Monads are Burritos (and the like)

Scala Developers Barcelona - 11th March 2015

Jordi Aranda - **@jordi_aranda**

Motivation

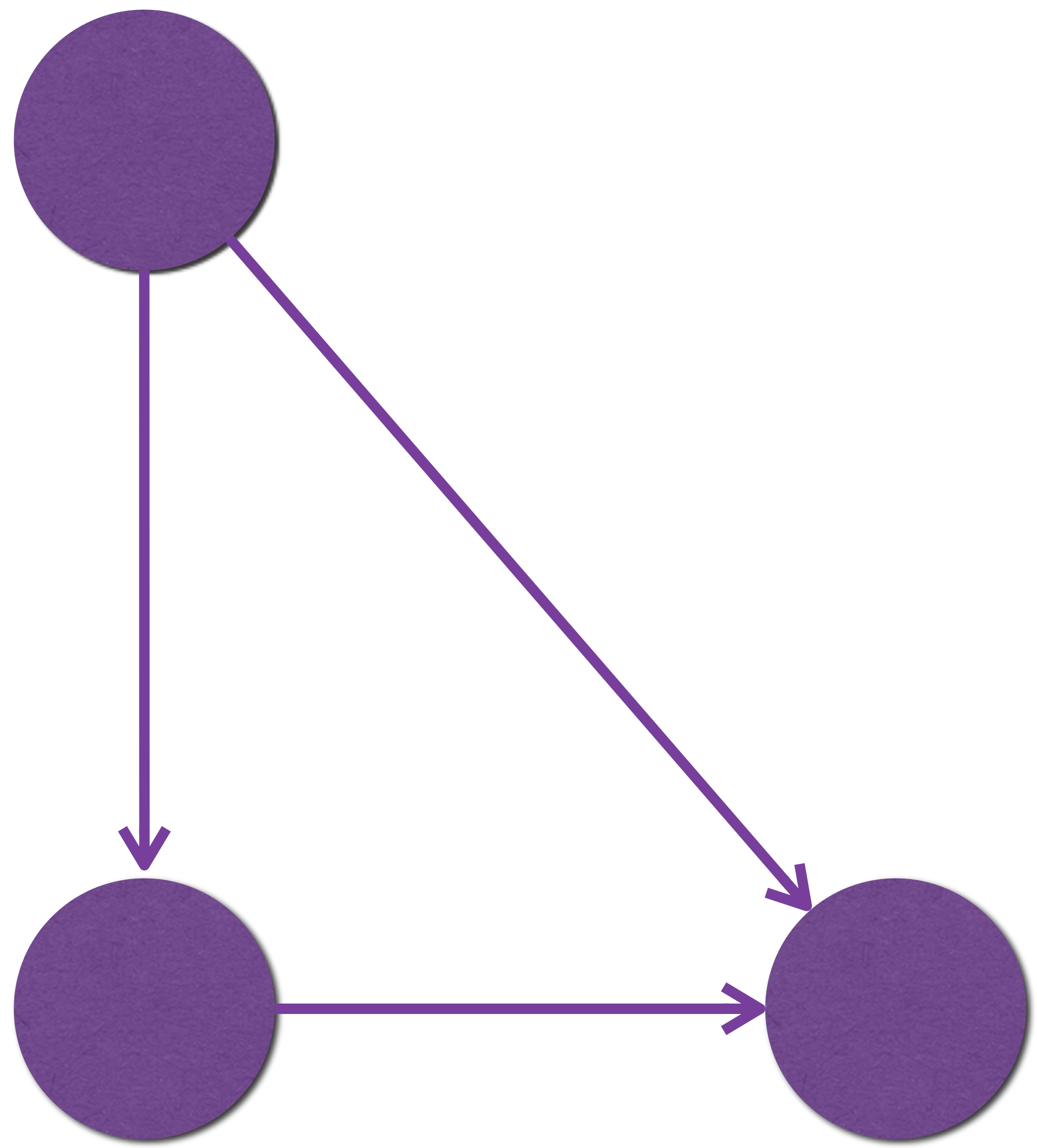
- Learn more about (scary) abstract algebra and category theory concepts
- Seeing those concepts in more places
- Promote and expand general interest

"*There is no Burrito: we all must find our own*" - **Unknown**

Schedule

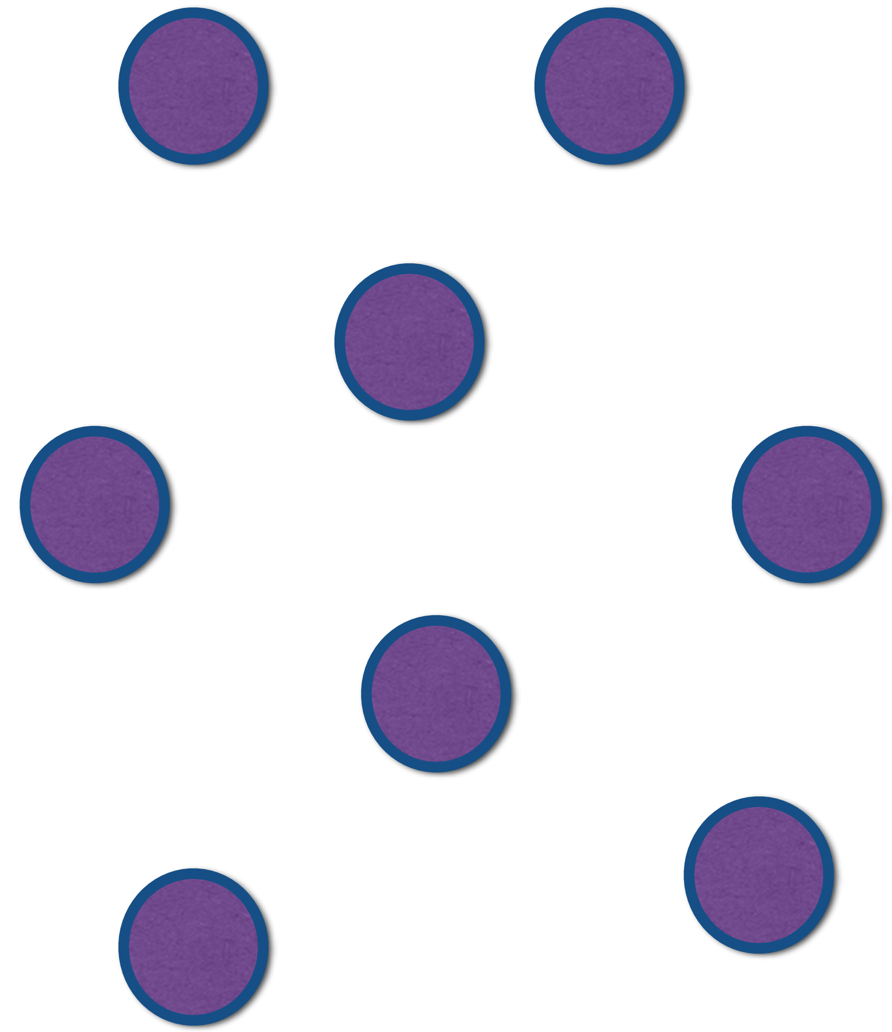
- Category Theory concepts
- Abstractions: Typeclass pattern
- Some typeclasses for glorious good

