$$\begin{tikzcd} A \arrow[r] \arrow[dr] & B \arrow[d] \\ & C \end{tikzcd}$$

Category Theory for beginners

Melbourne Scala User Group Feb 2015

@KenScambler

Abstract maths… for us?

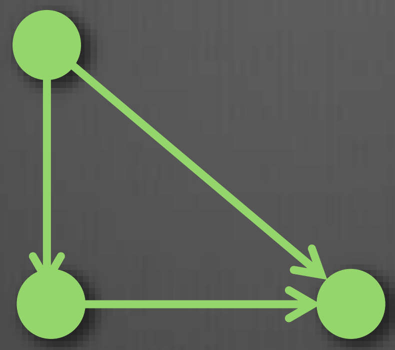
- Dizzyingly abstract branch of maths
- “Abstract nonsense”?
- Programming = maths
- Programming = abstraction
- Really useful to programming!

The plan

- Basic Category Theory concepts
- New vocabulary (helpful for further reading)
- How it relates to programming
- Category Theory as seen by maths versus FP

A bit of background

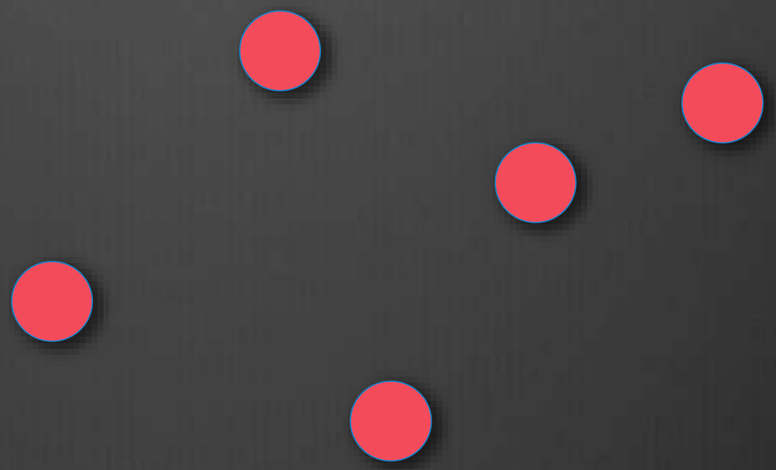
- 1940s Eilenberg, Mac Lane invent Category Theory
- 1958 Monads discovered by Godement
- In programming:
  - 1990 Moggi, Wadler apply monads to programming
  - 2006 “Applicative Programming with Effects” McBride & Paterson
  - 2006 “Essence of the Iterator Pattern” Gibbons & Oliveira



I. Categories

Category

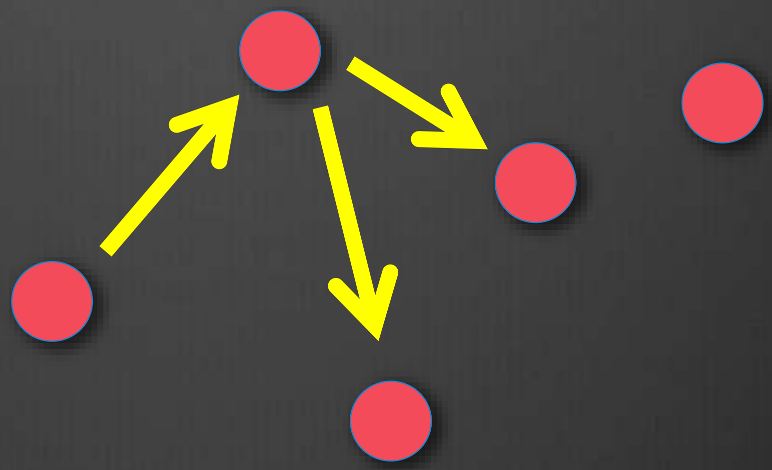
Objects



Category

Objects

Arrows or *morphisms*

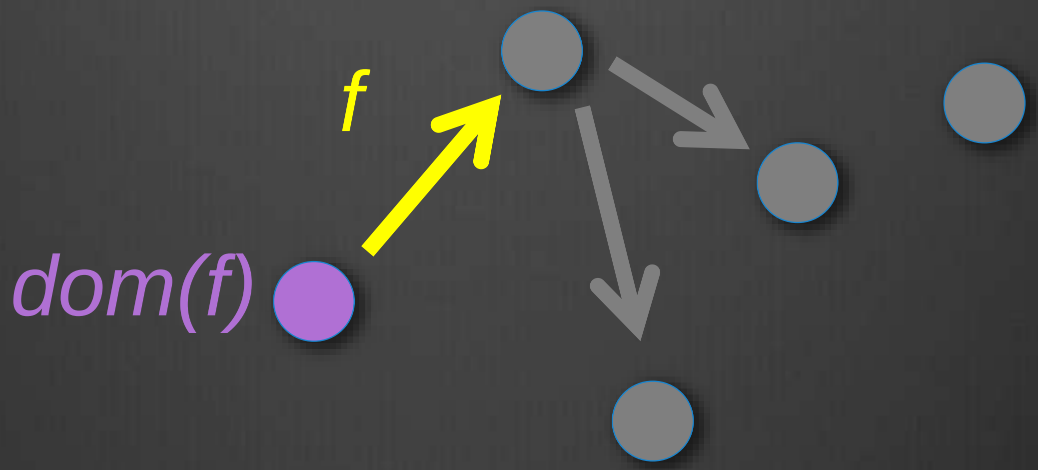


Category

Objects

Arrows

Domain

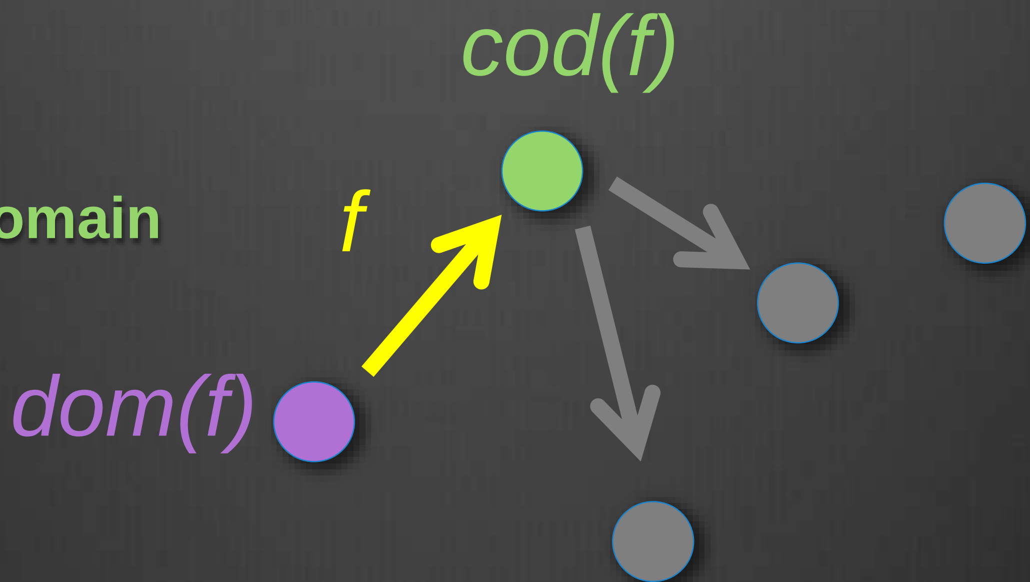


Category

Objects

Arrows

Domain/Codomain

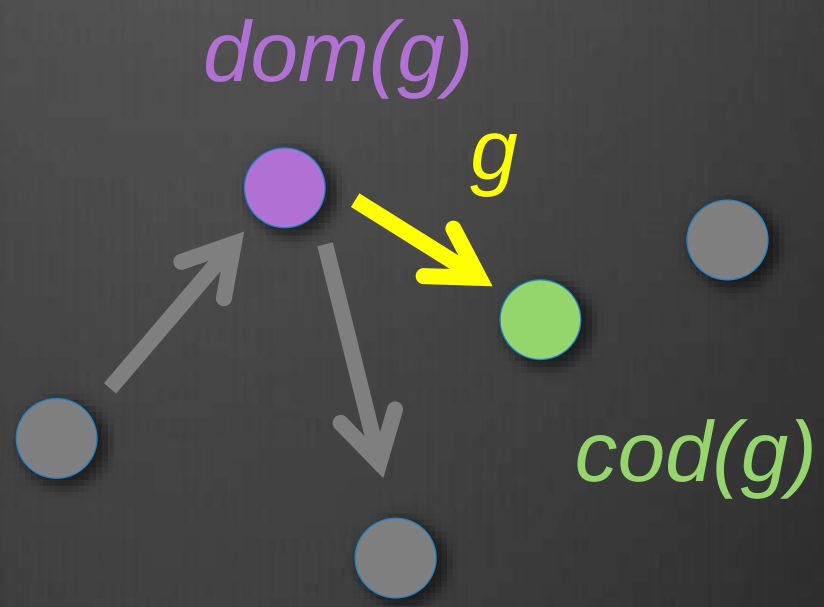


Category

Objects

Arrows

Domain/Codomain

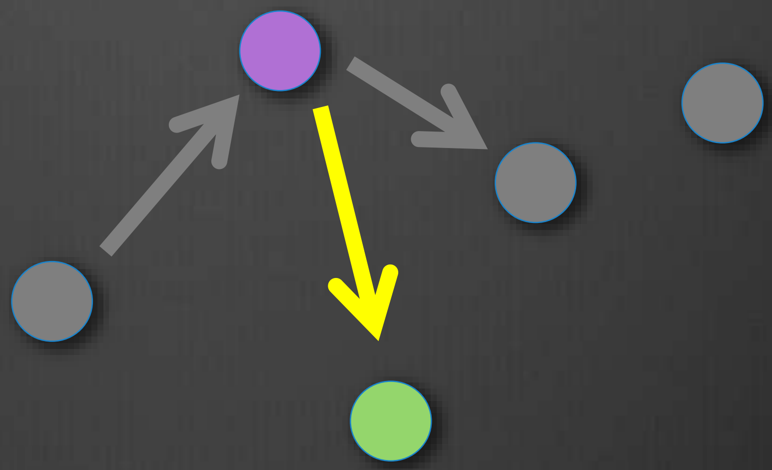


Category

Objects

Arrows

Domain/Codomain



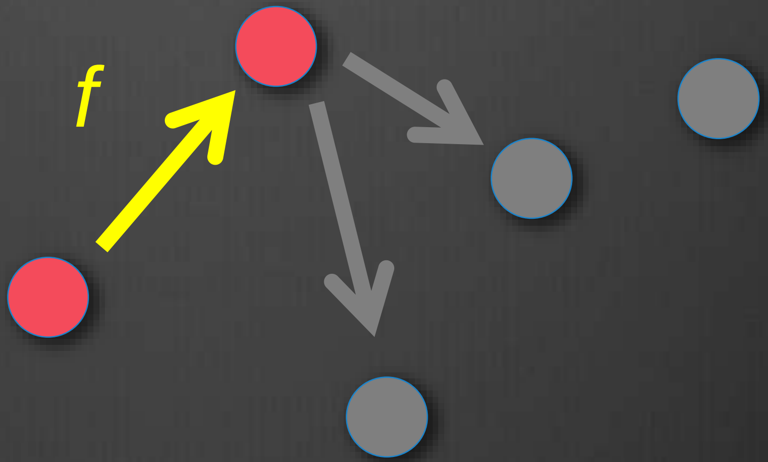
Category

Objects

Arrows

Domain/Codomain

Composition



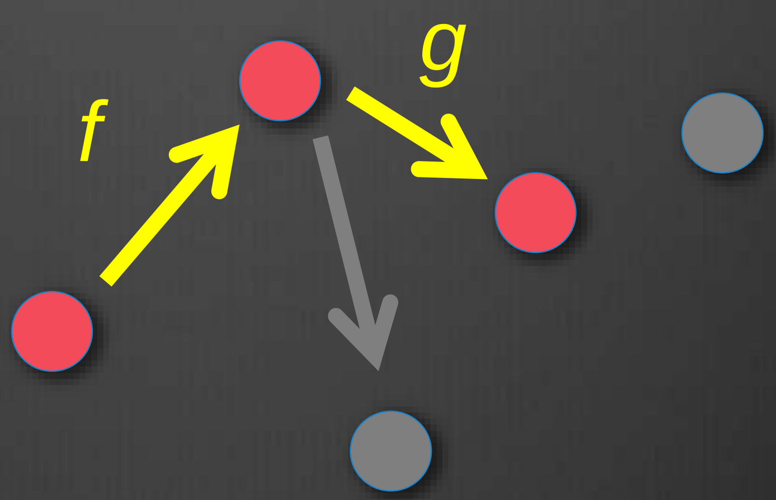
Category

Objects

Arrows

Domain/Codomain

Composition



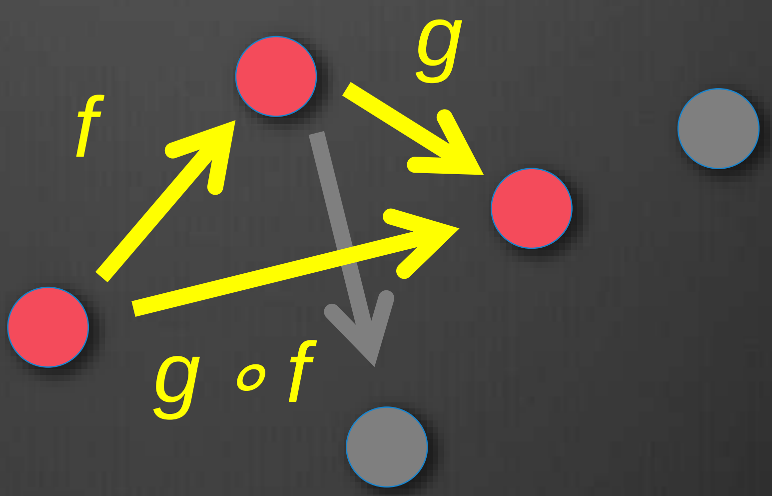
Category

Objects

Arrows

Domain/Codomain

Composition



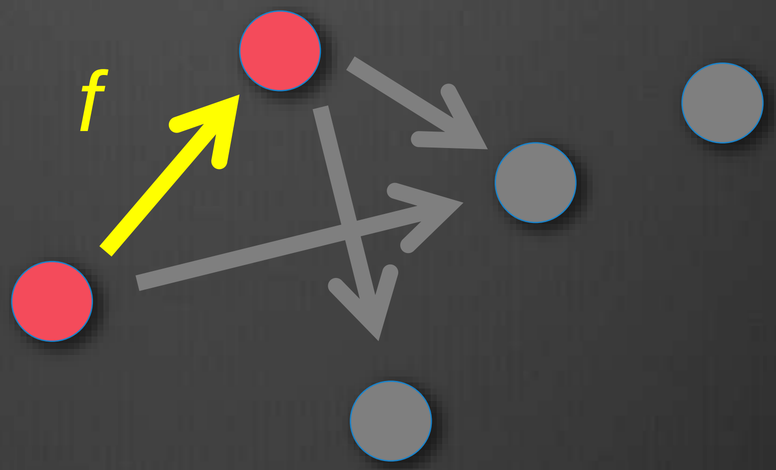
Category

Objects

Arrows

Domain/Codomain

Composition



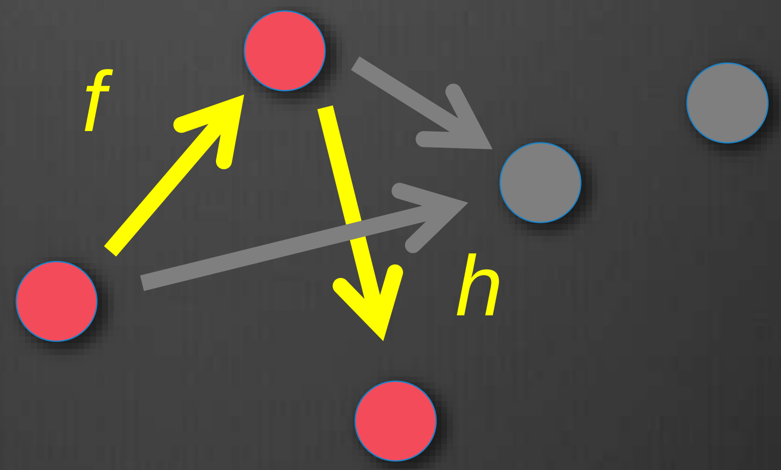
Category

Objects

Arrows

Domain/Codomain

Composition



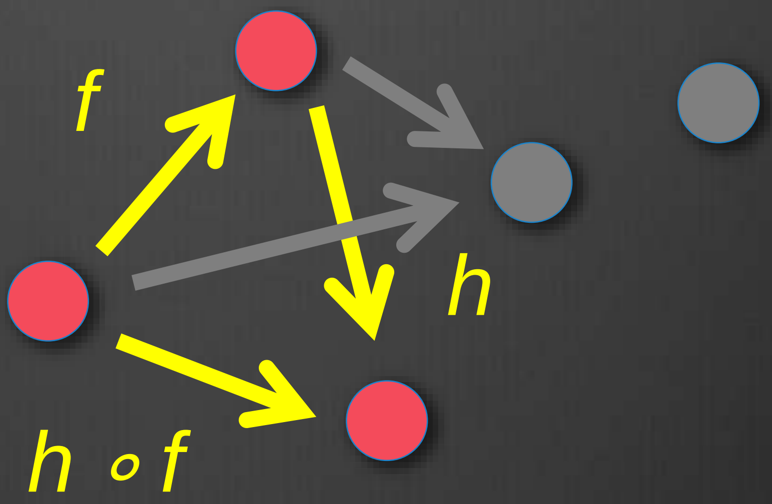
Category

Objects

Arrows

Domain/Codomain

Composition



Category

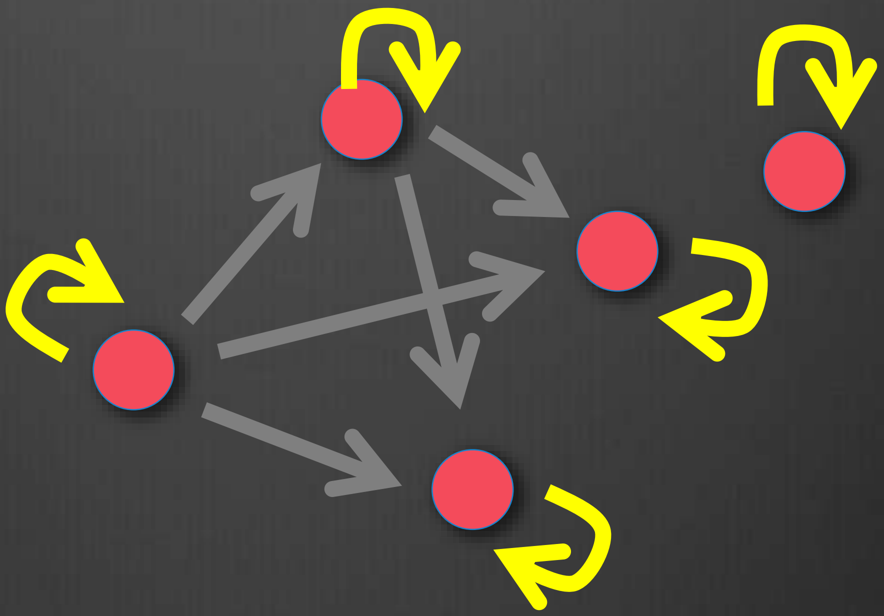
Objects

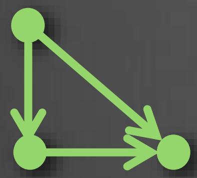
Arrows

Domain/Codomain

Composition

Identity





Category

Compose

$$\circ : (B \to C) \to (A \to B) \to (A \to C)$$

Identity

$$id : A \to A$$

Category Laws

Associative Law

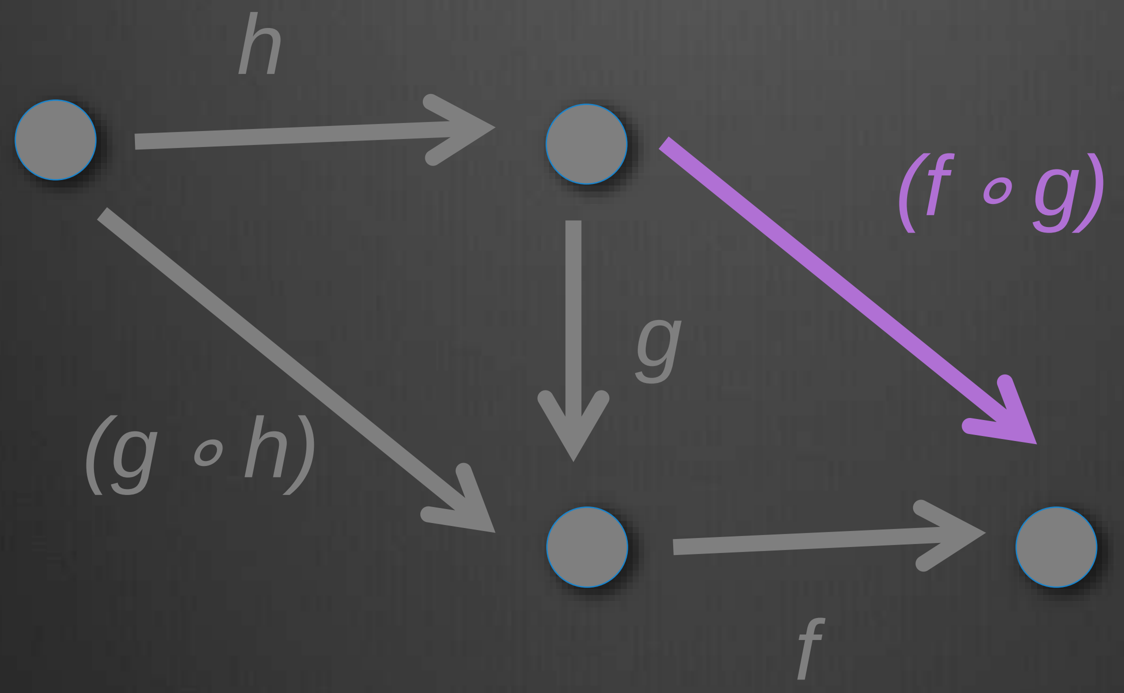
$$(f \circ g) \circ h = f \circ (g \circ h)$$

