scalaz

get functional

Me

Jason Zaugg / @retronym

Coder, Scala Evangelist @ EFG Financial Products, Zurich

Contributor to Scalaz, IntelliJ Scala Plugin

scalaz is...

scalaz is

an open source library to support functional programming in Scala.

http://scalaz.org

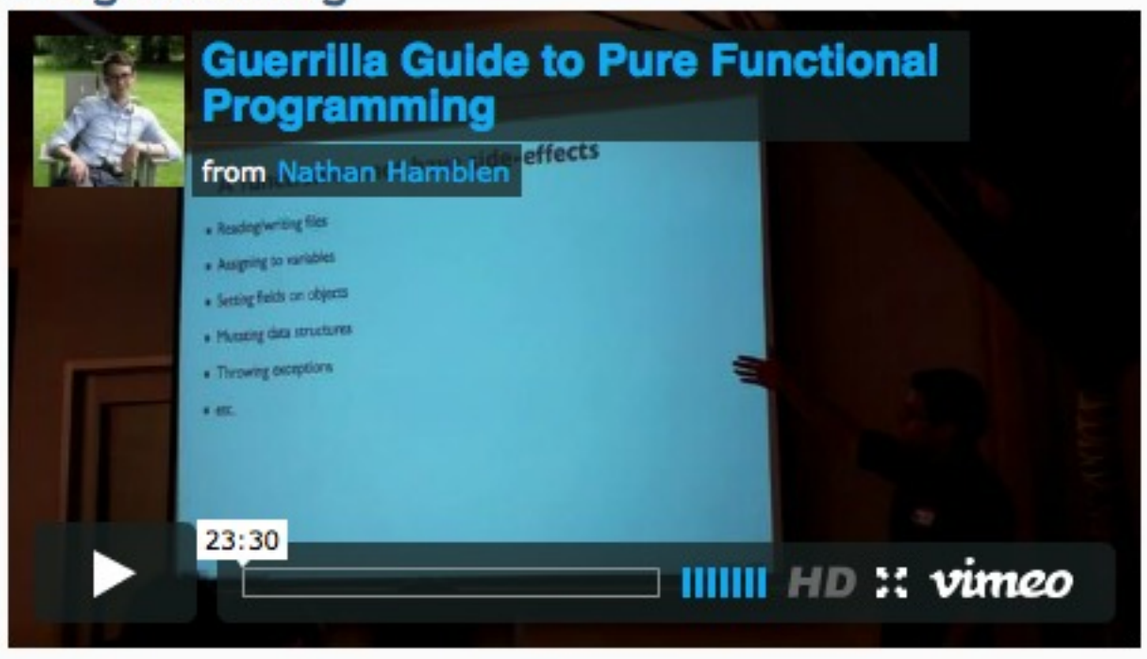
```
import Predef._
```

Runar “flatMap that shit” Bjarnason

http://www.nescala.org/2011/#guerrilla-guide-to-fp

Runar Bjarnason

on The Guerrilla Guide to Pure Functional Programming



functional programming is...

functional programming is

programming with functions

a **program** is...

a **program** is

a single, referentially transparent expression

an **expression** is...

an **expression** is

a combination of sub expressions, using the constructs of a language. It evaluates to a result.

a **referentially transparent** expression...

any occurrence of

a **referentially transparent** expression

within a program could be replaced by its result, without changing the meaning of the program

a **function** is...

a **function** *f*:*A* => *B* is

a relation between every value of type A to exactly one value of type B.

That's it, no side effects!

(otherwise, function calls are not RT)

a **type** is...

a **type** is

a set of values

side effects are...

EVIL!

for example,

side effects are

- I/O to disk, console, network
- mutating fields or data structures
- throwing exceptions

functional programming

is sometimes called

expression oriented programming

functions are the glue to build programs out of smaller programs.

Why FP?

Modularity

The degree to which the parts can be separated and recombined.

Compositionality

Understand the parts, and the connections, and you understand the whole.

Back to scalaz...

What's inside?

