First Intuition of Category Theory

wilberchao 2019/2/26

Agenda

- Definition of Category Theory
- Functor
- Natural Transformation

Definition of Category Theory

- Primitive
  - Object
  - Morphism
- Properties
  - Composition
  - Identity
  - Associativity

Object

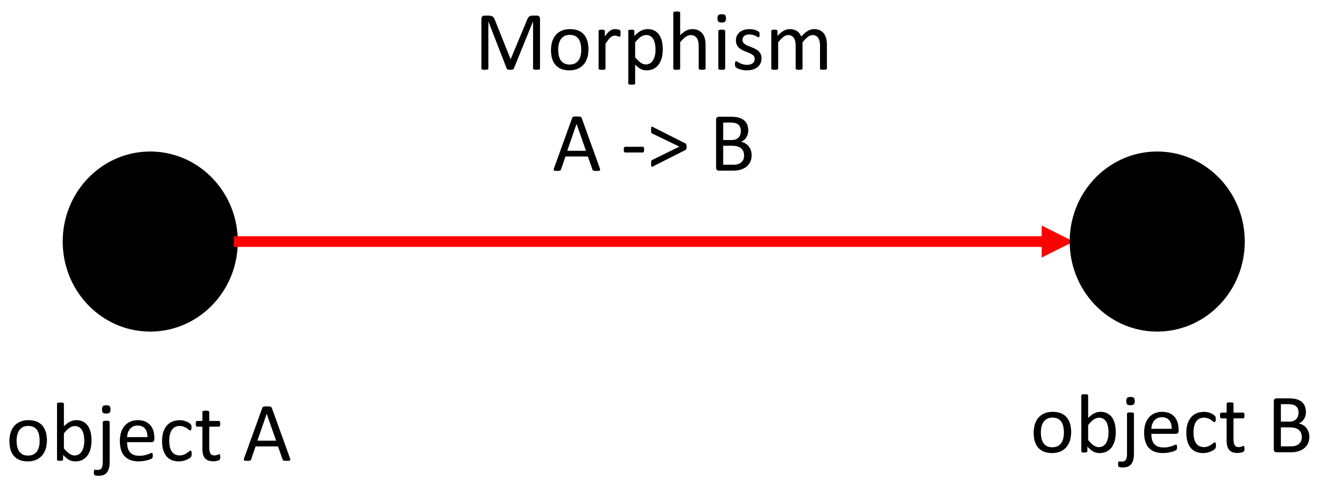
- Not Object in Object-Orient programming
- Not Class in Object-Orient programming
- Just a point
- Primitive
- No properties

Morphism

- Primitive
- No properties
- Just an arrow
  - Identifying start and end

Object - Morphism

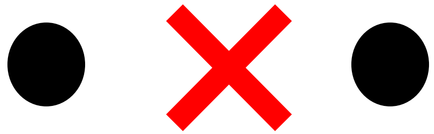
- The object is used to identify the start and end of the morphism
- The morphism shows the relationship between objects



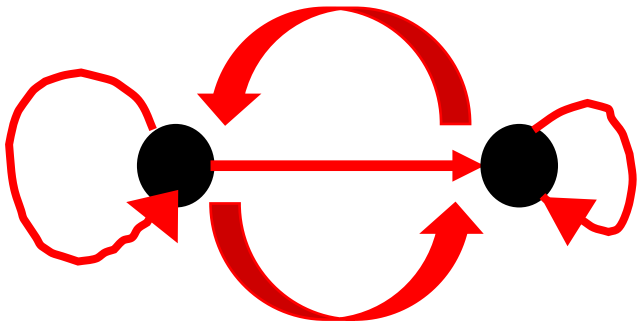
Object – Morphism

only one morphism

no morphism

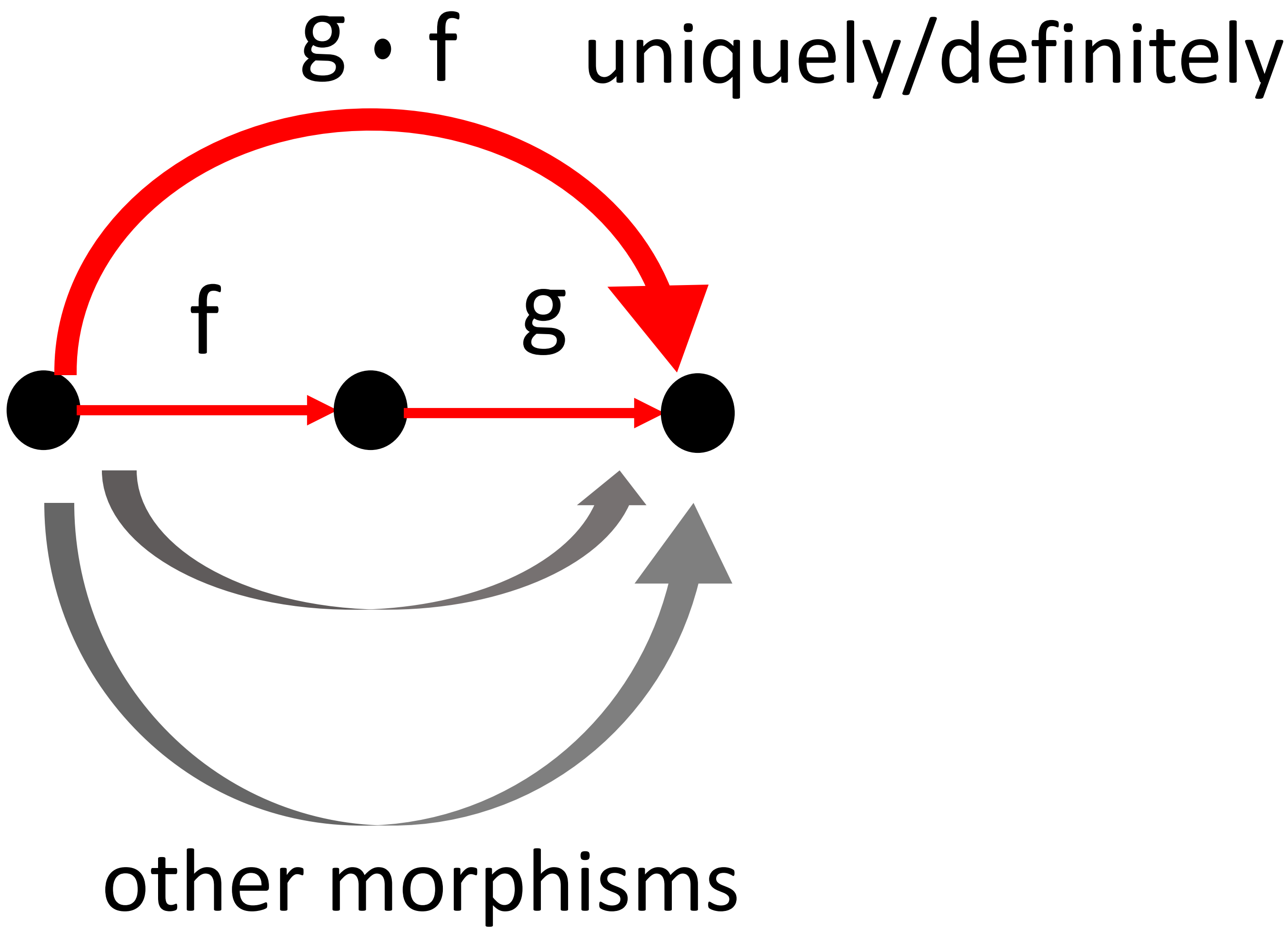


many morphisms



infinite morphisms

Composition

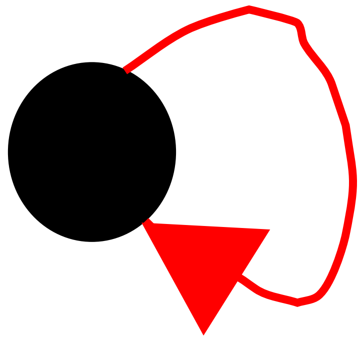


Composition

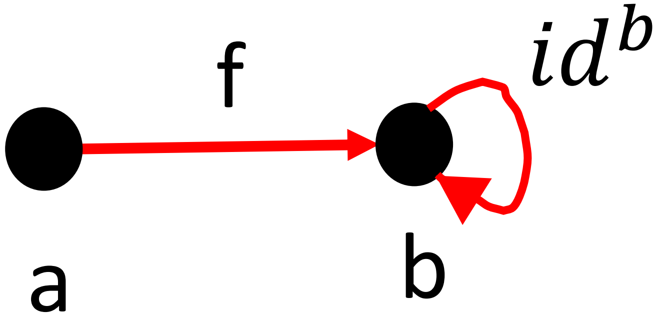
- Objects, morphisms don’t have structures
- Composition has structures
  - Encoding everything into compositions

Identity

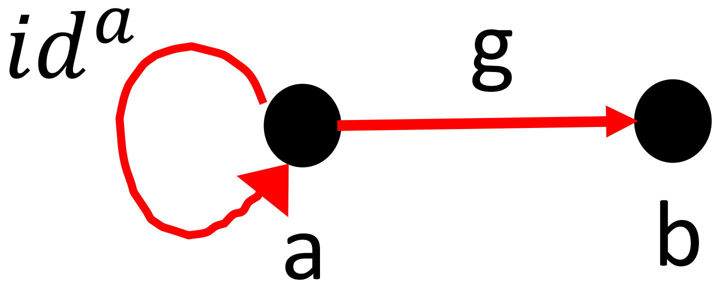
Every object has one identity morphism at least