Identity Laws

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

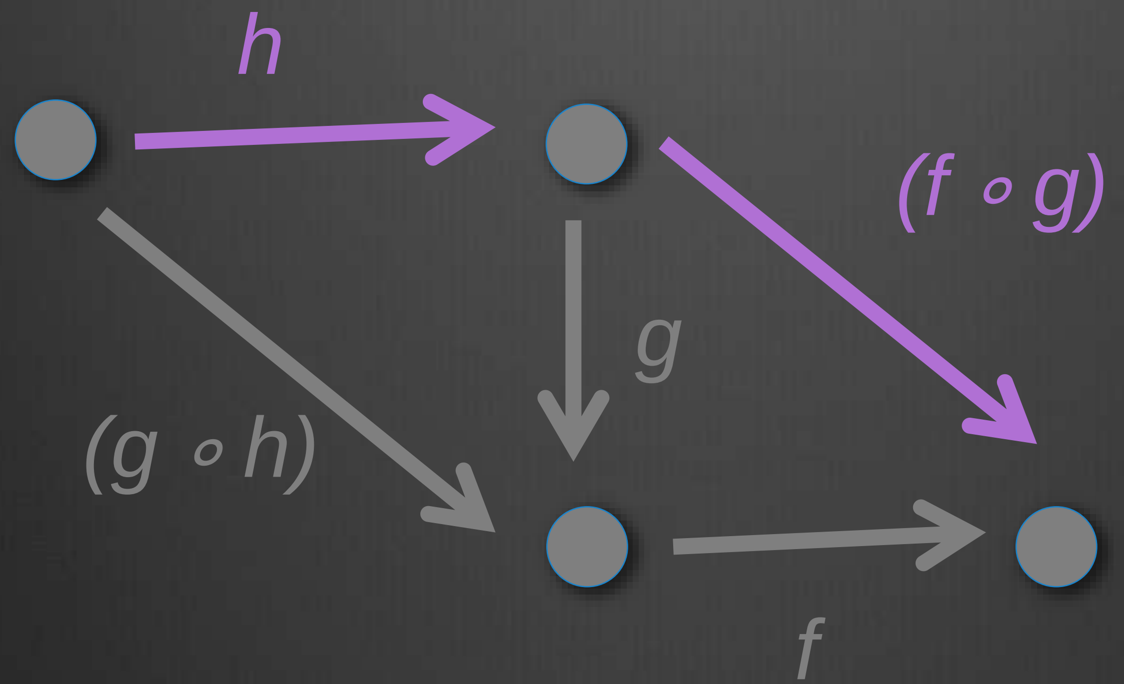
Associative law

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



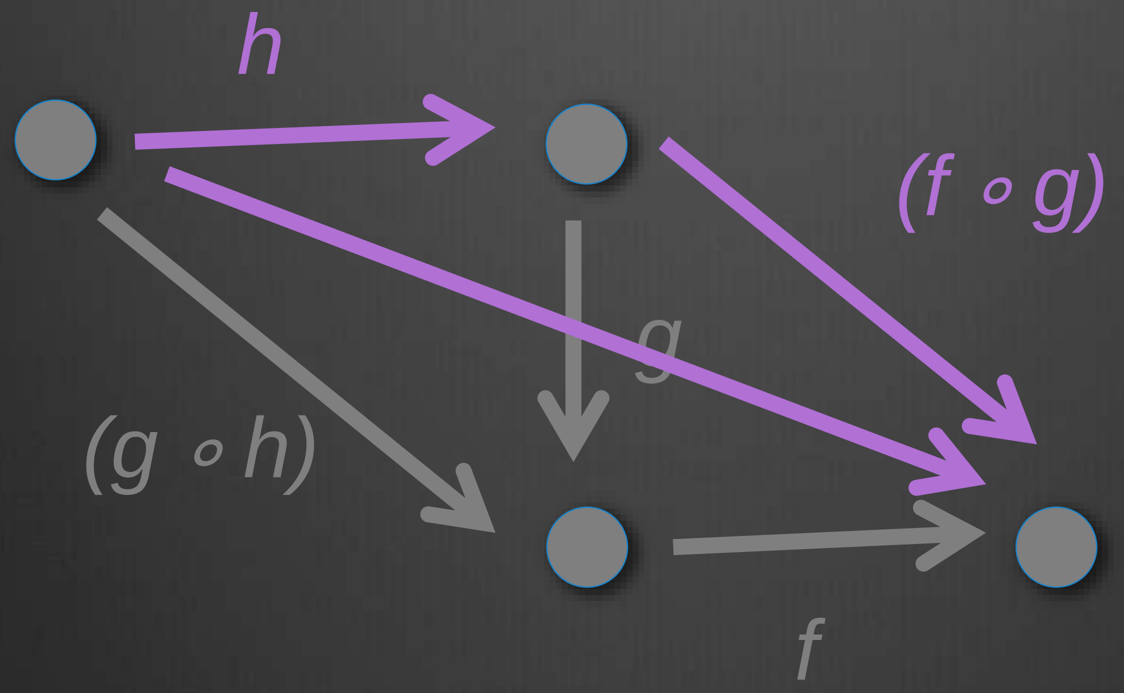
Associative law

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



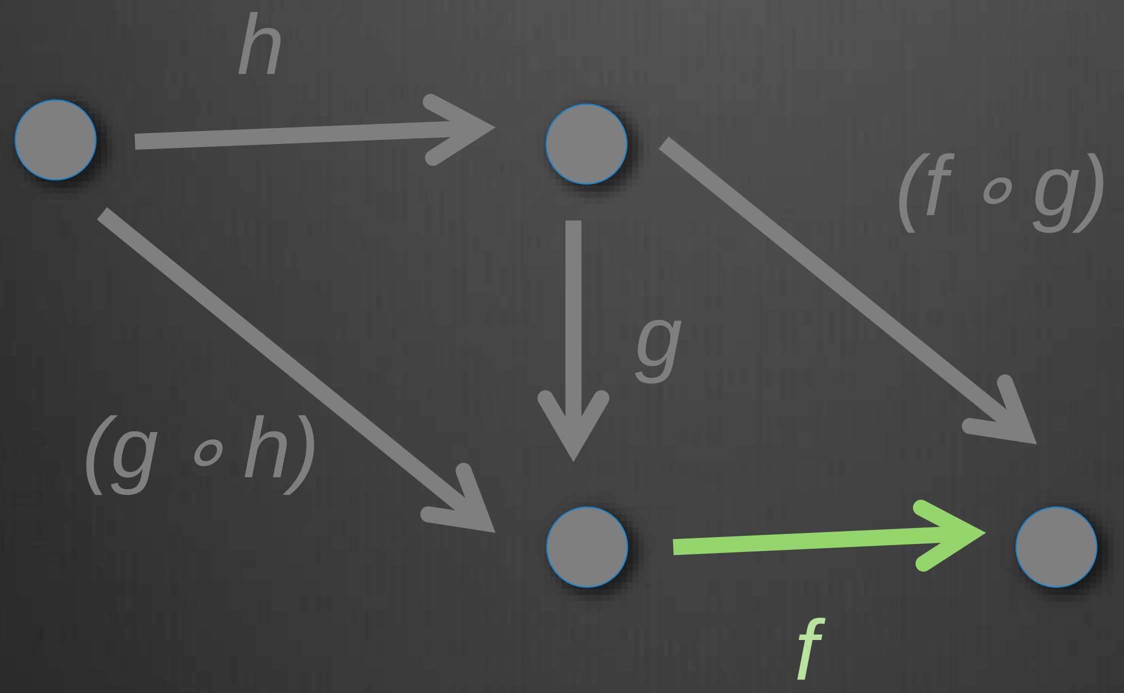
Associative law

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



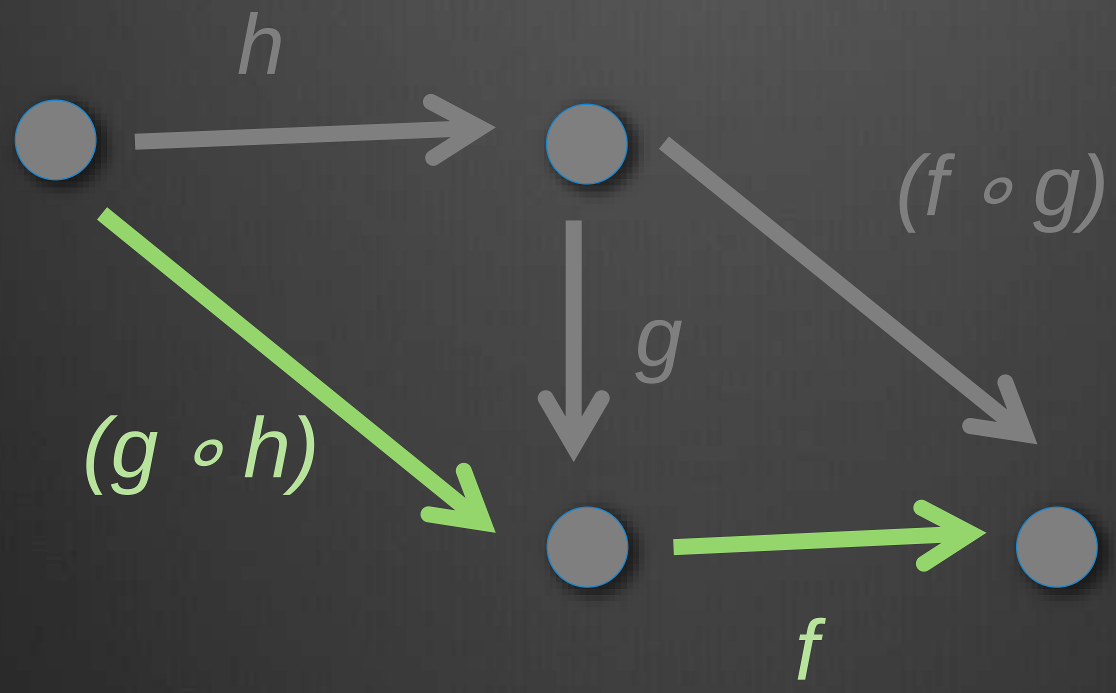
Associative law

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



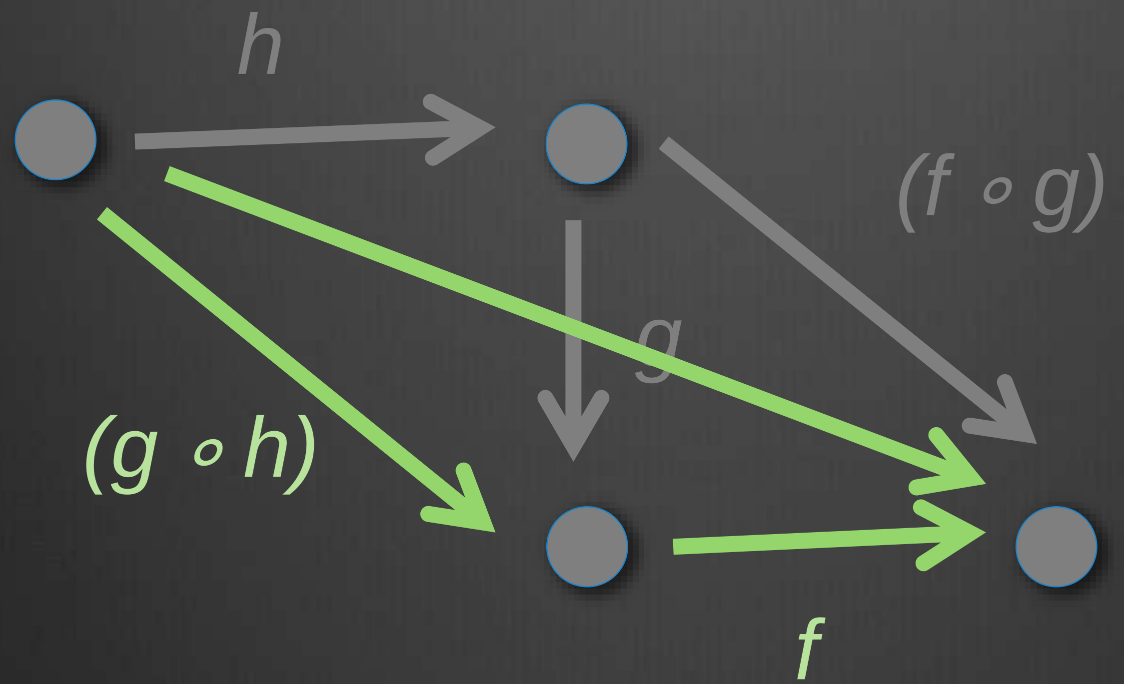
Associative law

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



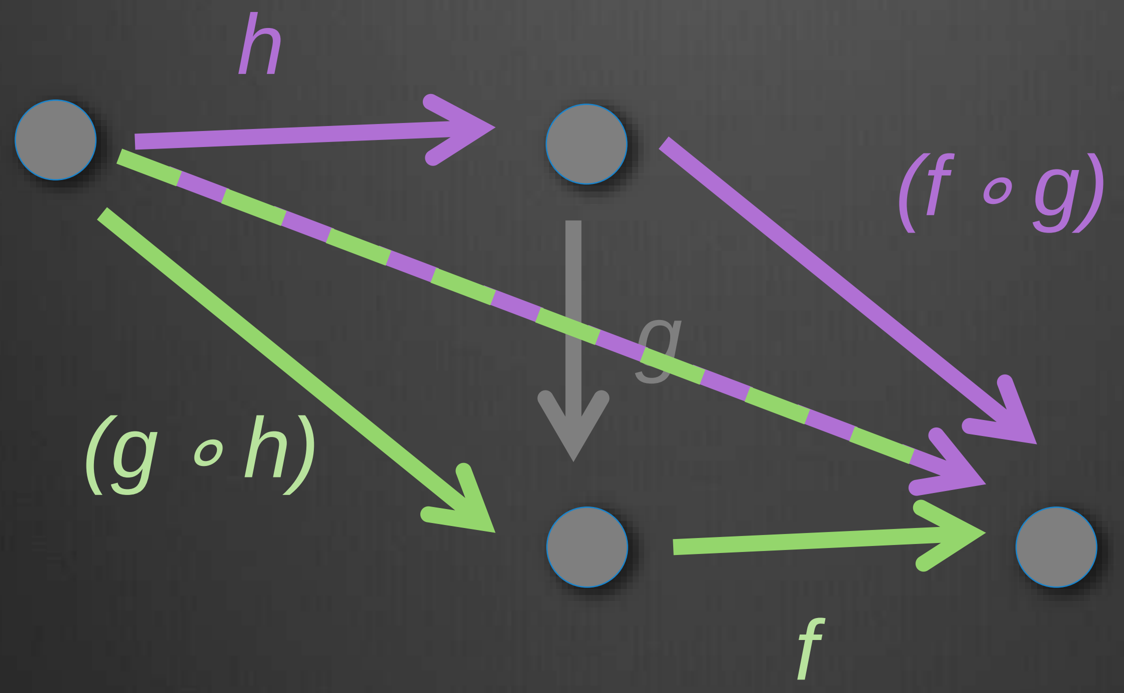
Associative law

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



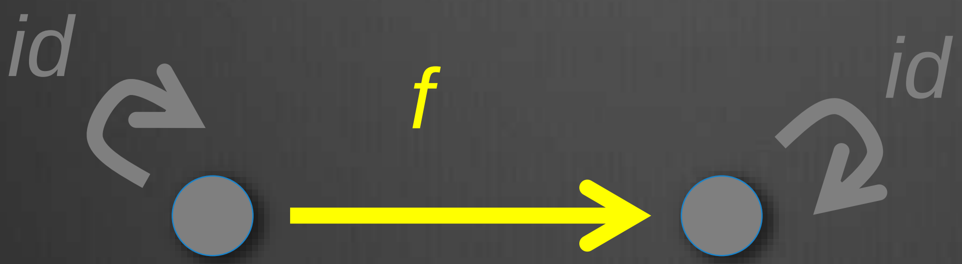
Associative law

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



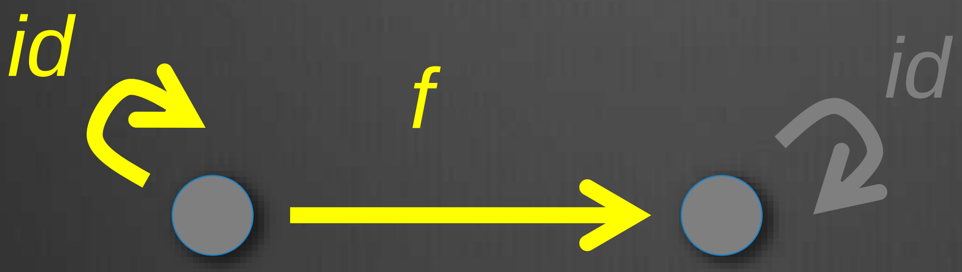
Identity laws

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



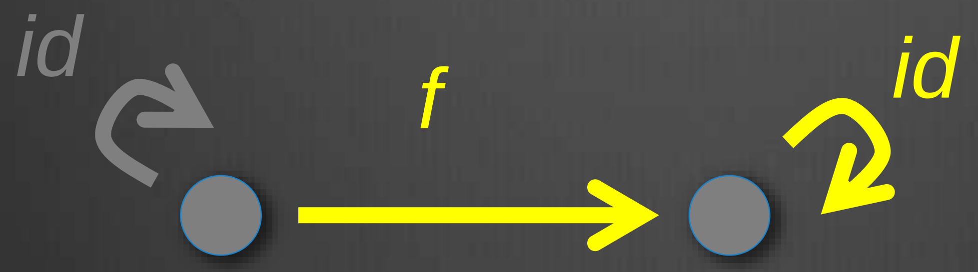
Identity laws

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



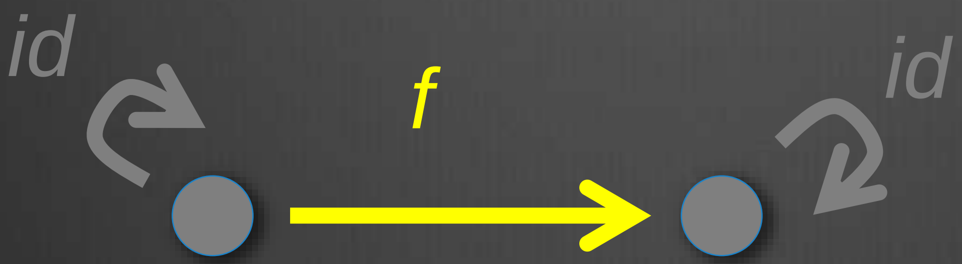
Identity laws

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



Identity laws

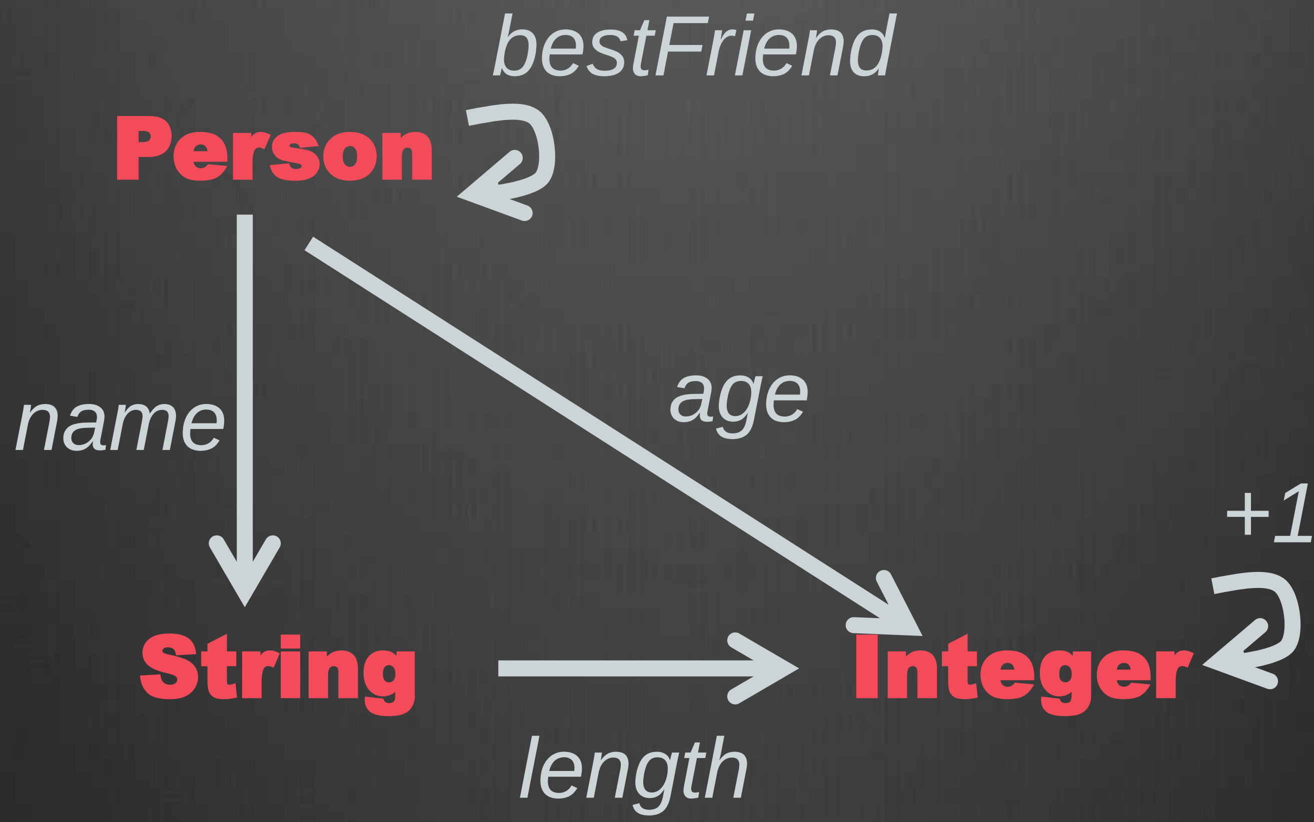
$$f \circ id \;=\; id \circ f \;=\; f$$



Examples

- Infinite categories
- Finite categories
- Objects can represent anything
- Arrows can represent anything
- As long as we have *composition* and *identity*!