- Type Classes + Instances
- Pure Functional Data Structures
- General Functions
- Implicit Pimps
- Concurrency: Actors / Promise

Getting Started

```
import scalaz._; import Scalaz._
// Profit!
```

Imports data types, functions, and necessary implicit conversions

Type Classes in Two Minutes

Alternative to subtype polymorphism

Recognize this from Java?

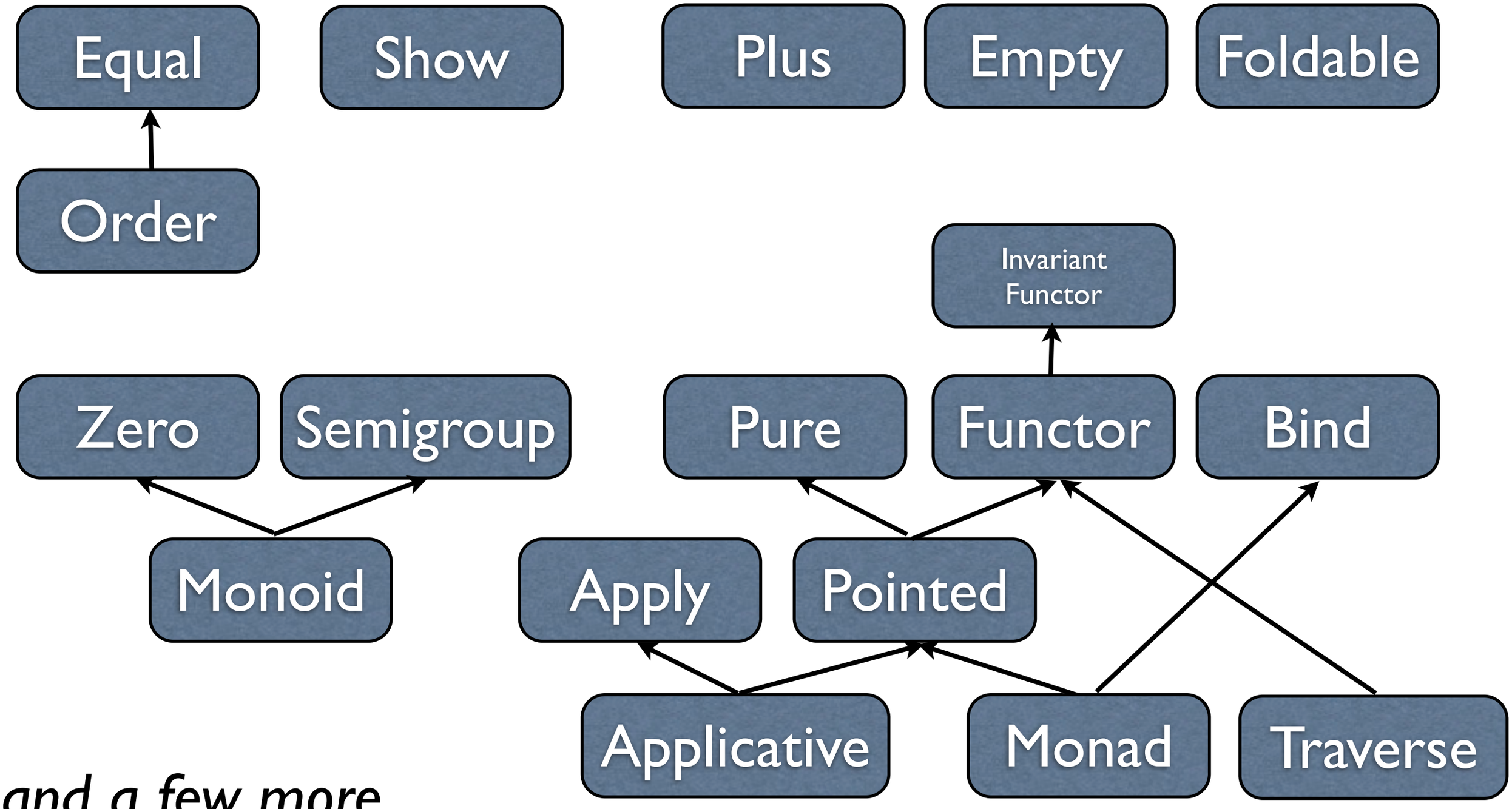
```
<T> void sort(
    List<T> ts,
    Comparator<? super T> comparator)
```

Type Class!

