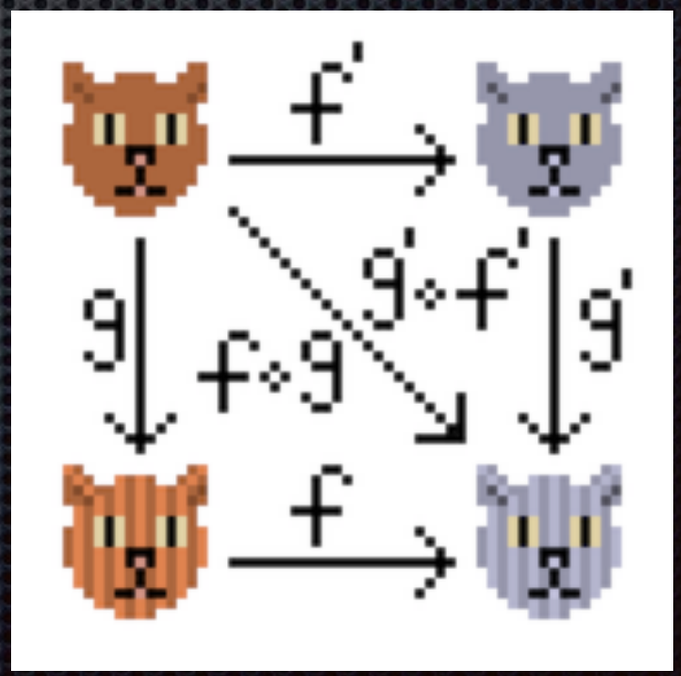
The Internet Was Made for Cats

The Typelevel Cats Project



Chicago-Area Scala Enthusiasts
Jan 21, 2016

Dean Wampler (not shown)

@deanwampler

github.com/deanwampler

Why Cats?

- “Provide a lightweight, modular, and extensible library that is approachable and powerful.”

Approachable

- Keep Cats approachable for people new to the concepts in this library.

Modular

- A tight core with only:
  - type classes.
  - bare minimum data structures they need.
  - type class instances for those data structures and standard library types.

Type classes

```
package cats.will.play.data

/**
 * A simple Complex number class.
 */
case class Complex(real: Double, imag: Double)
```

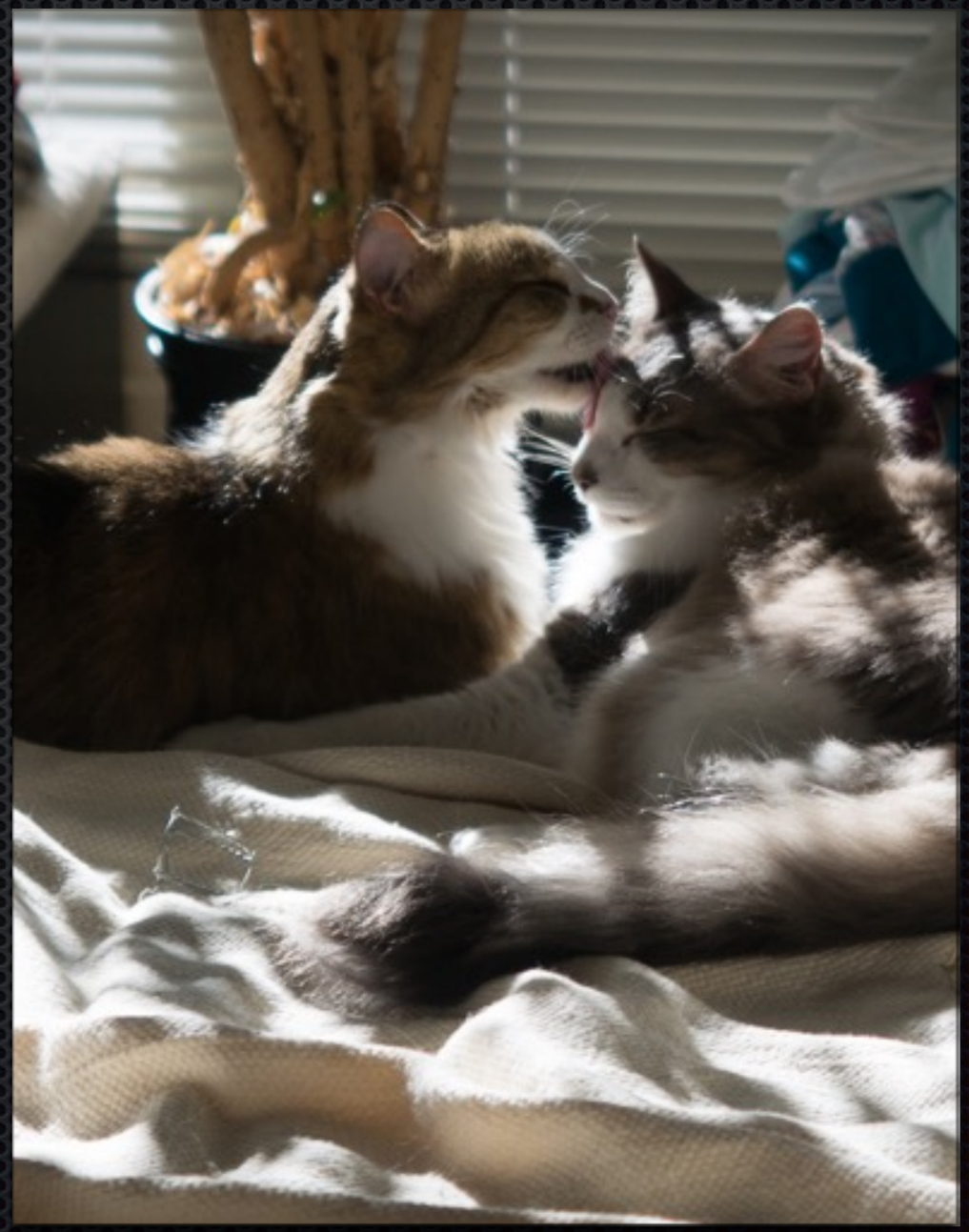
Type classes

```
import cats.will.play.data._

trait ToJSON[A] {
  def toJSON: String
}

val c = Complex(1.1, 2.2)
c.toJSON    // ERROR: no toJSON method!
```

```
implicit class ComplexToJSON(c: Complex)
    extends ToJSON[Complex] {
  def toJSON: String =
    s"""{"real": ${c.real}, "imag": ${c.imag}}"""
}

c.toJSON    // Success!
```