Sets & functions



Sets & functions

- Infinite arrows from composition
- $+1 \circ$ length $\circ$ name
- bestFriend $\circ$ bestFriend
- bestFriend $\circ$ bestFriend $\circ$ bestFriend
- $+1 \circ$ age $\circ$ bestFriend

Sets & functions

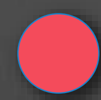
- **Objects**
- **Arrows**
- **Composition**
- **Identity**

Sets & functions

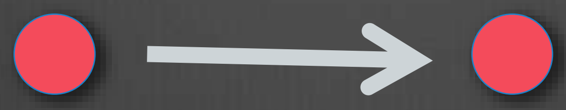
- **Objects** = sets (or types)
- **Arrows** = functions
- **Composition** = function composition
- **Identity** = identity function

Zero

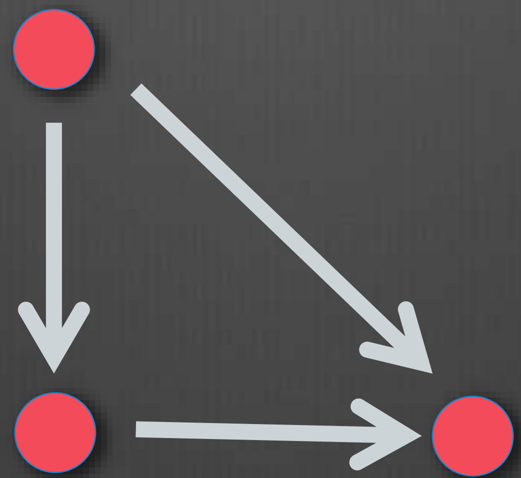
One



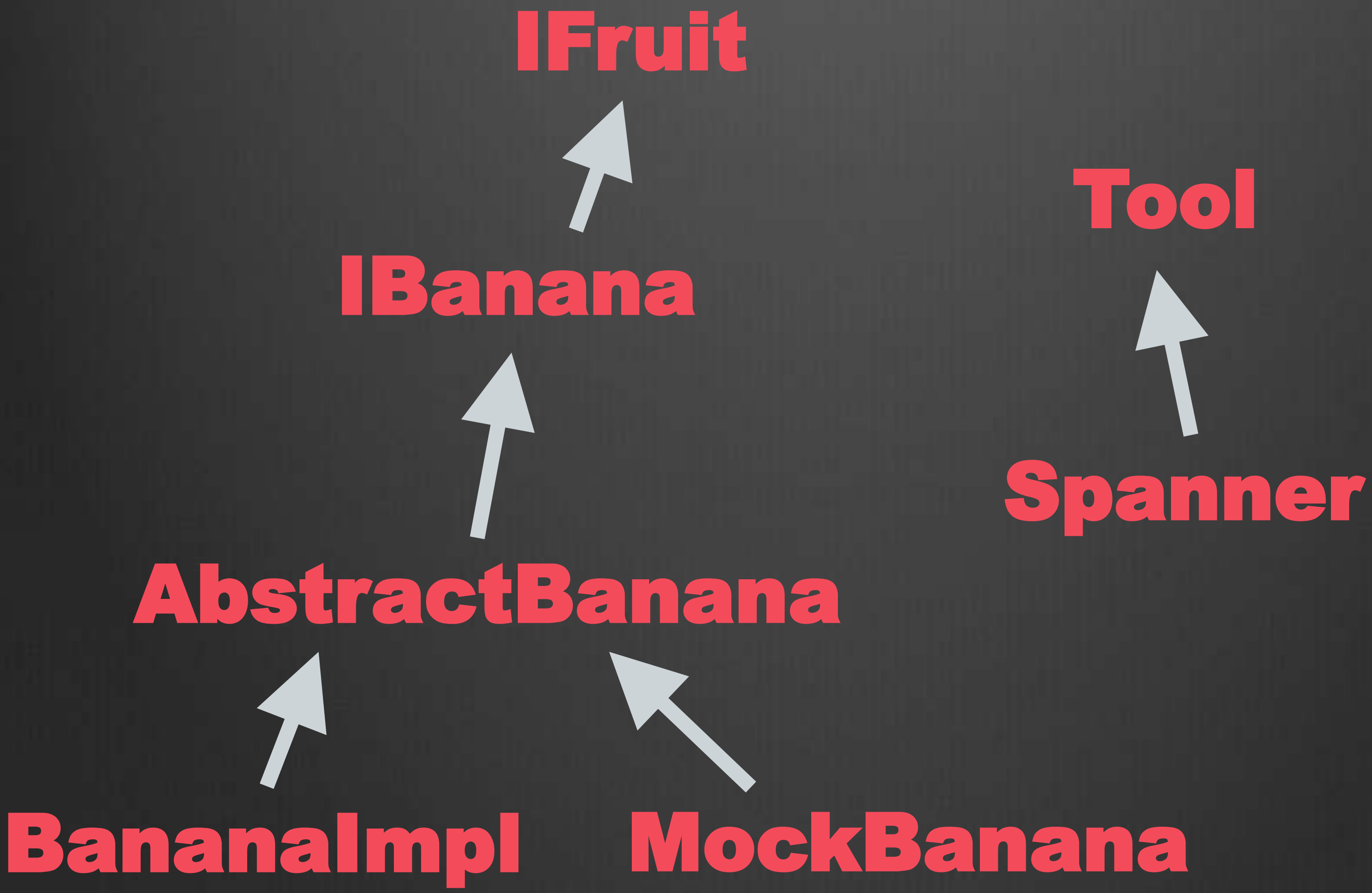
Two



Three



Class hierarchy



Class hierarchy

- **Objects**
- **Arrows**
- **Composition**
- **Identity**

Class hierarchy

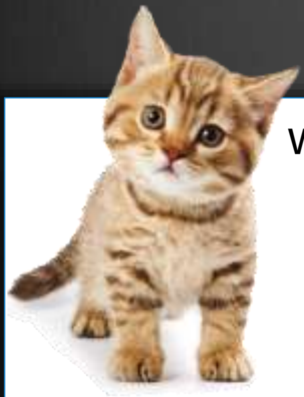
- **Objects** = classes
- **Arrows** = “extends”
- **Composition** = “extends” is transitive
- **Identity** = trivial

✓

~~Class hierarchy~~
Partially ordered sets (posets)

- **Objects** = elements in the set
- **Arrows** = ordering relation **≤**
- **Composition** = **≤** is transitive
- **Identity** = trivial

World Wide Web



www.naaawcats.com

No dogs allowed!

www.robodogs.com

See here for more robots

www.coolrobots.com

BUY NOW!!!!

World Wide Web

- **Objects** = webpages
- **Arrows** = hyperlinks
- **Composition** = Links don’t compose
- **Identity**

~~World Wide Web~~
Graphs

- **Objects** = nodes
- **Arrows** = edges
- **Composition** = Edges don’t compose
- **Identity**

"Free Category" from graphs!

- **Objects** = nodes
- **Arrows** = **paths (0 to many edges)**
- **Composition** = aligning paths end to end
- **Identity** = you're already there ✓

Categories in code

```
trait Category[Arrow[_,_]] {

  def compose[A,B,C](
    c: Arrow[B,C],
    d: Arrow[A,B]): Arrow[A,C]

  def id[A]: Arrow[A,A]

}
```

Category of Types & Functions

```
object FnCat
    extends Category[Function1] {

  def compose[A,B,C](
    c: B => C,
    d: A => B): A => C = {

      a => c(d(a))
  }

  def id[A]: A => A = (a => a)
}
```

Category of Garden Hoses

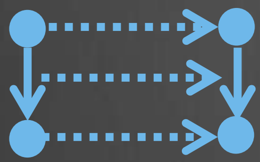
```scala
sealed trait Hose[In, Out] {
  def leaks: Int
  def kinked: Boolean

  def >>[A](in: Hose[A, In]):
                Hose[A, Out]
  def <<[A](out: Hose[Out, A]):
                Hose[In, A]

}
```

Category of Garden Hoses

[live code example]

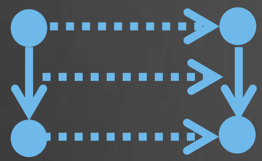
Categories embody the principle of strongly-typed composability

II. Functors



Functors

- Functors map between categories
- Objects → objects
- Arrows → arrows
- Preserves composition & identity



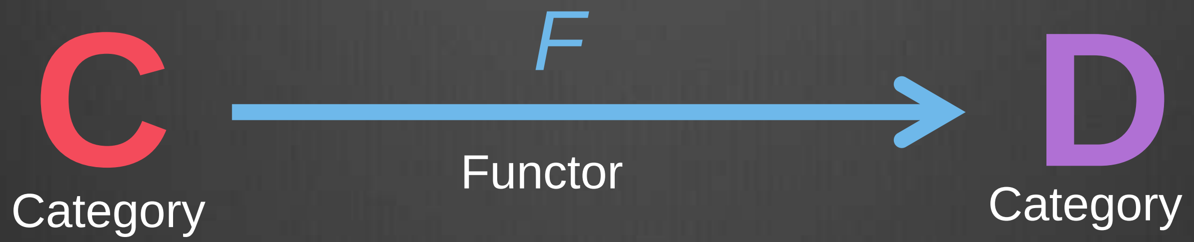
Functor laws

Composition Law

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

Identity Law

$$F(id_A) = id_{F(A)}$$


C
Category
F
Functor
D
Category

Cat

Category of categories

$$\mathbf{C} \xrightarrow[\text{Functor}]{F} \mathbf{D}$$

Category — Category

Cat

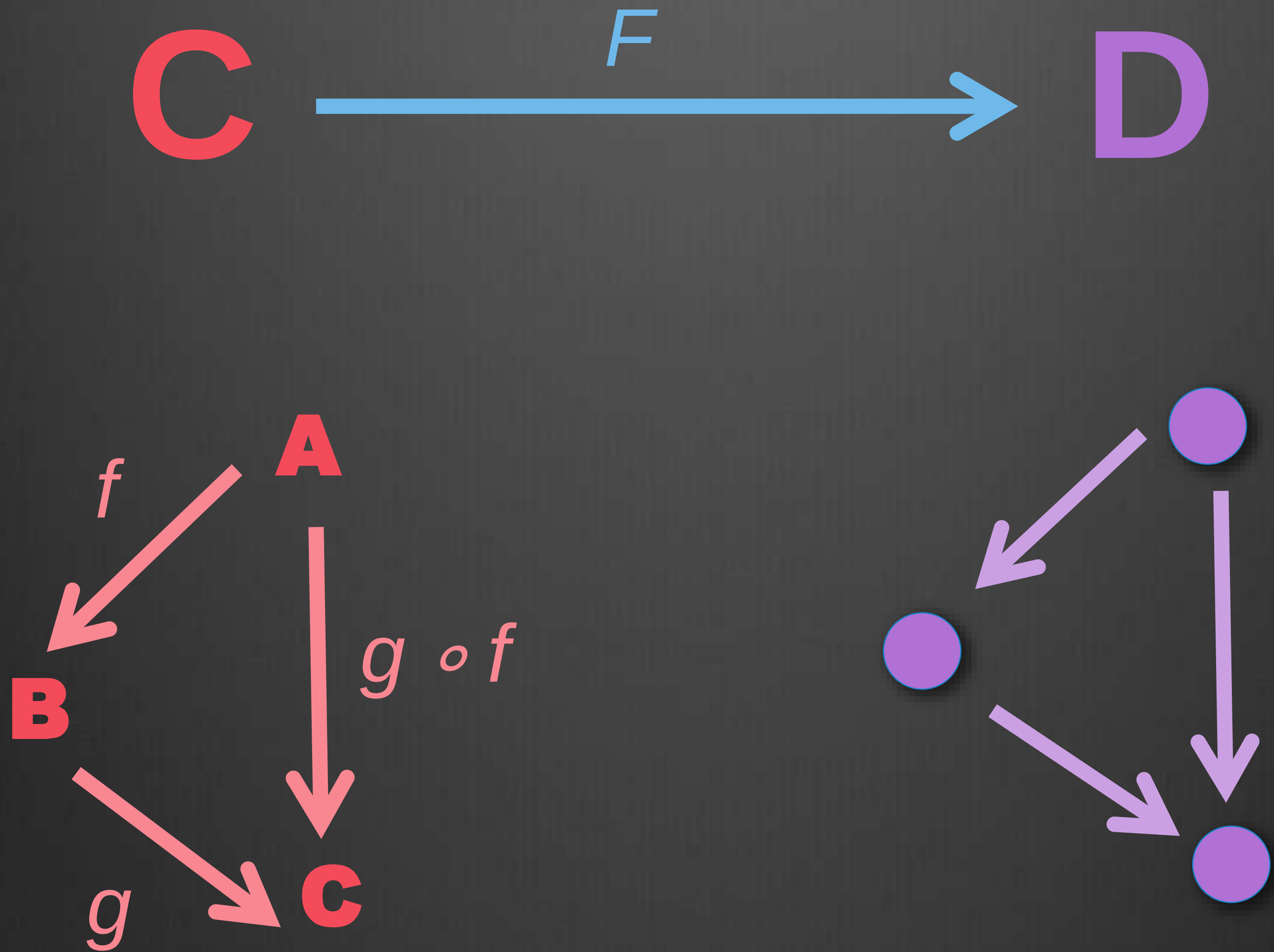
Category of categories

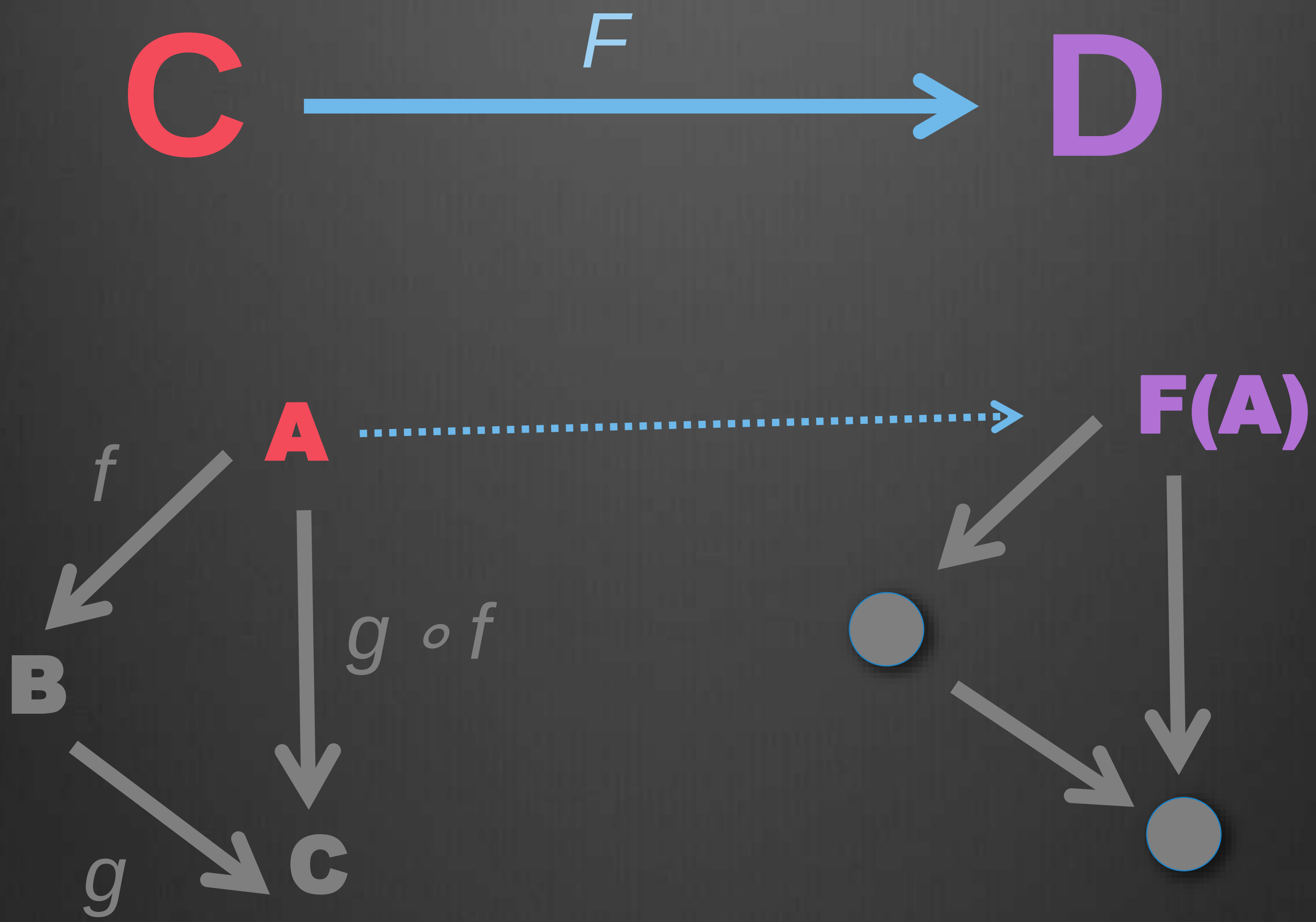
Objects = categories
Arrows = functors
Composition = functor composition
Identity = Identity functor