Categories



Category

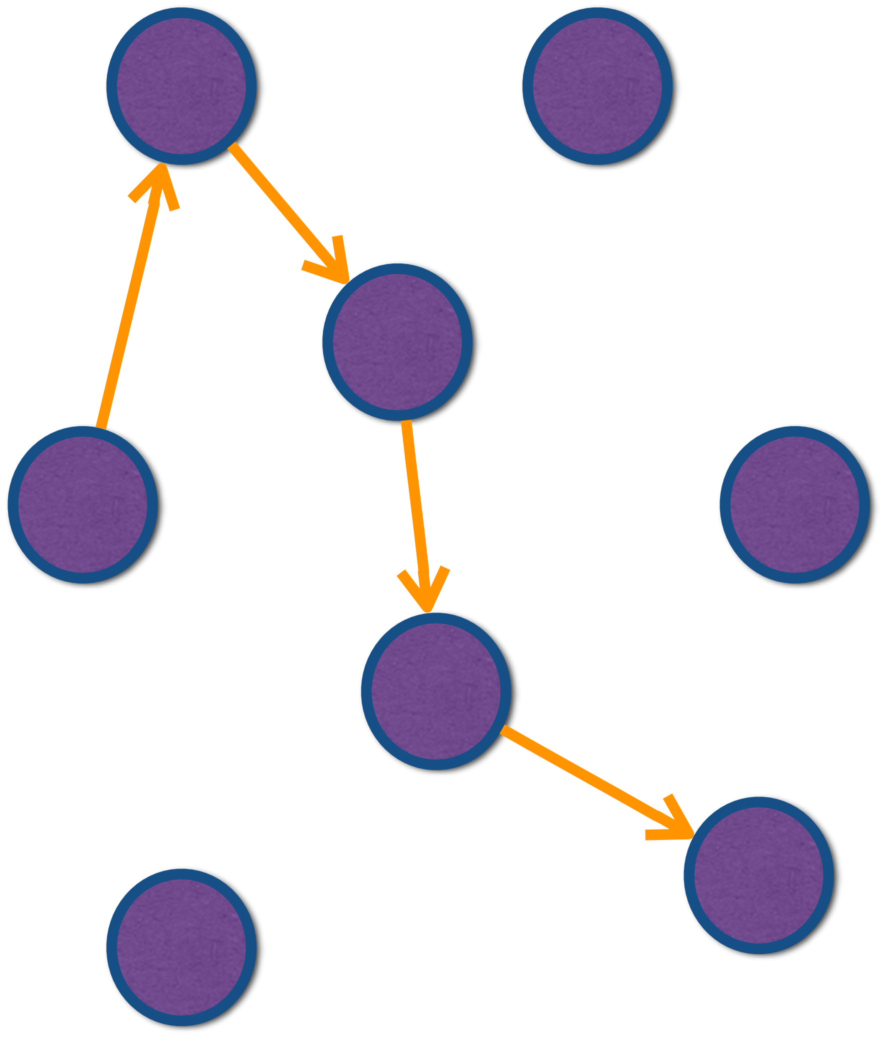
Objects



Category

Objects

Morphisms / Arrows

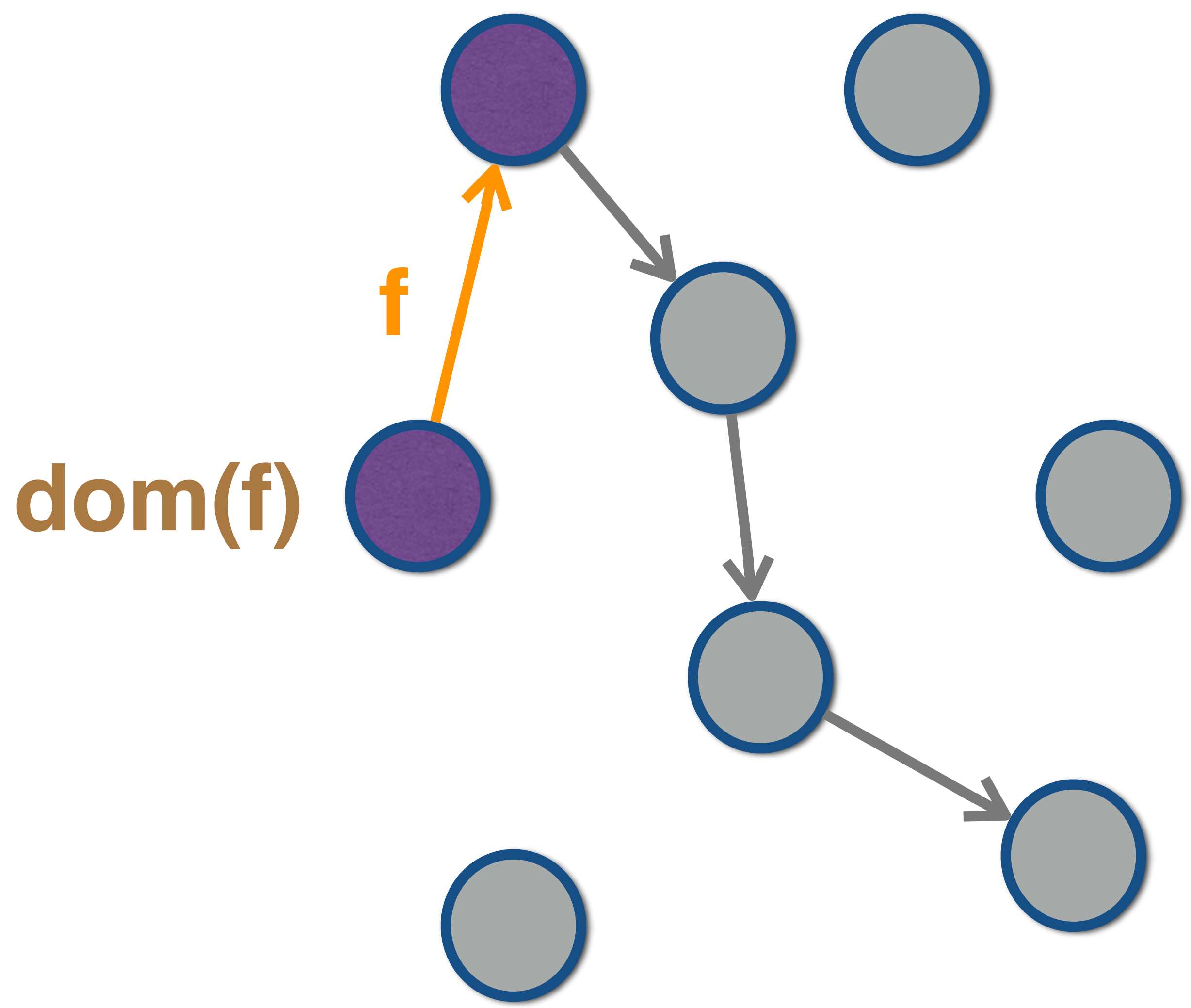


Category

Objects

Arrows

Domain

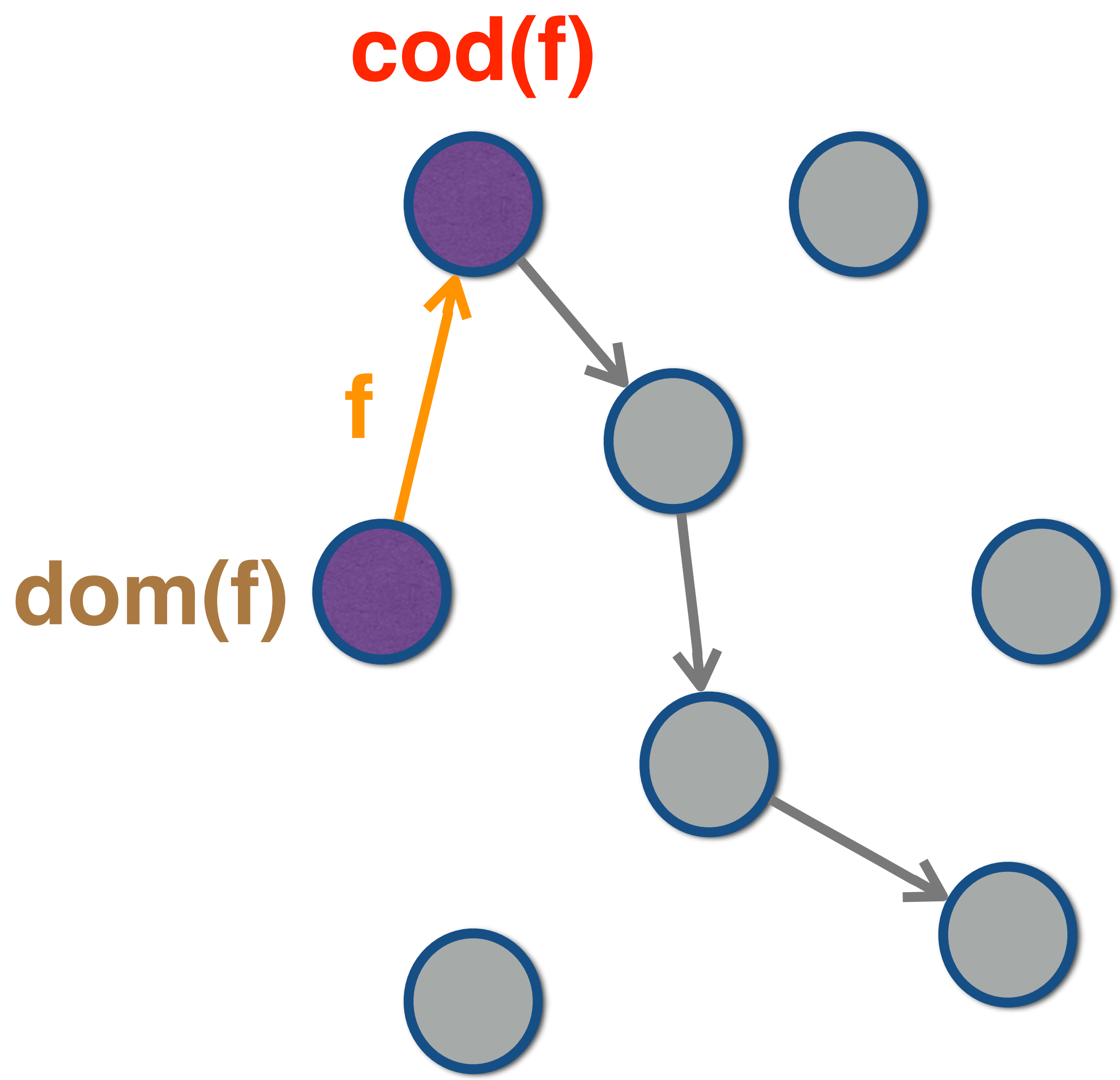


Category

Objects

Arrows

Domain/**Codomain**

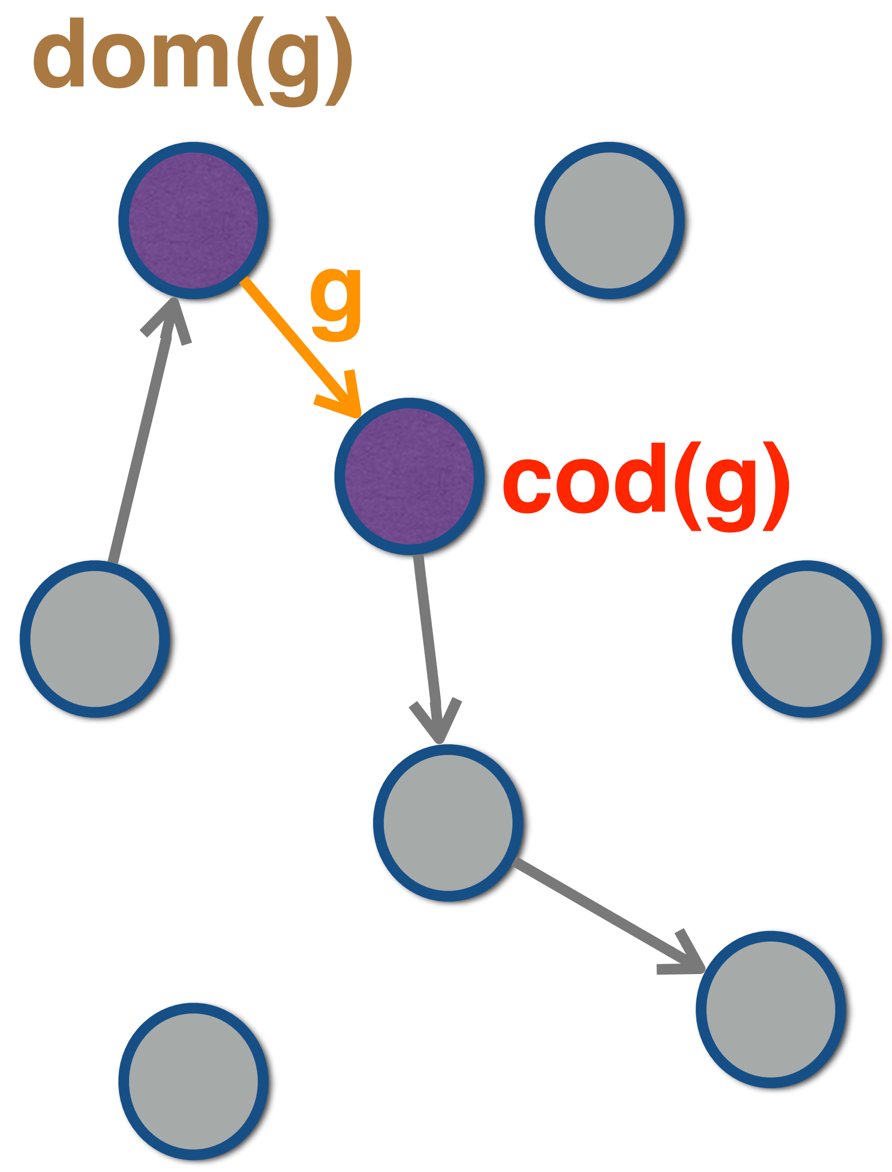


Category

Objects

Arrows

Domain/Codomain



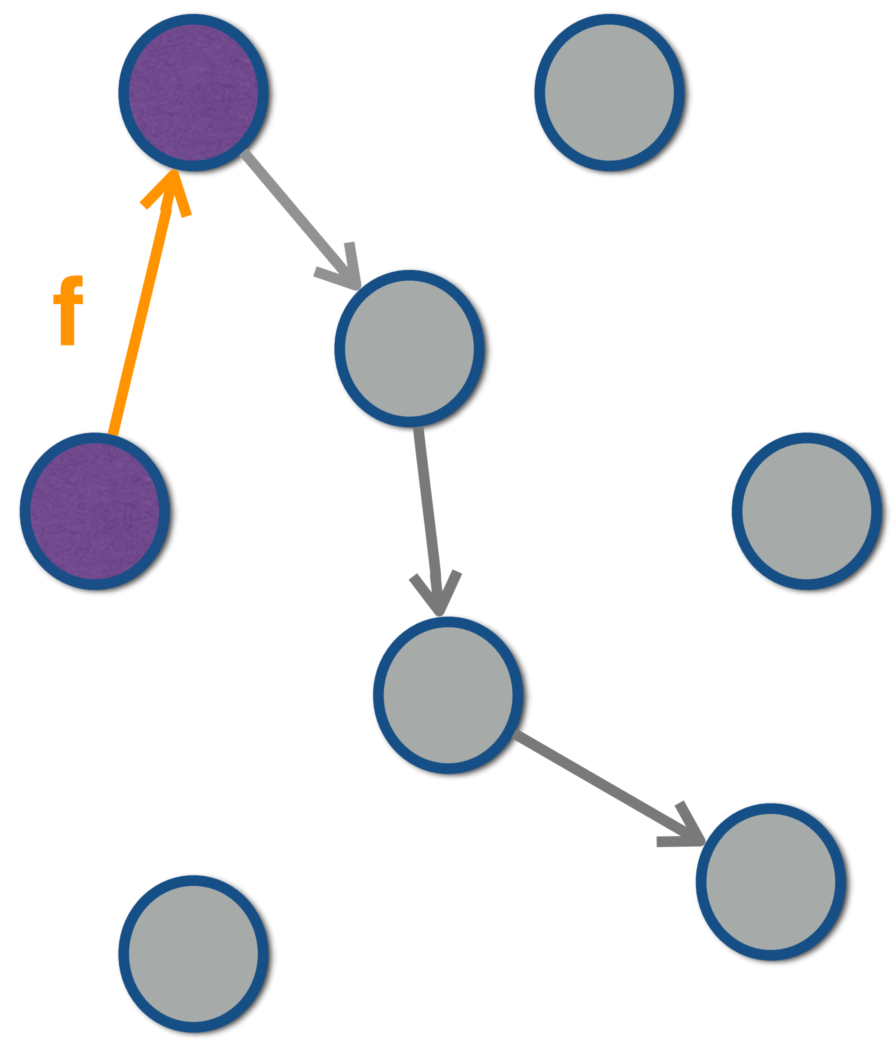
Category

Objects

Arrows

Domain/Codomain

