CATEGORY THEORY:

An Abstraction For Anything

@alissapajer

TALK OUTLINE

(1) git history as a Category

(2) terminal and initial objects

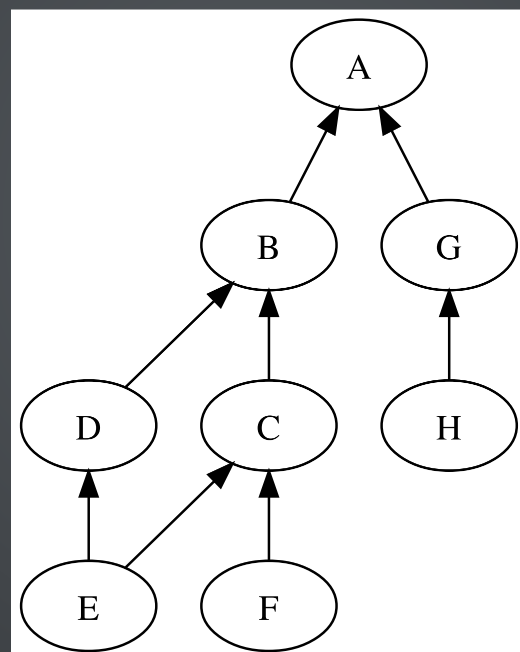
(3) polymorphism as a Natural Transformation

(4) currying as a Natural Transformation

(5) covariance and contravariance of Functors

DIRECTED ACYCLIC GRAPHS

git commit history is a DAG!



DIRECTED ACYCLIC GRAPHS

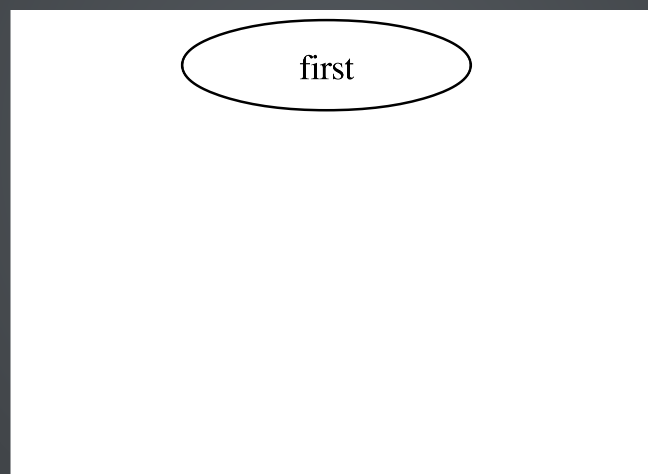
```scala
val currentId = new AtomicInteger(0)
```

```scala
sealed trait Commit {
  def id: Int
}
```

```scala
case class IndivCommit(parent: Commit) extends Commit {
  lazy val id: Int = currentId.getAndIncrement()
}

case class MergeCommit(left: Commit, right: Commit) extends Commit {
  lazy val id: Int = currentId.getAndIncrement()
}

case object FirstCommit extends Commit {
  lazy val id: Int = currentId.getAndIncrement()
}
```

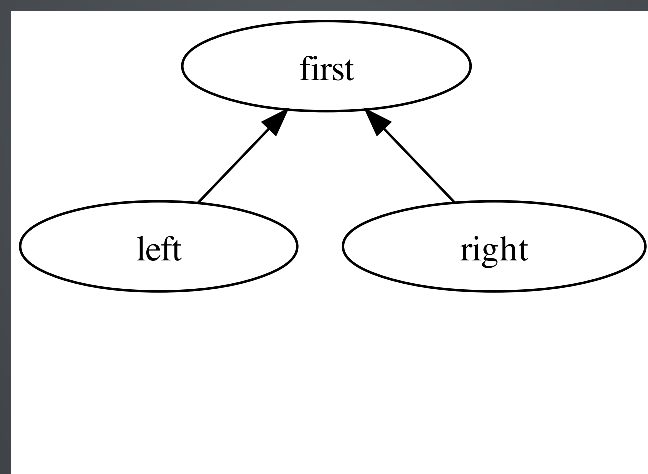
GIT MERGE GRAPH

```
val first = FirstCommit
```