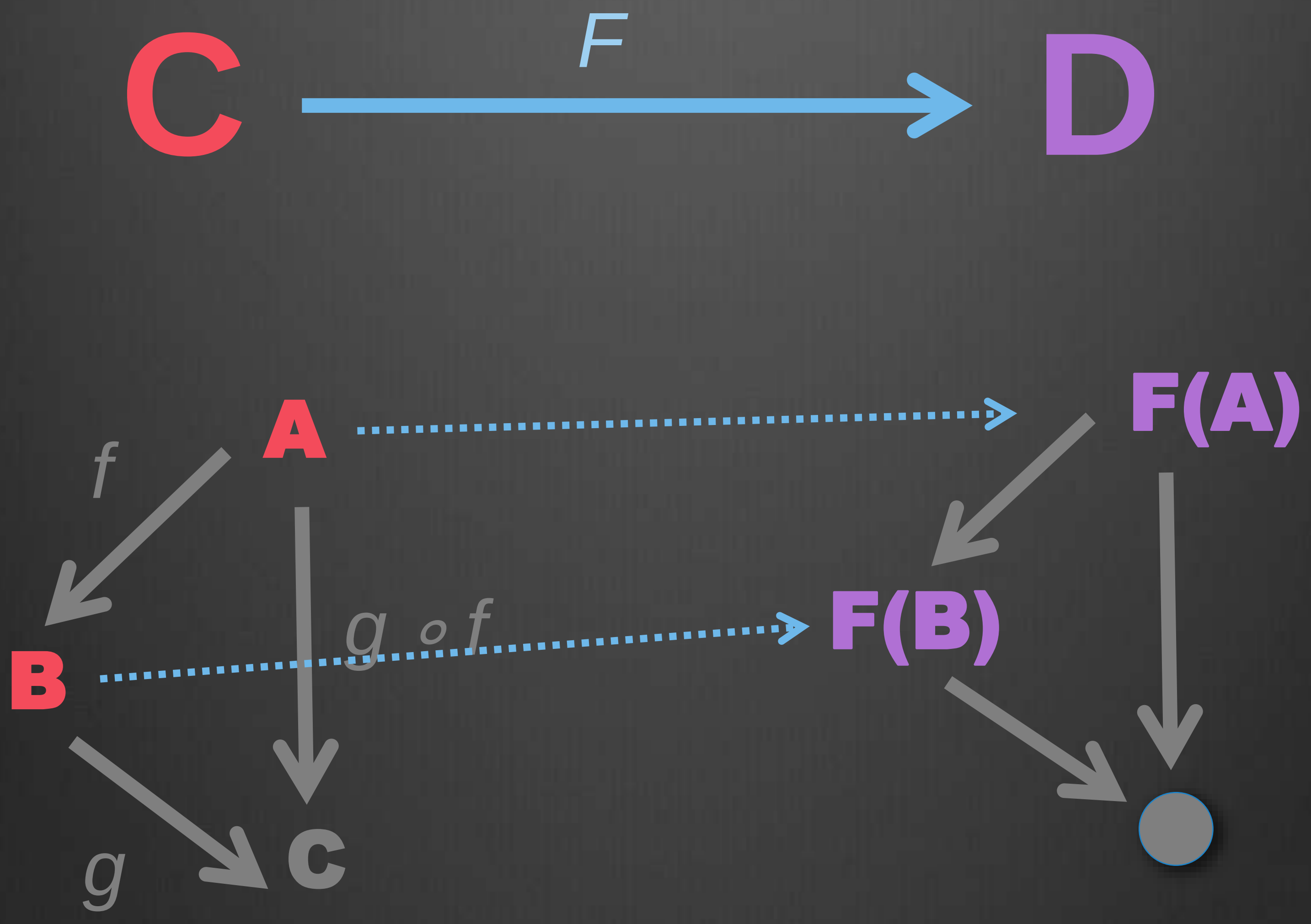
✓

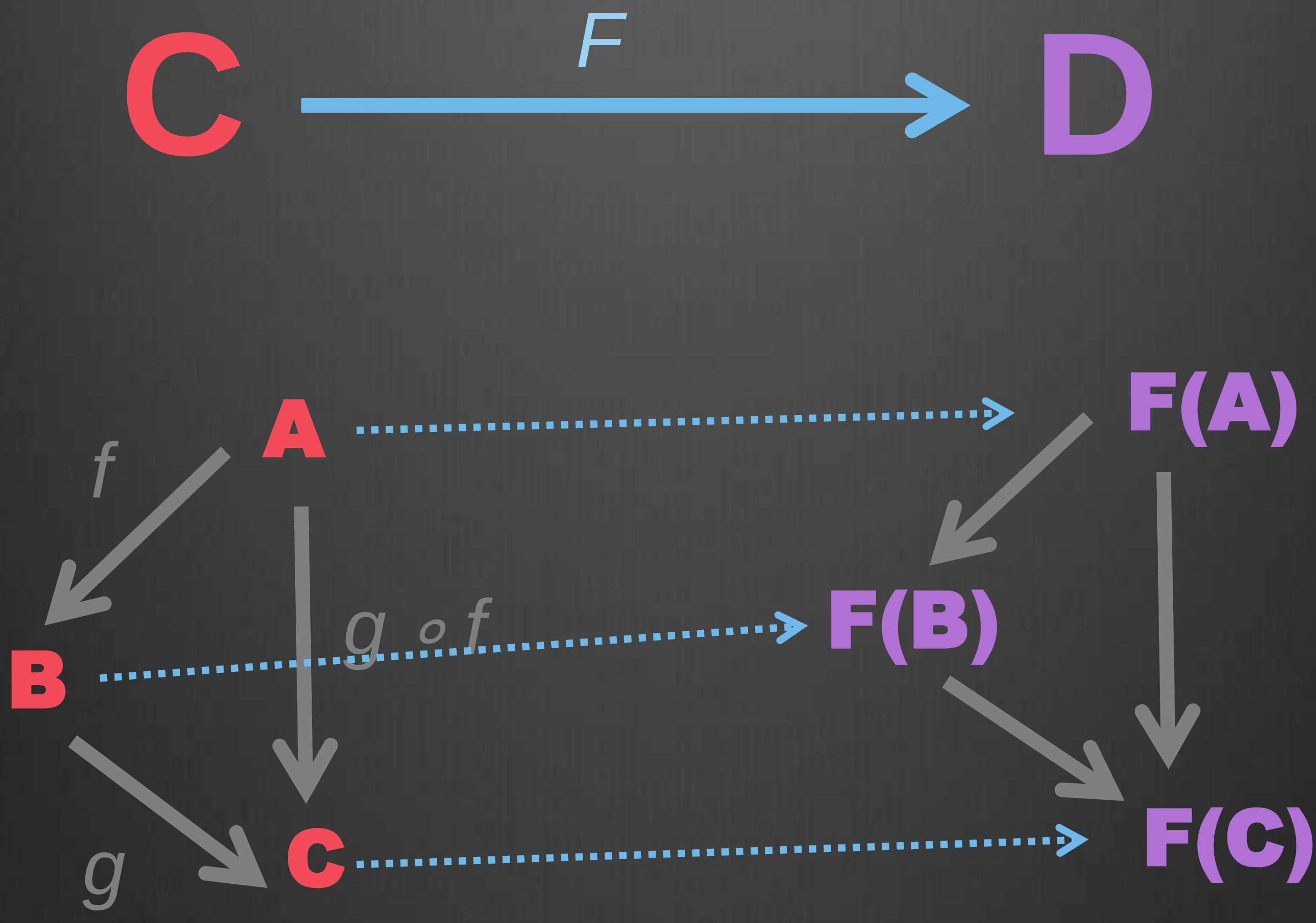
$$\mathbf{C} \xrightarrow[\text{Functor}]{F} \mathbf{D}$$

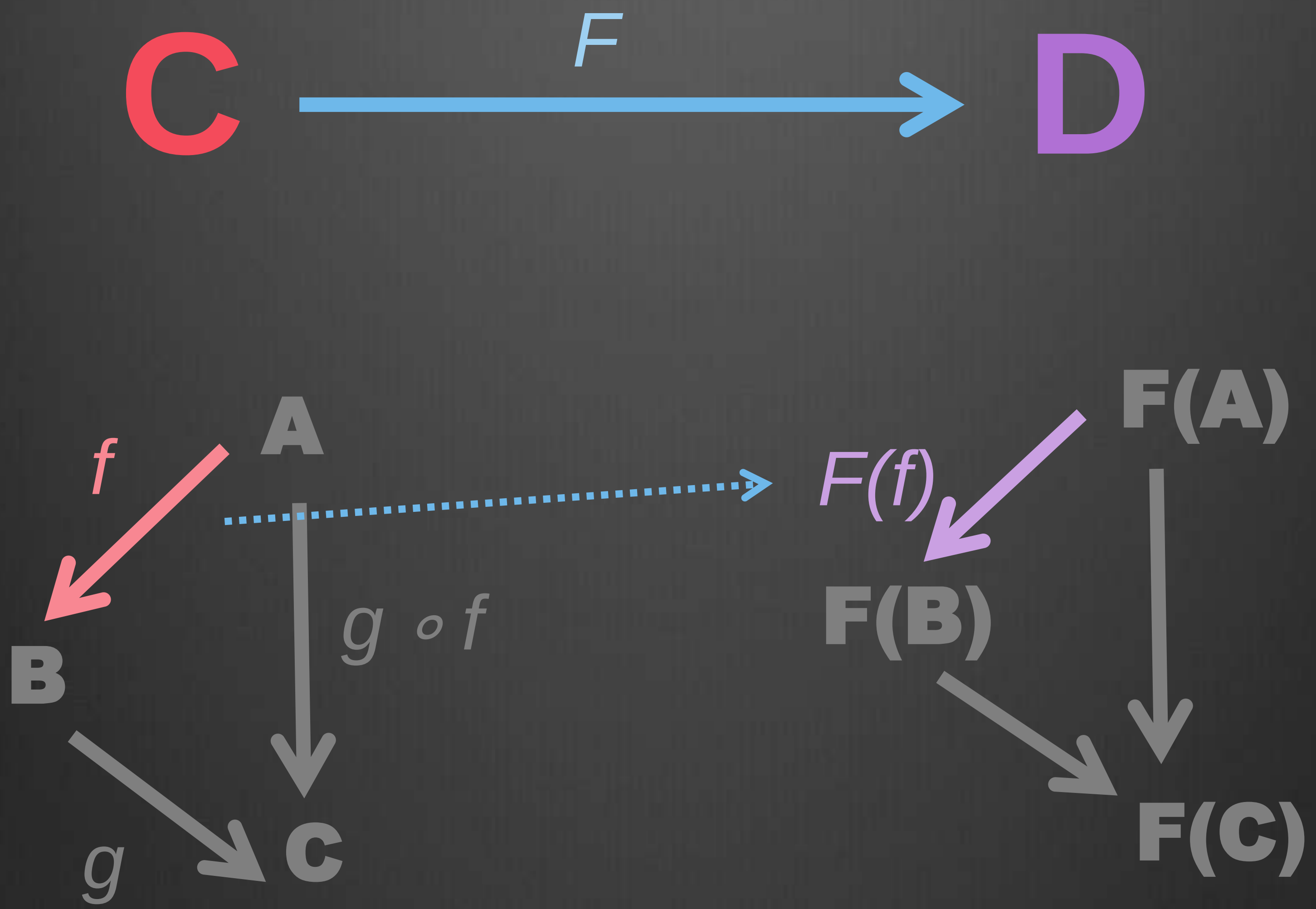
Category | Category


C
F
D
A
f
B
g ∘ f
g
C

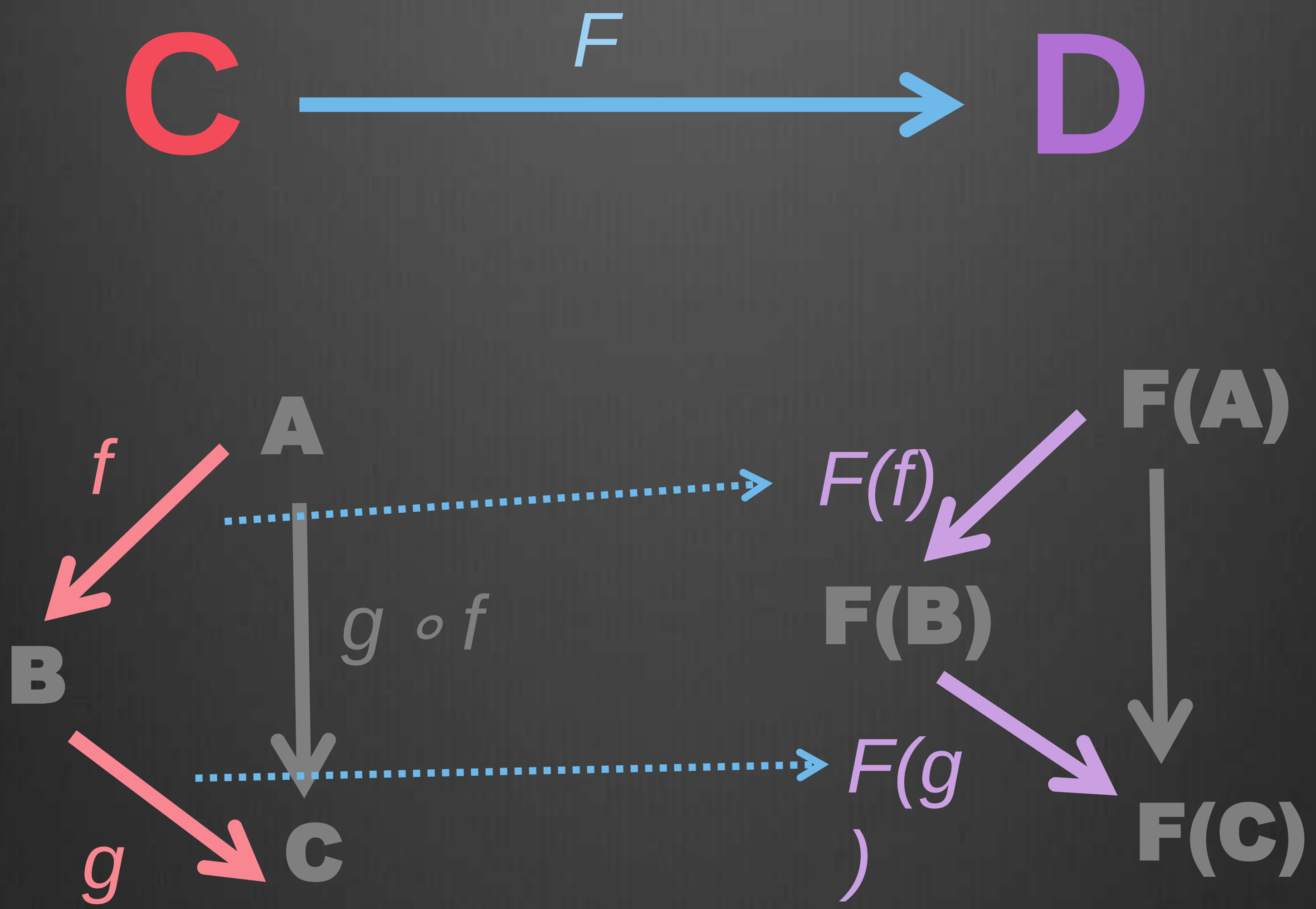

C
F
D
A
F(A)
f
B
g ∘ f
g
C

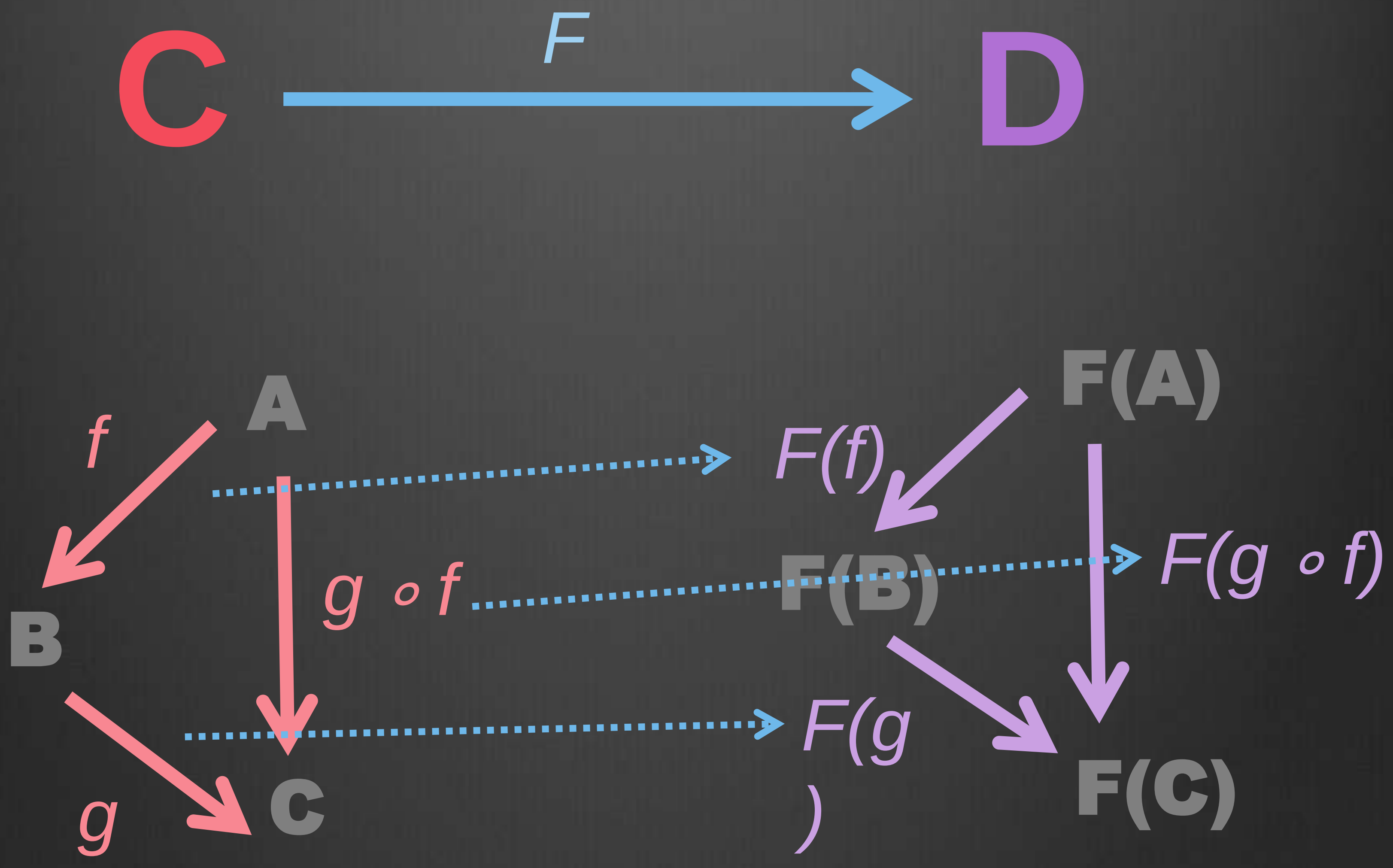

C
F
D
A
f
B
g ∘ f
g
c
F(A)
F(B)


C
F
D
A
F(A)
f
F(B)
g ∘ f
B
g
C
F(C)



C
F
D
A
f
B
g ∘ f
g
C
F(f)
F(A)
F(B)
F(C)

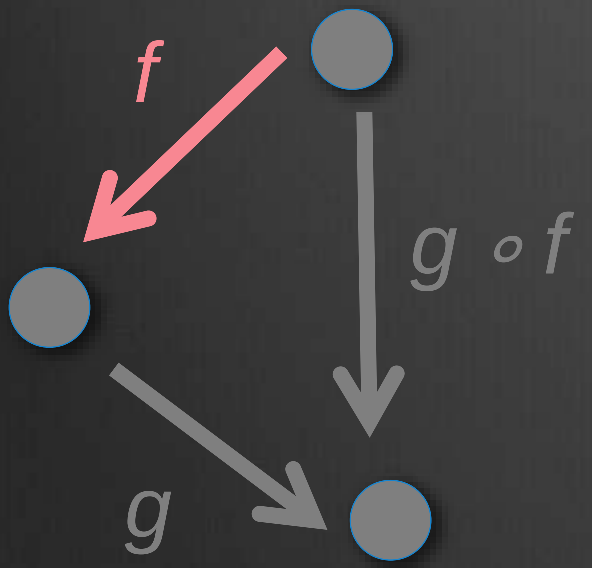

C
F
D
A
f
B
g
C
g ∘ f
F(f)
F(g
)
F(A)
F(B)
F(C)

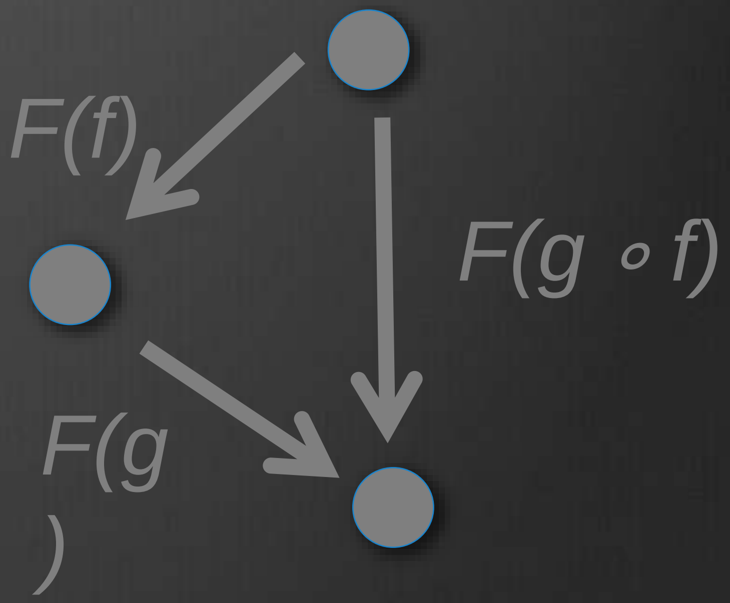


C
F
D
A
f
g ∘ f
B
g
C
F(f)
F(A)
F(B)
F(g ∘ f)
F(g
)
F(C)