Documented

- Good documentation an essential project goal.
- Important for approachability.
  - Code well documented.
  - External documentation.
  - Large corpus of compiler verified examples of how the software can be used.
- A good place to start contributing!



Efficient

- Pure functions and writing efficient code in Scala can be at odds.
- Attempting to keep Cats as efficient as possible without unnecessary sacrifices of purity and usability.
- Where sacrifices are made, strive to make these obvious and well documented.

In the Cat House...

Getting Started

- non.github.io/cats - getting started, e.g., tutorial
- Scaladocs
- `"org.spire-math" %% "cats" % "0.3.0"`

What's Category Theory?

- "The study of abstract algebras of functions."

- Category Theory, 2nd Ed. Steve Awodey

What's a Category? (1/2)

- A class of *objects* (not in the OOP sense).
- A class of *morphisms* (a.k.a. maps or arrows).
  - f: a → b

What's a Category? (2/2)

- A binary operation ∘ for composition:
  - Associative
    - $(a \circ b) \circ c = a \circ (b \circ c)$
  - Identity
    - $1_x: x \to x$
    - for $f: a \to b$
      - $1_a \circ f = f \circ 1_b$

Functor - “map”

Functor

```
scala> Seq(1,2,3,4,5).map(i => i.toDouble)
res1: Seq[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)

scala> Vector(1,2,3,4,5).map(i => i.toDouble)
res2: scala.collection.immutable.Vector[Double] =
Vector(1.0, 2.0, 3.0, 4.0, 5.0)

scala> Option(1).map(i => i.toDouble)
res3: Option[Double] = Some(1.0)
```

Functor

- But this `object.method(...)` format obscures something profound...

Functor

```
trait Functor[A, B, G[A], H[B]] {
  def mapf(f: A => B): G[A] => H[B]
}

class SeqSetF[A,B] extends Functor[A,B,Seq,Set] {
  def mapf(f: A => B): Seq[A] => Set[B] =
    seq => Set(seq.map(f) :_*)
}
```

A Functor is morphism between categories.
It transforms object morphisms.

Functor

```
scala> val ss = new SeqSetF[Int,(Int,String)]
ss: SeqSetF[Int,(Int, String)] = SeqSetF@79c85f41

scala> val fss = ss.mapf{
  i => val i3 = i%3; (i3, i3.toString)}
fss: Seq[Int] => Set[(Int, String)] = <function1>

scala> fss(1 to 10)
Set[(Int, String)] = Set((1,1), (2,2), (0,0))
```

Functor: Properties (“Laws”)

- Identity: For categories G and H:
  - for all objects x in G Functor(I_x) → $I_{Functor(x)}$
- For all f1: a → b and f2: b → c
  - Functor(f1 ∘ f2) = Functor(f1) ∘ Functor(f2)