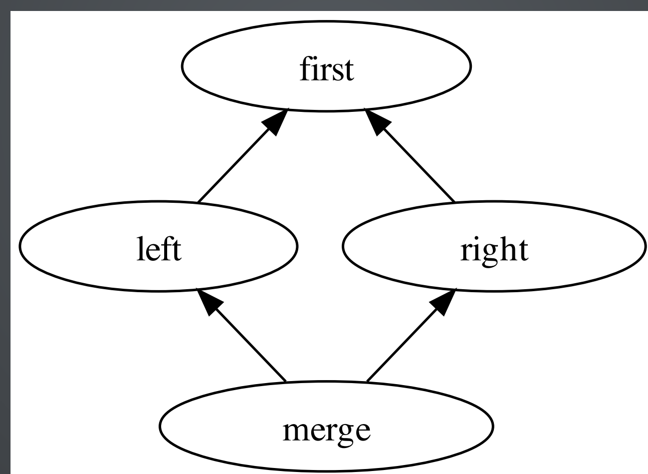
GIT MERGE GRAPH

```
val first = FirstCommit

val left = IndivCommit(first)

val right = IndivCommit(first)
```



GIT MERGE GRAPH

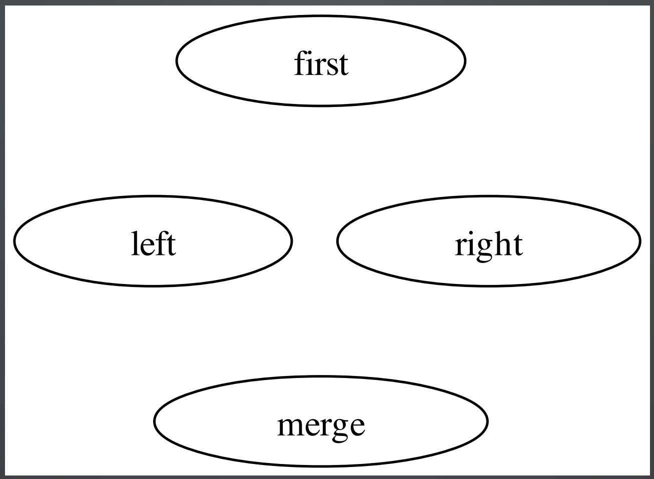
```
val first = FirstCommit

val left = IndivCommit(first)

val right = IndivCommit(first)

val merge = MergeCommit(left, right)
```



CREATING A CATEGORY

1. A collection of objects

The nodes in our graph of commits will be the objects.