Identity



$id^b \circ$ f = f   left identity



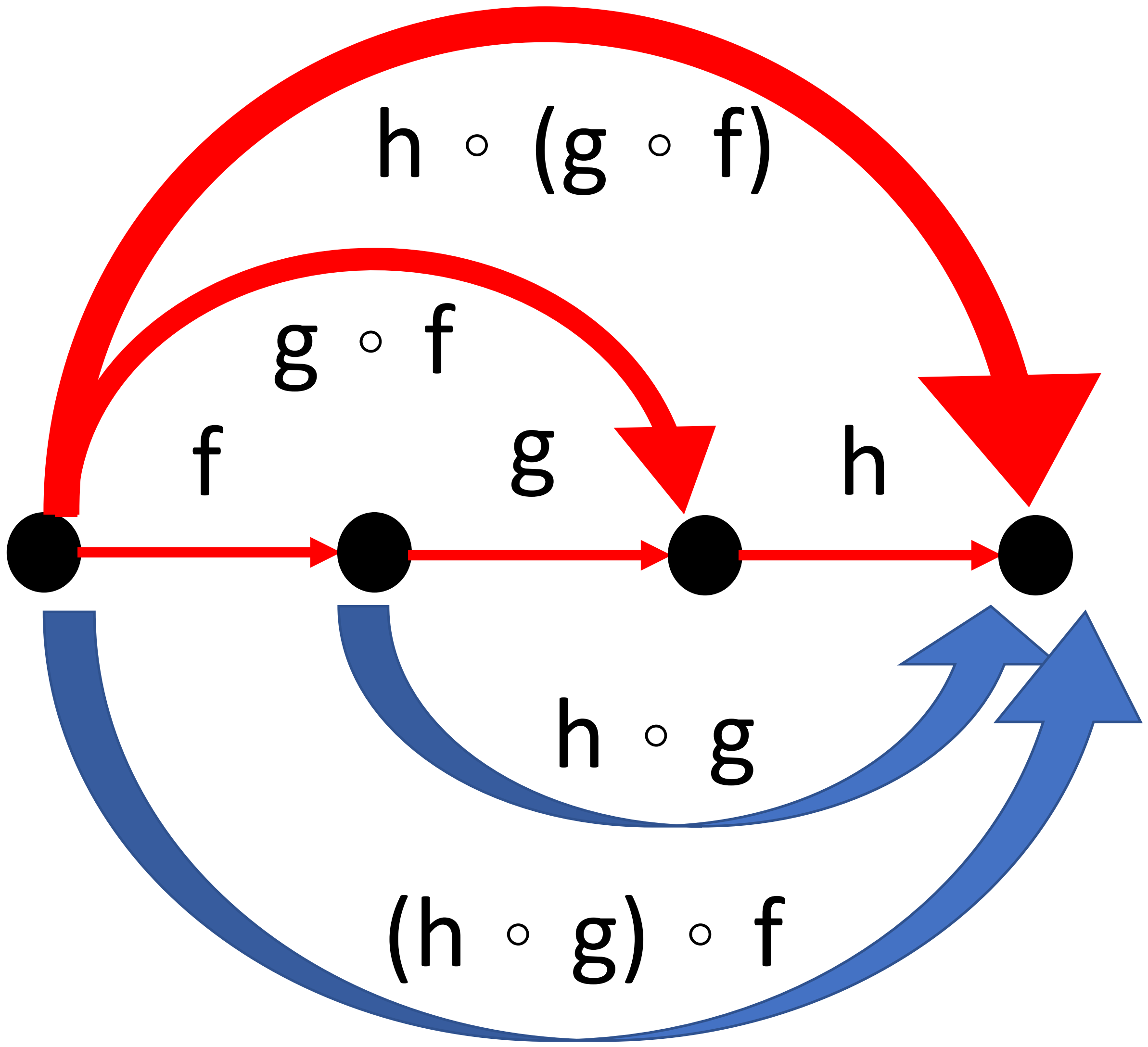
$g \circ id^a$ = g   right identity

Associativity

$h \circ (g \circ f)$

$=$

$(h \circ g) \circ f$



Set Category

- Object: Set
- Morphism: functions between sets

Scala Category

- Object: Type
- Morphism: Function

```
scala> val f:Int => String = a => a.toString
f: Int => String = $$Lambda$1079/671384775@5a6f6cac

scala> val g: String => Long = s => 0L
g: String => Long = $$Lambda$1080/202968316@77ce8bc5

scala> f andThen g
res2: Int => Long = scala.Function1$$Lambda$1078/1861236708@335cdd2

scala> g compose f
res3: Int => Long = scala.Function1$$Lambda$1218/860285190@443ac5b8
```

Why Referential transparency

- Making functions produce a value rather than effects
  - val f: JDBC => IO[Read Mysql]
  - val g: JDBC => Future[Read Mysql]
- We can compose functions which produce value
- We can't compose functions which produce effect
  - Effects disobey the associativity of composition

Functor ?

`Functor` is a type class that abstracts over type constructors that can be `map`'ed over. Examples of such type constructors are `List`, `Option`, and `Future`.

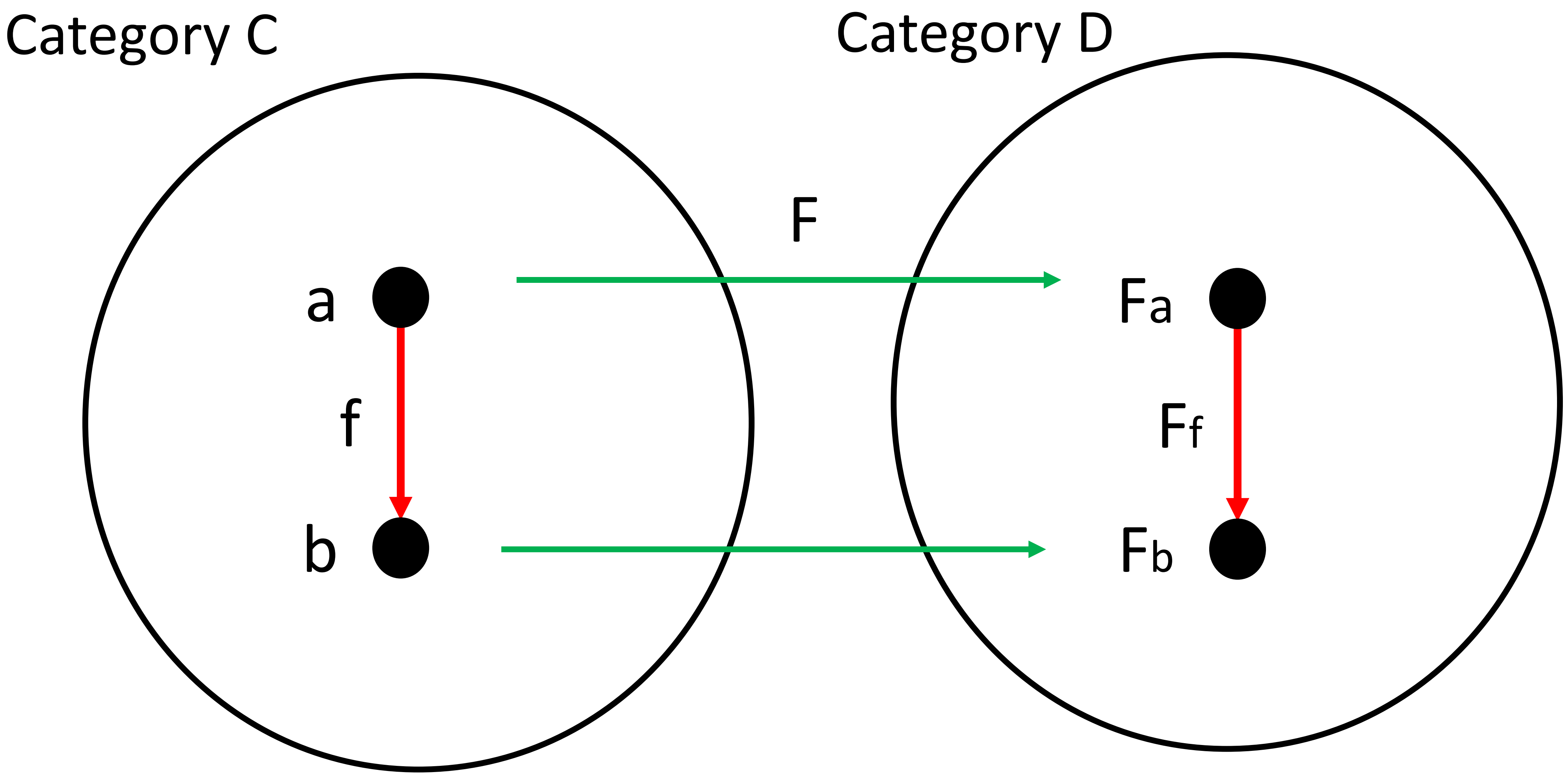
```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

// Example implementation for Option
implicit val functorForOption: Functor[Option] = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa match {
    case None    => None
    case Some(a) => Some(f(a))
  }
}
```