Functor in Cats

```
import cats._  // e.g., Functor[F[_]]

scala> implicit val listFunctor: Functor[List] =
     | new Functor[List] {
     |   def map[A,B](fa: List[A])(f: A => B):List[B] =
     |     fa map f
     | }
listFunctor: cats.Functor[List] = $anon$1@634cd876

scala> val len: String => Int = _.length
len: String => Int = <function1>

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res: List[Int] = List(4, 5)
```

Additional Methods

```
val len: String => Int = _.length

val lenList: List[String] => List[Int] =
  Functor[List].lift(len)
```

```
scala> val list = List("qwer", "adsfg")
scala> lenList(list)
res: List[Int] = List(4, 5)
```

Lift a function from A => B to F[A] => F[B]

Additional Methods

```
scala> Functor[List].fproduct(list)(len).toMap
res: Map(qwer -> 4, adsfg -> 5)
```

Pair up a with f(a), b with f(b), …

Additional Methods

```
scala> implicit val listFunctor: Functor[List] =
     | new Functor[List] {
     |   def map[A,B](fa: List[A])(f: A => B) =
     |     fa map f
     | }
```

```
scala> implicit val optFunctor: Functor[Option] =
     | new Functor[Option] {
     |   def map[A,B](fa: Option[A])(f: A => B):
     |     Option[B] = fa map f
     | }
```

Compose functor F[_] and functor G[_] to create a new functor F[G[_]]

Additional Methods

```
scala> val listOpt = Functor[List] compose Functor[Option]
listOpt: cats.Functor[[X]List[Option[X]]] = …

scala> listOpt.map(List(Some(1), None, Some(3)))(_ + 1)
res: List[Option[Int]] = List(Some(2), None, Some(4))
```

```
scala> val optList = Functor[Option] compose Functor[List]
optList: cats.Functor[[X]Option[List[X]]] = …

scala> optList.map(Some(List(1, 2, 3)))(_ + 1)
res: Option[List[Int]] = Some(List(2, 3, 4))
```

Apply

Apply, the Feline Way

- Extends Functor with an “ap” method
- Functor map: A → B
- Apply ap: F[A → B]

Apply

```
implicit val optionApply: Apply[Option] = new Apply[Option] {
  def ap[A, B](fa: Option[A])(f: Option[A => B]): Option[B] =
    fa.flatMap (a => f.map (ff => ff(a)))

  def map[A,B](fa: Option[A])(f: A => B):Option[B] = fa map f
}
```

```
implicit val listApply: Apply[List] = new Apply[List] {
  def ap[A, B](fa: List[A])(f: List[A => B]): List[B] =
    fa.flatMap (a => f.map (ff => ff(a)))

  def map[A,B](fa: List[A])(f: A => B): List[B] = fa map f
}
```

Apply

```
val intToString: Int => String = _.toString
val double: Int => Int = _ * 2
val addTwo: Int => Int = _ + 2

scala> Apply[Option].ap(Some(1))(Some(intToString))
res: Option[String] = Some(1)

scala> Apply[Option].ap(Some(1))(Some(double))
res: Option[Int] = Some(2)

scala> Apply[Option].ap(None)(Some(double))
res: Option[Int] = None

scala> Apply[Option].ap(Some(1))(None)
res: Option[Nothing] = None

scala> Apply[Option].ap(None)(None)
res: Option[Nothing] = None
```

Apply

```
val countVowels =
 (s:String) => s.replaceAll("[^aeiou]*", "").length

val list = List("qwer", "adsfg", “foobar")

scala> Apply[List].ap[String,Int](list)(List(_.length))
res: List[Int] = List(4, 5, 6)

scala> Apply[List].ap[String,Int](list)(
     |   List(_.length, s=>countVowels(s)))
res: List[Int] = List(4, 1, 5, 1, 6, 3)

scala> Apply[List].ap[String,String](list)(
     |   List(_.toLowerCase, _.toUpperCase))
res: List[String] = List(qwer, QWER, adsfg, ADSFG, foobar, FOOBAR)
```

Additional Methods

```
val add2 = (a: Int, b: Int) => a + b
val add3 = (a: Int, b: Int, c: Int) => a + b + c

scala> Apply[Option].ap2(Some(1), Some(2))(
     |   Some(add2))
res: Option[Int] = Some(3)

scala> Apply[Option].ap3(Some(1), Some(2),
     |   Some(3))(Some(add3))
res: Option[Int] = Some(6)
```

Functor's map and compose, plus ap2, ap3, ..., ap22

Additional Methods

```
scala> Apply[Option].map3(Some(1), Some(2),
     |   Some(3))(Some(add3))
res: Option[Int] = Some(6)
```

```
scala> Apply[Option].tuple3(Some(1), Some(2),
     |   Some(3))
res: Option[(Int, Int, Int)] = Some((1,2,3))
```

map2, map3, ..., map22
tuple2, tuple3, ..., tuple22

Additional Methods

```
import cats.syntax.apply._

def f1(a: Option[Int], b: Option[Int]) =
    (a |@| b) map (_ * _)

def f2(a: Option[Int], b: Option[Int]) =
    Apply[Option].map2(a, b)(_ * _)

f1(Some(2), Some(3))  // Some(6)
f2(Some(2), Some(3))  // Some(6)
f1(None,    Some(3))  // None
```

Builder syntax: f1 and f2 are equivalent.