In Scala, we use implicits parameters to automatically pass the type class instance.

Example: Type Class and Instance

```
trait Pure[P[_]] {
  def pure[A](a: => A): P[A]
}

implicit def Tuple1Pure = new Pure[Tuple1] {
  def pure[A](a: => A) = Tuple1(a)
}

implicitly[Pure[Tuple1]].pure(1)

1.pure[Tuple1]
```



and a few more...

Monoid

A pair of functions:

```
def append(a1: A, a2: A): A
def zero: A
```

Satisfying some laws, e.g:

```
append(a, zero) == a  (for all a)
```

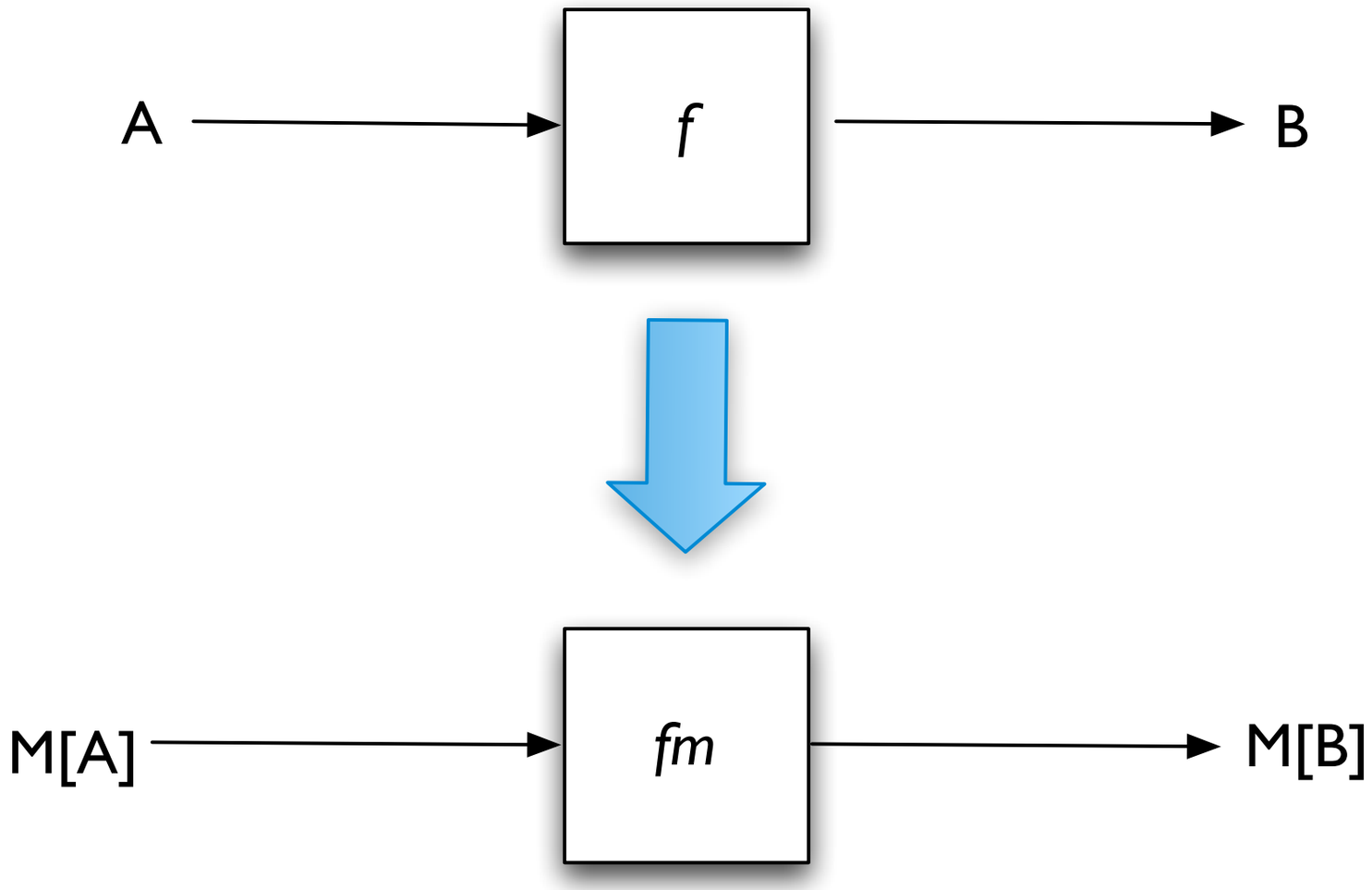
Monoid

Let’s build one!