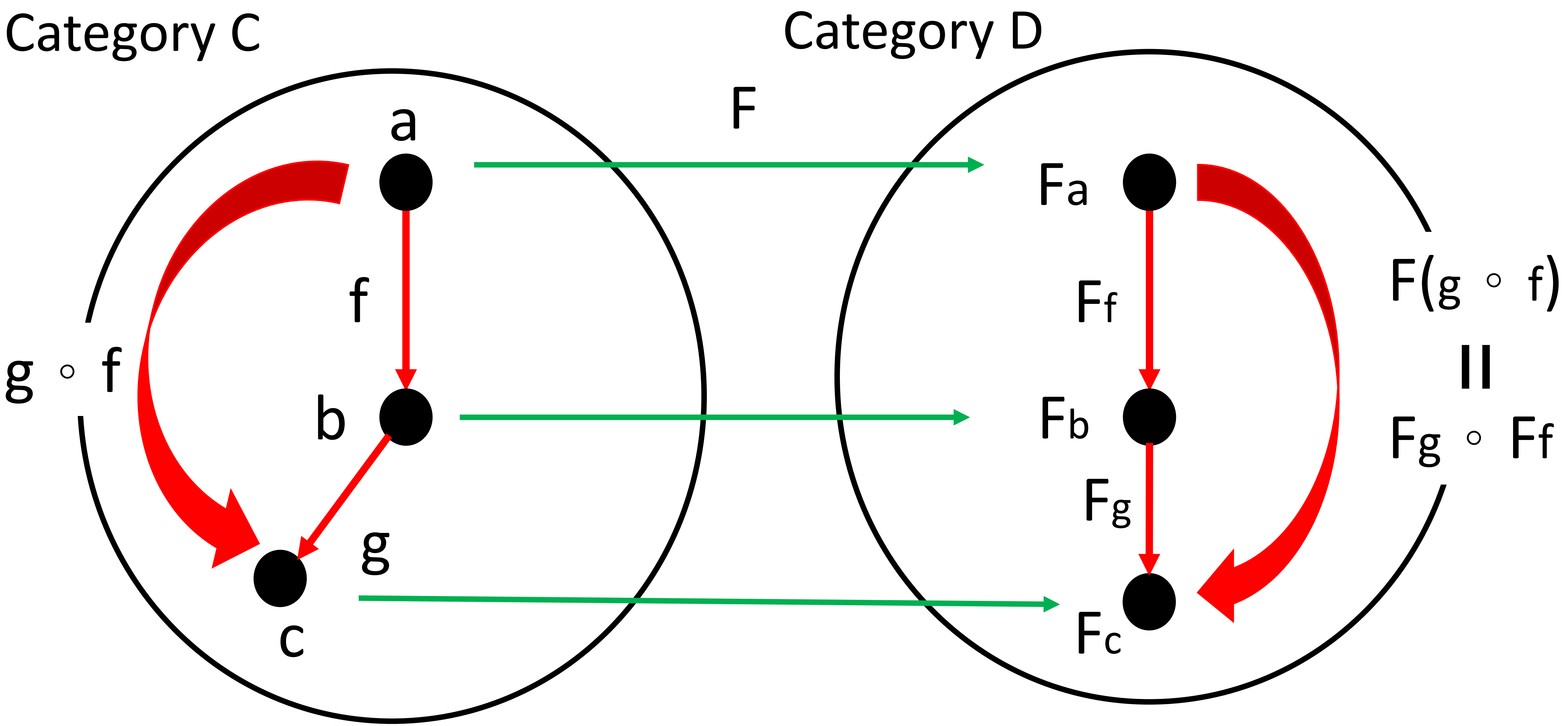
Functor

- Mappings between categories
  - Object
  - Morphism
- Preserving Structure

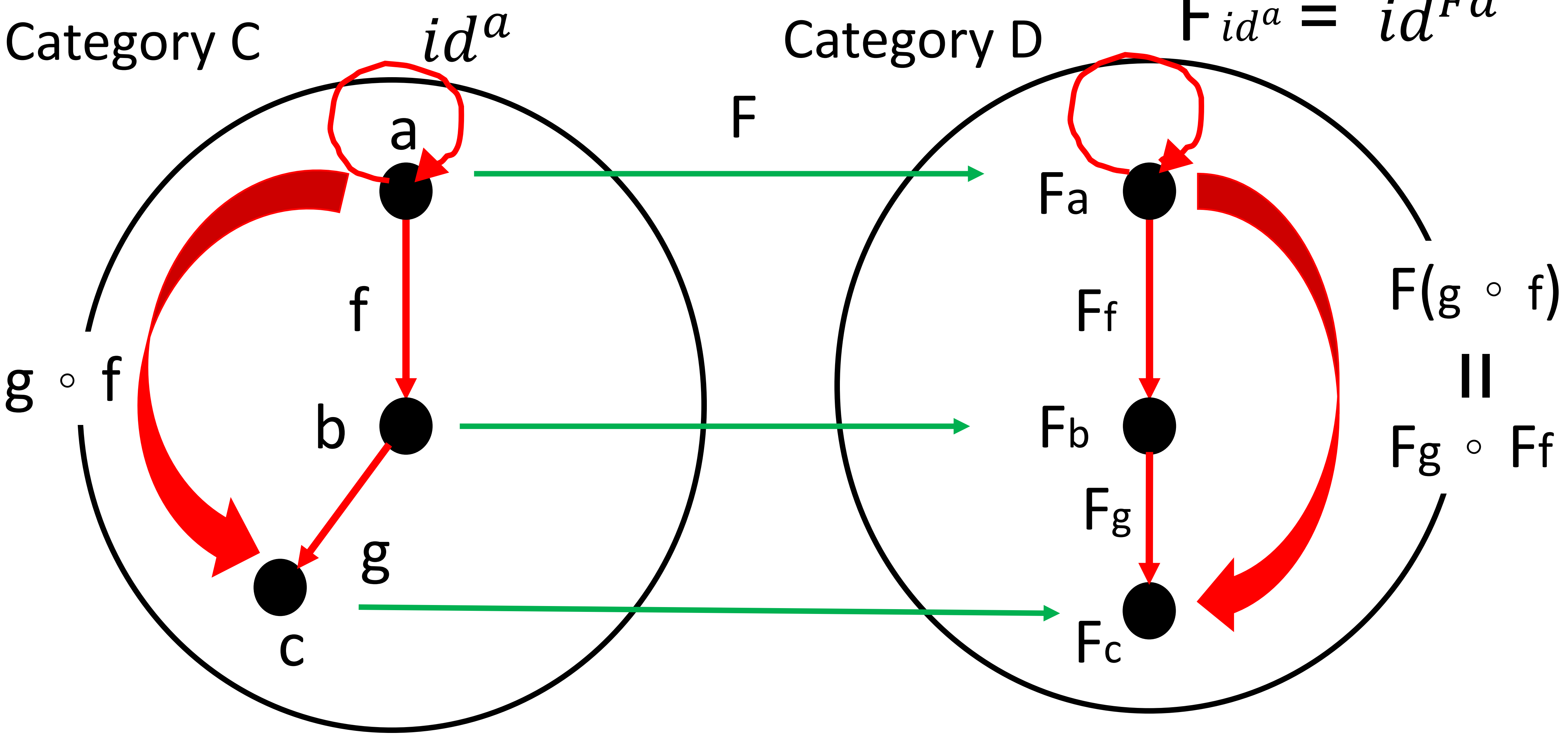
Functor Mapping



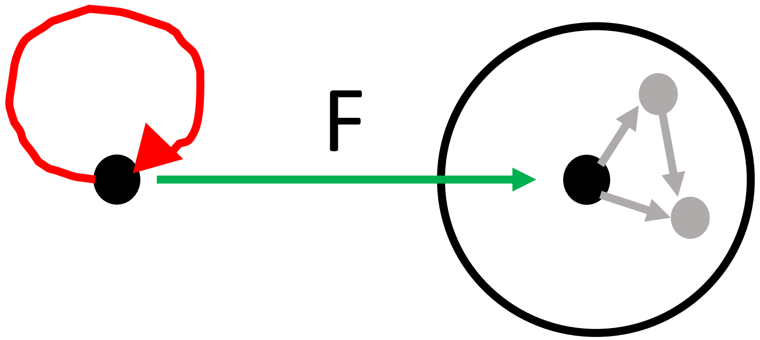
Functor Preserving Structure



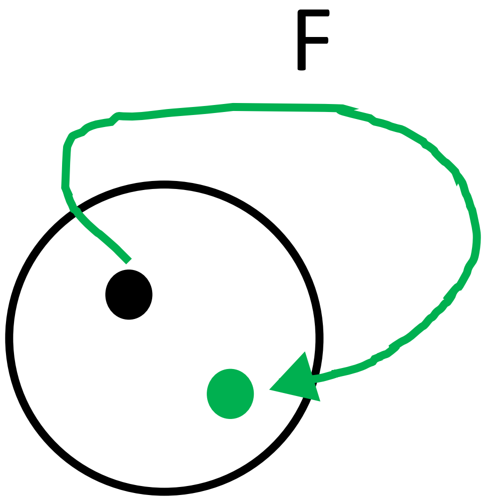
Functor Identity



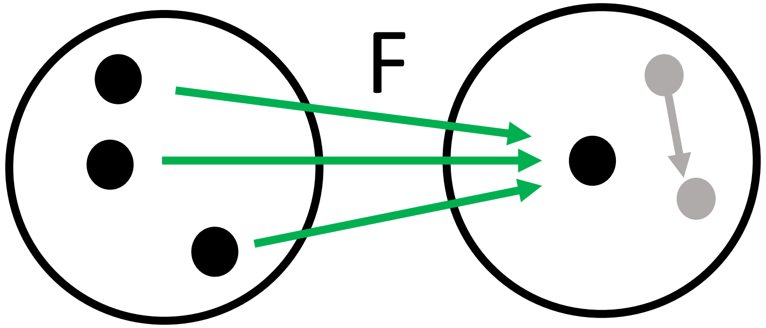
Functors



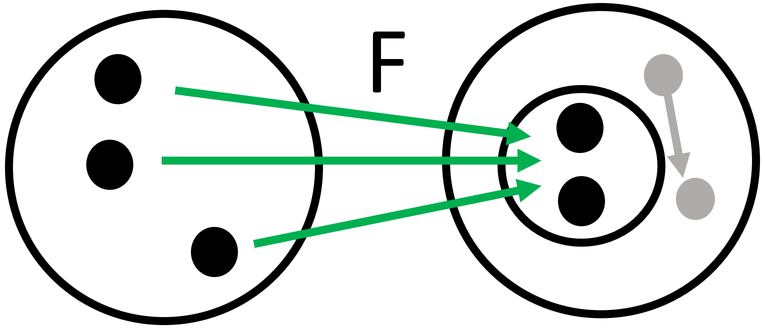
Functor maps a single object category to another point in another category



