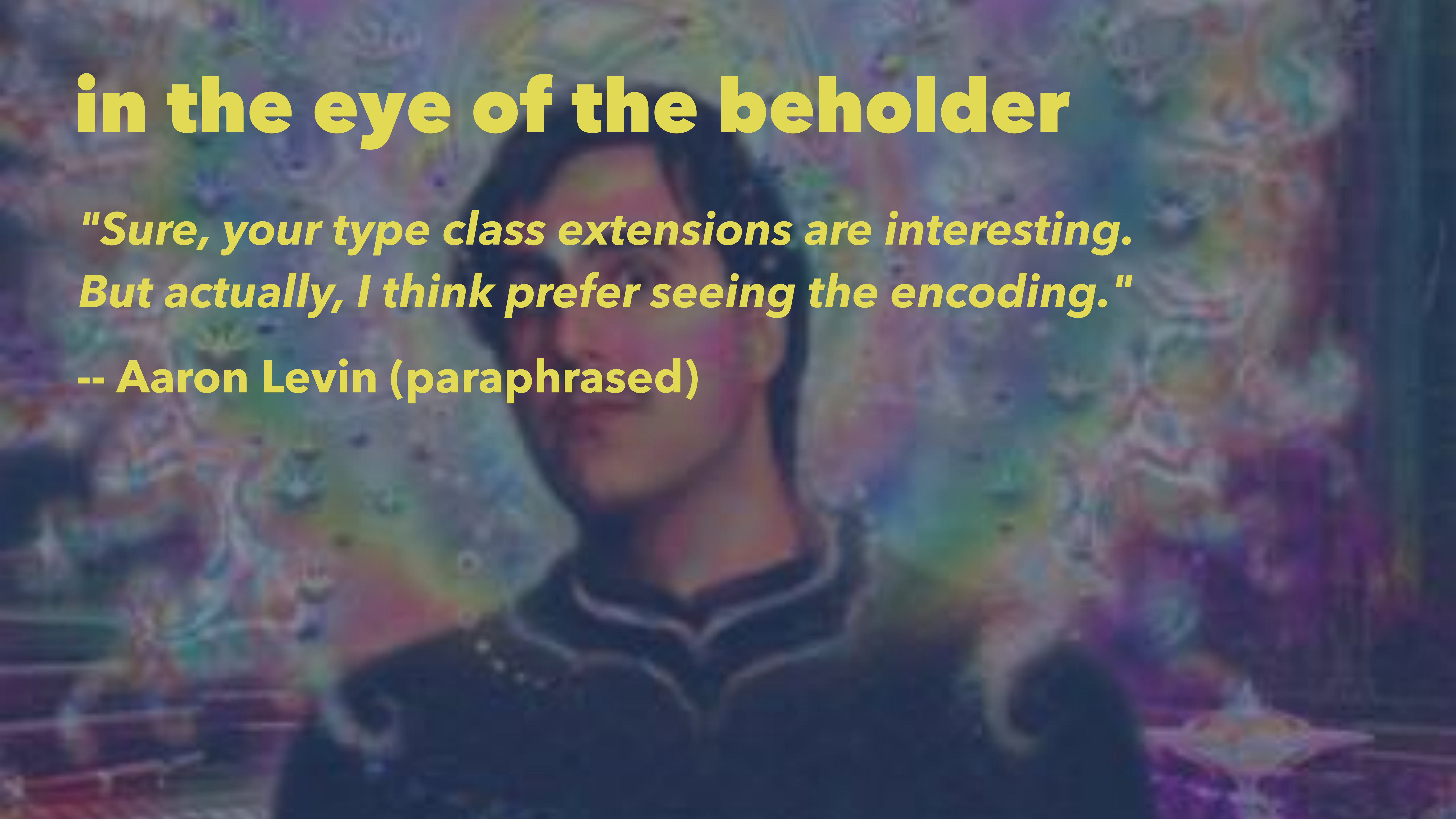
and also...

in the eye of the beholder

in the eye of the beholder

"Sure, your type class extensions are interesting. But actually, I think prefer seeing the encoding."

-- Aaron Levin (paraphrased)



encodings as pedagogy

Encodings have pedagogical value:

- Express new ideas in familiar language semantics.
- Usually provide a smoother ramp for newcomers.
- Offer flexibility in implementation.
- Encodings are often more composable (vs e.g. macros)

(These were not points I had really considered.)

trade-offs

Extensions are better when:

```
* Encodings aren't possible
* Encodings are too horrible to use
* Concepts are ubiquitous enough
* Collateral damage is minimal
```

Encodings are better when:

```
* Flexibility is needed
* No broad agreement on design
* Minimize disruption to third-parties
* Encourage compositionality
```


let's revisit the dog example

cast back your mind

Slightly more idiomatic Dog type with JSON encoder:

```
case class Dog(
  name: String, breed: String, age: Int, weight: Double, wellTrained: Boolean)

object Dog {
  implicit val dogEncoder: Encoder[Dog] =
    Encoder.instance { (d: Dog) =>
      Json.obj(
        "name"        -> Encoder[String](d.name),
        "breed"       -> Encoder[String](d.breed),
        "age"         -> Encoder[String](d.age),
        "weight"      -> Encoder[String](d.weight),
        "wellTrained" -> Encoder[String](d.wellTrained))
    }
}
```