Apply: Properties (“Laws”)

- *Functors* properties
- *Monoids* properties (we’ll come back to those).

Quickly: Applicative

- Adds a *pure* method to Apply:
  - `def pure[A](x: A): F[A]`

```
import cats._

scala> Applicative[Option].pure(1)
res: Option[Int] = Some(1)

scala> Applicative[List].pure(1)
res: List[Int] = List(1)

scala> (Applicative[List] compose Applicative[Option]).pure(1)
res: List[Option[Int]] = List(Some(1))
```

Monoid

Monoid:
Not Your Father's Addition

- Monoid abstracts over *addition*:
  - A binary operation + for composition (where have we seen this before?):
    - Associative (but not necessarily commutative):
      - (a + b) + c = a + (b + c)
    - There exists an identity element:
      - 1 + x = x = x + 1

Monoid:
Not Your Father's Addition

- Many data structures fit this model:
  - (https://en.wikipedia.org/wiki/Monoid#Examples)
    - Numbers (well, except for round off…)
      - addition with 0 as the identity
      - multiplication with 1 as the identity
    - Strings (concatenation)

Monoid:
Not Your Father's Addition

- Many data structures fit this model:
  - bloom filters
  - hyperloglog
  - ... many other "data" data structures

Monoid in Cats

```
import cats._
import cats.std.all._

scala> Monoid[String].empty
res: String = ""

scala> Monoid[String].combineAll(List("a","b","c"))
res: String = abc

scala> Monoid[String].combineAll(List())
res: String = ""
```

Monad

Fear the Monad!

- Work with values in a context, such as enclosing state, containers, etc.
- Extends *Applicative* with *flatten*. (Think *flatMap*)
  - Flattens F[F[A]] to F[A].

Monad

```
import cats._

scala> Option(Option(1)).flatten
res: Option[Int] = Some(1)

scala> Option(None).flatten
res: Option[Nothing] = None

scala> List(List(1),List(2,3)).flatten
res: List[Int] = List(1, 2, 3)
```

Monad

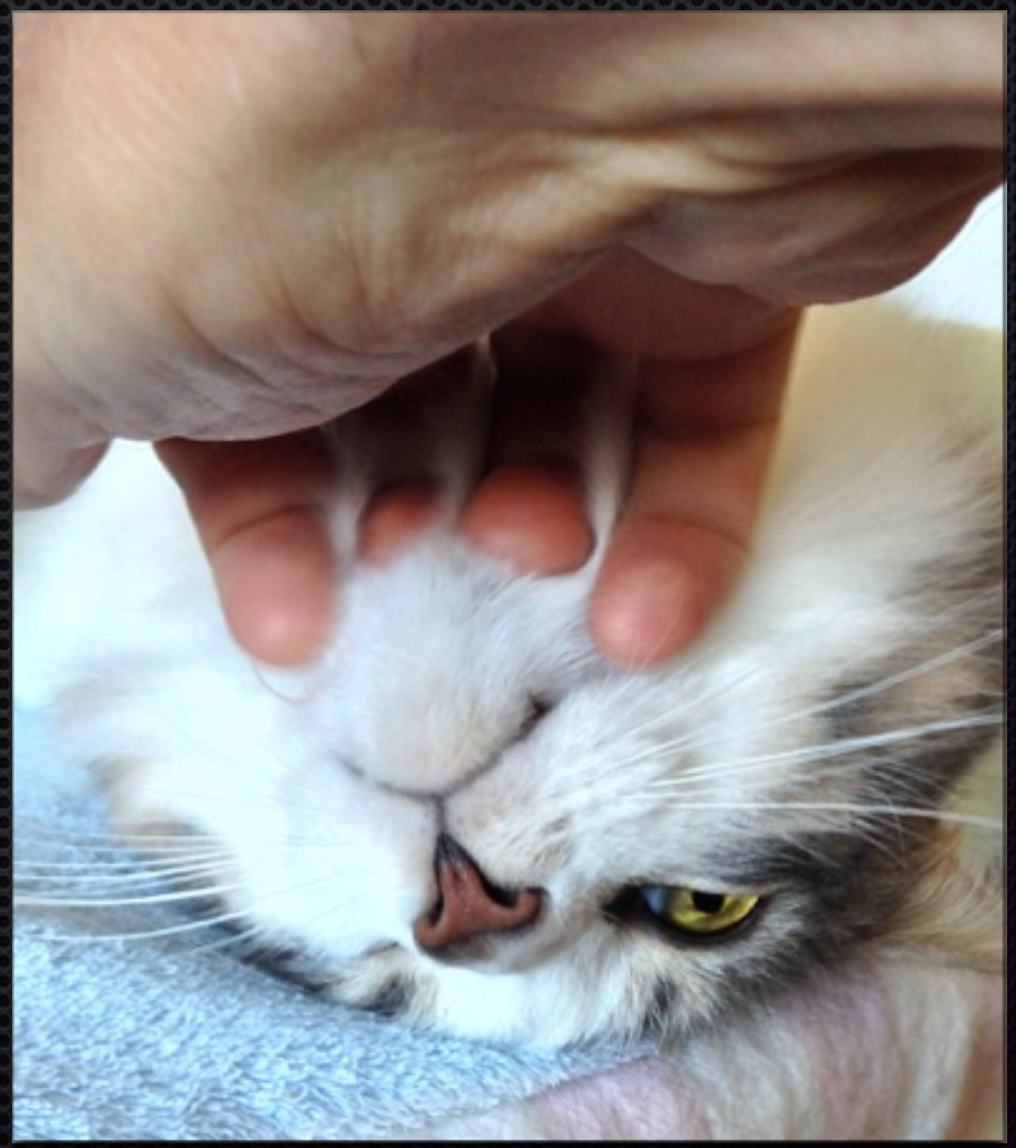
```
implicit def optionMonad(
  implicit app: Applicative[Option]) =
    new Monad[Option] {
  override def flatMap[A, B](fa: Option[A])(
   f: A => Option[B]): Option[B] =
     app.map(fa)(f).flatten  // Use option's flatten

   // Reuse from Applicative.
   override def pure[A](a: A):Option[A] =app.pure(a)
  }

Monad[Option].flatMap(Option(1))(i => Some(i+2))
// Some(3)
```