Functor maps a category to itself

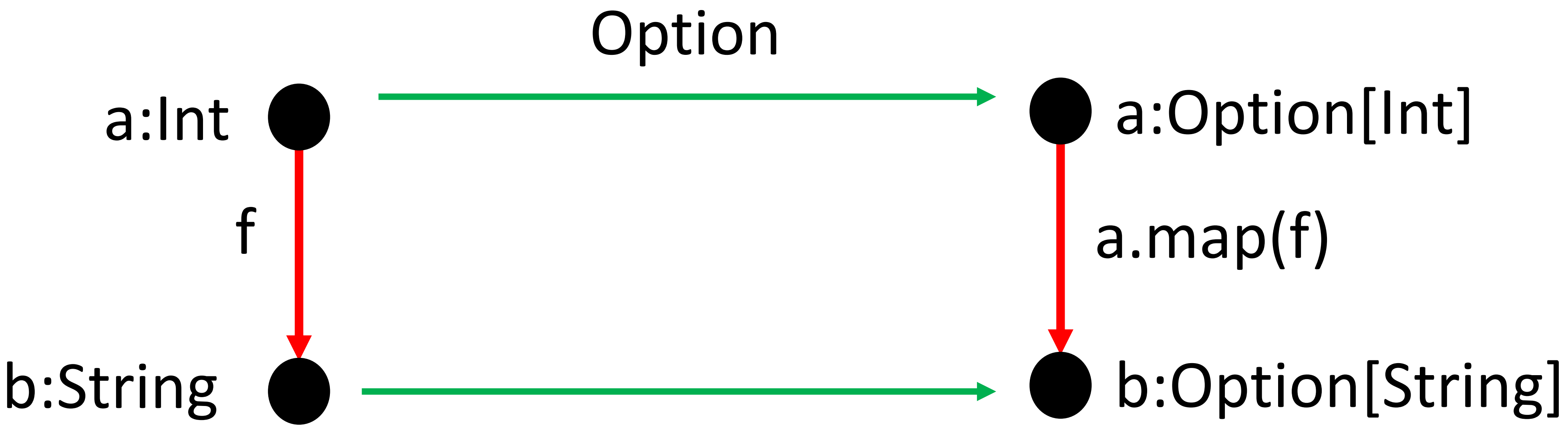


Functor maps whole category to an object in another category



Functor maps whole category to an subset of another category

Functor in Scala



Functor in Cats

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

// Example implementation for Option
implicit val functorForOption: Functor[Option] = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa match {
    case None    => None
    case Some(a) => Some(f(a))
  }
}
```

Functor law

A `Functor` instance must obey two laws:

- Composition: Mapping with `f` and then again with `g` is the same as mapping once with the composition of `f` and `g`
  - `fa.map(f).map(g) = fa.map(f.andThen(g))`
- Identity: Mapping with the identity function is a no-op