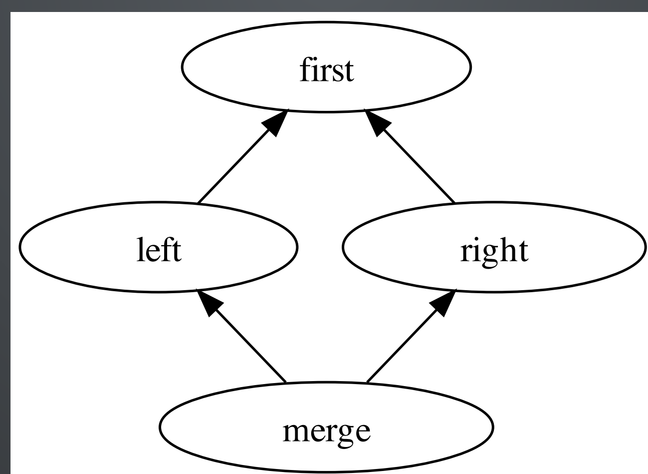


CREATING A CATEGORY

2. A set of morphisms or arrows between every two objects

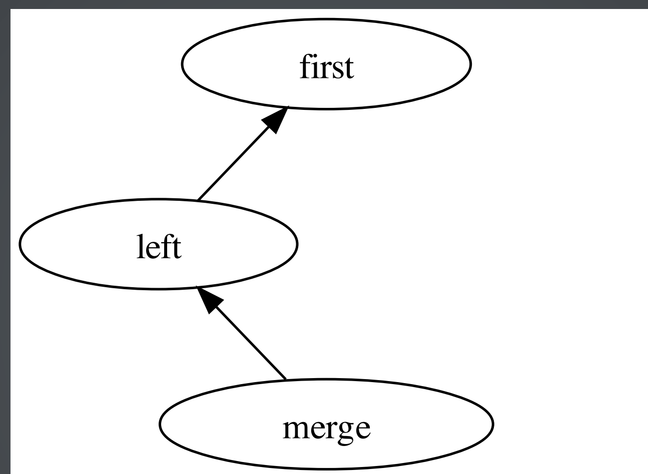
```
Hom(first, left) = { }
Hom(merge, first) = { ≤_MF }
Hom(left, left) = { ≤_LL }
...
```

morphisms governed by reachability



CREATING A CATEGORY

3. A way to compose morphisms



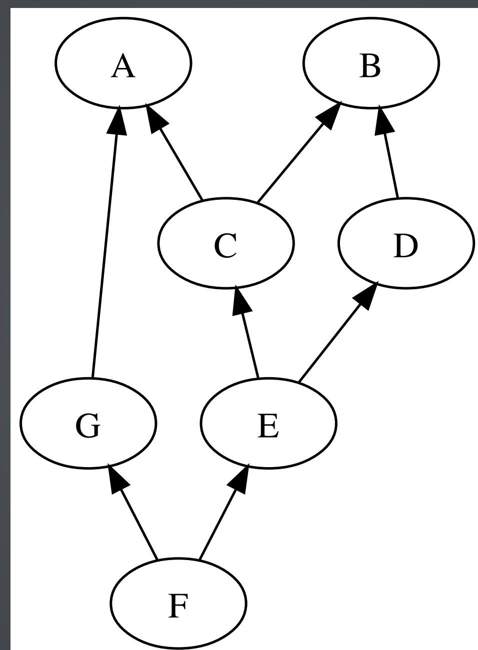

```
≤_ML * ≤_LF = ≤_MF
```

(a) compositional associativity holds
(b) an identity morphism exists for each object

TERMINAL AND INITIAL OBJECTS

Object F is an initial object:

For every object X, Hom(F, X) has size one



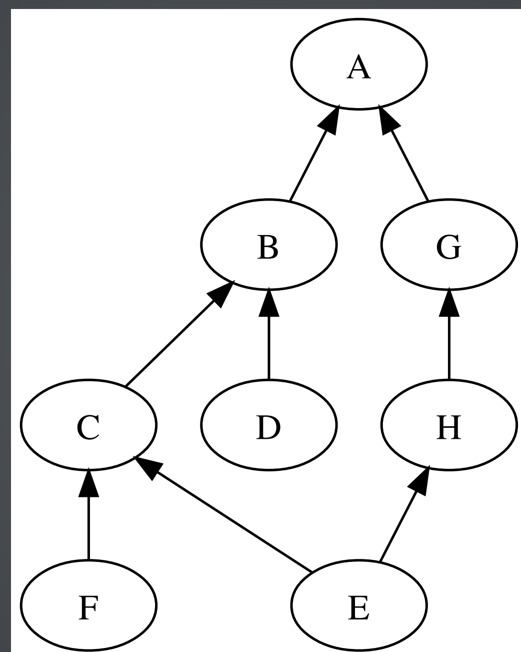
(i.e. F ≤ X for all X)

(i.e. all objects are reachable from F)

TERMINAL AND INITIAL OBJECTS

Object A is a terminal object:

For every object X, Hom(X, A) has size one



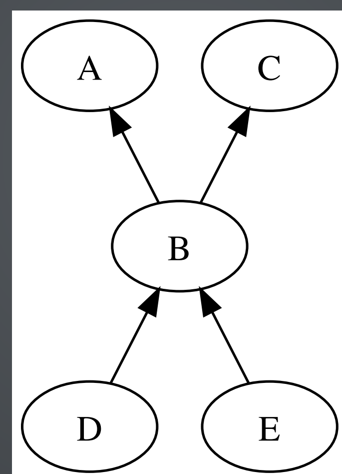
(i.e. X ≤ A for all X)

(i.e. all objects can reach to A)

TERMINAL AND INITIAL OBJECTS

Is a terminal or initial object guaranteed to exist in a DAG?