Composition



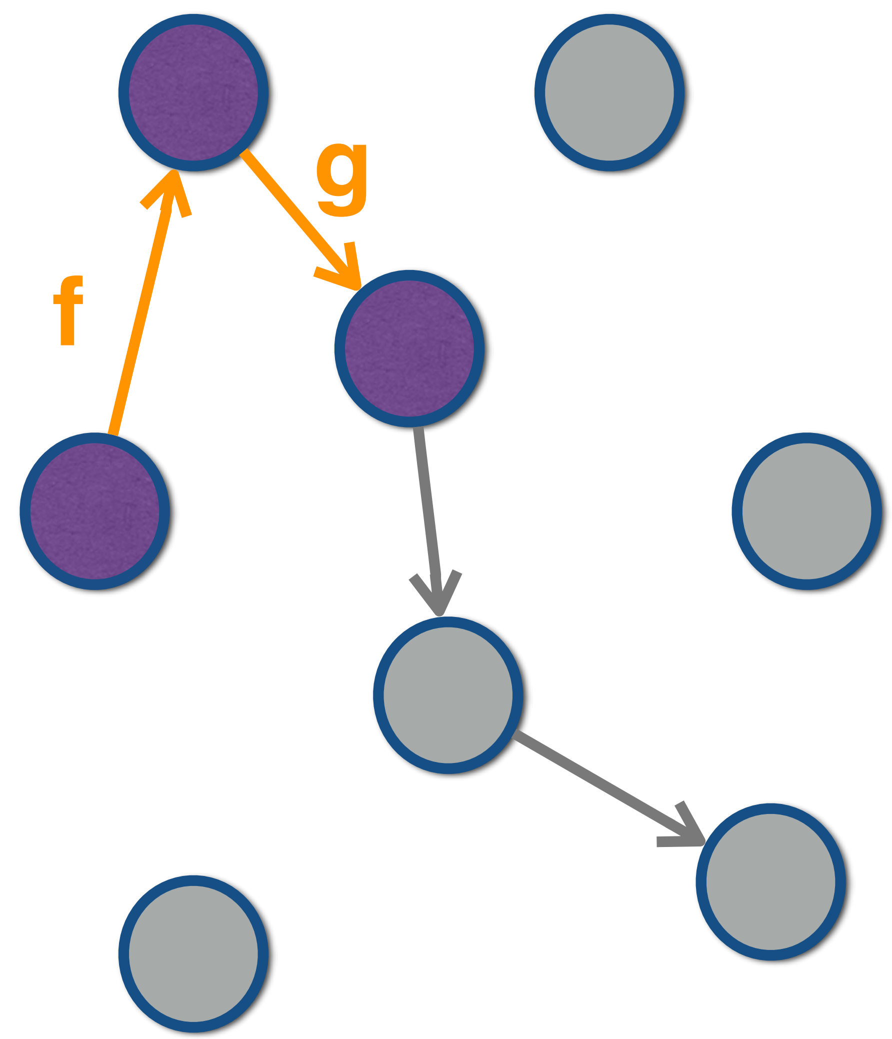
Category

Objects

Arrows

Domain/Codomain

Composition



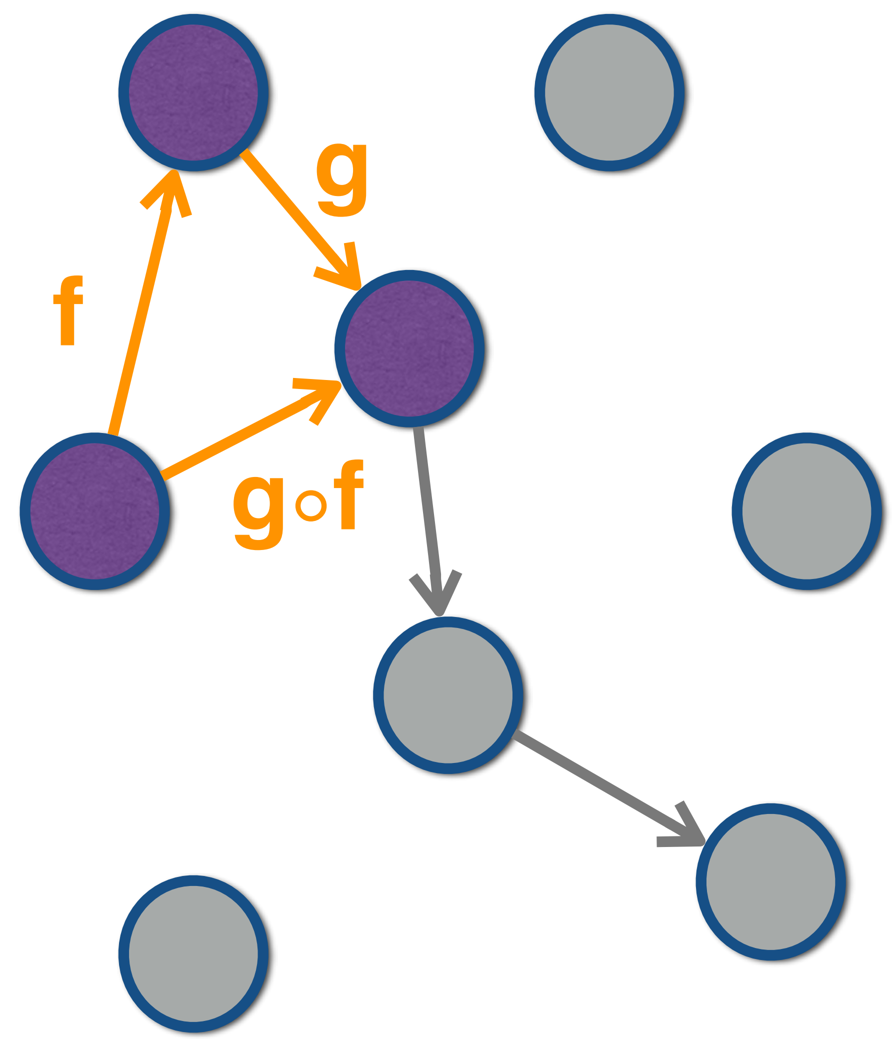
Category

Objects

Arrows

Domain/Codomain

Composition



Category

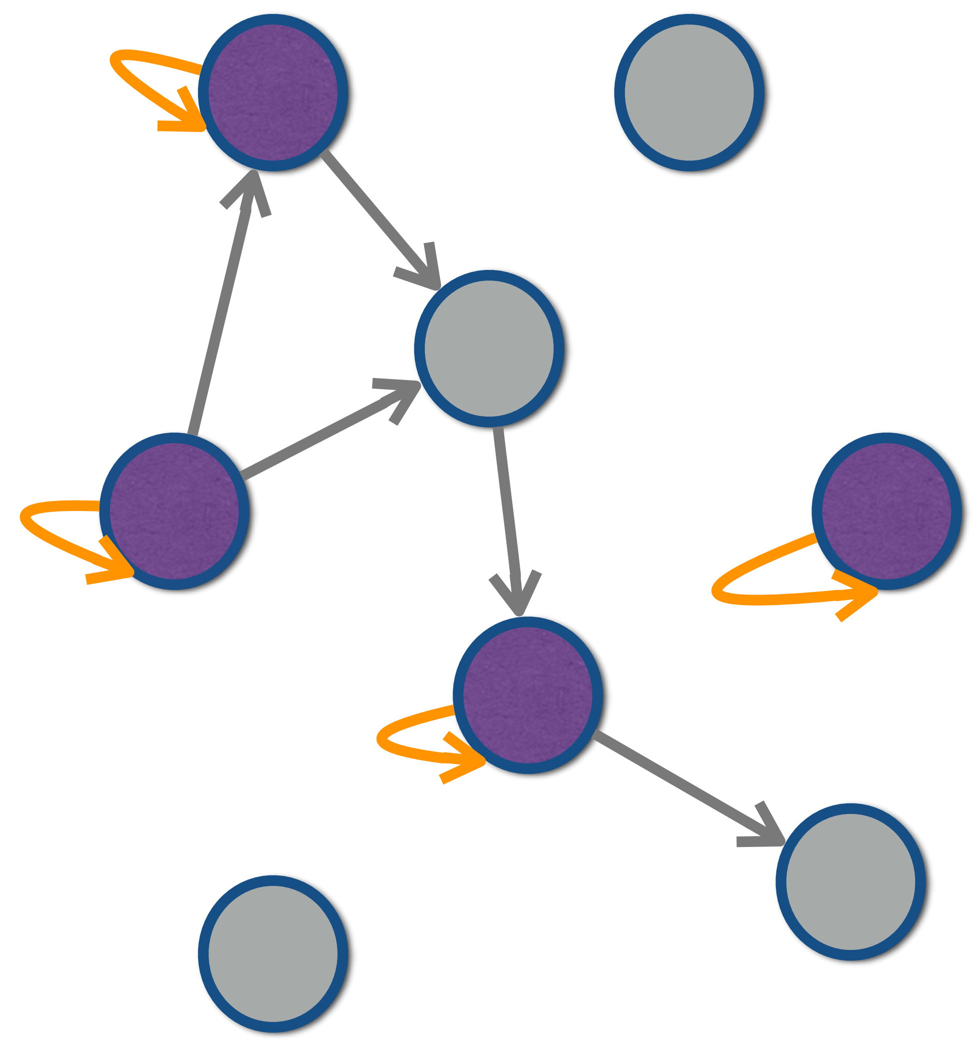
Objects

Arrows

Domain/Codomain

Composition

Identity



Category

- **Composition Law**:
  - : (B —> C) —> (A —> B) -> (A —> C)
- **Identity**:

id: A —> A

Category Axioms