Composition Law

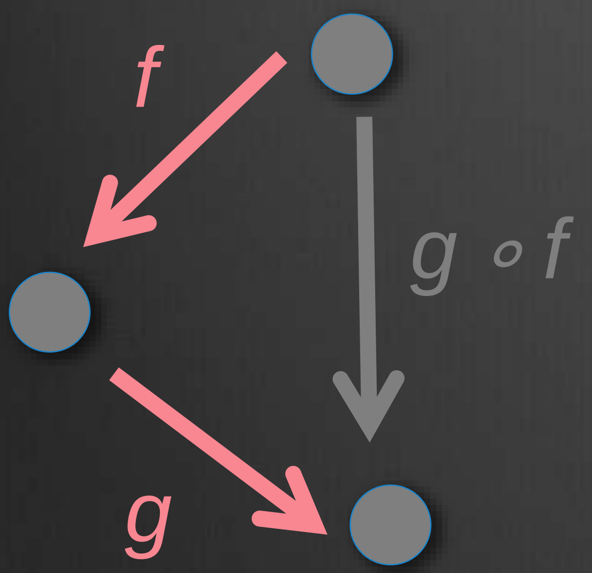
$F(g \circ f) = F(g) \circ F(f)$

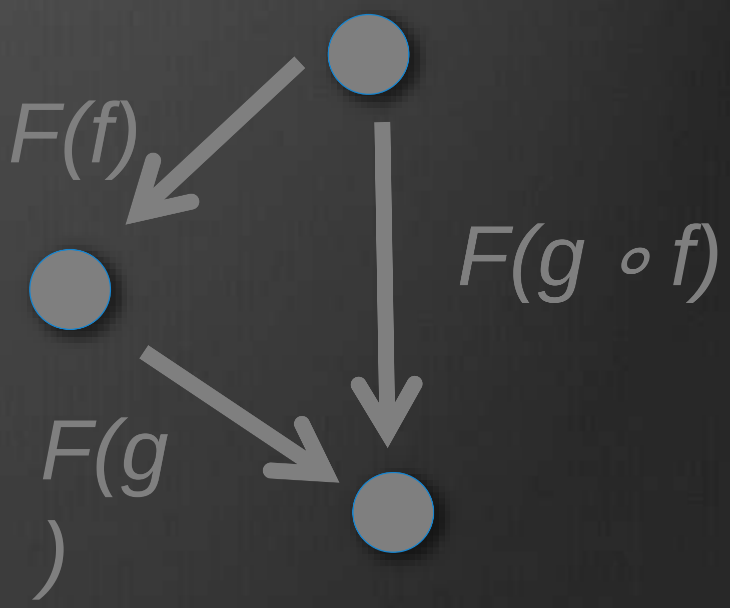




Composition Law

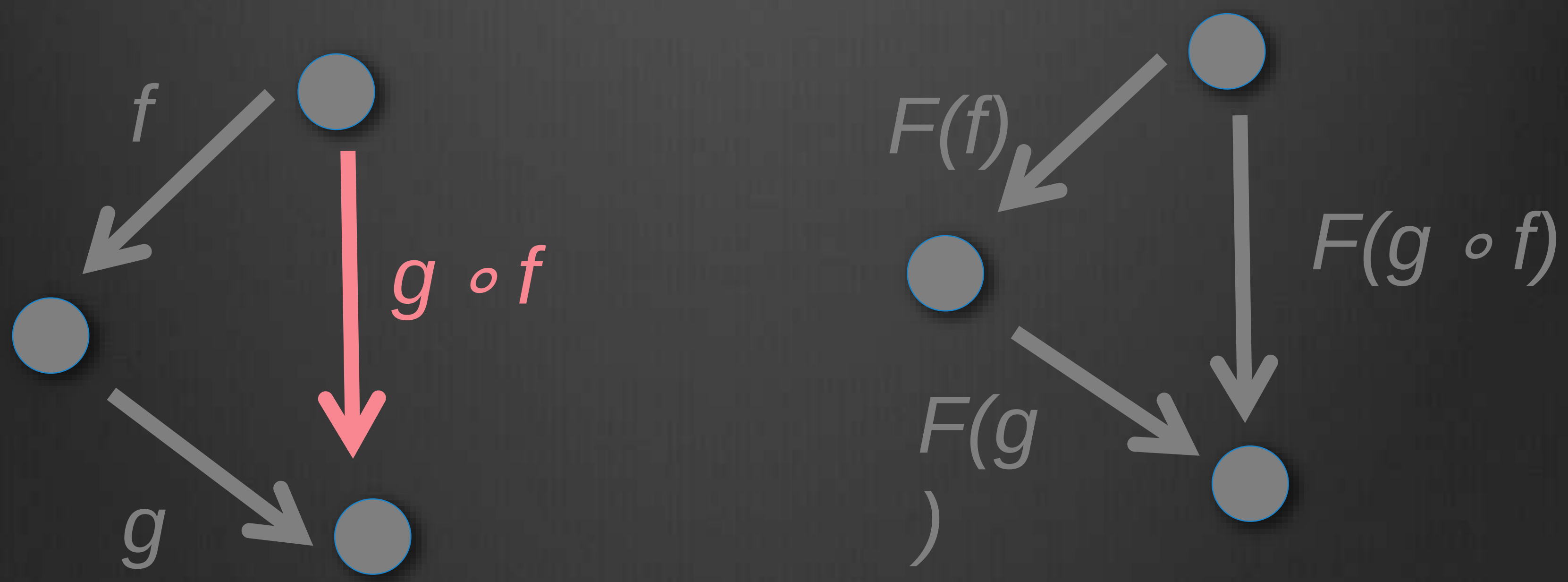
$$F(g \circ f) = F(g) \circ F(f)$$





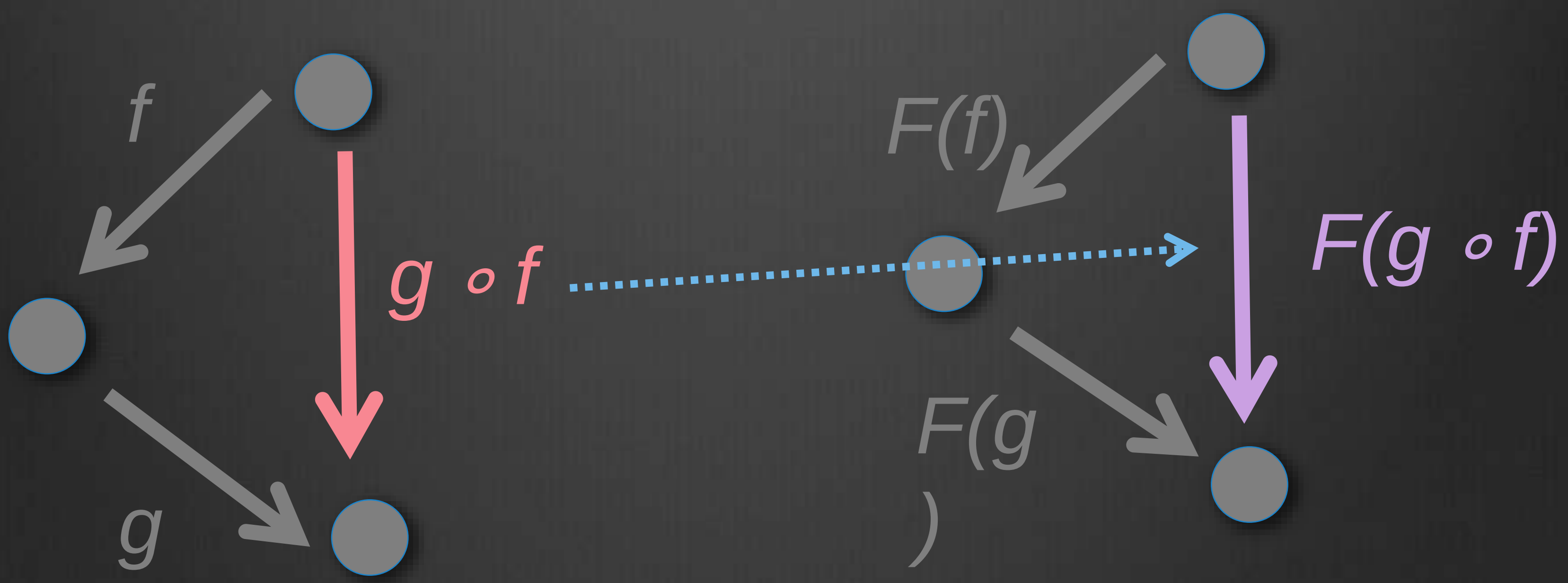
Composition Law

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



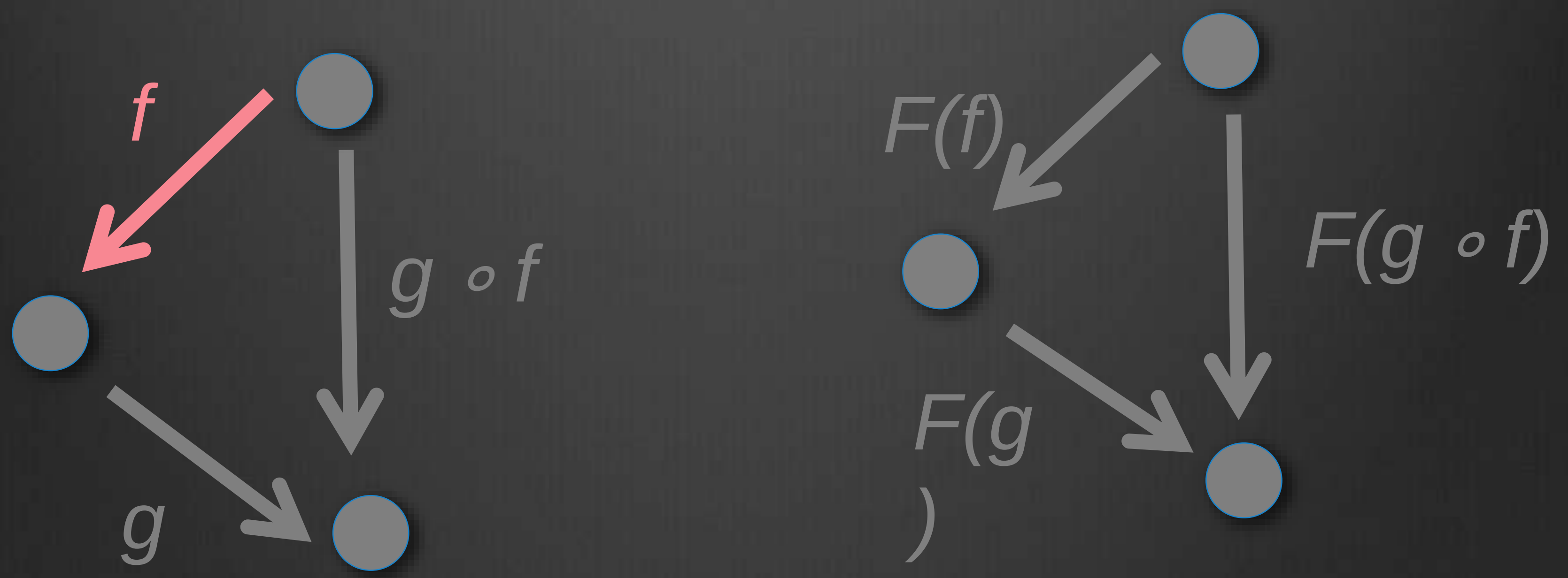
Composition Law

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



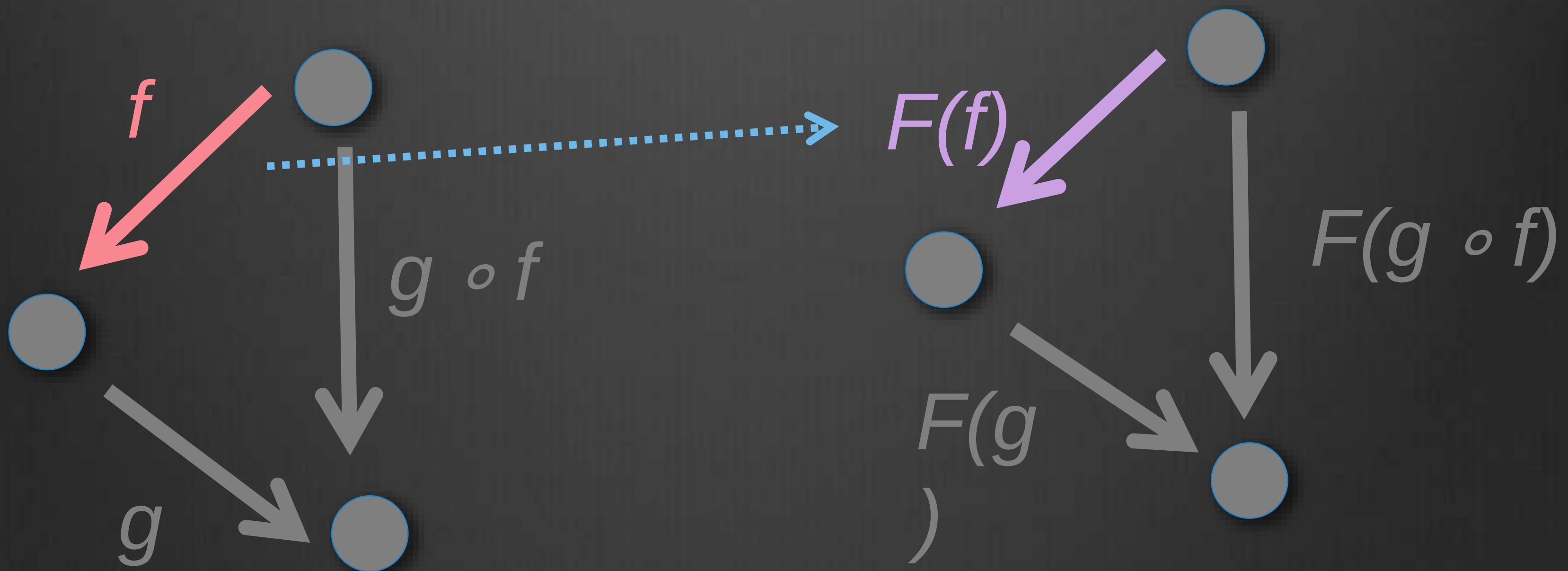
Composition Law

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



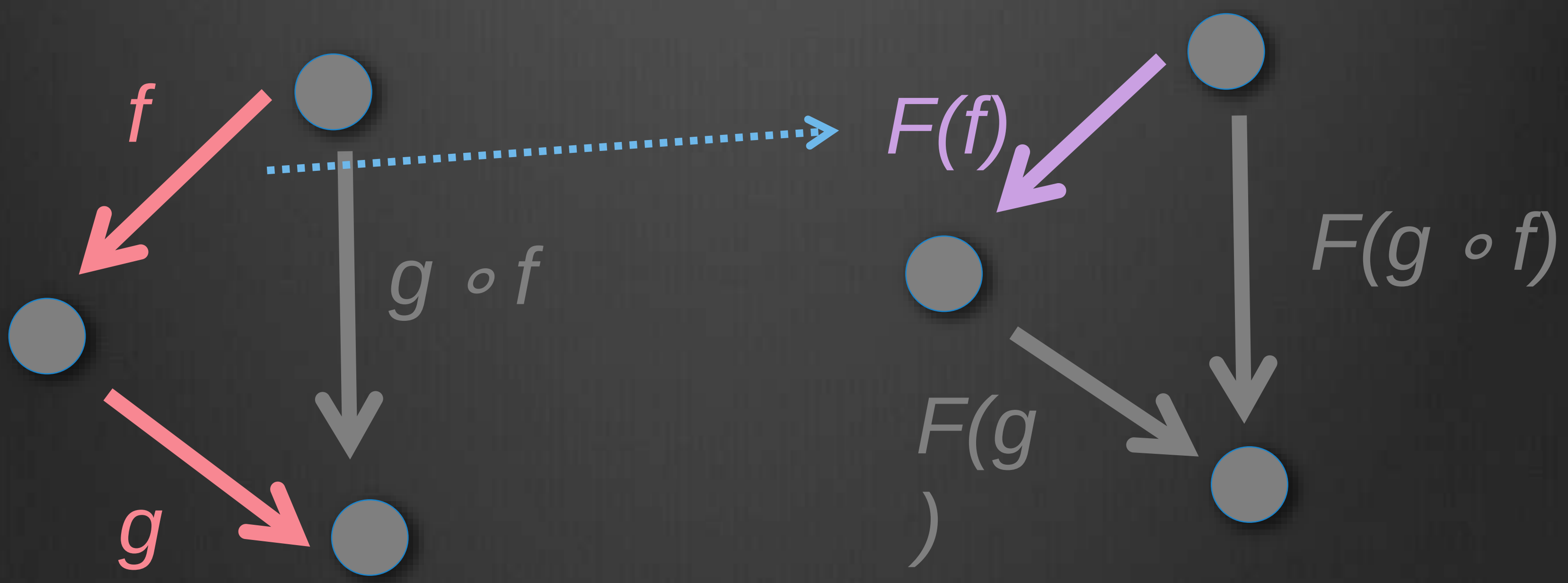
Composition Law

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



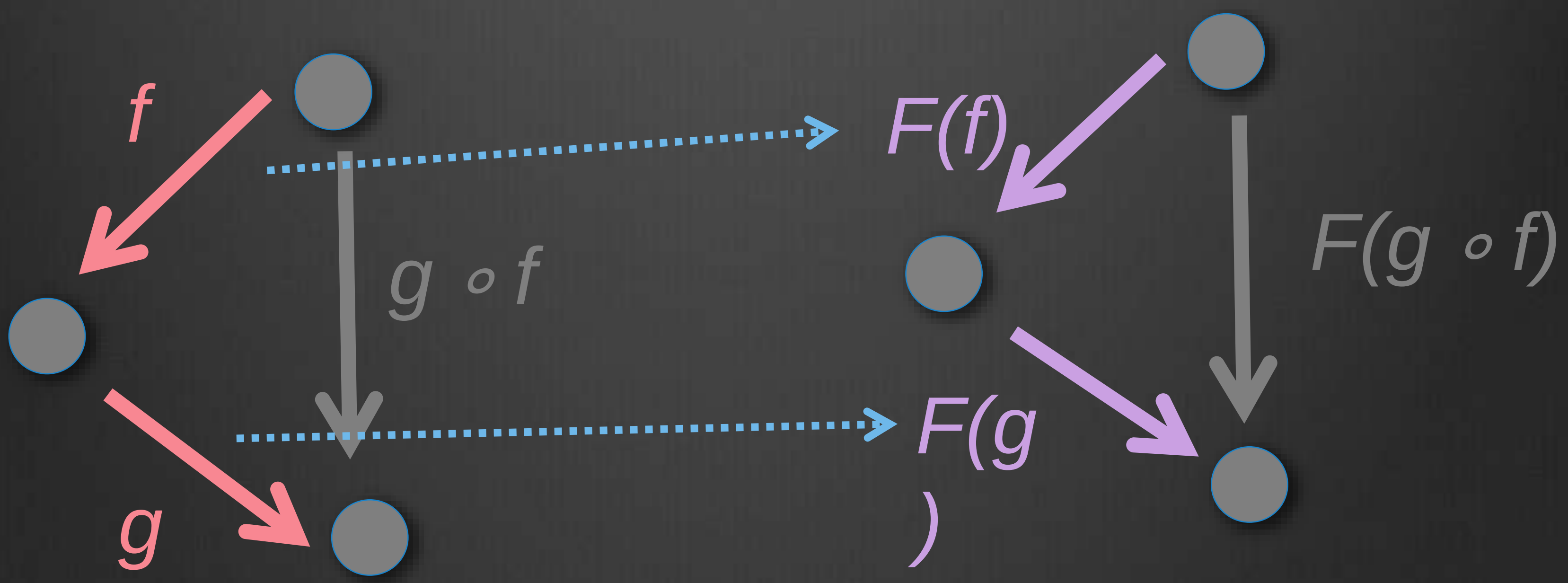
Composition Law

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



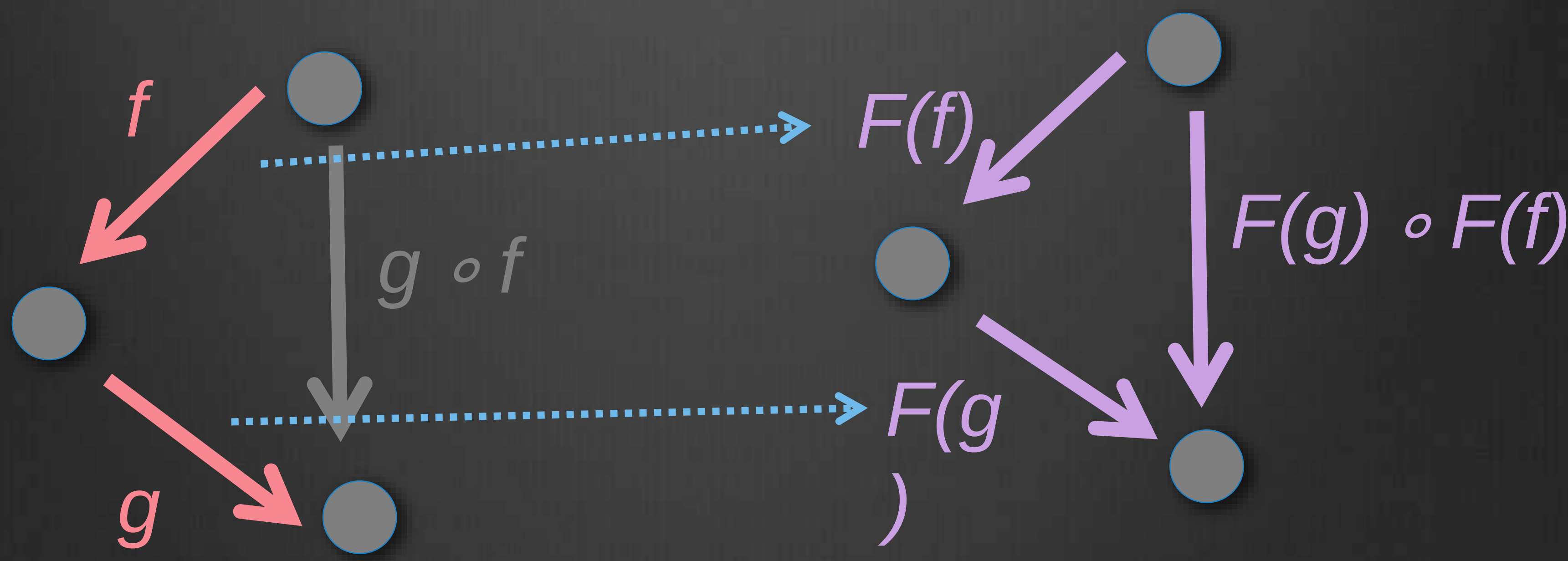
Composition Law

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