No!

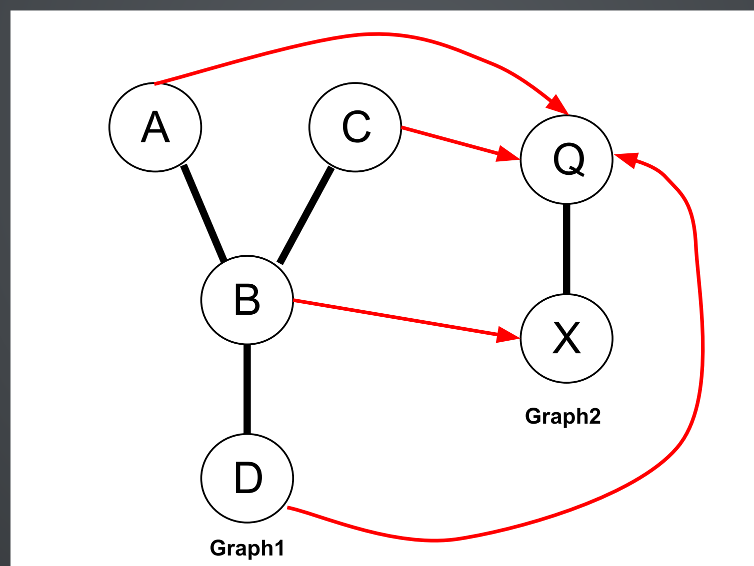


CATEGORY OF ALL THE GRAPHS

Objects: Graphs themselves

Morphisms: Structure-preserving graph homomorphisms

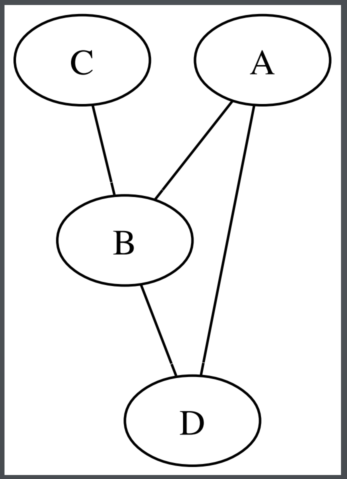
(i.e. adjacent nodes map to adjacent nodes)

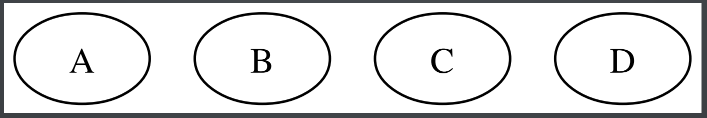


FORGETFUL FUNCTOR

Functors are morphisms between categories

Forget: **GRAPH** -> **SET**





OPTION FUNCTOR

Option[_] is parametrized on a Scala type

Option[_] gives us a way to convert from A to Option[A]

```
scala> def optionize[A](a: A): Option[A] = Some(a)
optionize: [A](a: A)Option[A]
```

Option[_] is a Functor on the category of Scala types

What about the morphisms from one type to another?