- **Associativity**:

$(f \circ g) \circ h = f \circ (g \circ h)$

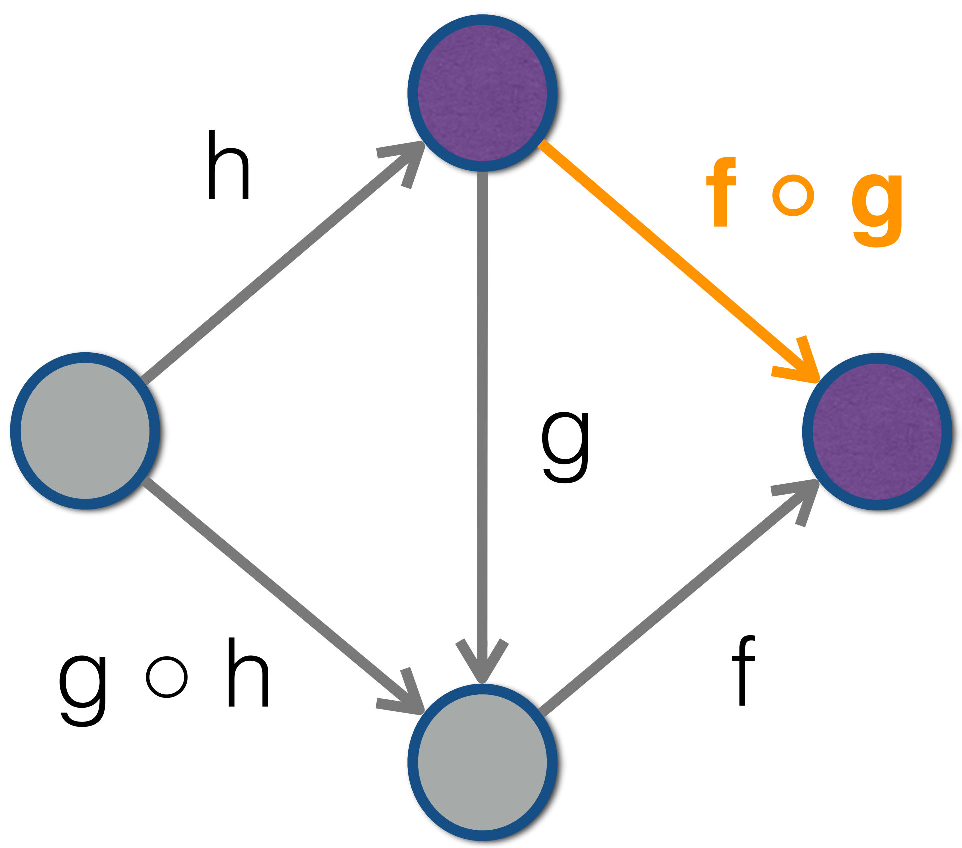
- **Identity**:

$id \circ f = f \circ id = f$

Category Axioms

- **Associativity**:

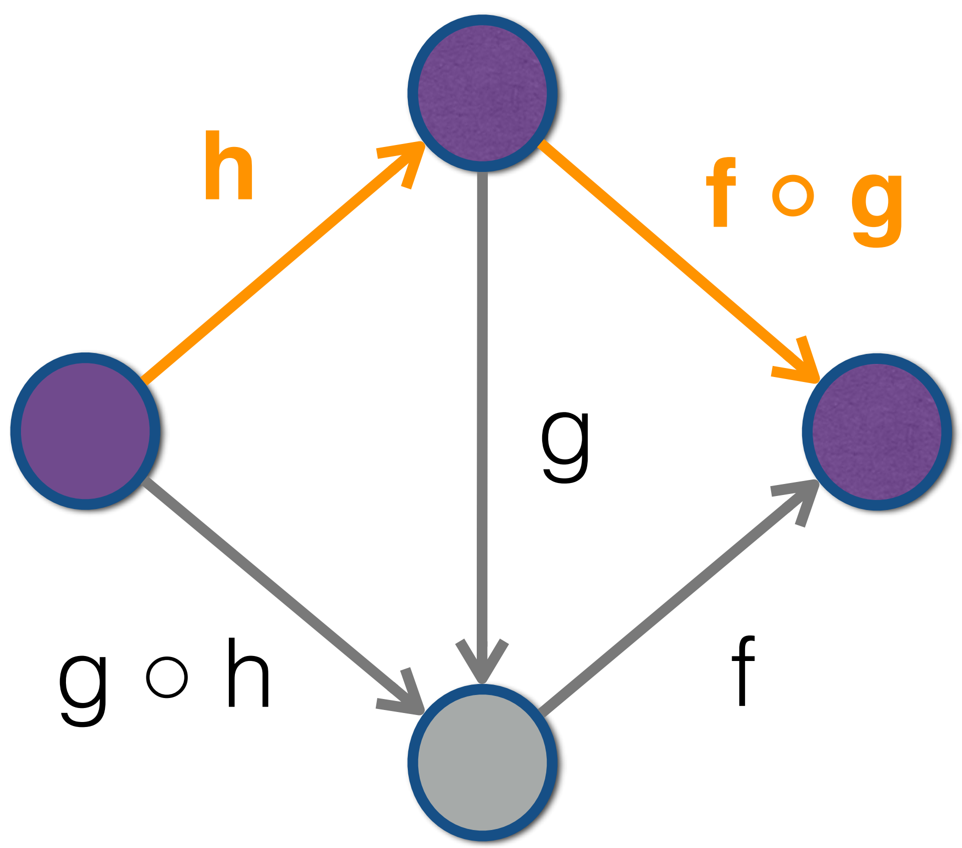
$(f \circ g) \circ h = f \circ (g \circ h)$



Category Axioms

- **Associativity**:

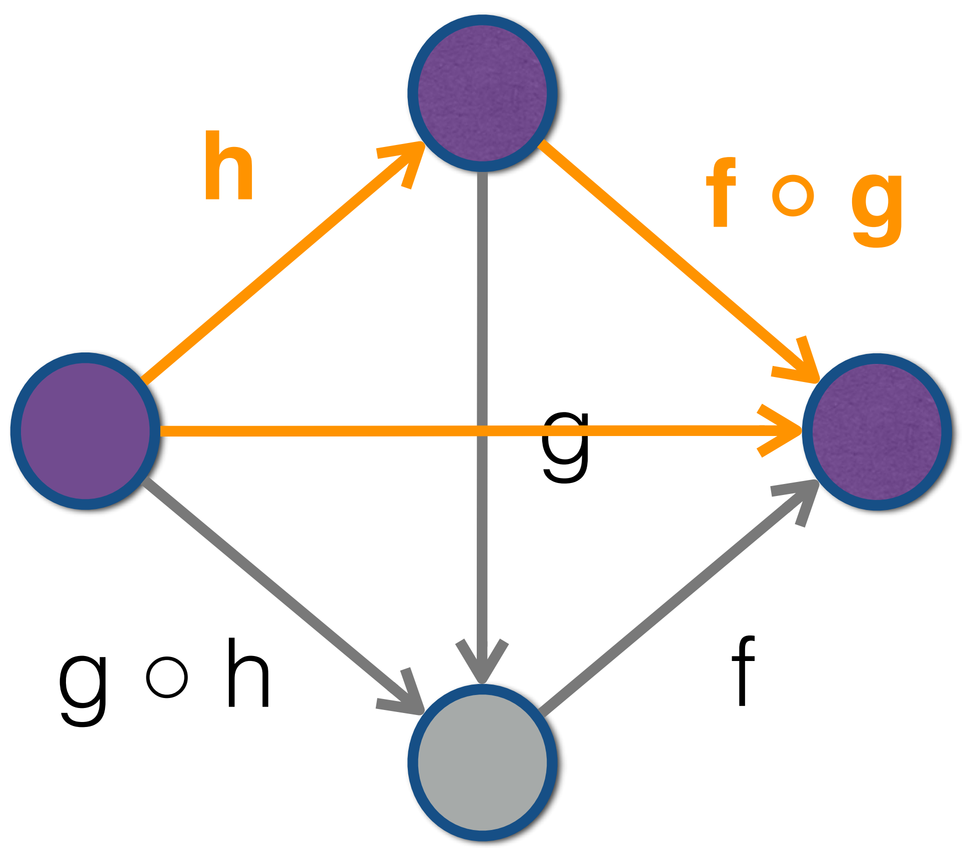
$(f \circ g) \circ h = f \circ (g \circ h)$



Category Axioms

- **Associativity**:

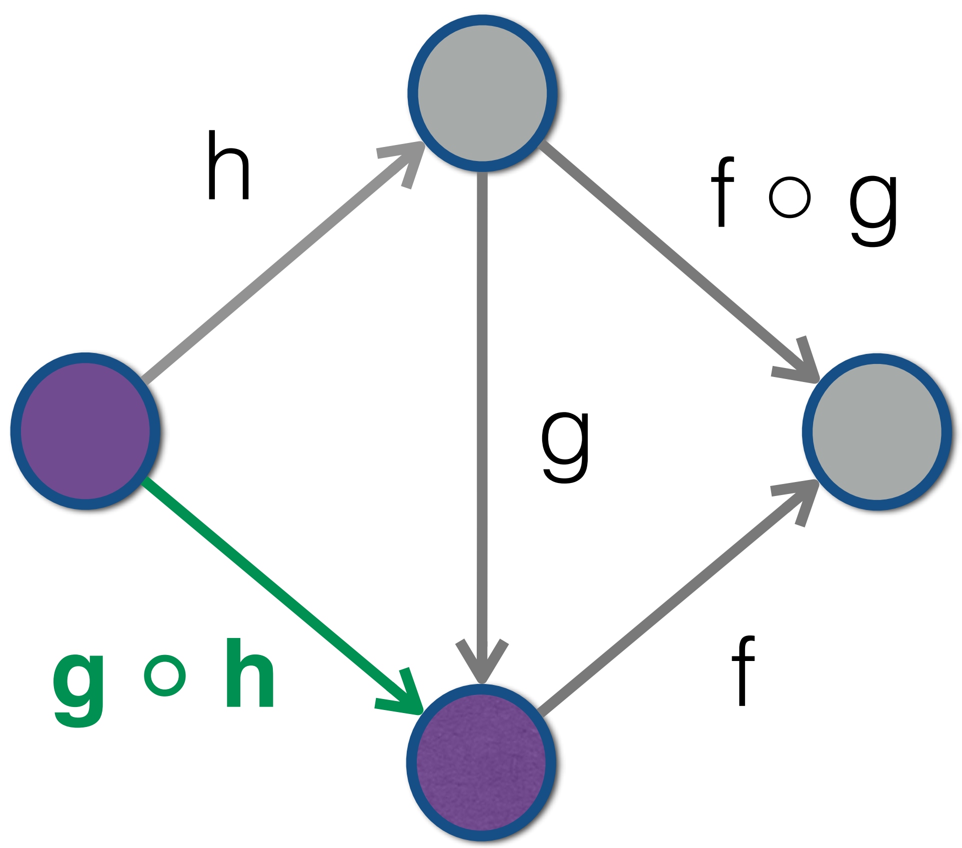
$(f \circ g) \circ h = f \circ (g \circ h)$



Category Axioms

- **Associativity**:

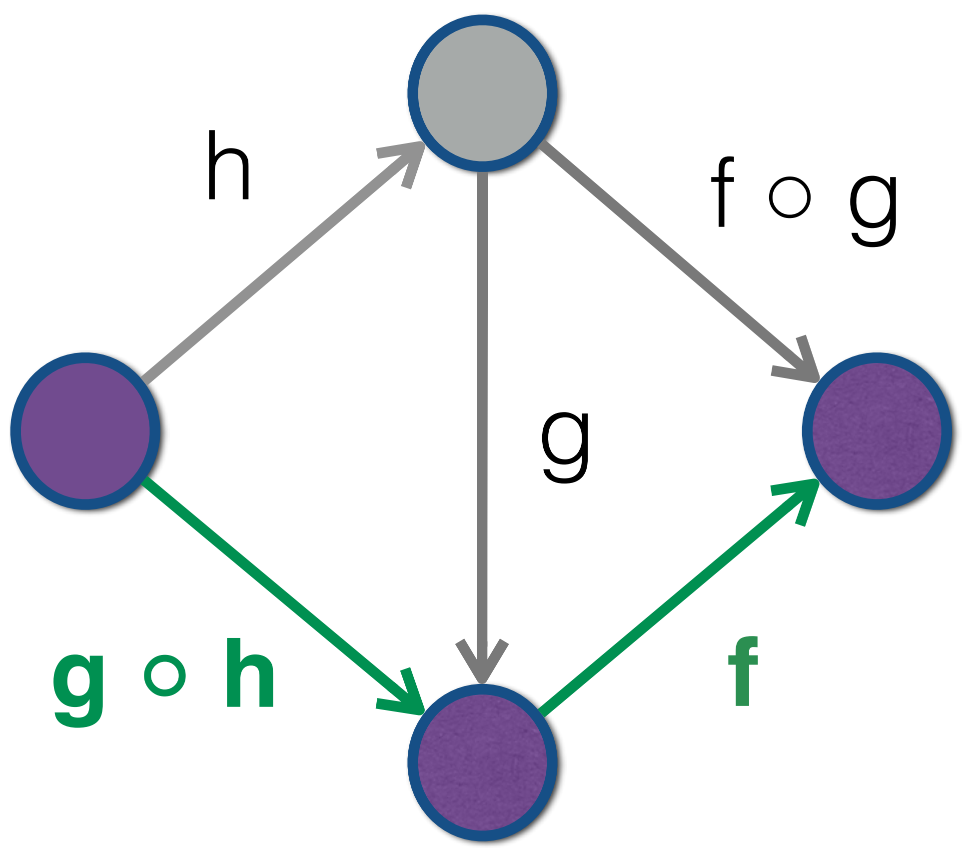
$(f \circ g) \circ h = f \circ \mathbf{(g \circ h)}$



Category Axioms

- **Associativity**:

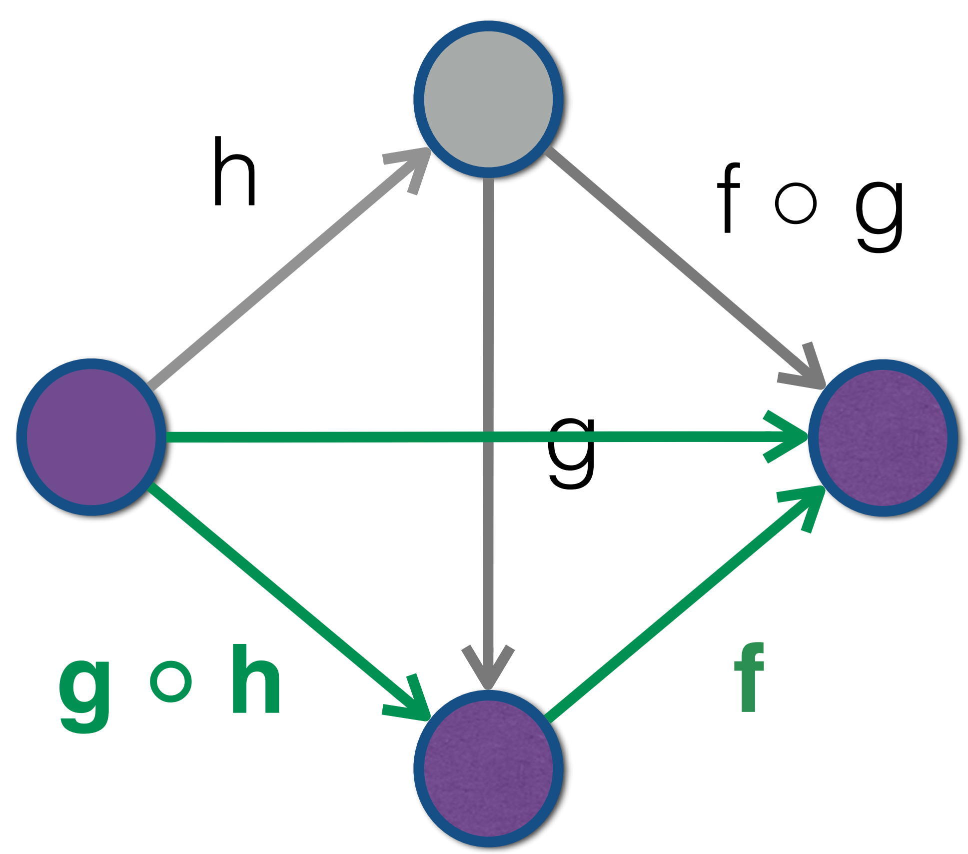
$(f \circ g) \circ h = \mathbf{f \circ (g \circ h)}$



Category Axioms

- **Associativity**:

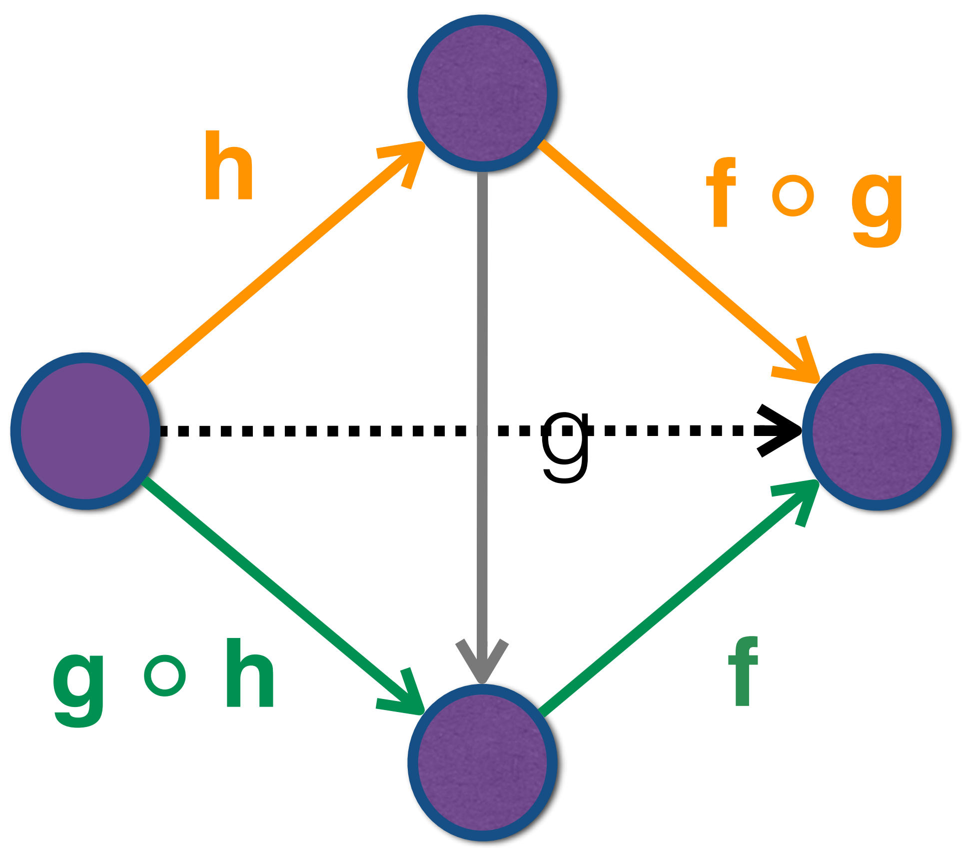
$(f \circ g) \circ h = \mathbf{f \circ (g \circ h)}$