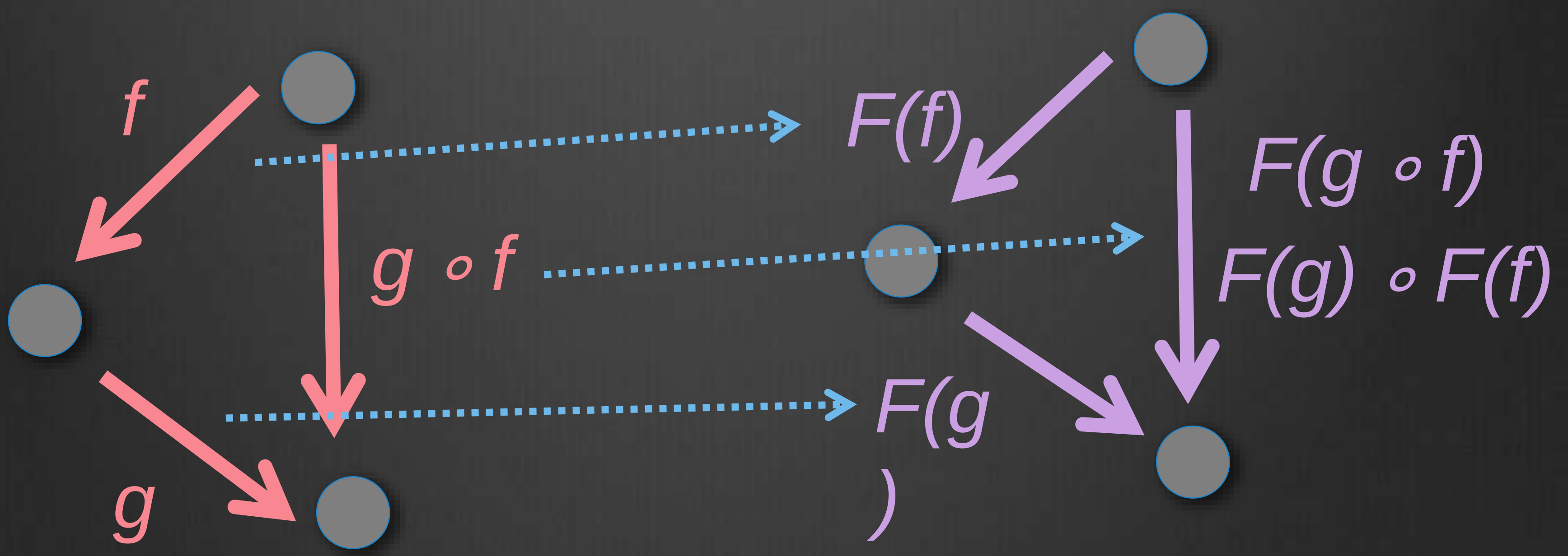
Composition Law

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



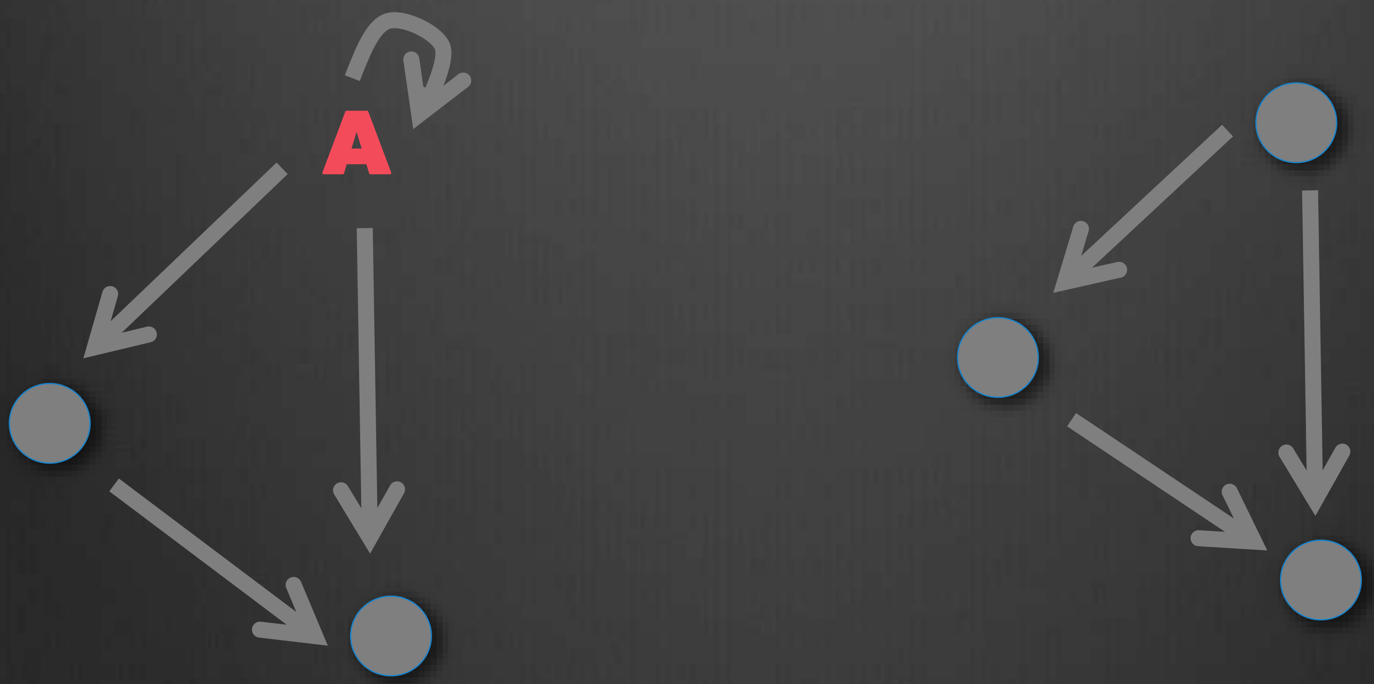
Composition Law

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



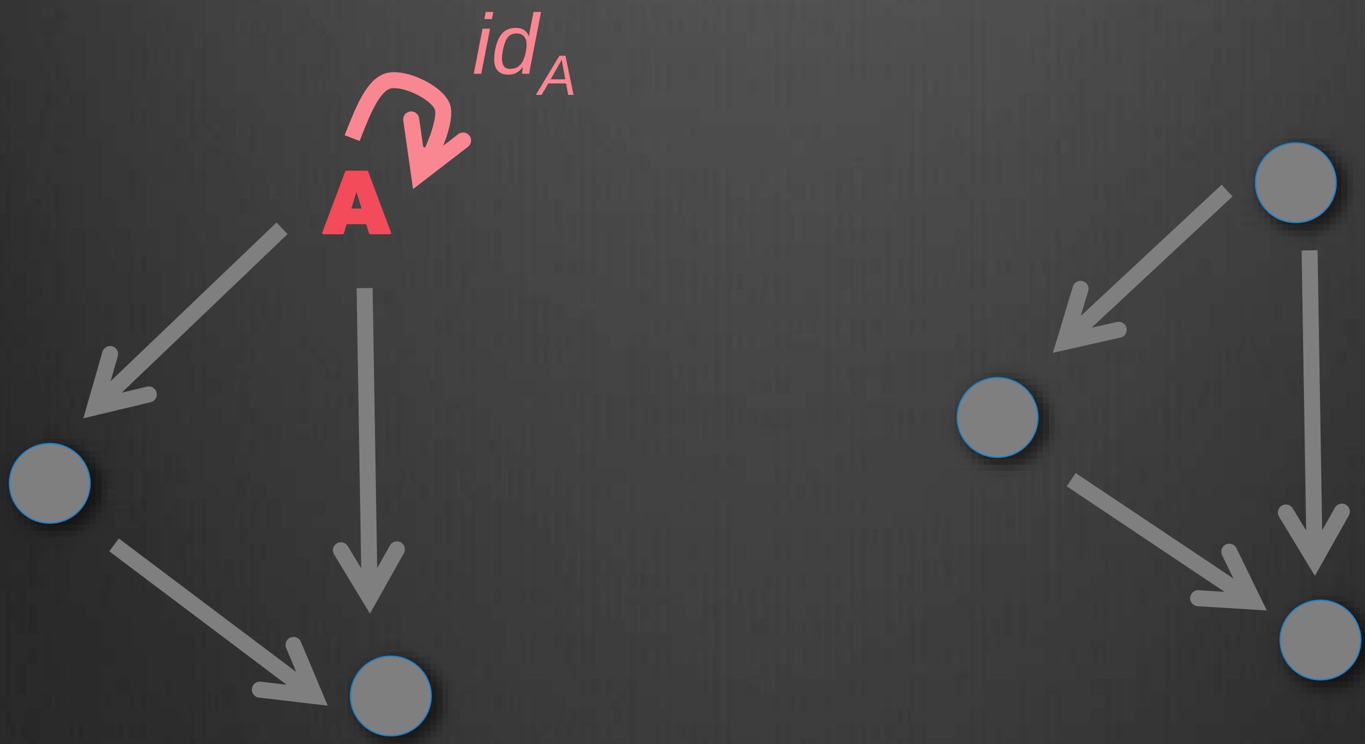
Identity law

$F(id_A) = id_{F(A)}$



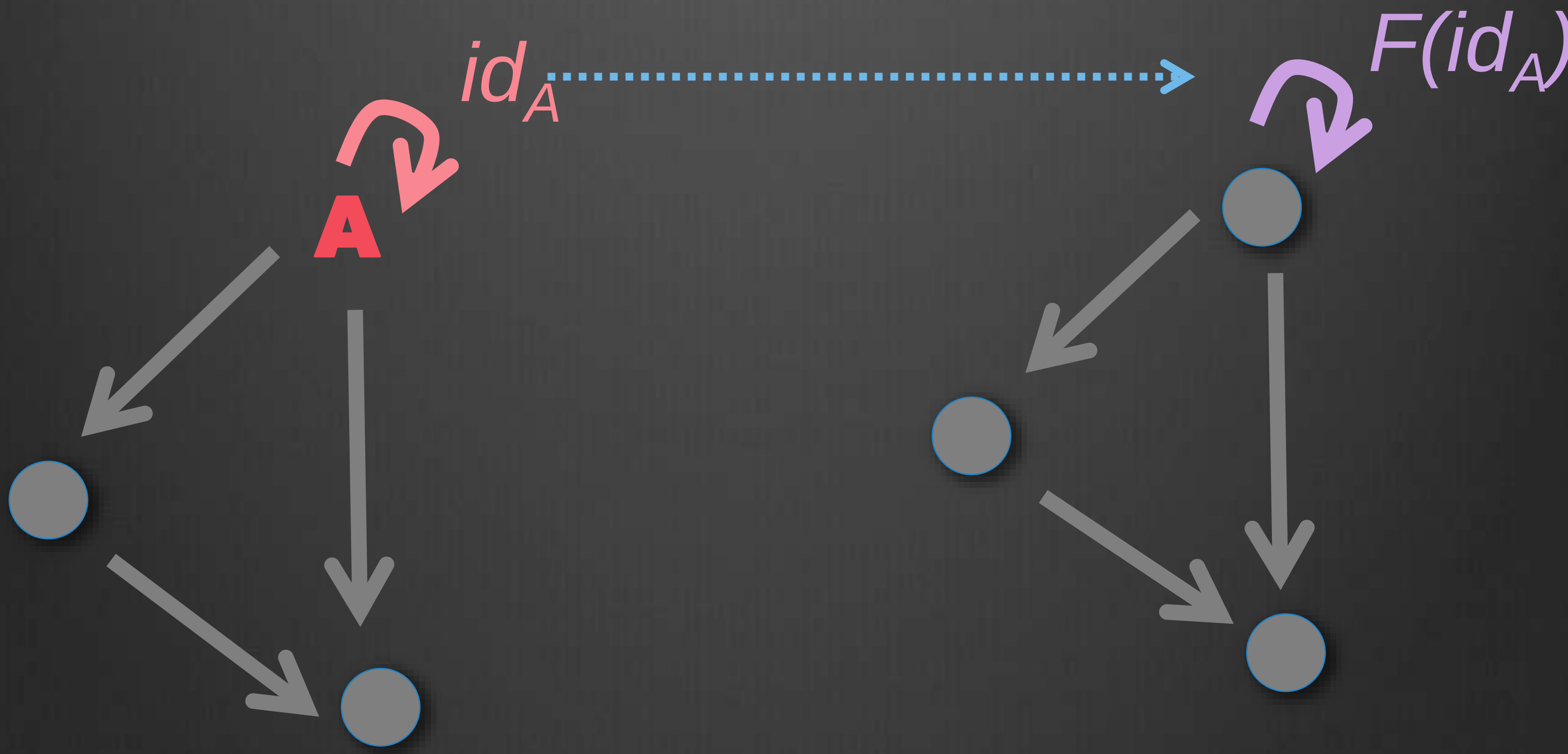
Identity law

$F(id_A) = id_{F(A)}$



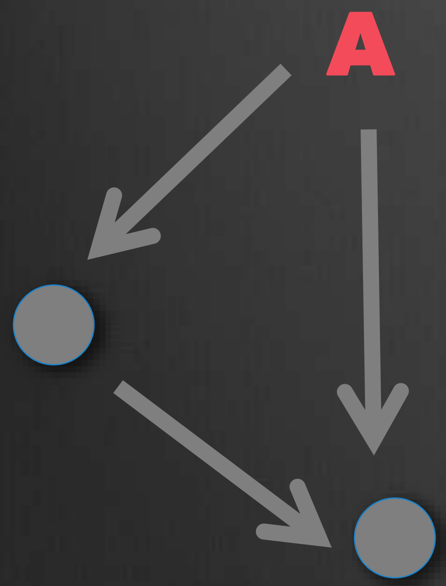
Identity law

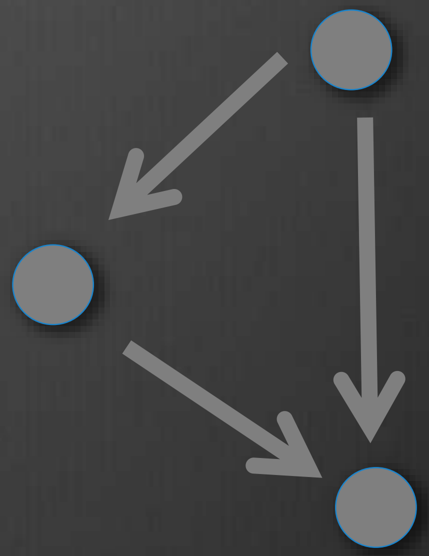
$$F(id_A) = id_{F(A)}$$



Identity law

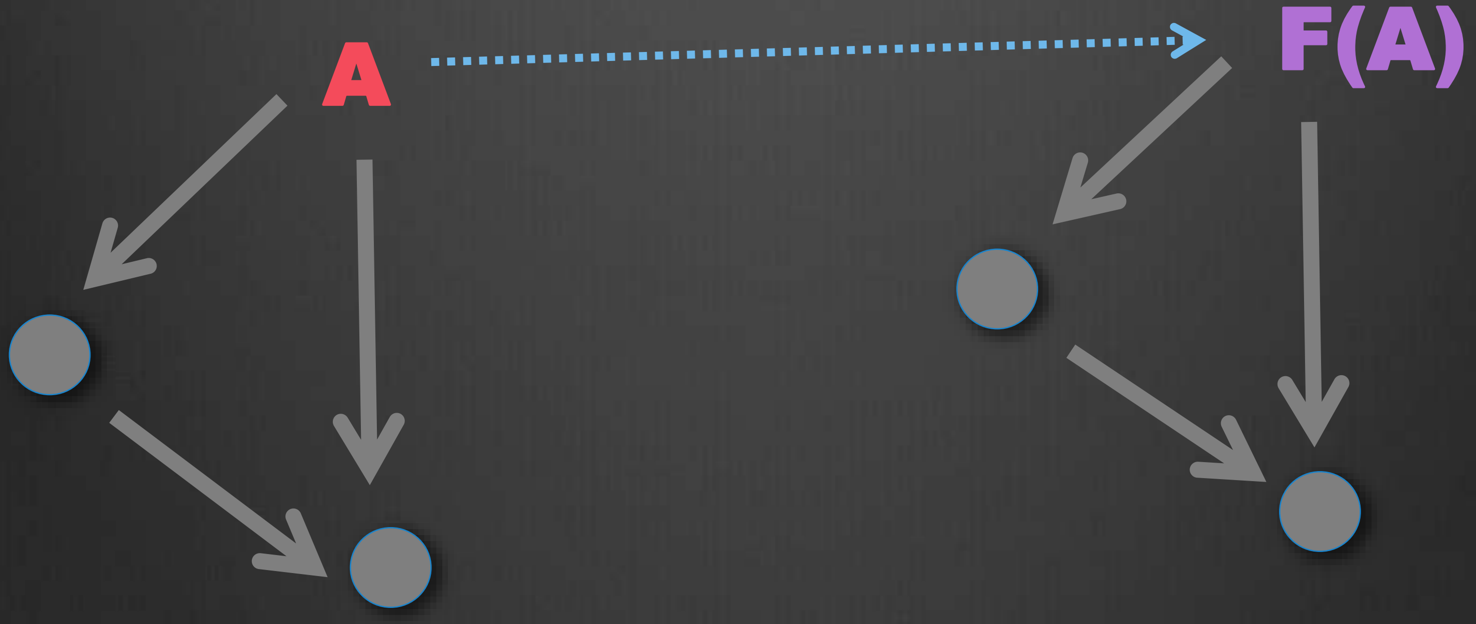
$F(id_A) = id_{F(A)}$





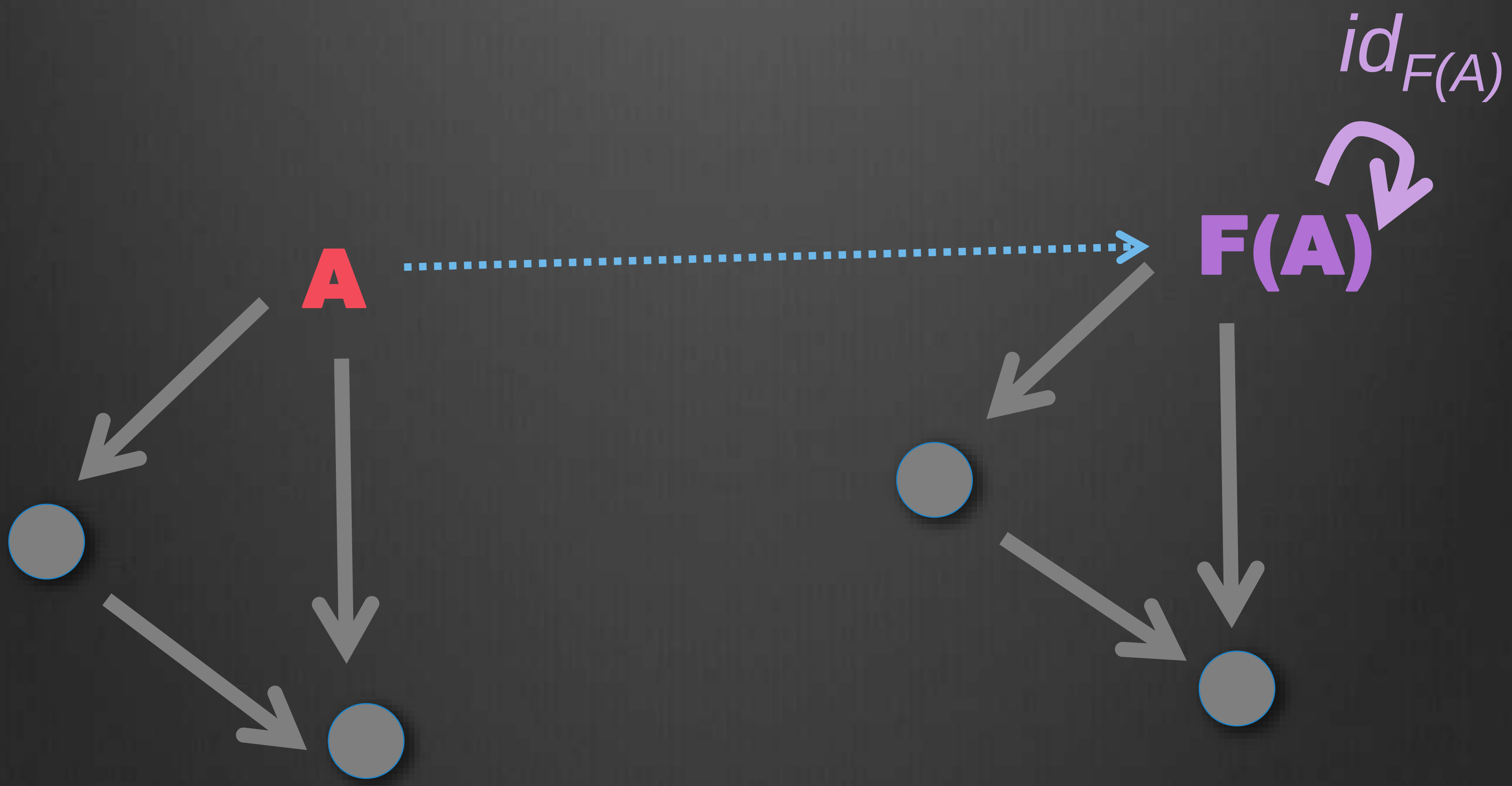
Identity law

$F(id_A) = id_{F(A)}$



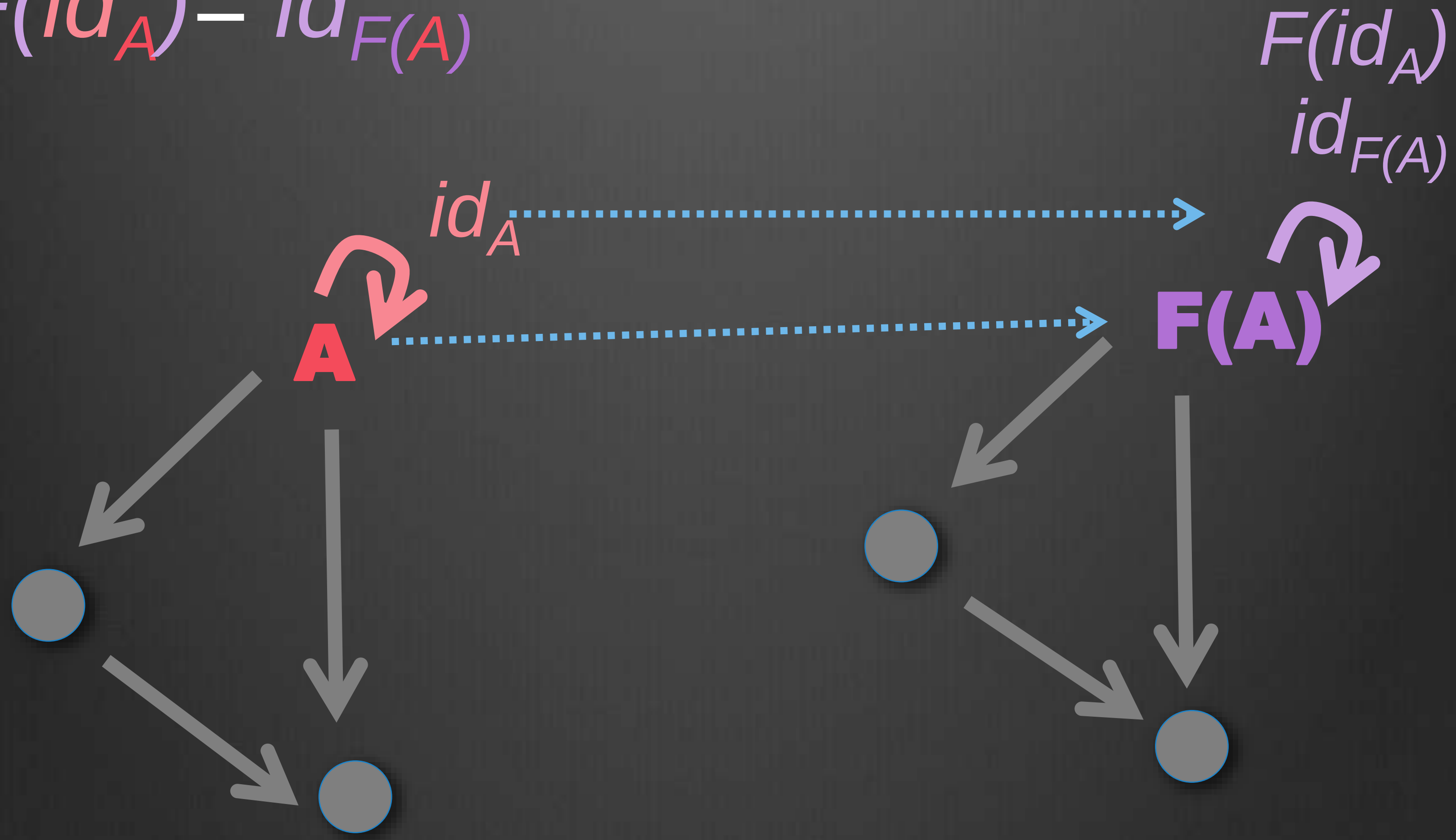
Identity law

$F(id_A) = id_{F(A)}$



Identity law

$F(id_A) = id_{F(A)}$



Terminology

homomorphism

Terminology

homomorphism

Same

Terminology

homomorphism

Same-shape-ism

Terminology

homomorphism

“structure preserving map”