Monoid: Examples

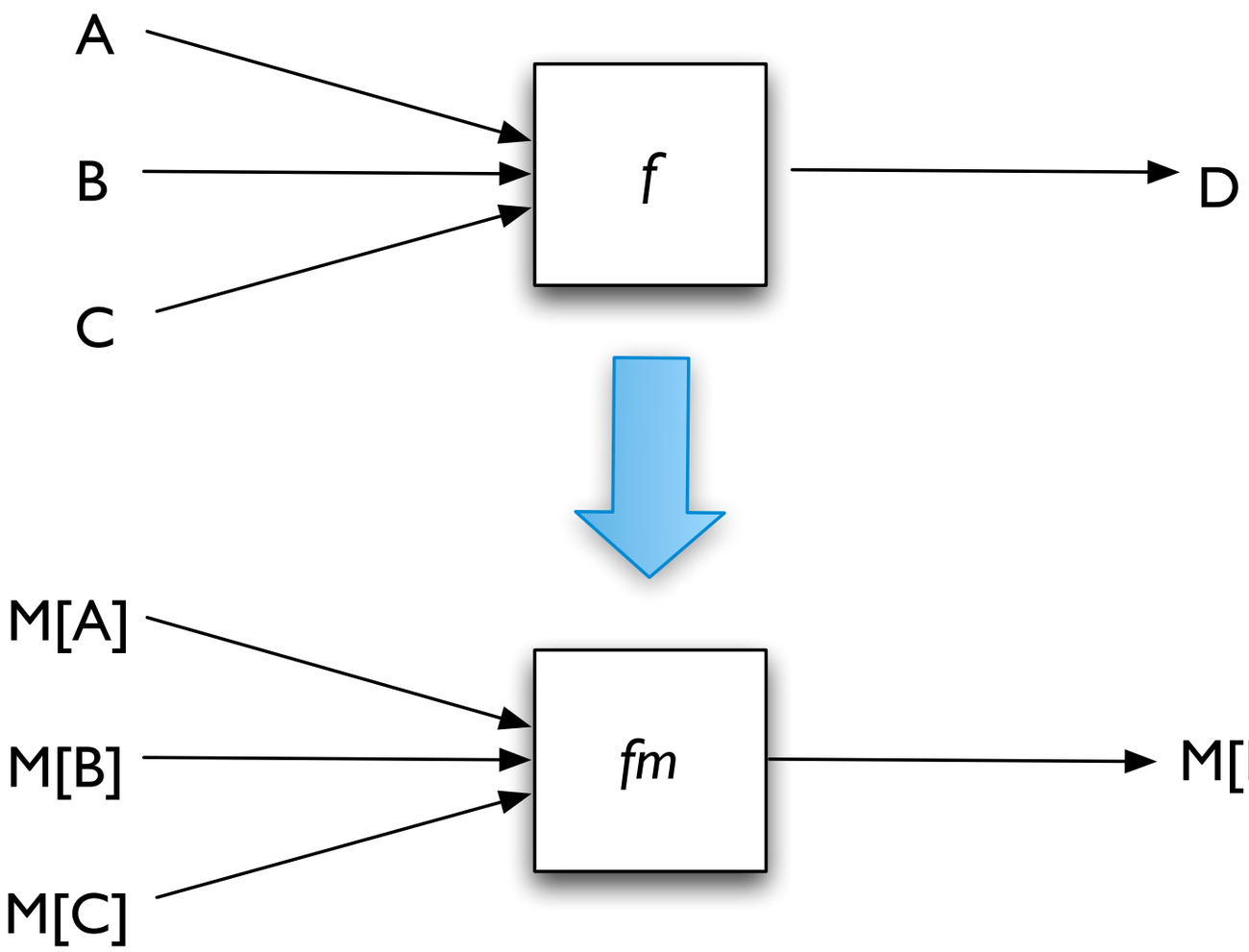
```
((1, "a") |+| (2, "b")) assert_===( (3, "ab"))

List(1.some, 2.some, none[Int]).sumr assert_===(Some(3))

Seq(0, 1, 2).foldMap(x => (x, x * x)) assert_===((3, 5))
```