Category Axioms

- **Associativity**:

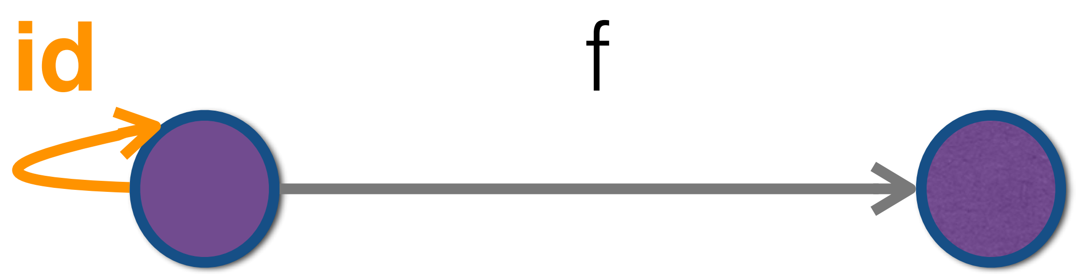
$$(f \circ g) \circ h = f \circ (g \circ h)$$



Category Axioms

- **Identity**:

$f \circ \mathbf{id} = id \circ f = f$



Category Axioms

- **Identity**:

$\mathbf{f} \circ \mathbf{id} = \mathrm{id} \circ f = f$

Category Axioms

- **Identity**:

$f \circ id = id \circ \mathbf{f} = f$

Category Axioms

- **Identity**:

$f \circ id = \mathbf{id} \circ \mathbf{f} = f$

Category Axioms

- **Identity**:

$f \circ id = id \circ f = \mathbf{f}$

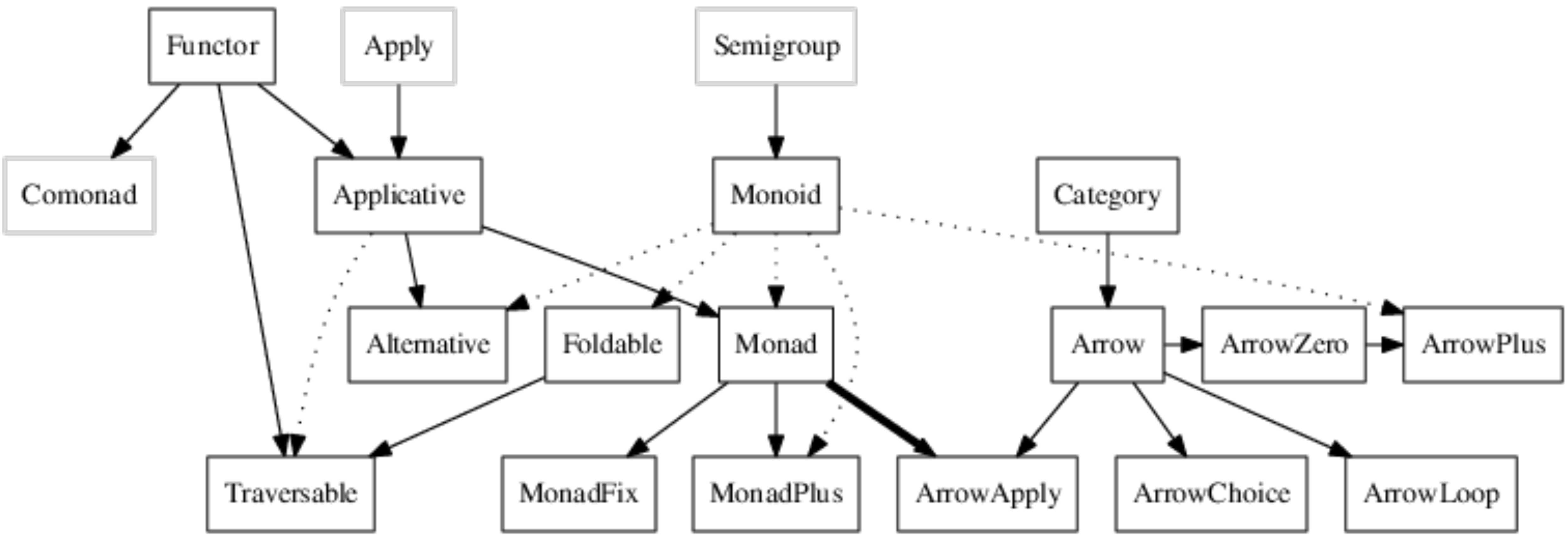
Typeclass pattern

- Coming from Haskell
- Category theory concepts are implemented using this pattern
- Provide ad-hoc polymorphism: different types supporting same operation
- Typeclasses can be seen as Java interfaces but more flexible

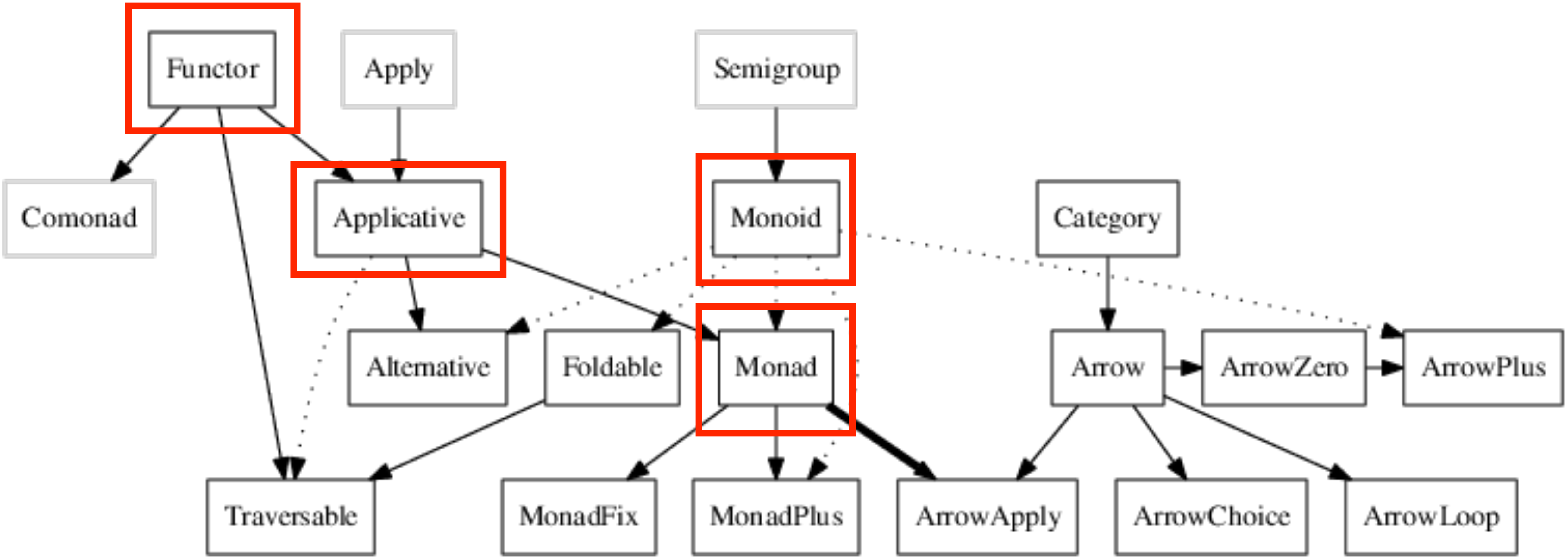
Typeclass pattern

```scala
// A textual representation for types
trait Show[A] {
  def shows(a: A): String
}

implicit val intShow = new Show[Int] {
  def shows(a: Int) = a.toString
}

// We use scala implicits
def shows[A](a: A)(implicit shower: Show[A]) =
                shower.shows(a)

// We can also use context bounds (fancy!)
def shows[A : Show](a: A) = implicitly[Show[A]].
                           shows(a)

shows(3)  // "3"
```

Typeclassopedia



Typeclassopedia

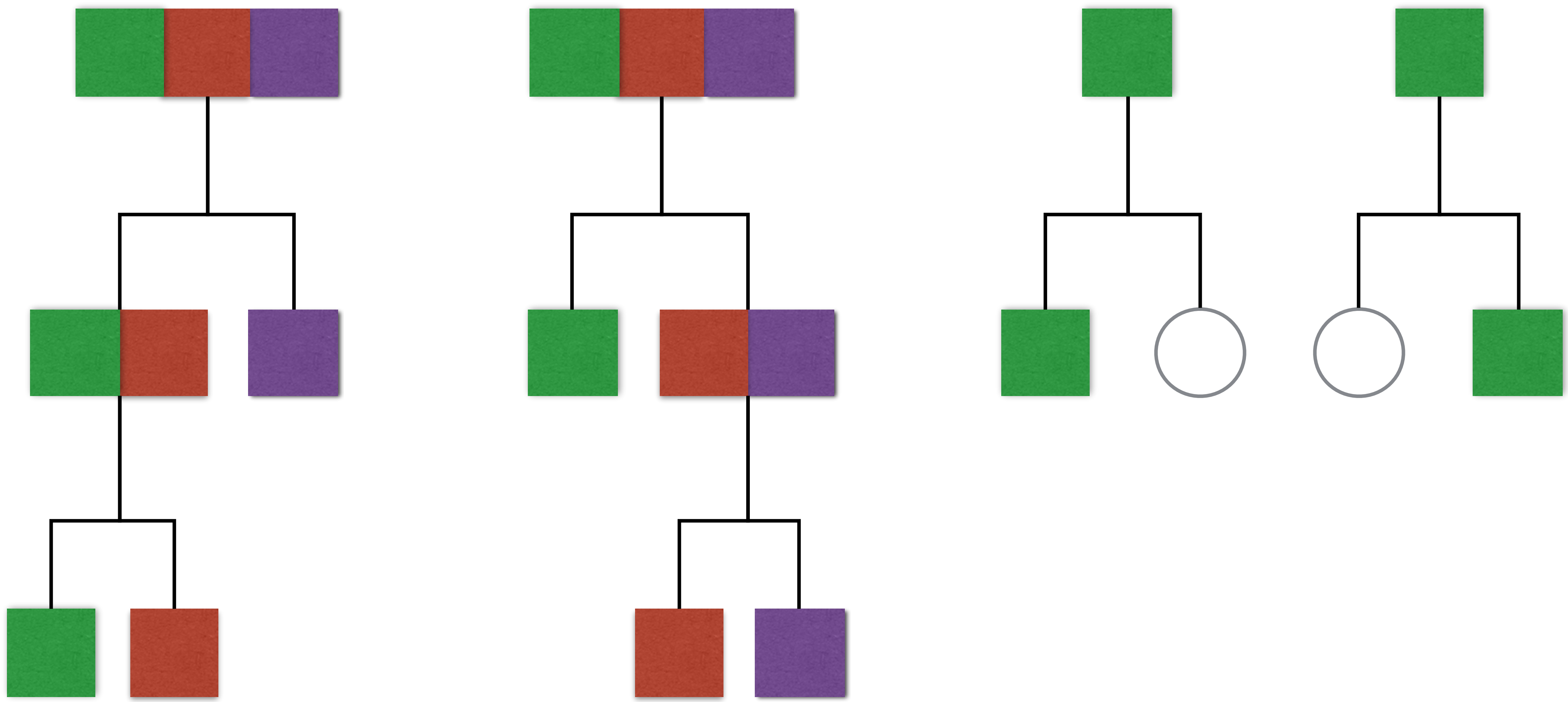


We'll cover **monoids**, **functors**, **applicatives** and **monads.** We'll skip category laws. We'll use **Scalaz** for better syntax.