Terminology

homomorphism

Functors are
“category homomorphisms”

Functors in code

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

Functors in code

Objects to objects

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

Functors in code

Arrows to arrows

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

Functors in code

Arrows to arrows

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

Functors laws in code

```
fa.map(f).map(g)

==

fa.map(g compose f)
```

Functors laws in code

```
fa.map(a => a) == fa
```

Terminology

endomorphism

Terminology

endomorphism

Within

Terminology

endomorphism

Within -shape-ism

Terminology

endomorphism

“a mapping from something back to itself”

Terminology

endo

“a mapping from something back to itself”

Endofunctors

In Scala, all our functors are actually *endofunctors*.

$$\text{Type} \xrightarrow[\text{Endofunctor}]{F} \text{Type}$$

Category — Category

Endofunctors

Luckily, we can represent any functor in our type system as some `F[_]`

$$\begin{array}{ccc} \textbf{Type} & \xrightarrow[\text{Endofunctor}]{F} & \textbf{Type} \\ \text{Category} & & \text{Category} \end{array}$$

List Functor

```
sealed trait List[+A]

case class Cons(head: A, tail: List[A])
  extends List[A]

case object Nil extends List[Nothing]
```

List Functor

```scala
sealed trait List[+A] {

  def map[B](f: A => B): List[B] =
    this match {
      case Cons(h,t) => Cons(f(h), t map f)
      case Nil => Nil
    }
  }
}
```

List Functor

```
potatoList
  .map(mashEm)
  .map(boilEm)
  .map(stickEmInAStew)
```

List Functor

```
userList
  .map(_.name)
  .map(_.length)
  .map(_ + 1)
  .map(_.toString)
```

Other functors

```
trait Tree[A]

trait Future[A]

trait Process[A]

trait Command[A]

X => A

(X, A)

trait Option[A]
```

Functors

- Fundamental concept in Category Theory
- Super useful
- Everywhere
- Staple of functional programming
- Write code that's ignorant of unnecessary context

III. Monoids

Monoids

Some set we'll call M

Compose

$\bullet : M \times M \rightarrow M$

Identity

$id : M$

Monoid Laws

Associative Law

$$(f \bullet g) \bullet h \;=\; f \bullet (g \bullet h)$$

Identity Laws

$$f \bullet id \;=\; id \bullet f \;=\; f$$

Category Laws

Associative Law

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

Identity Laws

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

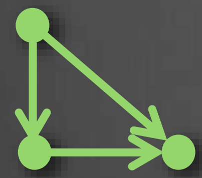
Monoids

Compose

$$\bullet : M \times M \to M$$

Identity

$$id : M$$



Category

Compose

$\circ : (B \to C) \to (A \to B) \to (A \to C)$

Identity

$id : A \to A$

Category with 1 object

Compose

$\circ : (A \rightarrow A) \rightarrow (A \rightarrow A) \rightarrow (A \rightarrow A)$

Identity

$id : A \rightarrow A$

Category with 1 object

Compose

$\circ : M \rightarrow M \rightarrow M$

Identity

$id : M$

Monoids are categories

Only one object

Each arrow is an element
in the monoid

Monoids are categories

Only one object

Objects = placeholder singleton
Arrows = elements of the monoid
Composition = •
Identity = id

Each arrow is an element in the monoid

$$\mathbf{M} \xrightarrow[\text{Monoid homomorphism}]{H} \mathbf{N}$$

$(S_M, \bullet_M, id_M)$

Monoid

$(S_N, \bullet_N, id_N)$

Monoid

Mon

Category of monoids

$$\mathbf{M} \xrightarrow{H} \mathbf{N}$$

Monoid homomorphism

$(S_{\mathrm{M}}, \bullet_M, id_M)$

Monoid

$(S_{\mathrm{N}}, \bullet_N, id_N)$

Monoid

Mon

Category of monoids

Objects = monoids
Arrows = monoid homomorphisms
Composition = function composition
Identity = Identity function

$$\mathbf{M} \xrightarrow{\quad H \quad} \mathbf{N}$$

Monoid homomorphism

$(S_M, \bullet_M, id_M)$ $(S_N, \bullet_N, id_N)$

Monoid Monoid

$$\mathbf{M} \xrightarrow{H} \mathbf{N}$$

“structure-preserving map”

Sets

$$\underset{\text{Set}}{S_M} \xrightarrow[\text{function}]{h} \underset{\text{Set}}{S_N}$$

Where *h* preserves composition & identity

Example

String length is a monoid homomorphism from `(String, +, "")` to `(Int, +, 0)`

Preserves identity

"".length == 0

Preserves composition

(str1 + str2).length = str1.length + str2.length

Monoids in code

```
trait Monoid[M] {
  def compose(a: M, b: M): M
  def id: M
}
```

Monoids in code

```
def foldMonoid[M: Monoid](
    ms: Seq[M]): M = {

  ms.foldLeft(Monoid[M].id)
             (Monoid[M].compose)

}
```

Int / 0 / +

```
import IntAddMonoid._

foldMonoid[Int](Seq(
  1,2,3,4,5,6))
```

➔ 21

Int / 1 / *

```
import IntMultMonoid._

foldMonoid[Int](Seq(
  1,2,3,4,5,6))
```

➔ 720

String / "" / +

```
foldMonoid[String](Seq(
  "alea",
  "iacta",
  "est"))
```

➔ ”aleaiactaest"

Endos / *id* / ∘

```
def mash: Potato => Potato
def addOne: Int => Int
def flipHorizontal: Shape => Shape
def bestFriend: Person => Person
```

A=>A / a=>a / compose

```
foldMonoid[Int => Int](Seq(
  _ + 12,
  _ * 2,
  _ - 3))
```

➔ (n: Int) => ((n + 12) * 2) - 3

Are chairs monoids?

Chair

Composition =

You can't turn two chairs into one

Identity =

Chair stack

Chair stack

Composition = stack them on top
Identity = no chairs

✓

Chair Stack is the *free monoid* of chairs

Protip: just take 0-to-many of anything, and you get a monoid for free

…almost

Real monoids don’t topple;
they keep scaling

Monoids embody the principle of

weakly-typed composability

IV. Products & sums

Algebraic Data Types

```
List[A]
  - Cons(A, List[A])
  - Nil
```

```
Option[A]
  - Some(A)
  - None
```

```
Wiggles
  - YellowWiggle
  - BlueWiggle
  - RedWiggle
  - PurpleWiggle
```

```
BusinessResult[A]
  - OK(A)
  - Error
```

```
Address(Street, Suburb,
        Postcode, State)
```

Algebraic Data Types

```
  Cons(A × List[A])
+ Nil
```

```
  Some(A)
+ None
```

```
  OK(A)
+ Error
```

```
  YellowWiggle
+ BlueWiggle
+ RedWiggle
+ PurpleWiggle
```

```
Street × Suburb × Postcode × State
```

Algebraic Data Types

$A \times List[A] + 1$

$A + 1$

$A + 1$

4

$Street \times Suburb \times Postcode \times State$

Algebraic Data Types

`A × List[A] + 1`

`A + 1`

isomorphic

`A + 1`

`4`

`Street × Suburb × Postcode × State`

Terminology

isomorphism

Terminology

isomorphism

Equal

Terminology

isomorphism

Equal-shape-ism

Terminology

isomorphism

“Sorta kinda the same-ish” but I want to sound really smart

- Programmers

Terminology

isomorphism

“Sorta kinda the same-ish” but I want to sound really smart

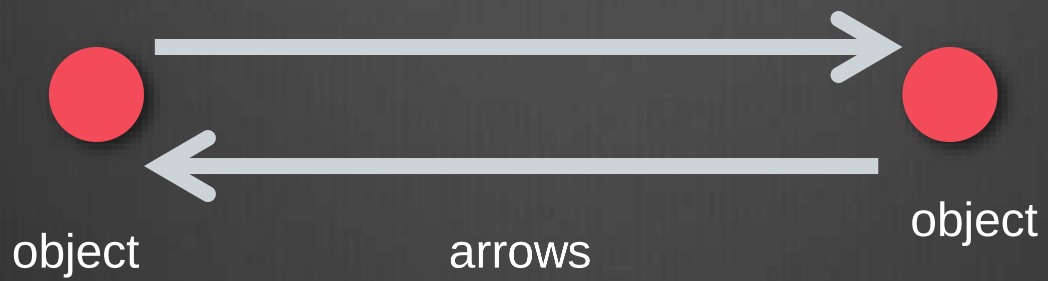
- Programmers

Terminology

isomorphism

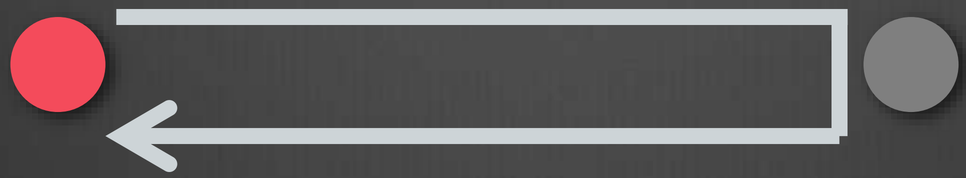
One-to-one mapping between two objects so you can go back-and-forth without losing information

Isomorphism



Isomorphism

Same as identity



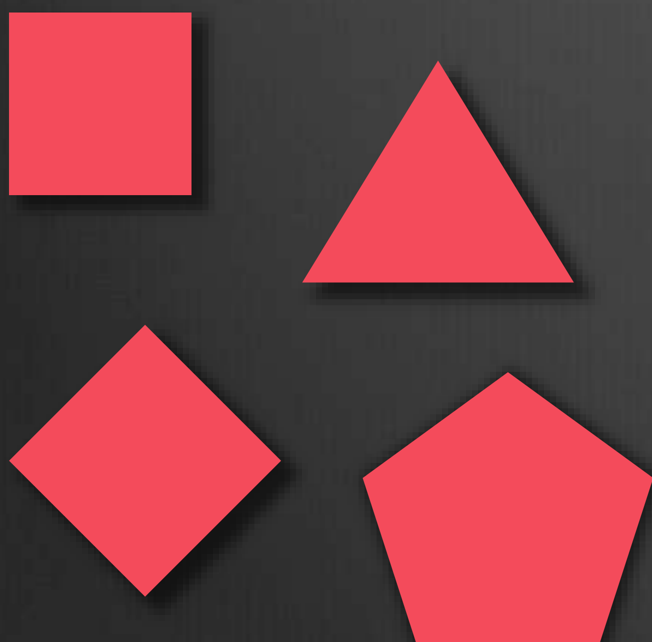
Isomorphism

These 4 Shapes ⇄ **Wiggles**

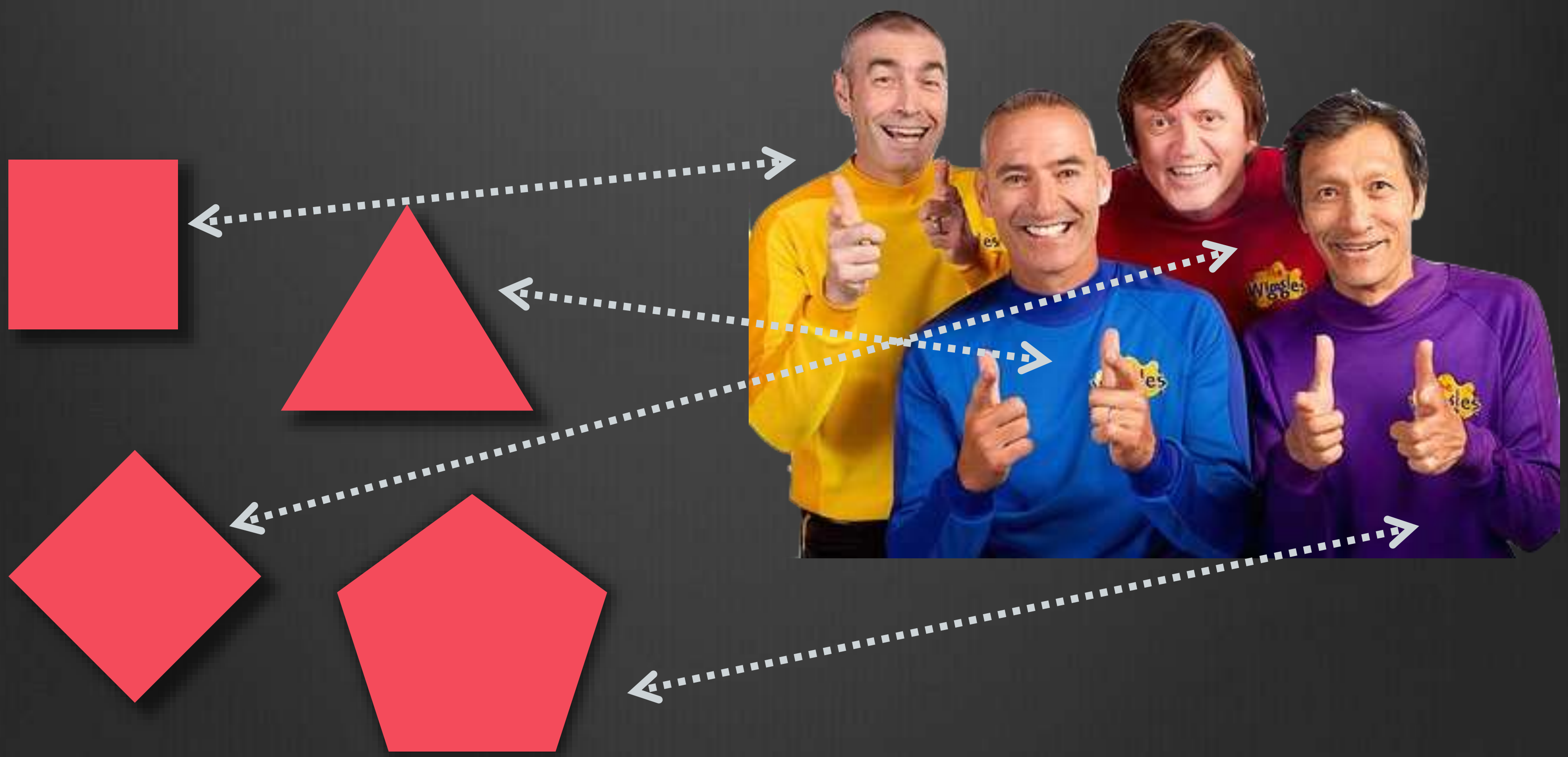
Set

functions

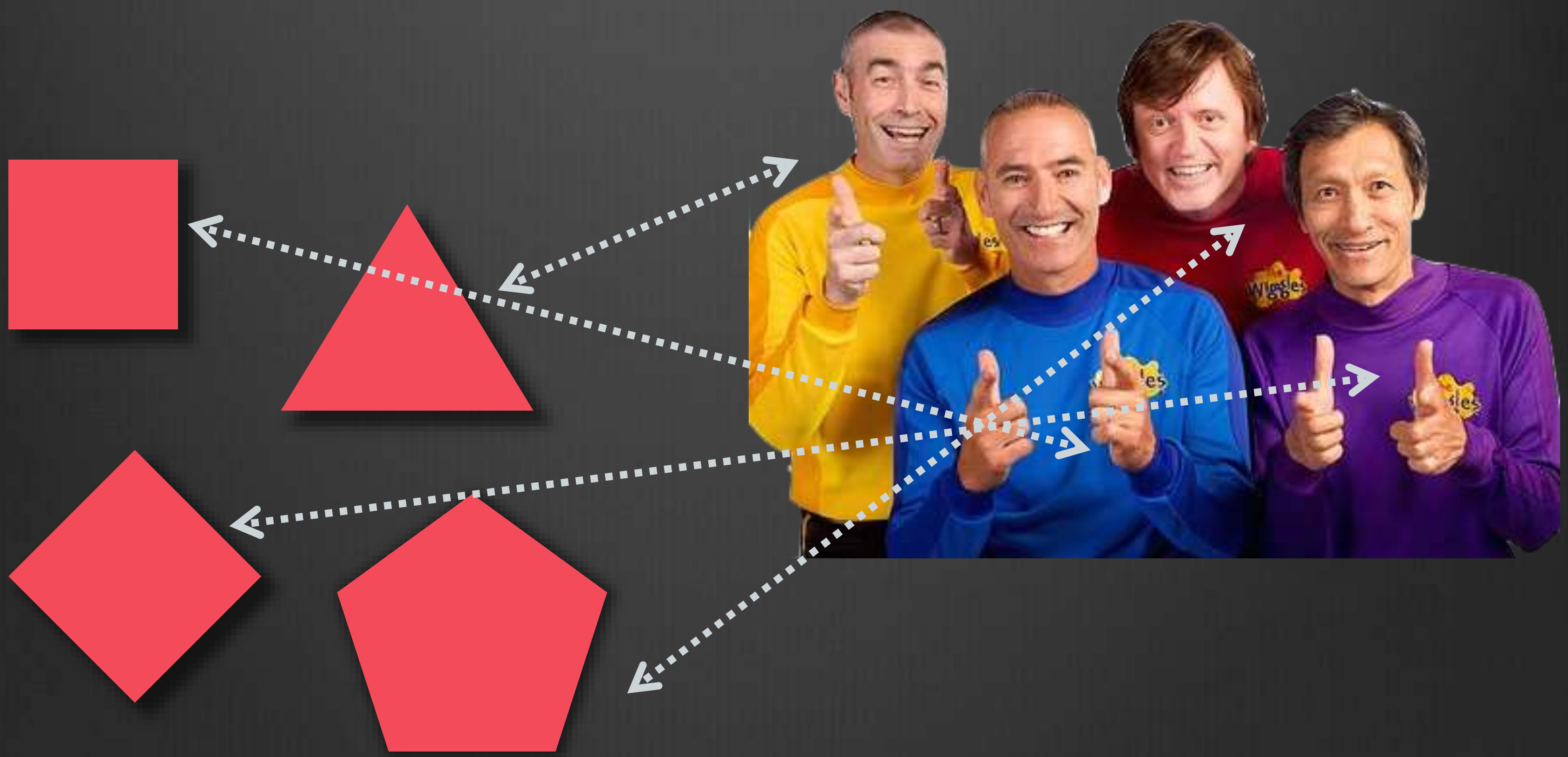
Set



These 4 Shapes ⇄ Wiggles



These 4 Shapes ⇄ **Wiggles**



There can be lots of isos between two objects!