Functor



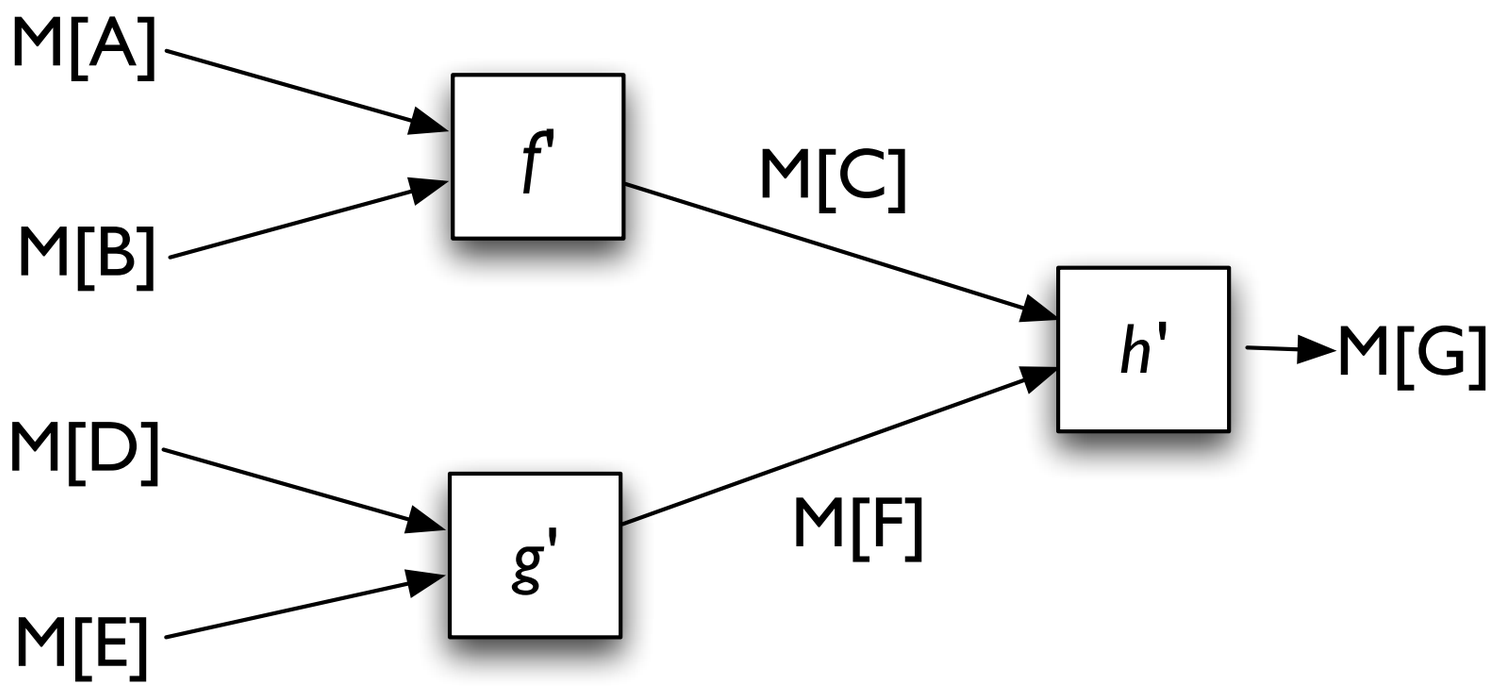

```
def fm(ma: M[A]) = ma map f
```

Applicative Functor




```
def fm(ma: M[A], mb: M[B], mc: M[C]): M[B]
          = (ma |@| mb |@| mc)(f)
```