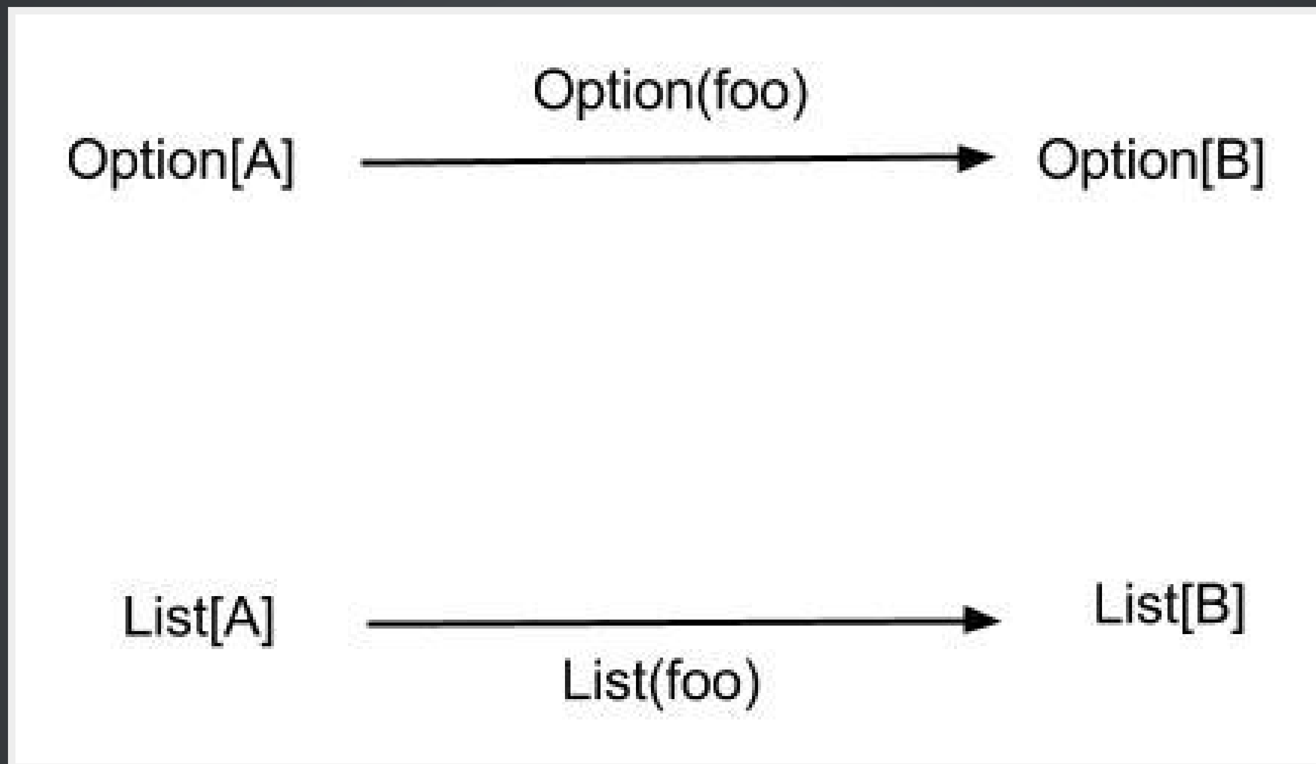
OPTION FUNCTOR

```
scala> def optionF[A, B](f: A => B)(optA: Option[A]): Option[B] = {
     |   optA match {
     |     case None => None
     |     case Some(a) => Some(foo(a))
     |   }
     | }
optionF: [A, B](f: A => B)(optA: Option[A])Option[B]
```

Option transforms morphisms and objects from the category of Scala types to itself

OPTION FUNCTOR

```
def foo[A, B](a: A): B = { ... }
```



How can we transform Option[T] into List[T]?