The familiar flatMap and we need pure, too.

Cat’s Goals: Use Modern Practices

simulacrum

- Minimizing type class boilerplate



without simulacrum

```
abstract class Addable[T](t1: T) {
  def add(t2: T): T
  def |+|(t2: T): T = add(t1, t2)
}

implicit class ComplexAddable(c1: Complex)
    extends Addable[Complex](c1) {
  def add(c2: Complex):Complex =
    Complex(c1.real + c2.real, c1.imag + c2.imag)
}

Complex(1.0, 2.0) |+| Complex(3.0, 4.0)
Complex(1.0, 2.0) add Complex(3.0, 4.0)

// => Complex(4.0, 6.0)
```

with simulacrum

```
// Compile separately (macros)
import simulacrum._
@typeclass trait Addable[T] {
  @op("|+|") def add(t1: T, t2: T): T
}

// In another compilation:
implicit val cAddable(c1: Complex) =
  new Addable[Complex] {
    def add(c2: Complex): Complex =
     Complex(c1.real + c2.real, c1.imag + c2.imag)
  }

import Addable.ops._

Complex(1.0, 2.0) |+| Complex(3.0, 4.0)
Complex(1.0, 2.0) add Complex(3.0, 4.0)
```

machinist

- Optimize implicit operators, by removing the intermediate objects typically used.