Applicative Functor




```
val mc = (ma |@| mb)(f)
val mf = (me |@| md)(g)
val mg = (mc |@| mf)(h)
```

Applicative Functor: Examples

```
scala> def foo(a: Int, b: String) =
  List.fill(a)(b).intersperse(".").collapse

scala> foo(10, "x")
res0: String = x.x.x.x.x.x.x.x.x.x

scala> (5.some |@| "x".some)(foo)
res1: Option[String] = Some(x.x.x.x.x)
```

Apply the *effectful* arguments in Options to the pure function `foo`

Mnemonic: @pplicative
See also: <*>, <**>, ...

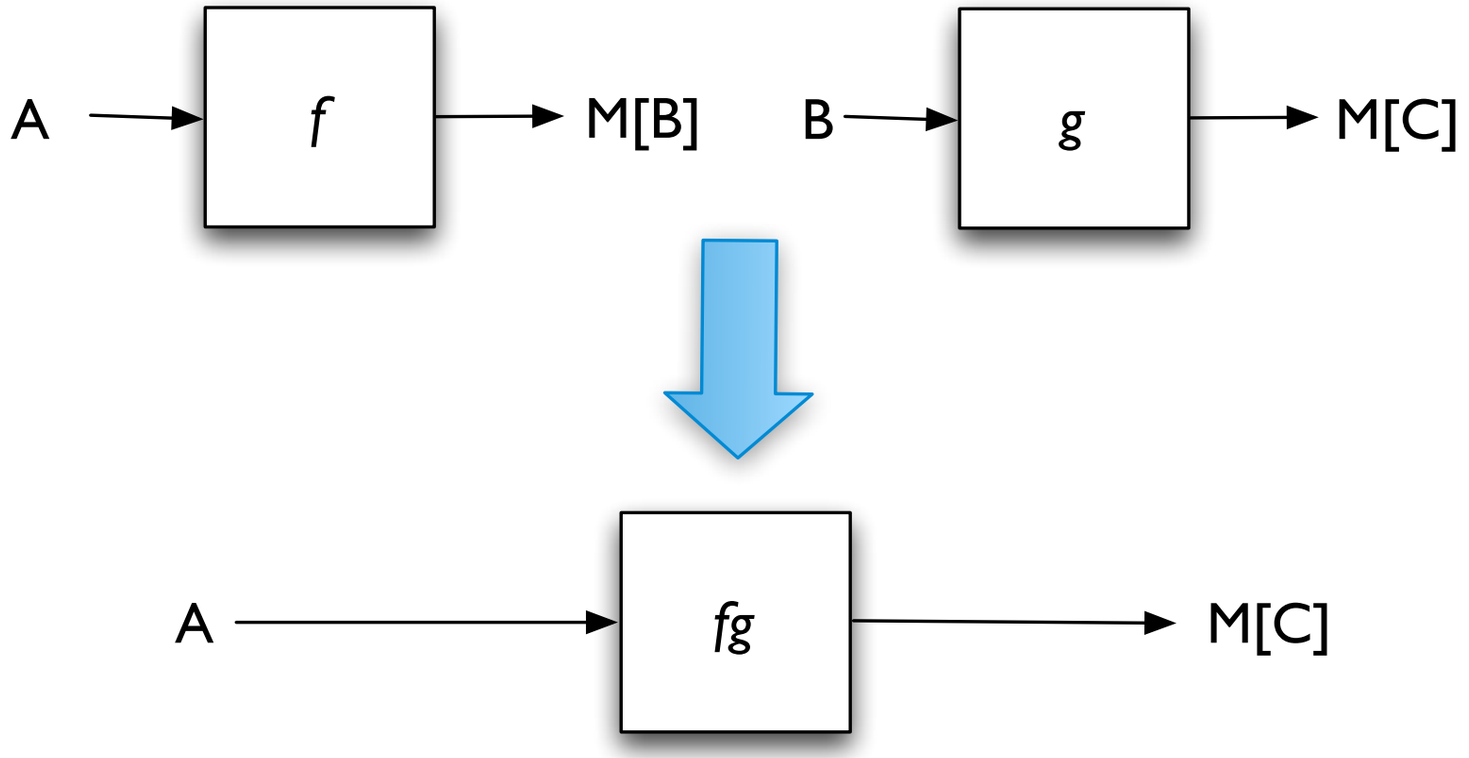
Applicative Functor: Examples

Promise is an async computation, ala Future

```
scala> Seq(promise(1), promise(2))
res5: Seq[Promise[Int]] =
   List(<promise>, <promise>)

scala> res5.sequence
res6: Promise[Seq[Int]] = <promise>

scala> res6()
res7: Seq[Int] = List(1, 2)
```