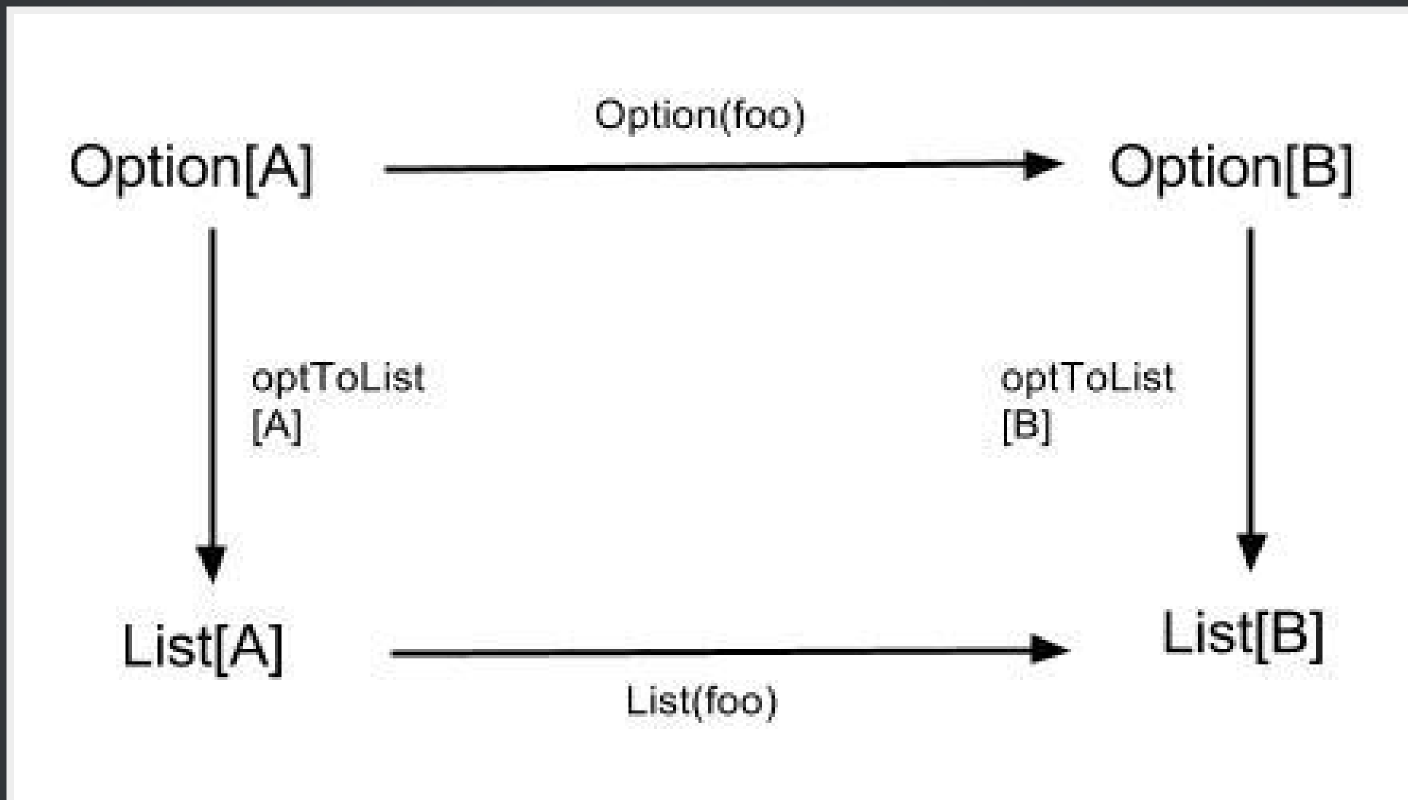
POLYMORPHIC FUNCTIONS

(parametric polymorphism)

```
def optToList[A](as: Option[A]): List[A] = {
     |   as match {
     |     case Some(a) => List[A](a)
     |     case None => List.empty[A]
     |   }
     | }
optToList: [A](as: Option[A])List[A]

scala> optToList[Int](Some(4))
res4: List[Int] = List(4)

scala> optToList[String](Some("foobar"))
res5: List[String] = List(foobar)

scala> optToList[Int](None)
res7: List[Int] = List()
```

NATURAL TRANSFORMATIONS

optToList[T] is a polymorphism and a Natural Transformation

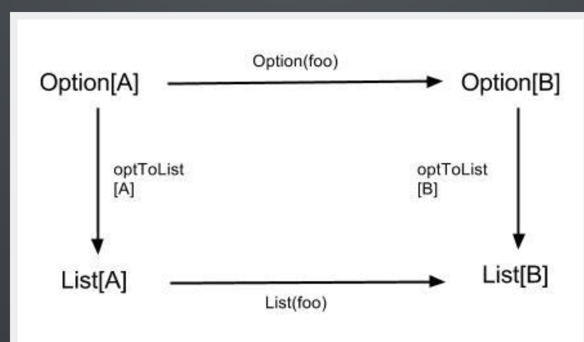


This diagram commutes

NATURAL TRANSFORMATIONS

For every object X
we have nat trans Option[X] -> List[X]
such that
for every morphism foo: X->Y
the diagram commutes

If we can define the same nat trans for every type A,
then this nat trans is a polymorphism!