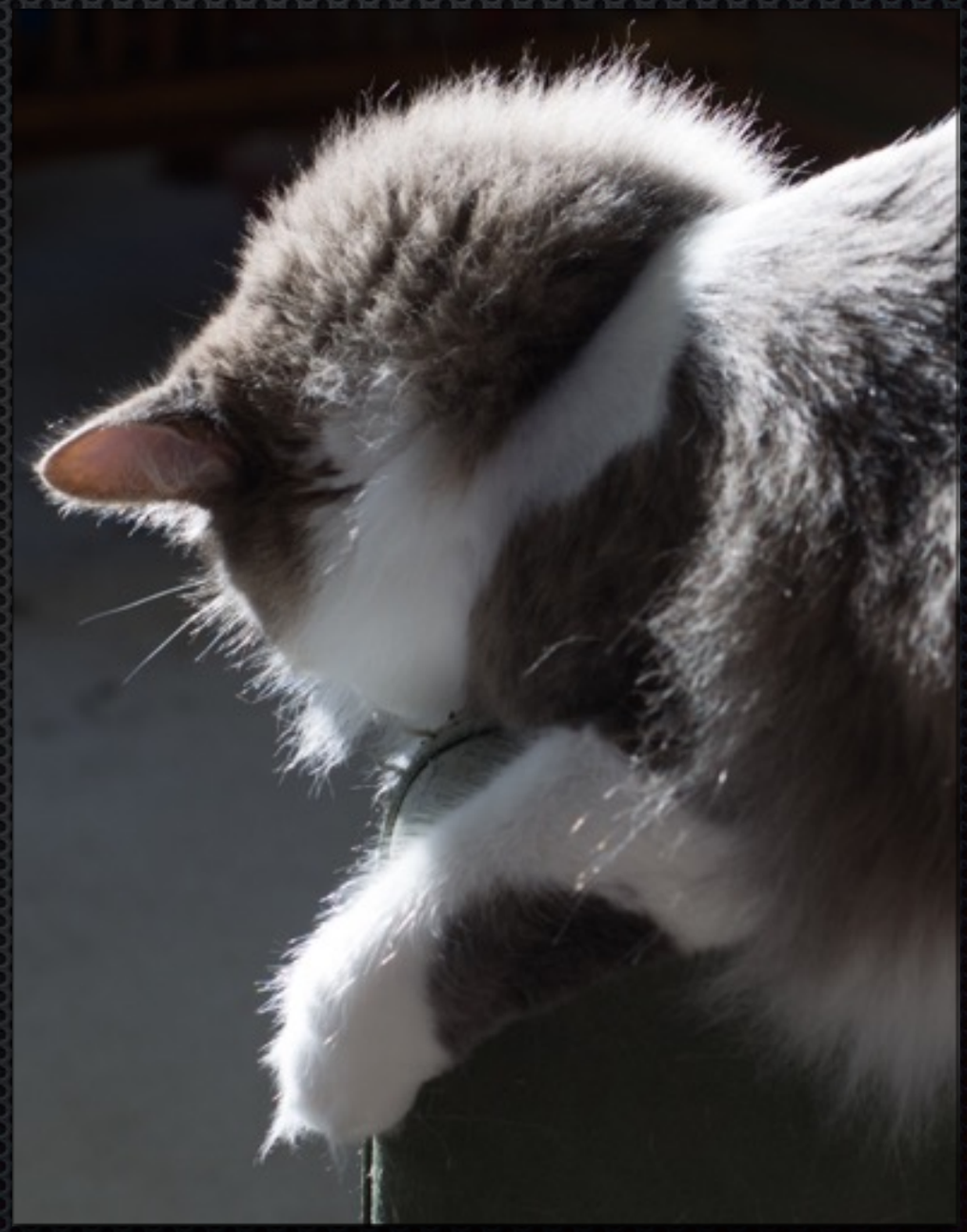
machinist

```
…
import machinist.DefaultOps

implicit val cAddable(c1: Complex) =
  new Addable[Complex] {
    def add(c2: Complex): Complex =
      macro DefaultOps.binop[Complex, Complex]
  }
```

```
Complex(1.0, 2.0) |+| Complex(3.0, 4.0)
Complex(1.0, 2.0) add Complex(3.0, 4.0)
```

scalacheck

- Property-based testing.



scalacheck

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Arbitrary.arbitrary

object AddableSpecification extends Properties("Addable") {

  implicit val complexGen = for {
    real <- arbitrary[Double]
    imag <- arbitrary[Double]
  } yield Complex(real, imag)