  - `fa.map(x => x) = fa`

Functor Law

- Functor of Option[_]
- Identity
  - map Option(a) Id_a = Id_a Option(a)
- Composition
  - map(Option(a))(g ∘ f) = (map g ∘ map f)(Option(a))

Functor Container

- Option[A], either None or Some(A)
  - You can just put a function into `map`
- List[A], either 0 or many A in a list
  - You can just put a function into `map`
- Future[A], either happened or pending
  - You can just put a function into `map`

Natural Transformation ?

Natural Transformation

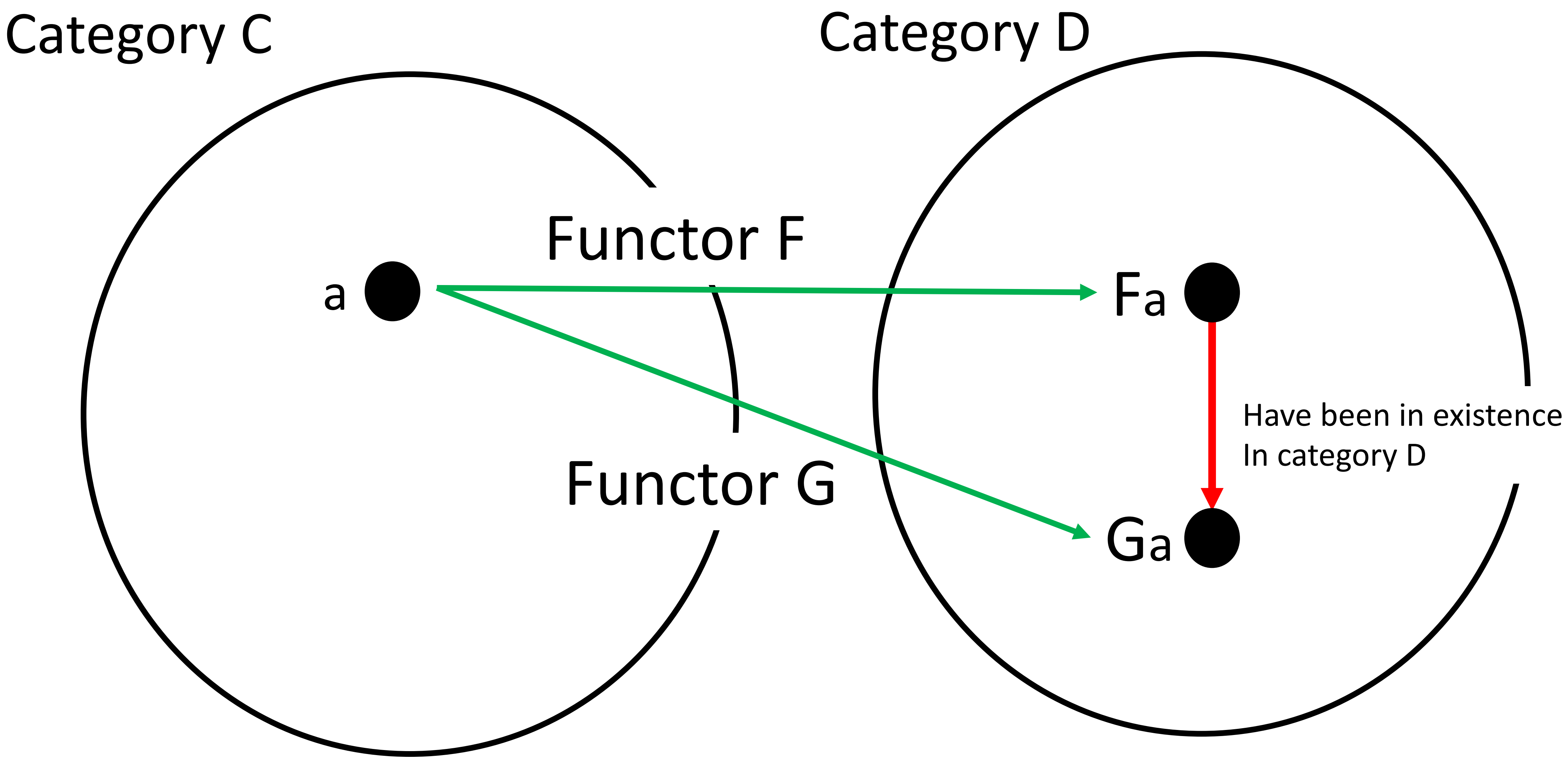
I think we now have enough ammunition on our hands to tackle naturality. Let's skip to the middle of the book, section 7.4.

> *A natural transformation is a morphism of functors. That is right: for fix categories* **C** *and* ***D****, we can regard the