CURRYING

(aka *schönfinkelzation)*

```
def uncurried(x: Int, y: Int): Int = x + y
uncurried: (x: Int, y: Int)Int

scala> def curried(x: Int): Int => Int = (y: Int) => x + y
moreCurried: (x: Int)Int => Int

scala> uncurried(2, 3)
res5: Int = 9

scala> curried(2)(3)
res4: Int = 5
```

Currying provides another example of a natural transformation

HOM FUNCTORS

Hom(X, Y) is the set of morphisms from object X to object Y

Hom(_, Y) is a functor

```
Hom(_, B): SomeCategory => Set
```

Apply Hom(_, B) to an object A

```
Hom(A, B)
```

Apply Hom(_, B) to a morphism f: P -> Q

```
Hom(Q, B) => Hom(P, B)
```

HOM FUNCTORS

Apply Hom(_, B) to a morphism f: X->Y

```
def f[X, Y](x: X): Y = { ... }
```

the "logical" (but wrong) idea

```
Hom(X, B) => Hom(Y, B)

X->Y,  X->B
```

the correct idea: contravariance

```
Hom(Y, B) => Hom(X, B)

X->Y, Y->B
```

COMIC INTERLUDE

CURRYING

```
curry[A]: Hom(A x B, C) => Hom(A, Hom(B, C))
```

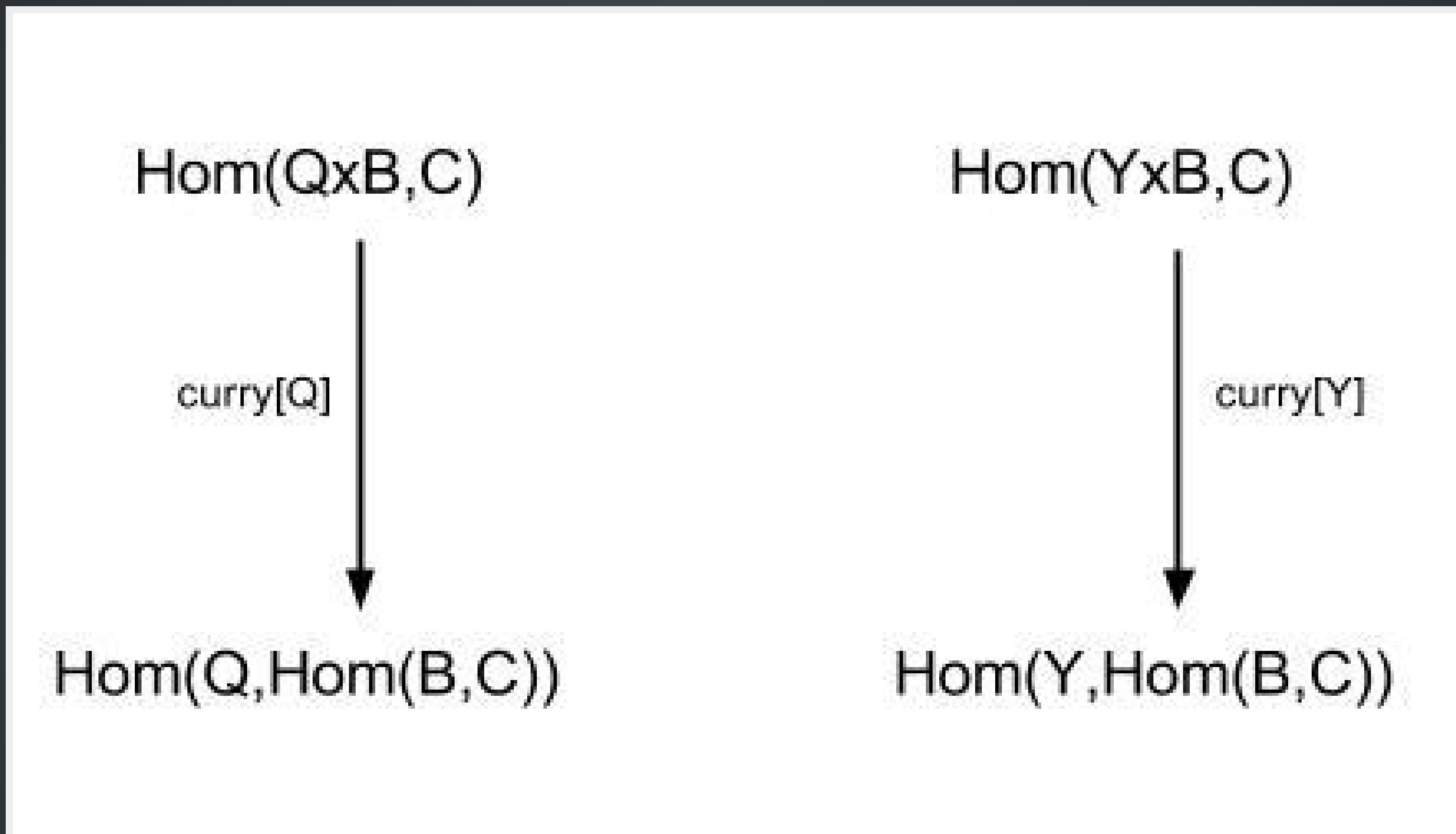
```
curry[_]: Hom(_ x B, C) => Hom(_, Hom(B, C))
```