  property("|+|") = forAll(complexGen, complexGen) {
    (a: Complex, b: Complex) =>
      val c = a |+| b
      (c.real == a.real + b.real) &&
      (c.imag == a.imag + b.imag)
  }
}
```

discipline

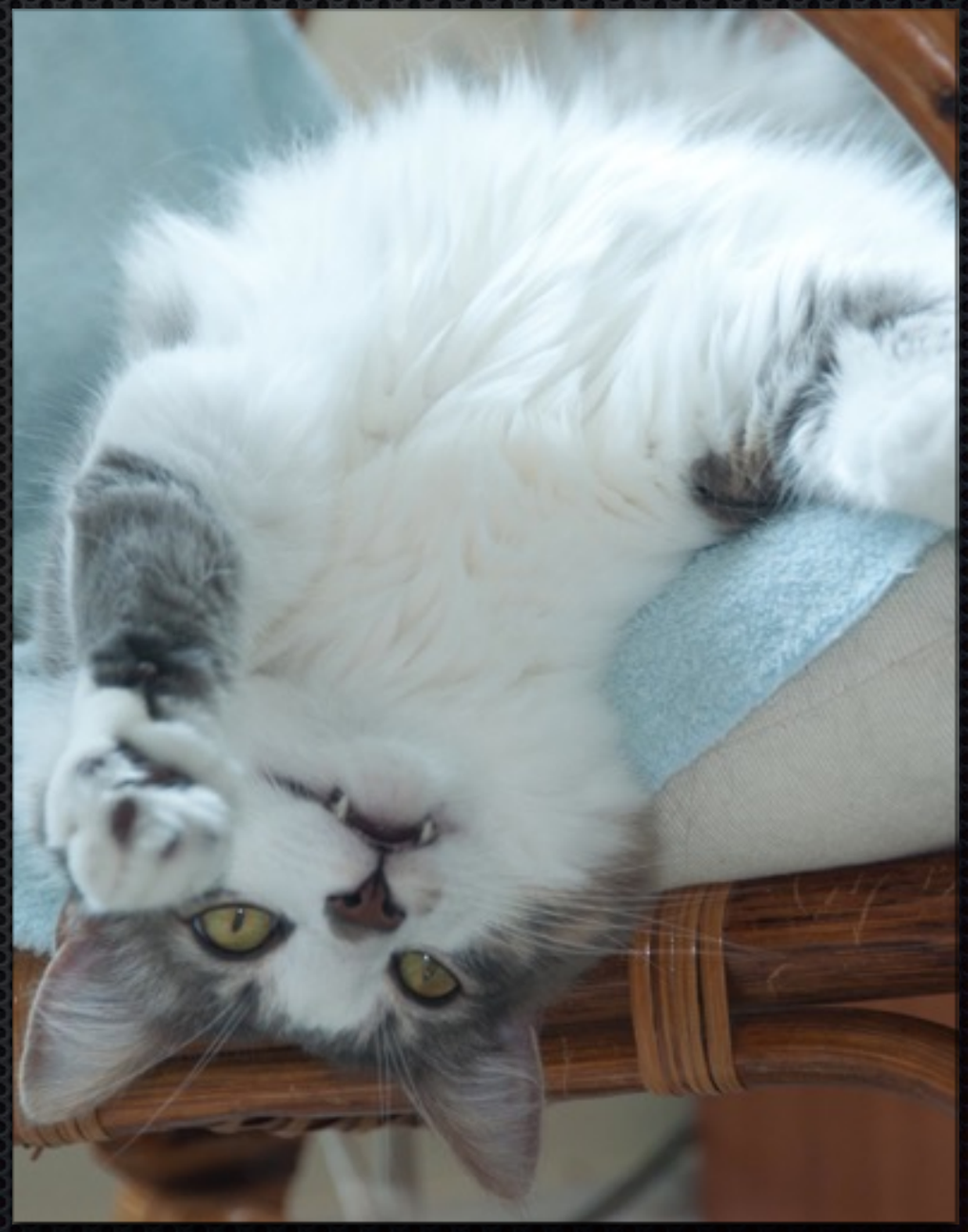
- For encoding and testing *laws*.
- Extensions to scalacheck.

discipline

```scala
// from cats.MonoidalTests.scala
import org.typelevel.discipline.Laws

trait MonoidalTests[F[_]] extends Laws {
  def laws: MonoidalLaws[F]

  def monoidal[A : Arbitrary, B : Arbitrary, C : Arbitrary](
    implicit iso: MonoidalTests.Isomorphisms[F],
    ArbFA: Arbitrary[F[A]],
    ArbFB: Arbitrary[F[B]],
    ArbFC: Arbitrary[F[C]],
    EqFABC: Eq[F[(A, B, C)]]
  ): RuleSet = {
    new DefaultRuleSet(
      name = "monoidal",
      parent = None,
      "monoidal associativity" -> forAll(
        (fa: F[A], fb: F[B], fc: F[C]) => iso.associativity(
          laws.monoidalAssociativity(fa, fb, fc)))
    )
  }
}
```

Kind Projector

- Type Lambdas



without kind projector

```
trait Mapp[A,+M[_]] {
  def mapp[B](f: A => B): M[B]
}

object Mapp {
  // Easy:
  implicit class SeqMapp[A](seq: Seq[A]) extends Mapp[A,Seq] {
    def mapp[B](f: A => B): Seq[B] = seq map f
  }

  // Uh…
  implicit class RightMapp[L,R1](either: Either[L,R1])
    extends Mapp[R1, ({type Lam[X] = Either[L,X]})#Lam] {
      def mapm[R2](f: R1 => R2): Either[L,R2] = either match {
        case r: Right[L,R1] => r.right.map(f)
        case l: Left[L,R1] => Left[L,R2](l.left.get)
      }
    }
}
```

with kind projector

```
trait Mapp[A,+M[_]] {
  def mapp[B](f: A => B): M[B]
}

object Mapp {
  // Easy:
  implicit class SeqMapp[A](seq: Seq[A]) extends Mapp[A,Seq] {
    def mapp[B](f: A => B): Seq[B] = seq map f
  }