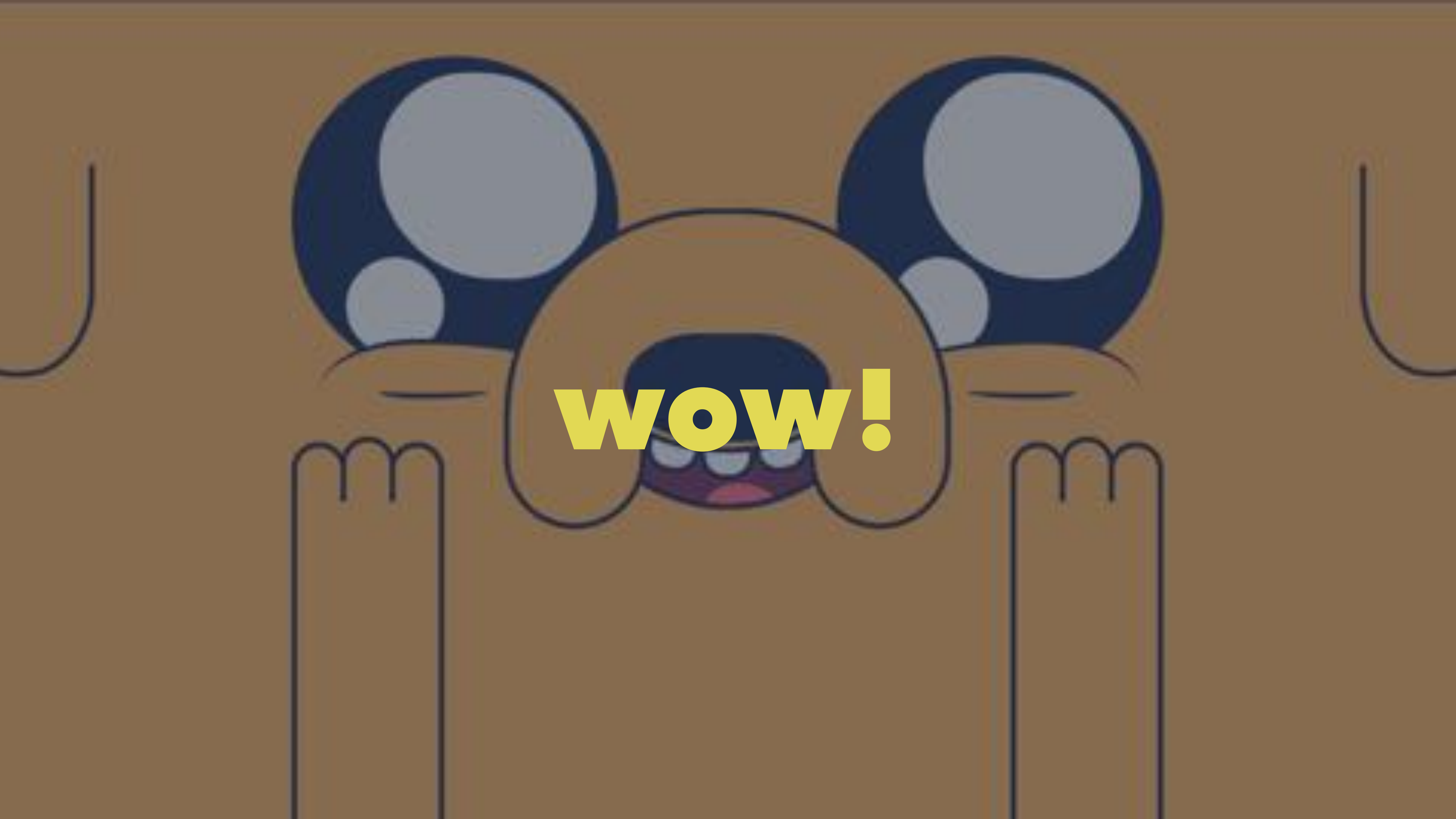
cast back your mind

Shazam!

```
import io.circe._, io.circe.generic.semiauto._

case class Dog(
  name: String, breed: String, age: Int, weight: Double, wellTrained: Boolean)

object Dog {
  implicit val dogEncoder: Encoder[Dog] = deriveEncoder
}
```


wow!

magic dogs

How does this work?

- `deriveEncoder` comes from *circe*
- use *shapeless* (plus a few macros)
- compile-time reflection
- derives types/values from type-level representation
- inspired by "Scrap your boilerplate", Idris, etc.

encoding or extension?

Shapeless makes heavy use of much of Scala's type system.

- Implicit search
- Type constructors and type members
- Path-dependent types
- Singleton types
- Existintial types
- ...and **macros**

encoding or extension?

Shapeless is both!

- Initially just encodings for `HList`, `Poly`, `Nat`, etc.
- Eventually broke down and used a macro.
- Continues to mostly stick to Scala proper.
- (But every time I look there are more macros!)

what did we learn?

encodings

- Draws on existing language knowledge
- Maps the boundaries of the language
- Preserves "space" -- plus flexibility to change
- Short-term payoff in terms of functionality.
- Established encodings: candidates for future extensions.

extensions

- Theoretically creates a new language
- (But often just involves tree rewriting or similar)
- Higher learning curve, but better ergonomics.
- Longterm investment in language.
- With great power comes great responsibility.
- (Also power corrupts, I think?)

so go forth and grow!

(in an ecologically sustainable way.)

Lambda World 1 October 2016

the end

Erik Osheim (@d6)

stripe