A functor transforms one category into another

Currying transforms one functor into another!

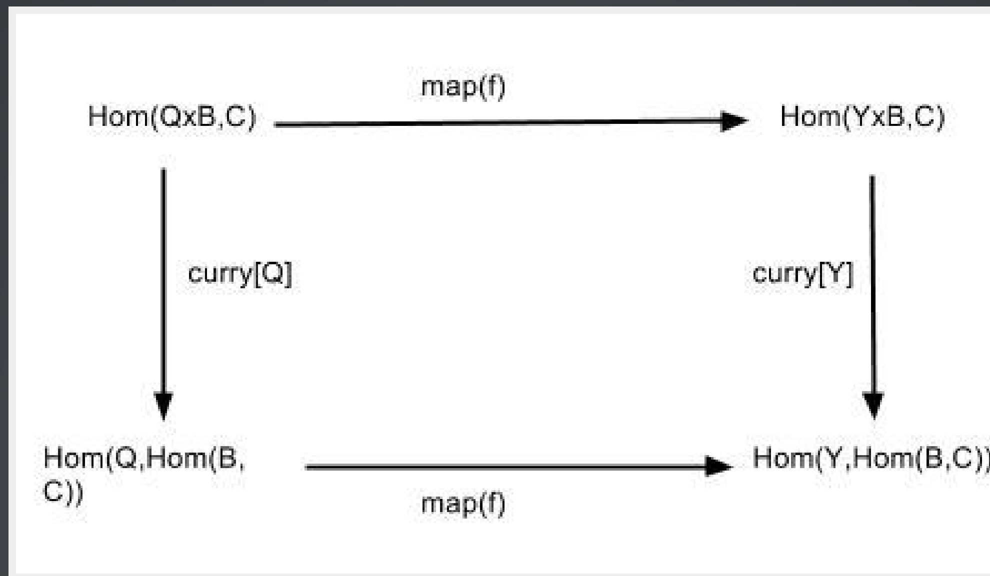
This sounds familiar...

CURRYING



Can we create a diagram that commutes?

CURRYING



For every object Q
we have nat trans Hom(QxB,C) -> Hom(Q,Hom(B,C))
such that
for every morphism f: Y->Q
the diagram commutes

CURRYING

Hom functor for type C instead

```
curry[C]: Hom(A x B, C) => Hom(A, Hom(B, C))
```

```
curry[_]: Hom(A x B, _) => Hom(A, Hom(B, _))
```

(google term: _ x B and Hom(B, _) are adjoint functors)

CURRYING

$$(A \times B) \Rightarrow C$$

```
def uncurried(x: A, y: B): C = { ... }
```

$$A \Rightarrow (B \Rightarrow C)$$

```
def curried(x: A): B => C = { ... }
```

There is a correspondence (isomorphism!) between the set of functions (A x B) -> C and the set of functions A -> (B -> C)

QUESTIONS?