Monoid

An associative operation |+| with a zero.
Why? We'll behaved "folds", no need to guarantee order!

Monoid.|+|



Monoid

Examples of monoids:

- Concatenation of lists, strings, etc.
- Set union
- “Numeric” addition and multiplication
- All sorts of data structures

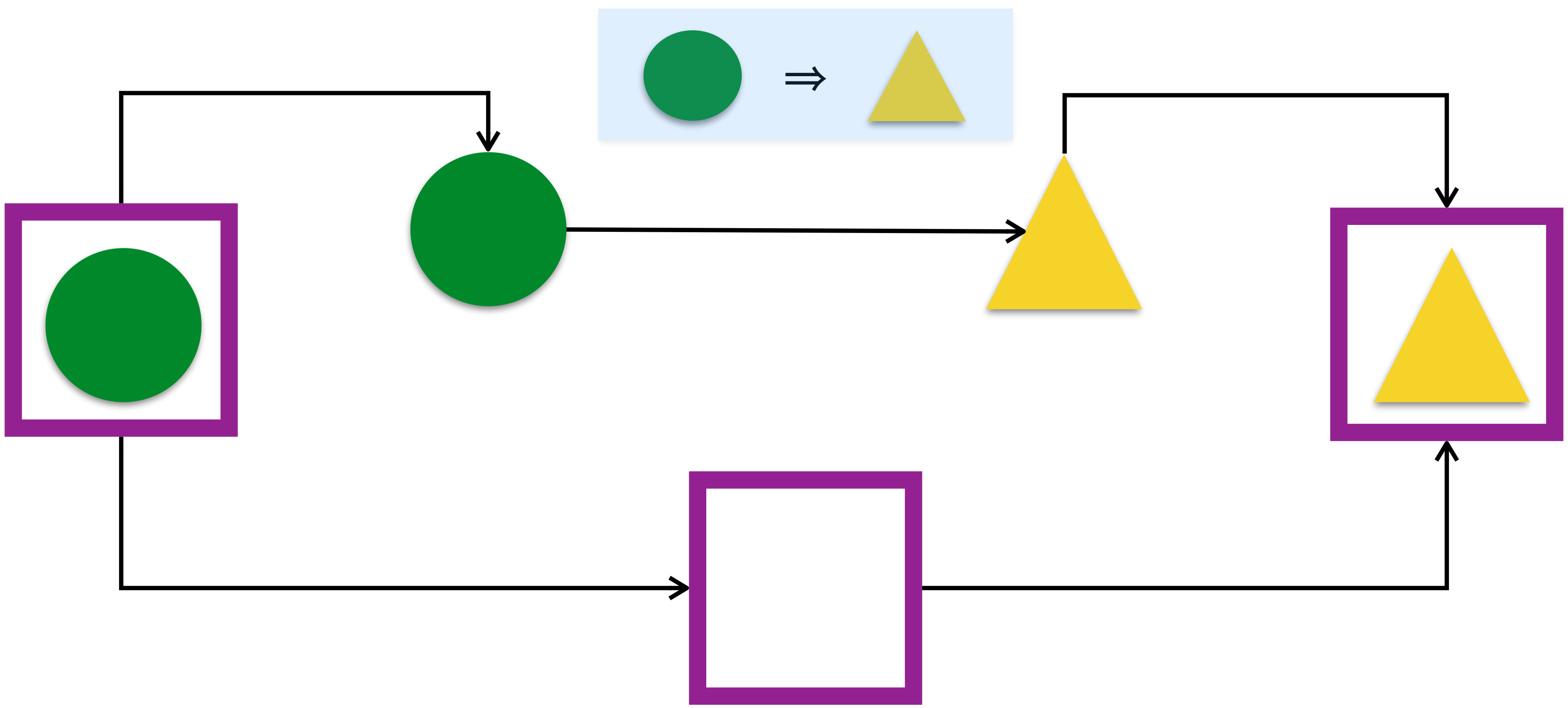
Monoid

```
trait Monoid[F] extends Semigroup[F] {
  /** also known as `id` */
  def zero: F

  /** `|+|` is an alias for `append` */
  def append(f1: F, f2: => F) : F

  // some other out-of-scope stuff
}

trait Semigroup[F] {
  def append(f1: F, f2: => F) : F

  // some other ouf-of-scope stuff
}

List(1,2,3) |+| List(4,5,6,7) // List(1,2,3,4,5,6,7)
3.some |+| 4.some |+| none[Int] // Some(7)
3.some |+| Monoid[Option[Int]].zero // Some(3)
```

Functor

A Functor is something you can *map* over

Functor.**map**



Functor

Examples of Functors:

- List
- Map
- Tree
- Option
- Function (yes, I know)

Functor

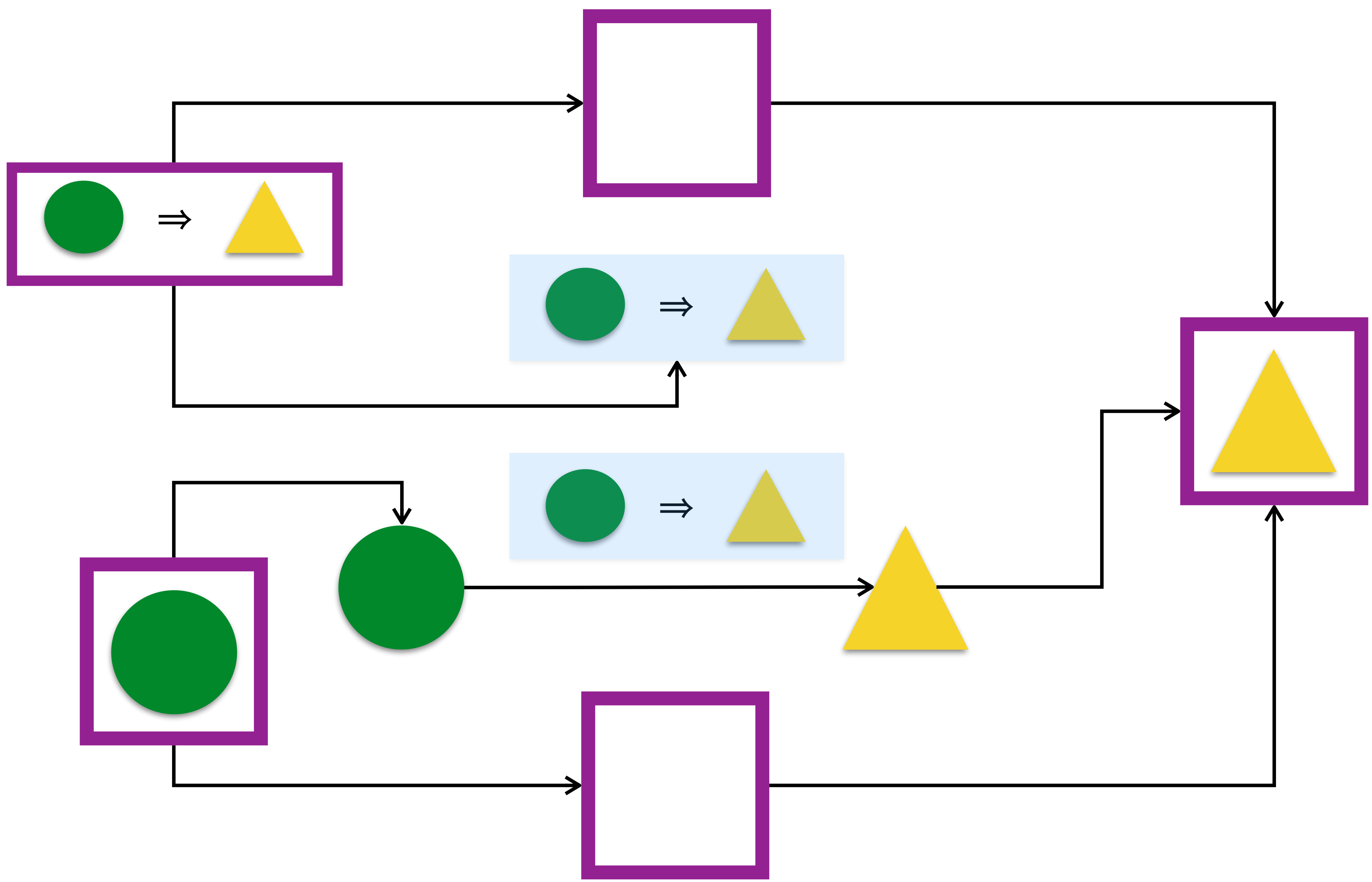
```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B) : F[B]
  // Other stuff out of scope
}

List(1,2,3) map {_ + 1} // List(2,3,4)

(1,2,3) map {_ + 1} // (2,3,4)

val h = ((x: Int) => x + 1) map {_ * 2}
h(4) // ???
```

Applicative

What if a function is actually within a context?

```
val f = List((x:Int) => x+1)
List(1,2,3) map f // throws error
```