  // Uh…
  implicit class RightMapp[L,R1](either: Either[L,R1])
    extends Mapp[R1, Either[?,R1]] {
      def mapm[R2](f: R1 => R2): Either[L,R2] = either match {
        case r: Right[L,R1] => r.right.map(f)
        case l: Left[L,R1] => Left[L,R2](l.left.get)
      }
    }
}
```

Algebra

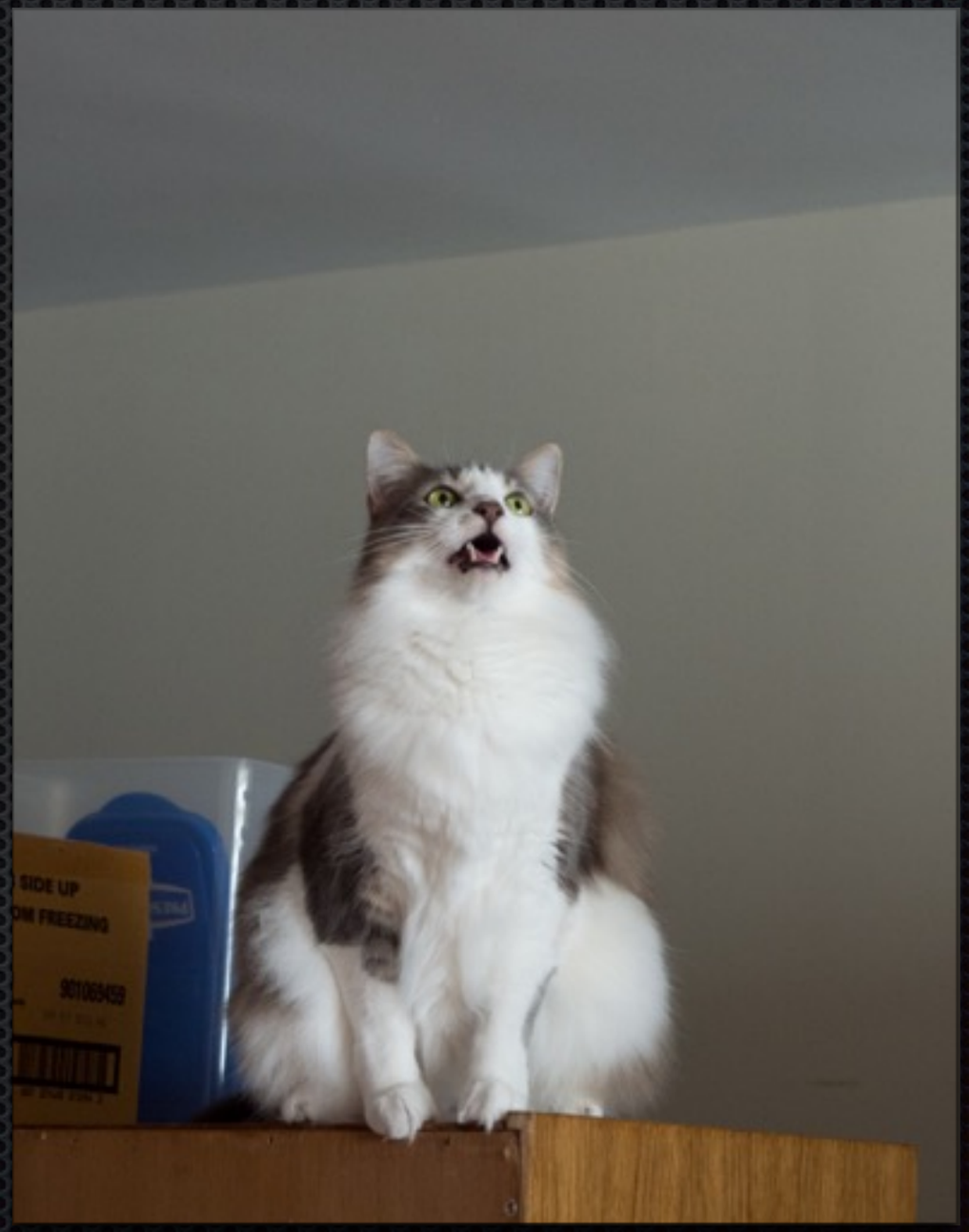
- For shared algebraic structures
  - (shared with Algebird)

Some Examples

- Monoid, Group & Semigroup, Ring & Semiring
  - (Some commute, some don't.)
- Lattice, Semilattice, Bounded Semilattice, etc.
- Others
- Helpers: Eq, Order, & PartialOrder

And finally.

- Pure functional subset of Scala
- Excellent documentation



The End

- typelevel.org - For all things leveled … and typed!
- Erik Osheim's (@d6) great talk (PDF) (video) at Scala World 2015, *Heuristics for approachable, modular, functional libraries*.
- polyglotprogramming.com/papers/Cats.pdf