Voila! A single Promise, of a Seq[Int]

(The Dreaded) Monad




```
def fg(a: A): M[C] = for {
  b <- f(a)
  c <- g(b)
} yield c

val fg = f >=> g // alternative
```

IO: Corralling Side Effects

```
scala> def ls(f: File): IO[List[File]] = io {
  ~Option(f.listFiles).map(_.toList)
}

scala> val cd = new File(".")
cd: File = .
```

Side Effect resulting in List[File]

```
scala> ls(cd)
res0: IO[List[File]] = <effect>

scala> res0.map(_.take(4))
res1: IO[List[File]] = <effect>

scala> res1.unsafePerformIO
res2: List[File] = List
(./.git, ./.gitignore, ./.idea, ./.idea_modules)
```

"The end of the universe"

Pimps at Work

```
1 === 1
List(1, 2, 3).collapse
```

```
Scalaz.IdentityTo[Int](1).===(1)(Equal.IntEqual)
Scalaz.maImplicit[List, Int](List(1, 2, 3)).collapse(
  Traverse.TraversableTraverse[List],
  Monoid.monoid[Int](Semigroup.IntSemigroup, Zero.IntZero)
)
```

Lister

- Larger program to show a few Scalaz features used in concert.
  - Monoids, Tree, TreeLoc, IO
  - Compare Lister.{Impure, Pure}
- https://github.com/retronym/scalaexchange-scalaz/blob/master/src/main/scala/sx/Lister.scala

Want to know more?

- Slides, Code
  github.com/retronym/scalaexchange-scalaz
- Mailing List
  groups.google.com/forum/#!forum/scalaz
- IRC
  #scalaz on FreeNode IRC

Recommended Reading

- Typeclassopedia
- Learn You A Haskell
- “Functional Programming in Scala” book is rumoured, stay tuned!

Bonus Slides

FP in Scala

no statements, just expressions

FP in Scala

function literals

functions are values

higher-order functions

FP in Scala

algebraic data types (albeit with clunky syntax)

FP in Scala

type parametric polymorphism

FP in Scala

implicit parameters for ad-hoc polymorphism

(aka type classes)

FP in Scala

Expressive type system

Type Inference

Type Constructor Polymorphism

FP in Scala: What's hard?

no side effect tracking (up to you!)

lazy evaluation tricky

subtyping a hinderance

temptation of mutable vars, data structures

```scala
case class Complex(real: Double, imaginary: Double)

object Complex {
  import scalaz._
  import Scalaz._

  implicit val Show: Show[Complex] = showA
  implicit val Equal: Equal[Complex] = equalA
  implicit val Zero: Zero[Complex] = zero(Complex(0, 0))
  implicit val Semig: Semigroup[Complex] = semigroup {
    case (Complex(r1, i1), Complex(r2, i2)) =>
      Complex(r1 + r2, i1 + i2)
  }
  implicit val OrderComplex: Order[Complex] = orderBy {
    case Complex(r, i) => (r, i)
  }
}
```

delegate to Any#
{toString, ==}

```scala
object ComplexTest extends Application {
  import scalaz._; import Scalaz._

  val (c1, c2) = (Complex(0, 1), Complex(0, 2))
  val cs = Seq(c1, c2)
  val csString = cs.shows
  c1 === c2
  c1 ≤ c2
  c1 ⊹ c2

  val sum = cs.∑

  // TraversableOnce#{min, max} are in the way.
  val (min, max) = (cs: MA[Seq, Complex]).pair
          .mapElements(_.min, _.max)
}
```