Applicative.<*>



Applicative

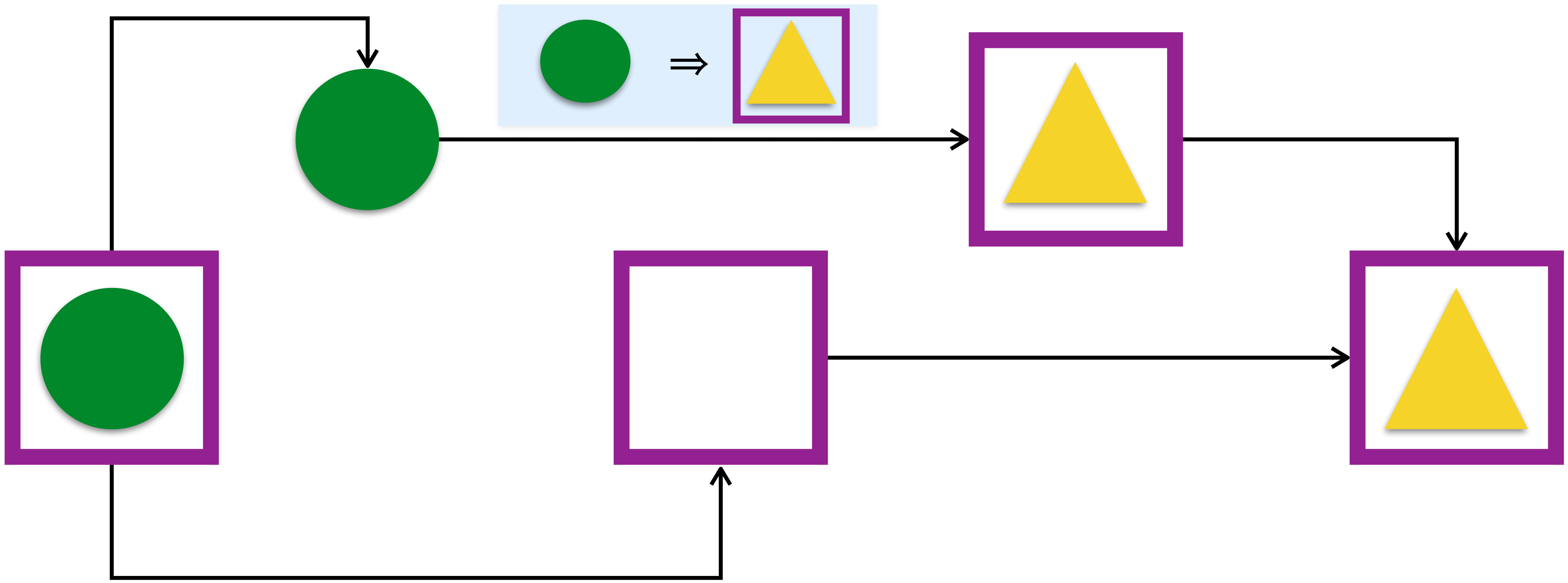
```
trait Applicative[F[_]] extends Apply[F] { self =>
  def point[A](a: => A): F[A]

  /** alias for `point` */
  def pure[A](a: => A): F[A] = point(a)

  // Some other out-of-scope stuff
}

trait Apply[F[_]] extends Functor[F] { self =>
  /** `<*>` is an alias for `ap` */
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]

  // Some other out-of-scope stuff
}

val f = List((x:Int) => x+1)
List(1,2,3) map f // throws error
List(1,2,3) <*> f // List(2,3,4)

// For one-function case, we have applicative builders
^(3.some, none[Int]) {_ * _}  // None
// Or even
(List(1,2) |@| List(3,4)) {_ * _}  // List(3,4,6,8)
```

Monad

What if we use functions whose return value is contextualized?

```
val params = Map("a" -> "1", "b" -> "2", "c" -> "3")

def parse(s: String) : Option[Int] = ...

params.get("a") map parse // Option[Option[Int]]… yuck!
```

Monad.>>=



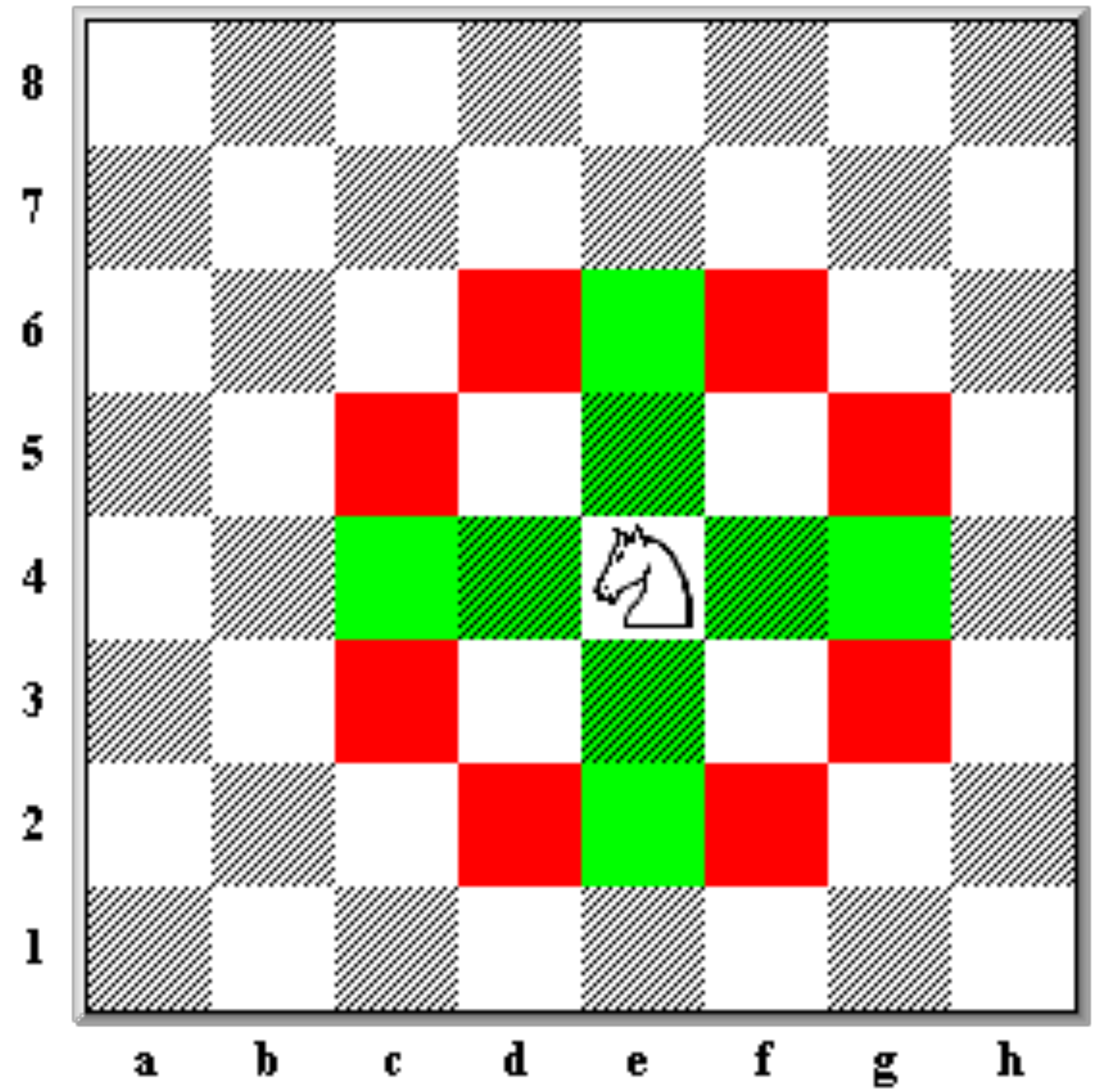
Monad

```scala
trait Monad[F[_]] extends Applicative[F] with Bind[F] {
  /** alias for `bind`, also known as `>>=` in haskell literature */
  def flatMap[A,B](fa: F[A])(f: A => F[B]) : F[B]

  // Some other out-of-scope stuff
}

trait Bind[F[_]] extends Apply[F] { self =>
 /** Equivalent to `join(map(fa)(f))`. */
 def bind[A,B](fa: F[A])(f: A => F[B]): F[B]

 // Some other out-of-scope stuff
}

val params = Map("a" -> "1", "b" -> "2", "c" ->"3")
val a = params.get("a")

a >>= parse // Some(1) :D

// Let's build a calculator!
a >>= parse >>= (a => (params.get("b") >>= parse) >>= (b => (params.get("c")
>>= parse) map (c => a + b + c))) // Some(6) :D
```

Monadic style… wait for it!

```
// We observe a pattern here! Let's refactor a little bit
a >>= parse >>= (a =>
    (params.get("b") >>= parse) >>= (b =>
        (params.get("c") >>= parse) map (c =>
            a + b + c
        )
    )
)

// Generally speaking
monadicA >>= (a =>
monadicB >>= (b =>
monadicC map (c =>
  a + b + c
)))

// Scala for-comprehensions are fancy!
for {
  a <- monadicA
  b <- monadicB
  c <- monadicC
} yield (a + b + c)
```

Monadic style: A Knight's quest



Monadic style: A Knight's quest

```
case class KnightPos(c: Int, r: Int) {
  def move : List[KnightPos] =
    for {
      KnightPos(c2, r2) <- List(
                                KnightPos(c+1, r+2),
                                KnightPos(c+2, r+1),
                                KnightPos(c+2, r-1),
                                KnightPos(c+1, r-2),
                                KnightPos(c-1, r-2),
                                KnightPos(c-2, r-1),
                                KnightPos(c-2, r+1),
                                KnightPos(c-1, r+2)
                                ) if (((1 |-> 8)
                                      contains c2)
                                      && ((1 |-> 8)
                                      contains r2))
    } yield KnightPos(c2, r2)
}

KnightPos(1,2).move // List(KnightPos(2, 4), KnightPos(3, 3), KnightPos(3, 1))
```

Monadic style: A Knight's quest

```scala
case class KnightPos(c: Int, r: Int) {
  . . .

  private def in3: List[KnightPos] =
    for {
      first <- move
      second <- first.move
      third <- second.move
    } yield third

  def canReachIn3(end: KnightPos) : Boolean = in3 contains end
}

KnightPos(1,2) canReachIn3 KnightPos(6,6) // true
KnightPos(1,2) canReachIn3 KnightPos(7,8) // false
```

Monad semantics

A monadic **for comprehension** is an embedded programming language with **semantics** defined by the **monad**:

- **Option**: Anonymous exceptions
- **Validation**: Descriptive exceptions
- **List**: Nondeterministic computation
- **Reader**: Read-only environment
- **Future**: Computation available at some point

Summary

- Functors are nearly everywhere
- Applicative Functors combine **independent** computations
- Monads combine (**possibly dependent**) computations
- These abstractions provide practical value, learn more about them and try to use them in **your** problems

Further reading

- Typeclassopedia
- Learning Scalaz
- Of Algebirds, Monoids, Monads and Other Bestiary for Large-Scale Data Analytics
- Life after Monoids
- Monads are not Metaphors
- Functors, Applicatives and Monads in Pictures


MONADS
MONADS EVERYWHERE
memegenerator.net