functors* **C** *=>* ***D*** *as the object of a new category, and the arrows between these objects are what we are going to call natural transformations.*

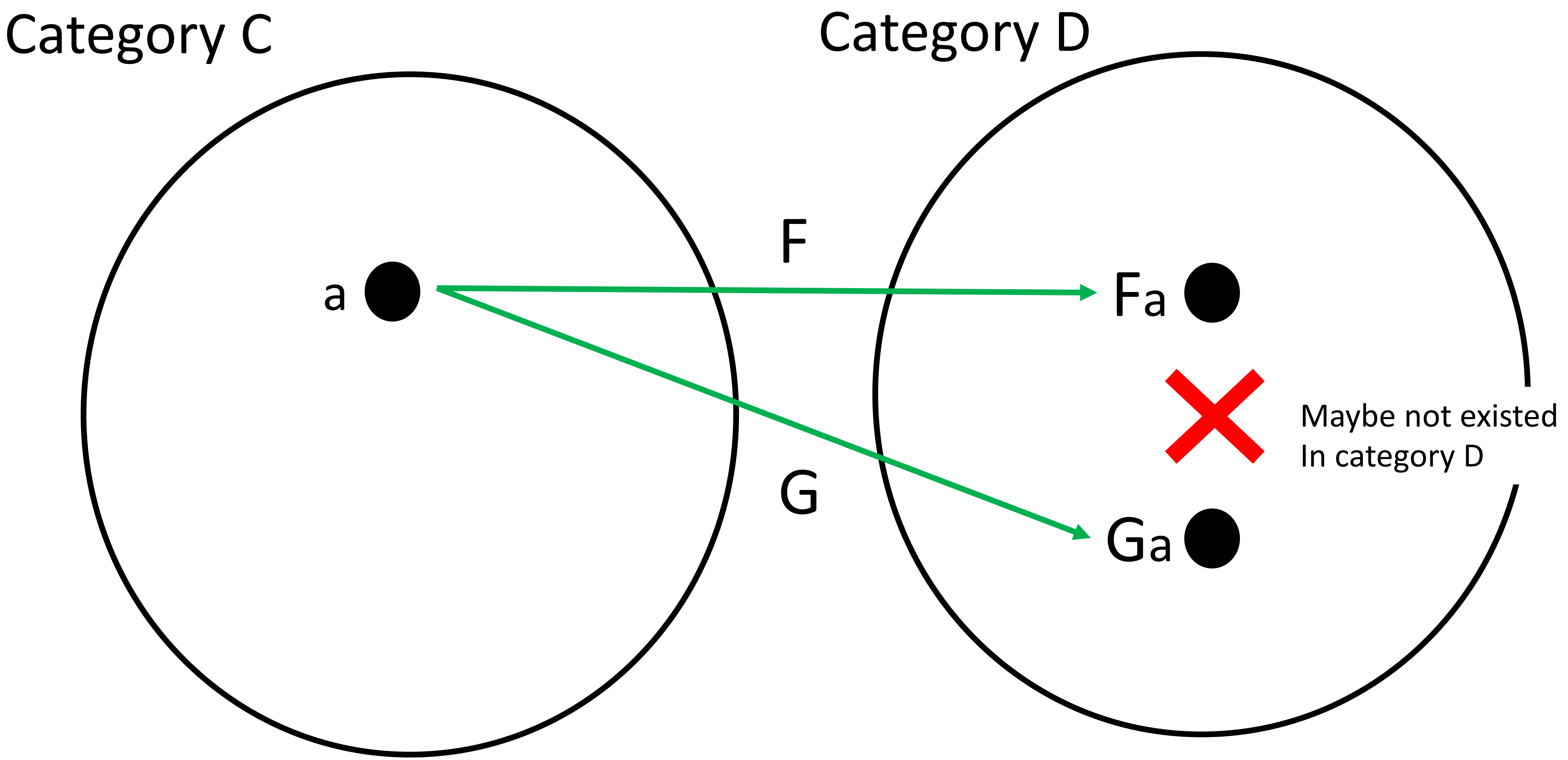
Natural Transformation

- Mapping between Functors
- Preserving structure

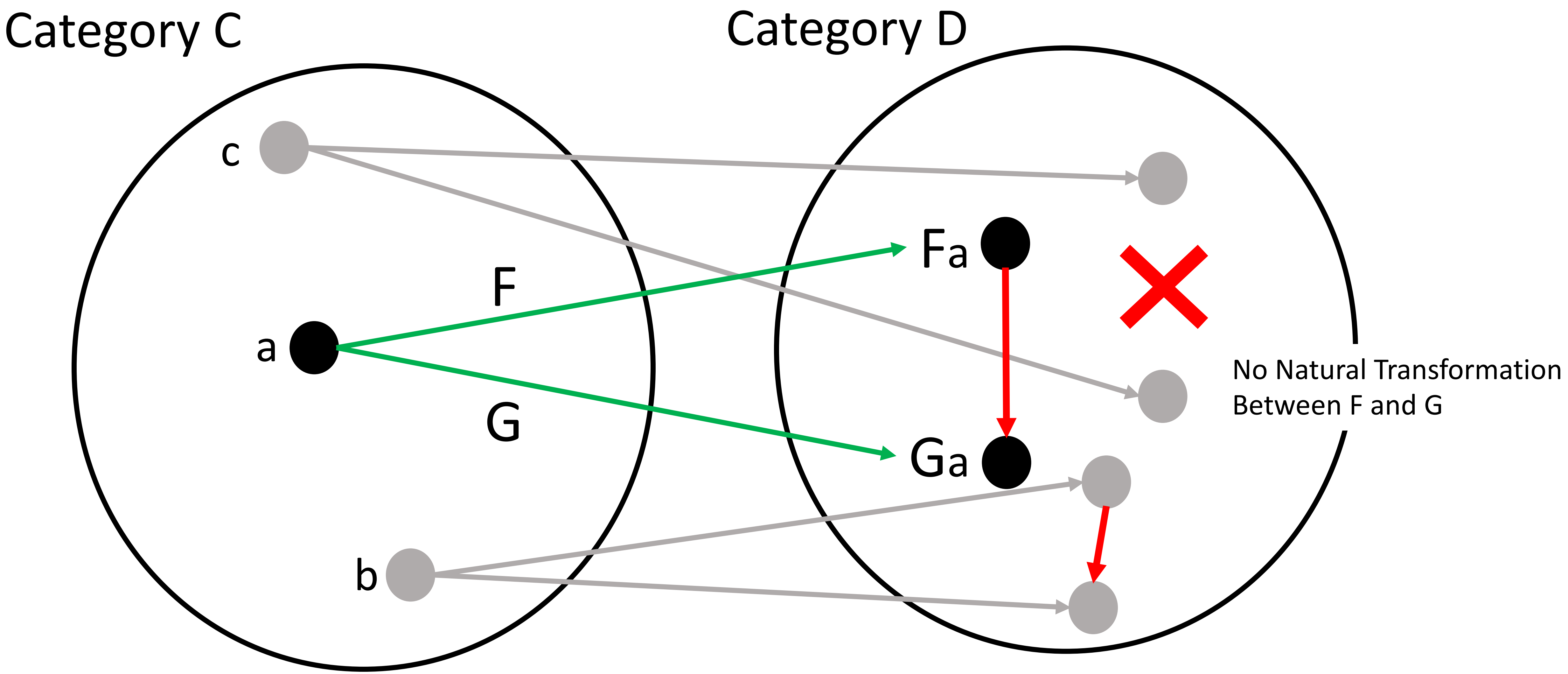
Natural Transformation



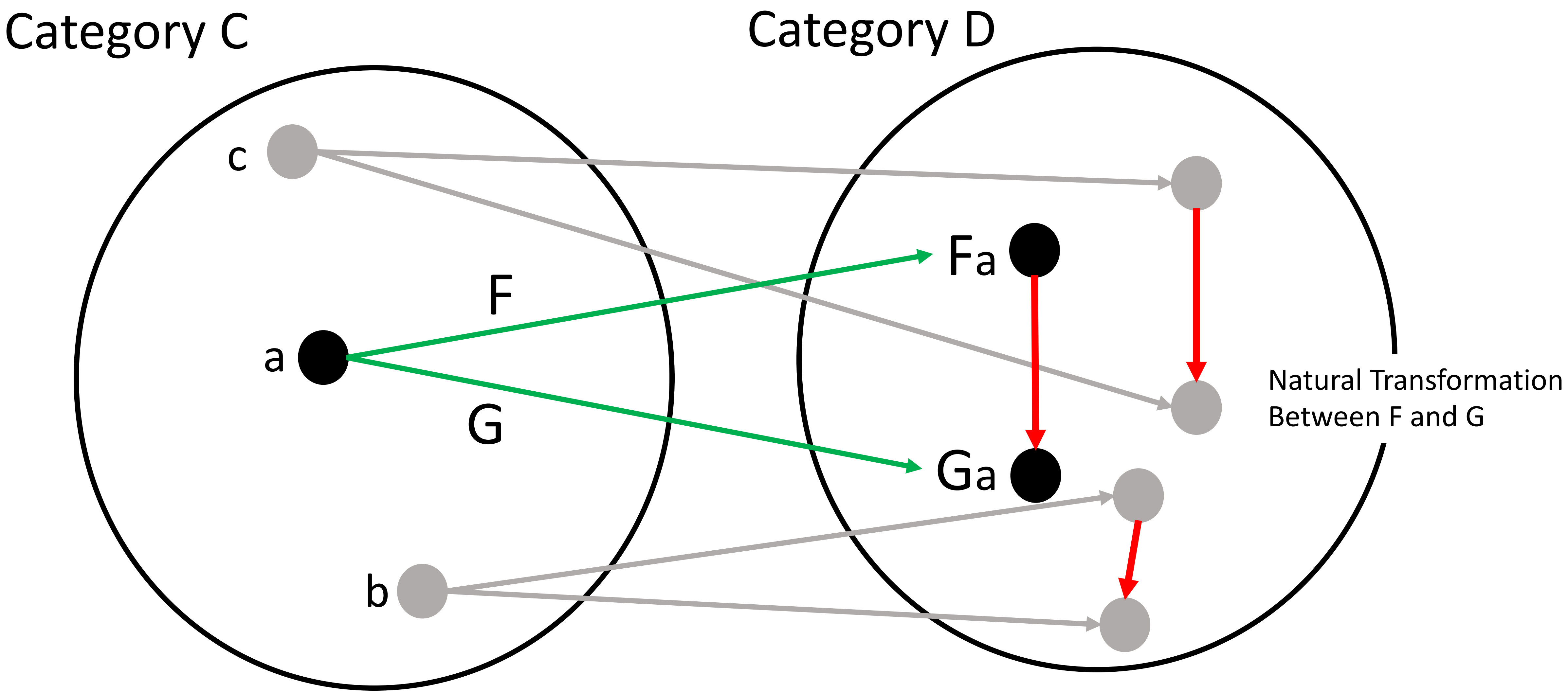
Natural Transformation



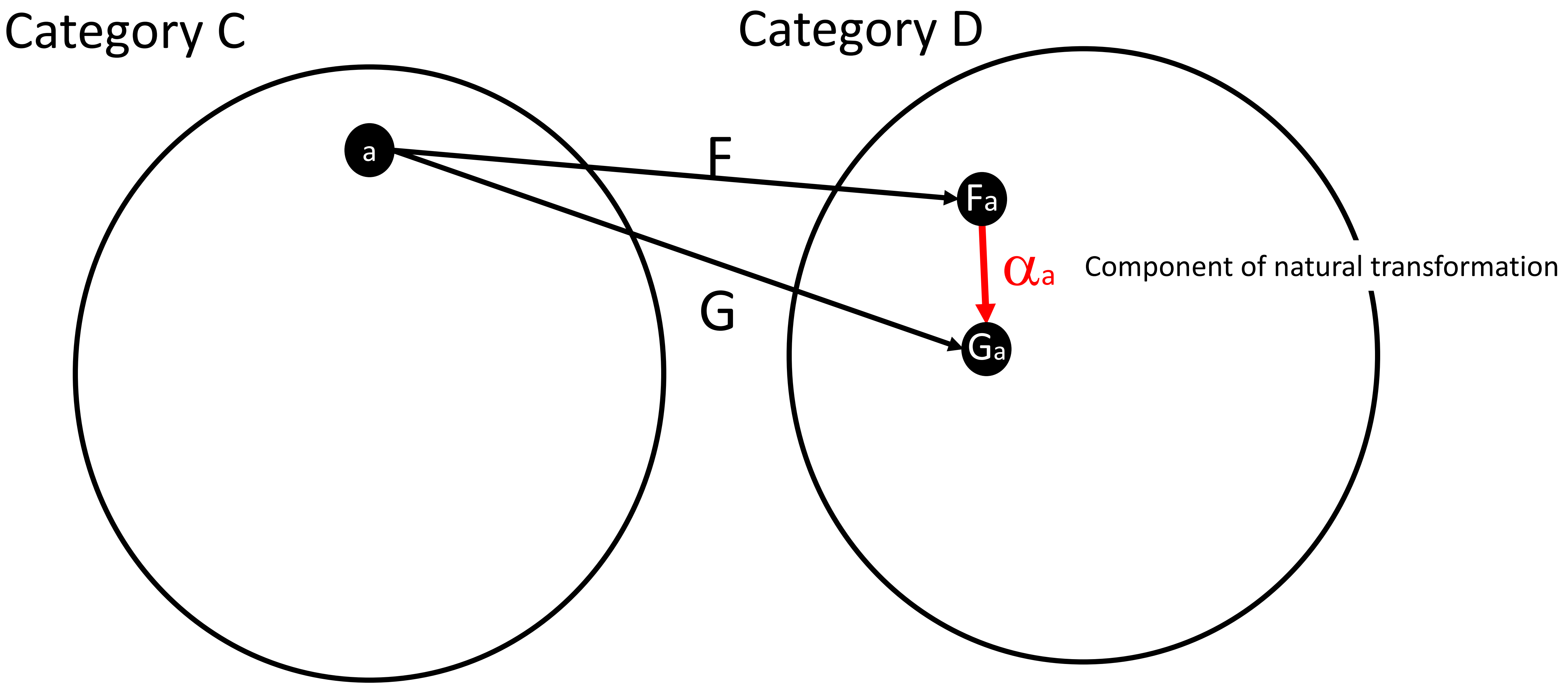
Natural Transformation



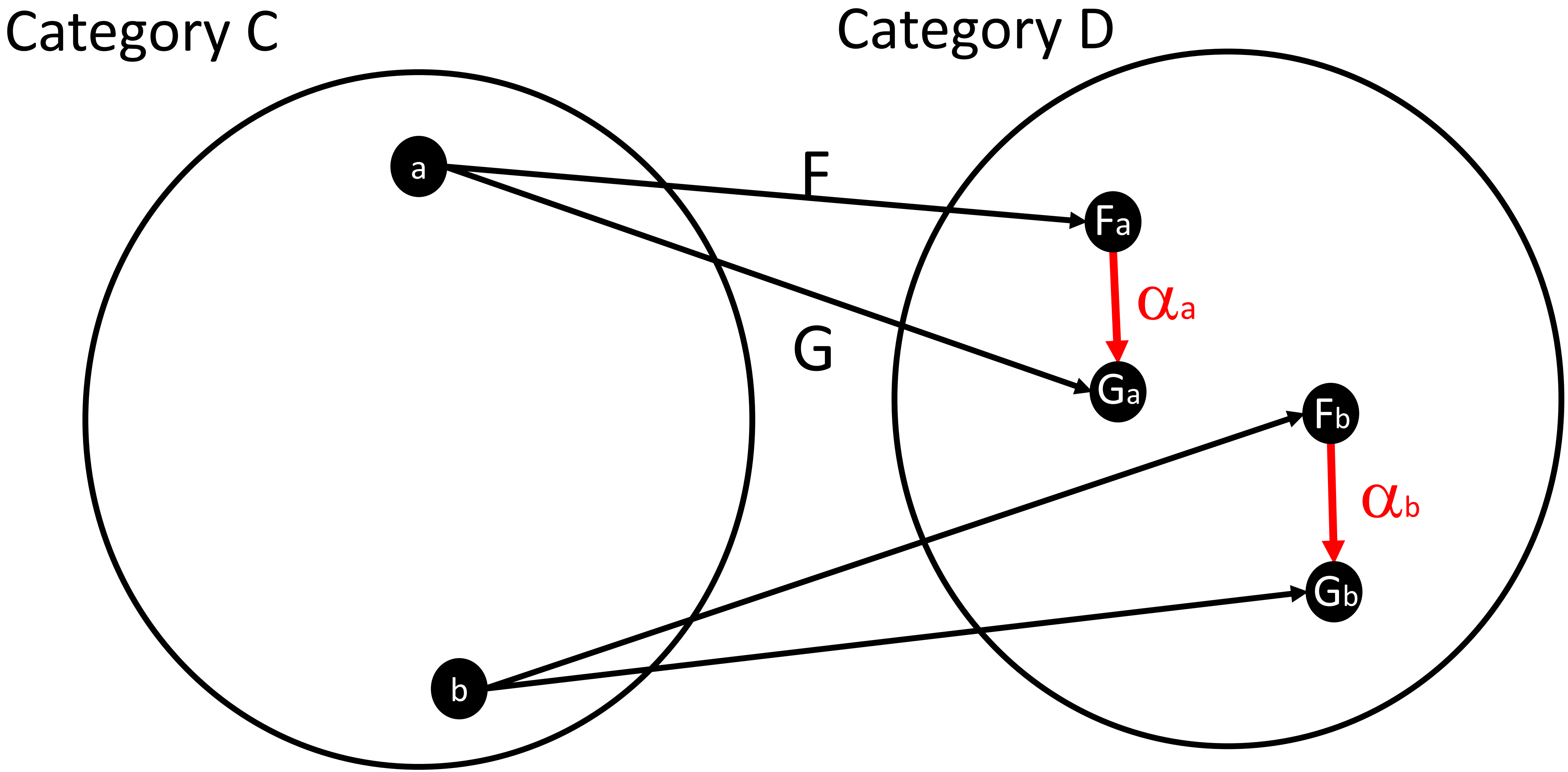
Natural Transformation



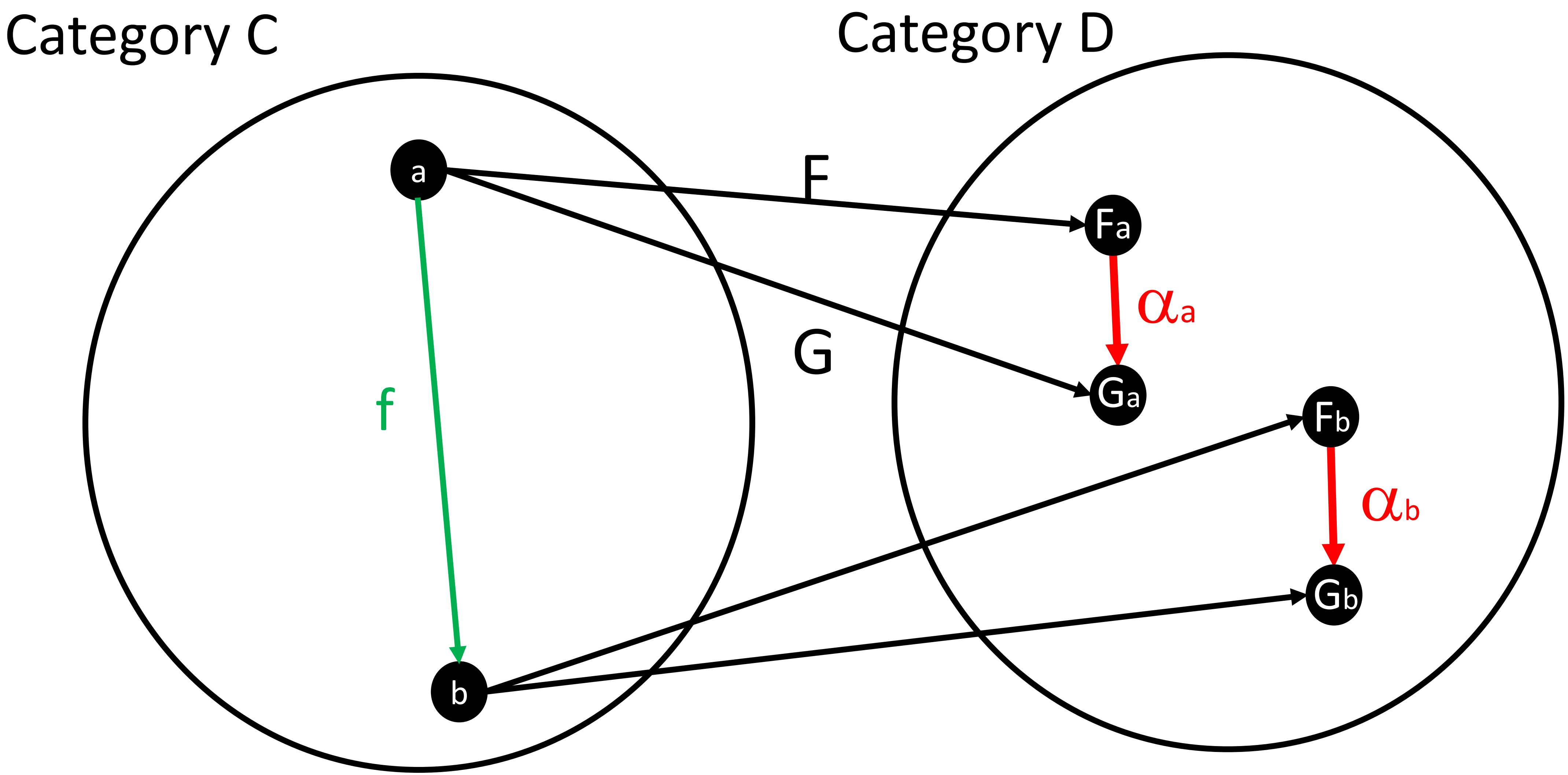
Natural Transformation – Preserving Structure



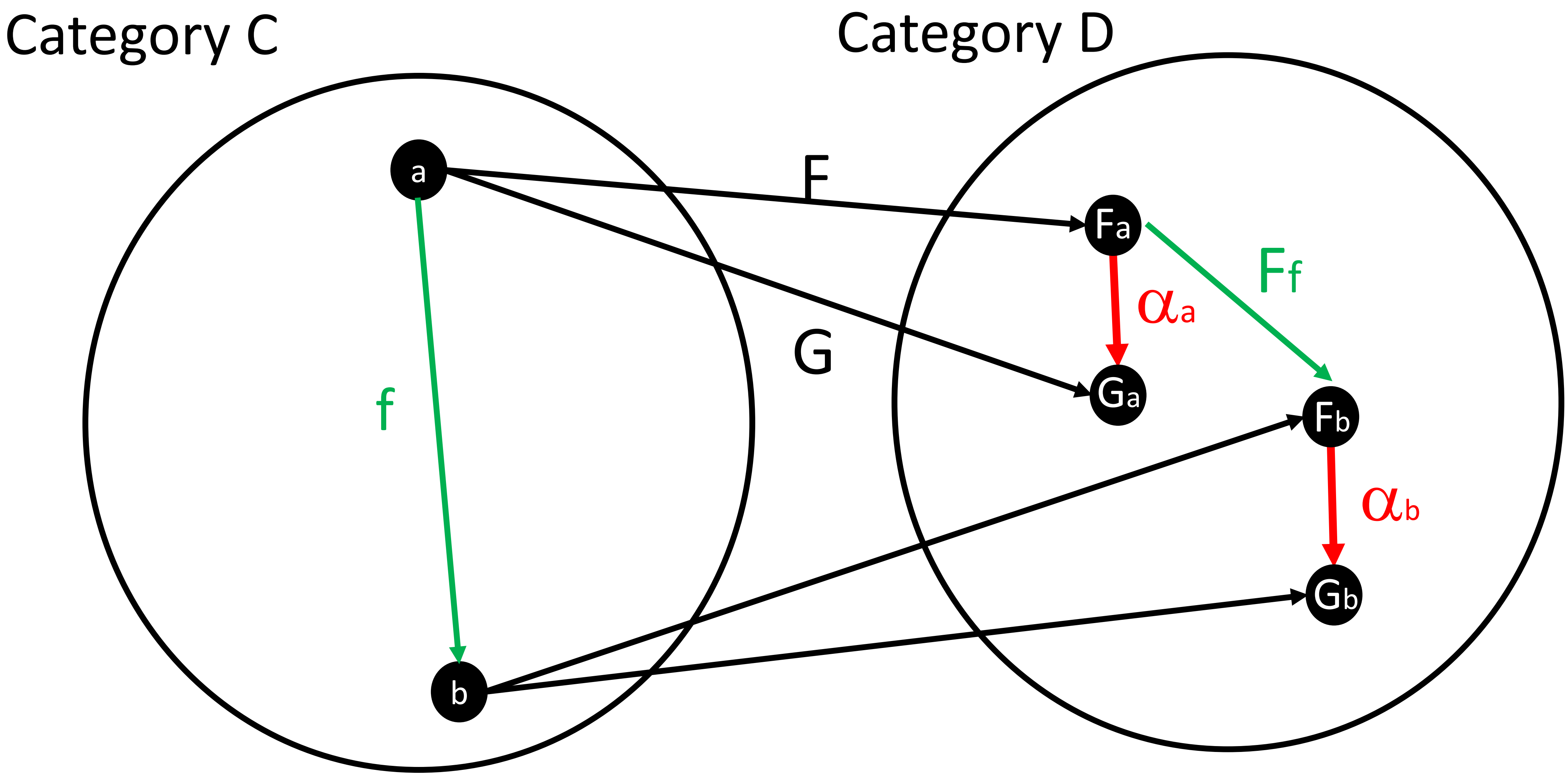
Natural Transformation – Preserving Structure



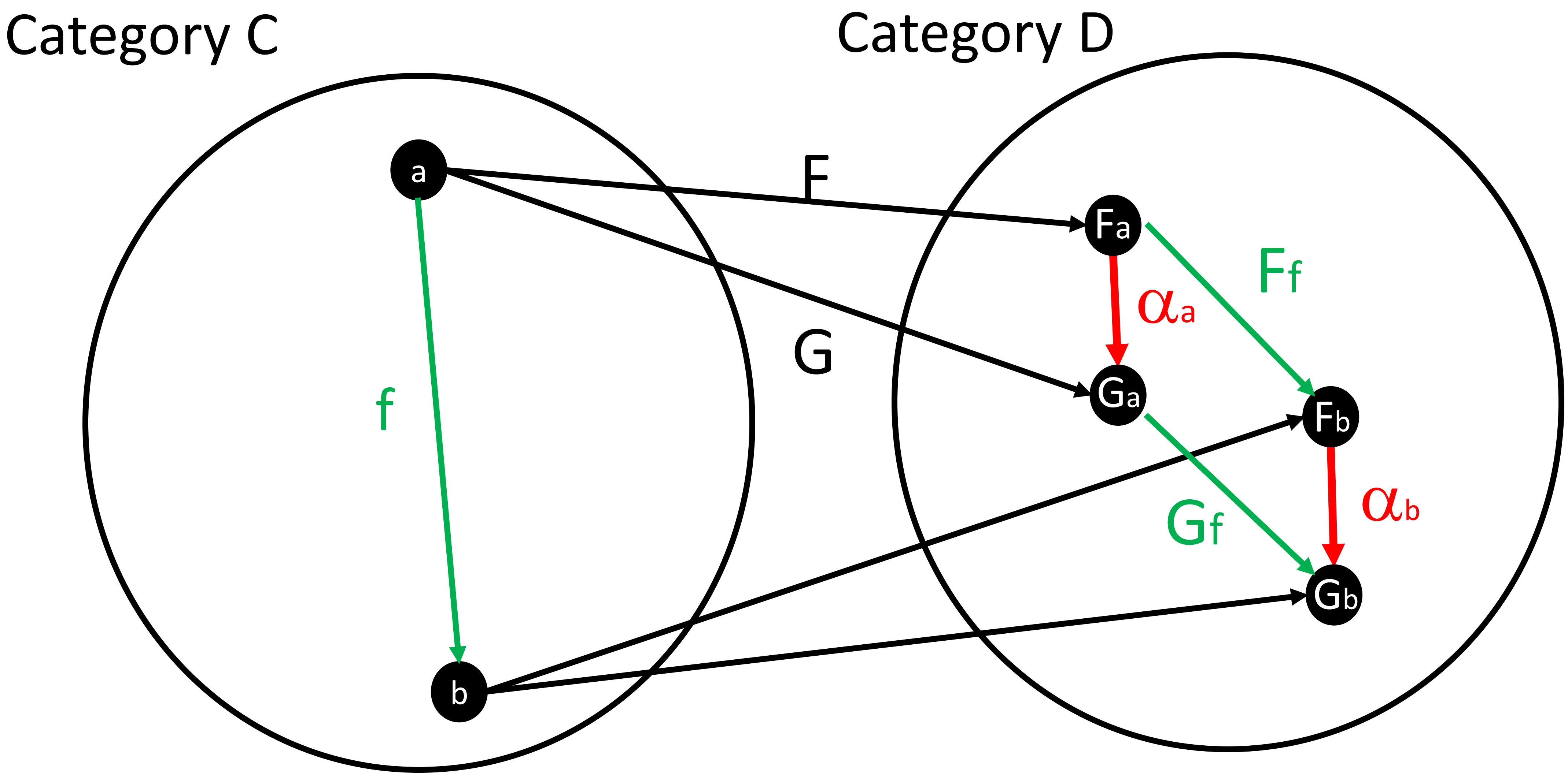
Natural Transformation – Preserving Structure



Natural Transformation – Preserving Structure



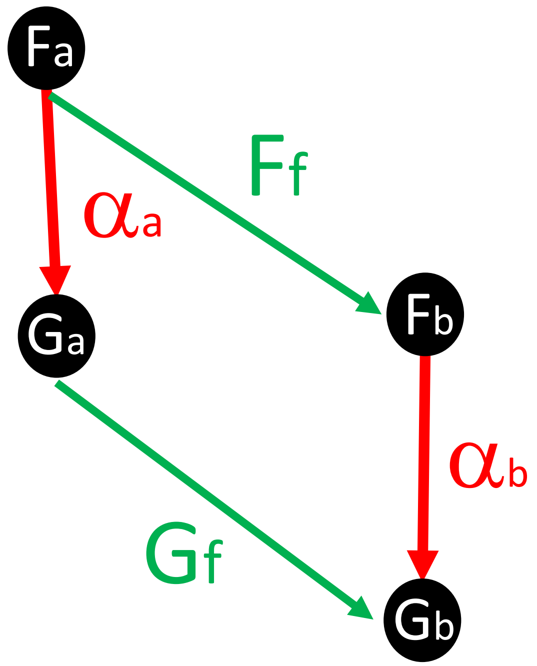
Natural Transformation – Preserving Structure



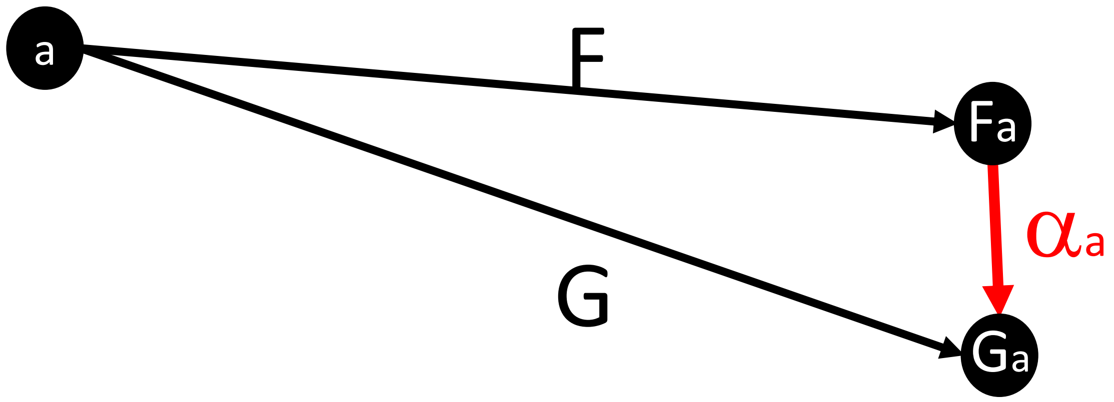
Natural Transformation – Preserving Structure

Relationship between F_f and G_f is Naturality Square

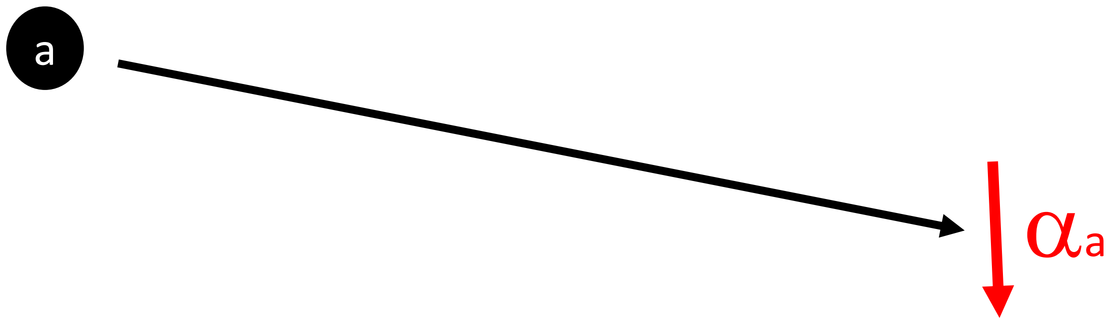
$$\alpha_b \circ F_f = G_f \circ \alpha_a$$



Natural Transformation

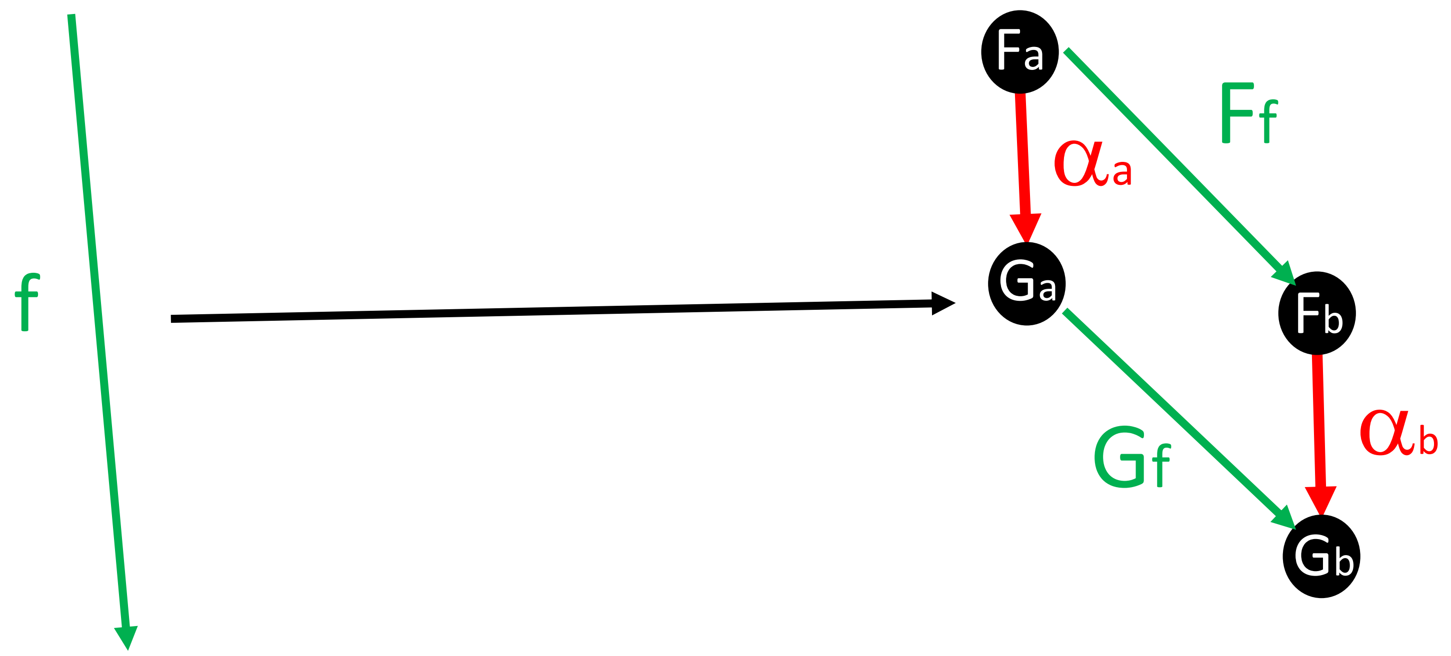


Natural Transformation



Mapping object a to morphism α_a

Natural Transformation

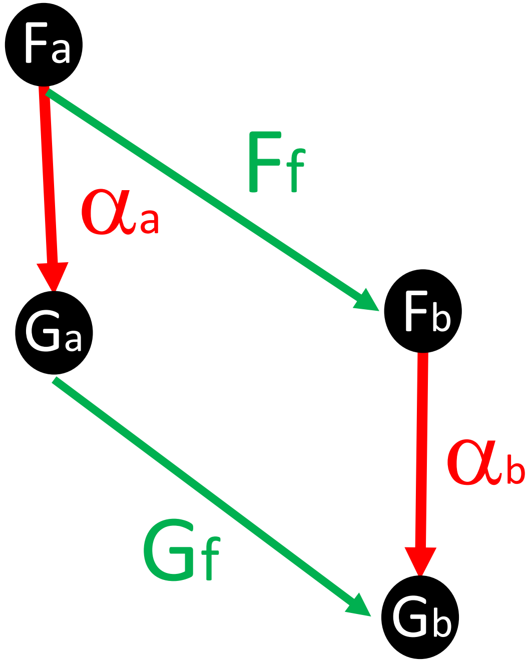


Mapping morphism f to a commuting diagram

Natural Transformation-Polymorphic function

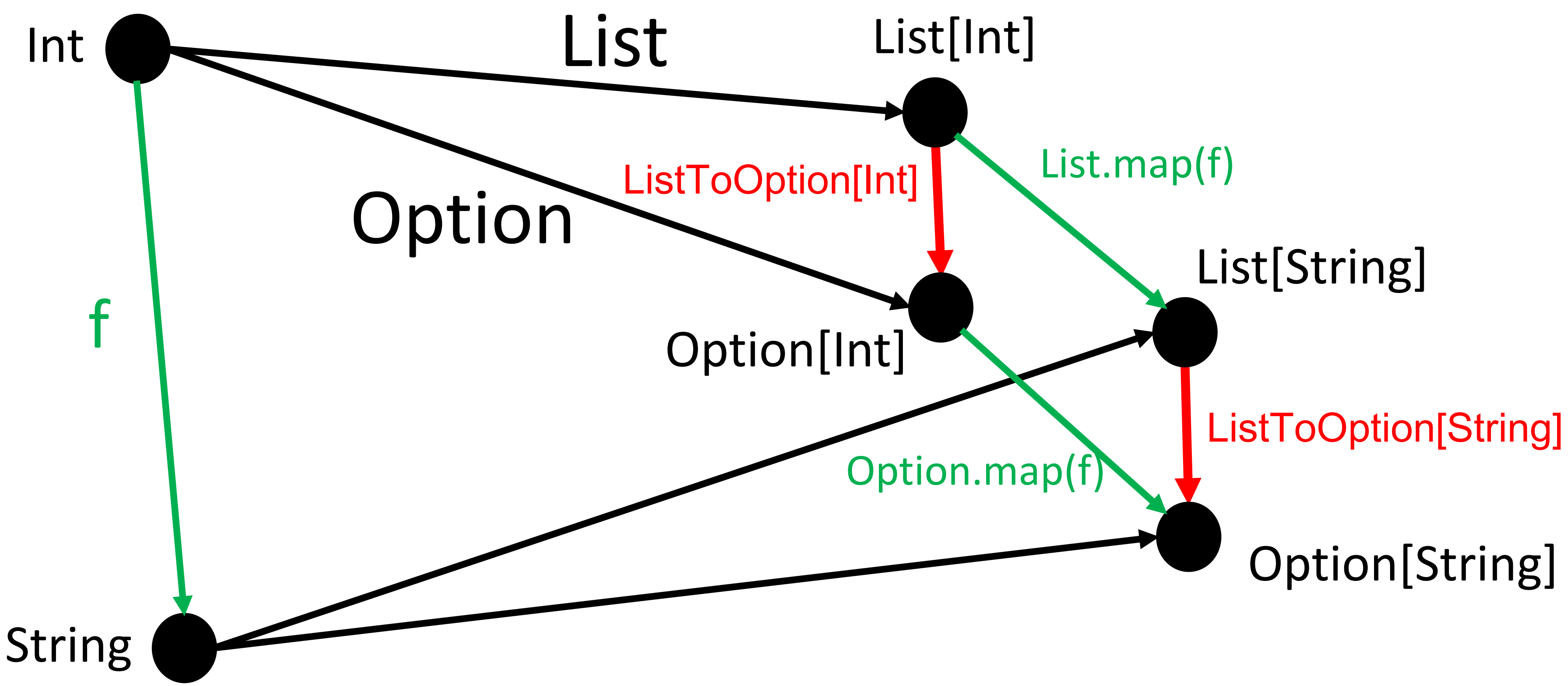
α : forall a, $F_a \Rightarrow G_a$

$\alpha_b \circ map_F\ f = map_G\ f \circ \alpha_a$



Natural Transformation-Parametric function

```
val ListToOption[A]: List[A] => Option[A] = list => list.headOption
```



Functor vs Natural Transformation

- Functor as a container
  - *map* modifies the content without reshaping the container
- Natural Transformation
  - To fulfill that mapping all objects from Functor F to Functor F
    - The content object is too polymorphic to modify it
  - Reshaping/transforming one container into another container without modifying content
  - Polymorphic/parametric functions

Reference

- Bartosz Milewski – Category theory for programmer
  - Category Theory 1.2: What is a category?
  - Category Theory 6.1: Functors
  - Category Theory 9.1: Natural transformations
- Category Theory for Programmers: The Preface

Thank you for your attention