If there's at least one, we can say they are *isomorphic*

or $A \cong B$

Products

$$A \xleftarrow{\mathit{first}} A \times B \xrightarrow{\mathit{second}} B$$

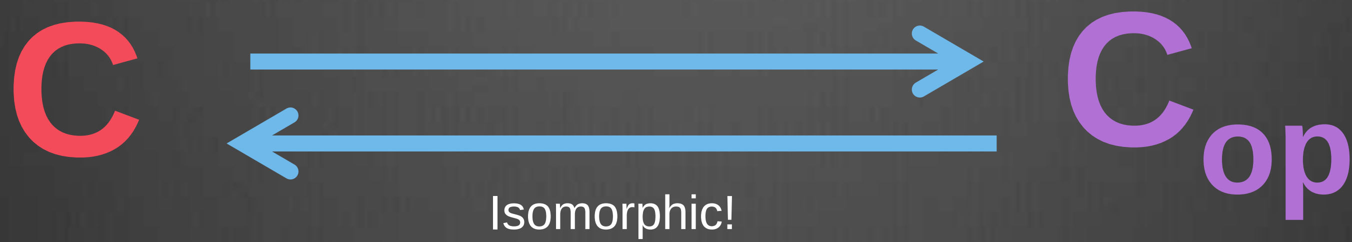
Given the product of A-and-B,
we can obtain both A and B

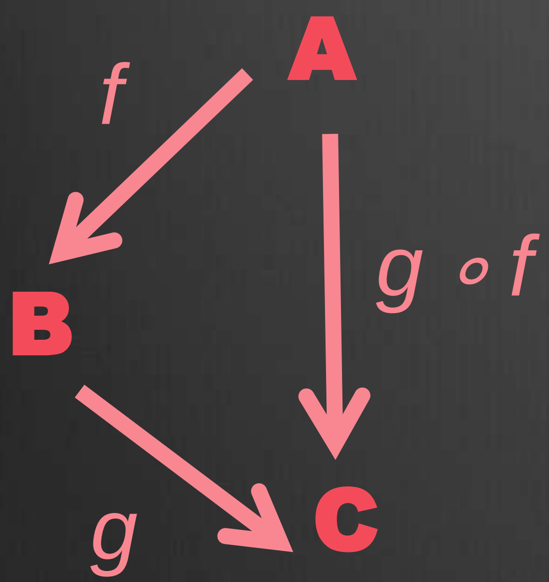
Sums

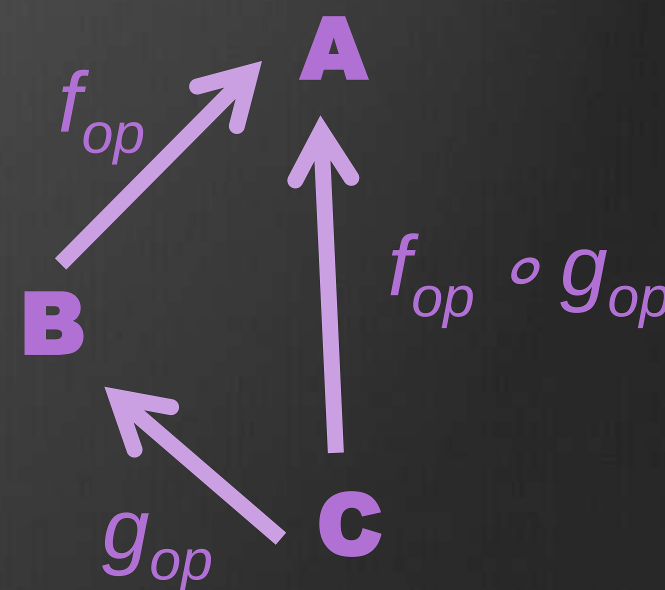
$$A \xrightarrow[\textit{left}]{} A + B \xleftarrow[\textit{right}]{} B$$

Given an A, or a B, we have the sum A-or-B

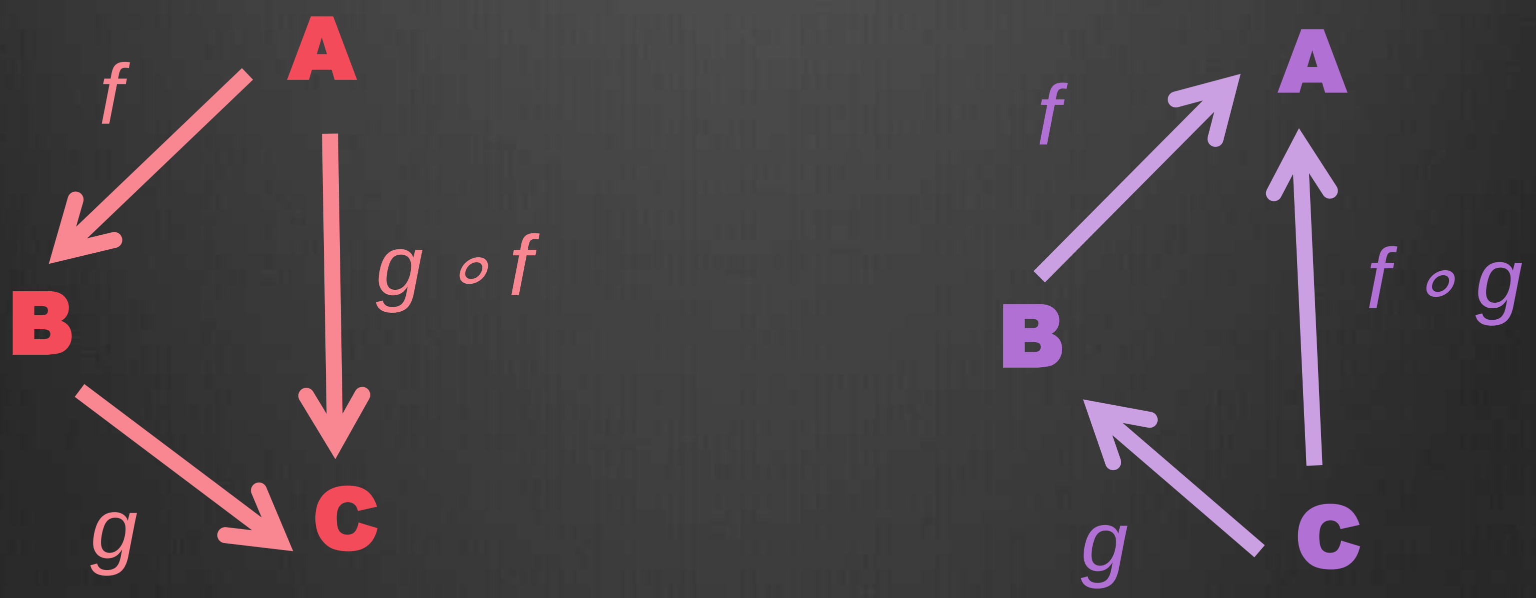
Opposite categories





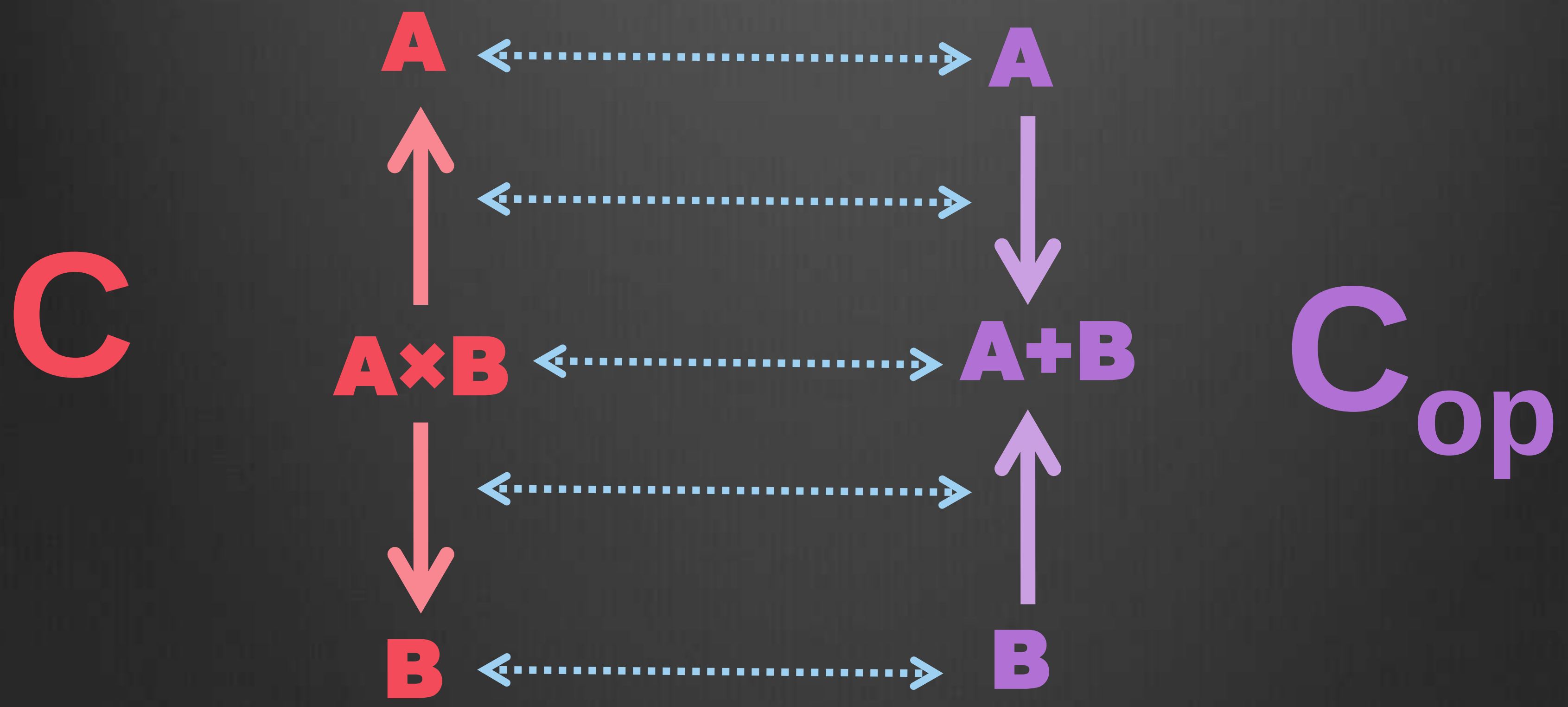


Just flip the arrows, and reverse composition!



A product in C is a sum in C_{op}

A sum in C is a product in C_{op}



Sums $\cong$ Products!

Terminology

An object and its equivalent in the opposite category are

dual

to each other.

Terminology

Often we call something's dual a

Co-(thing)

Terminology

Sums are also called

Coproducts

V. Composable systems

Growing a system

Banana

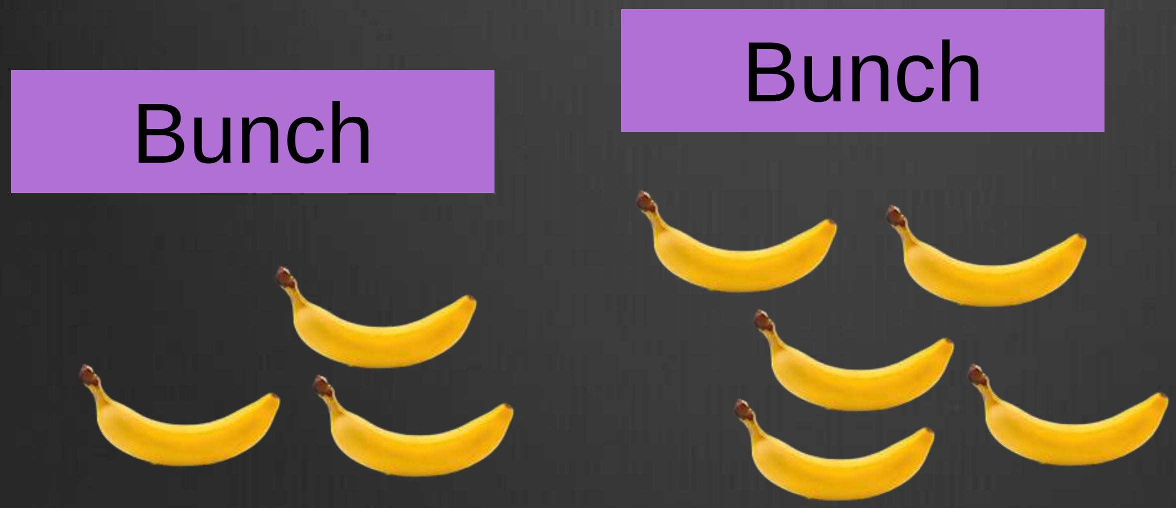
Growing a system

Growing a system

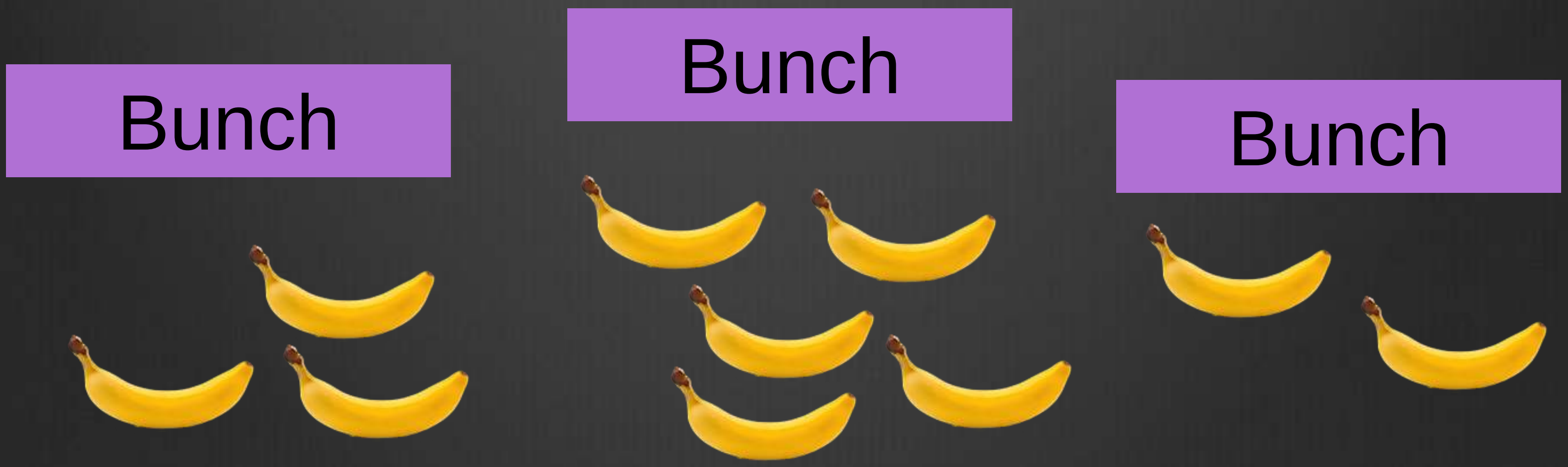
Growing a system

Bunch

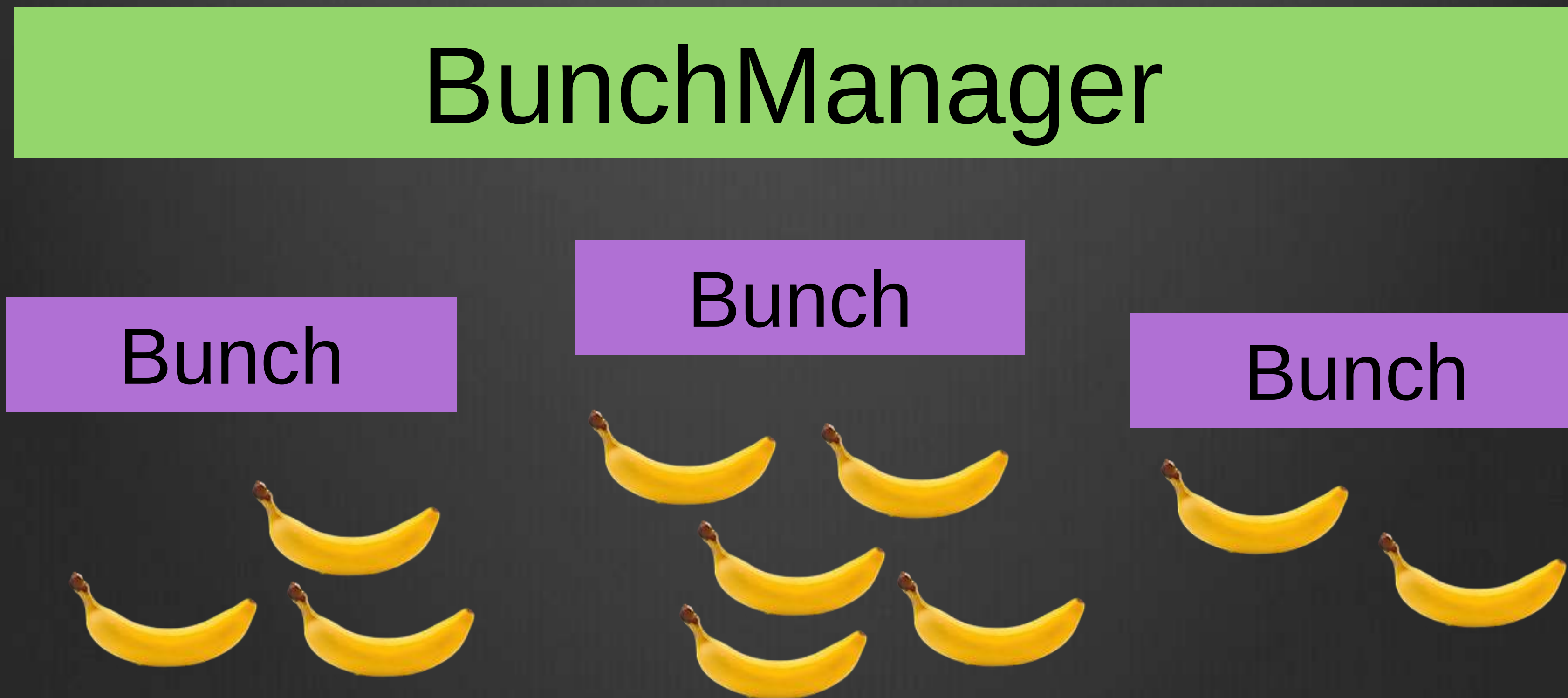
Growing a system



Growing a system



Growing a system



Growing a system

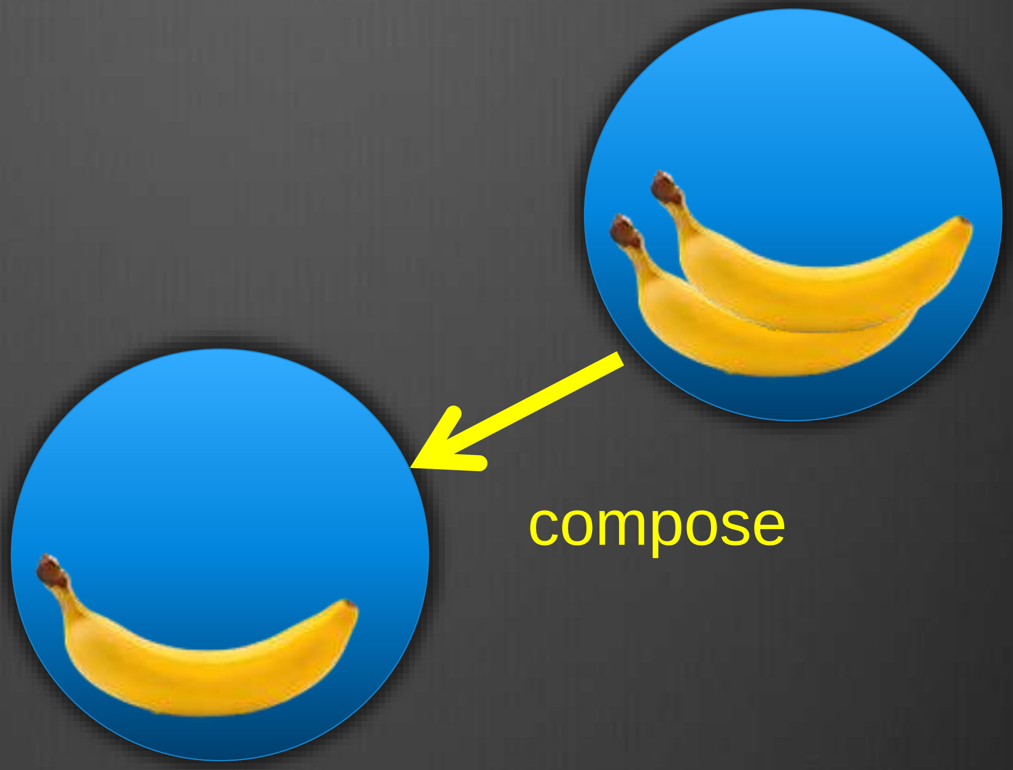
AnyManagers

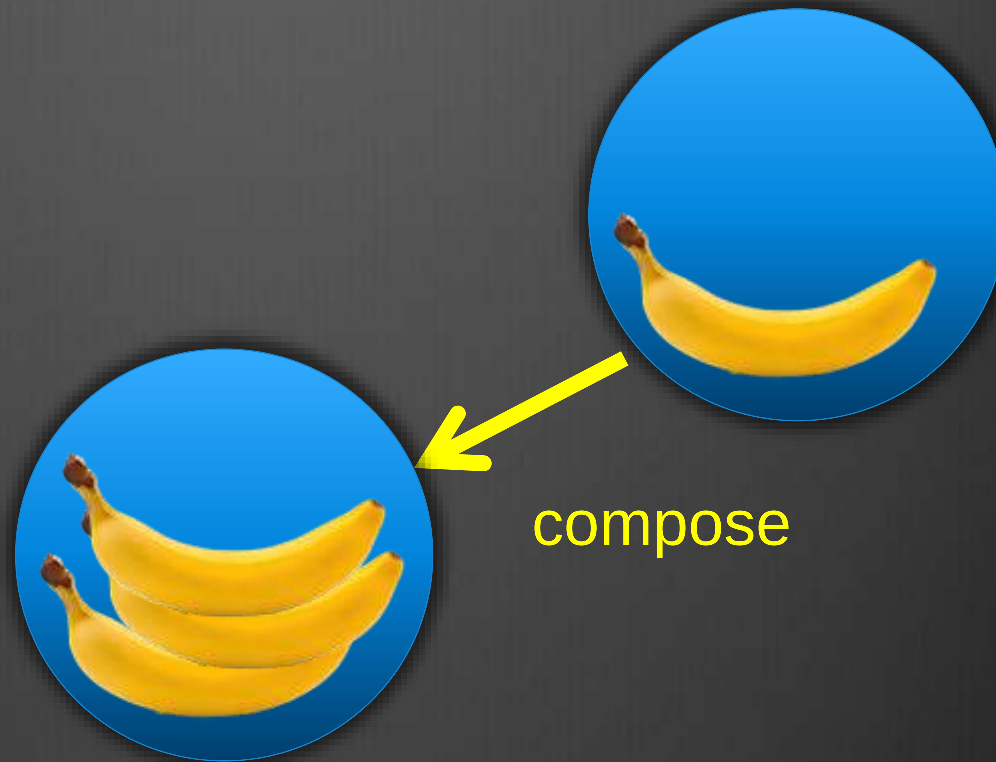
BunchManager

Bunch

Bunch

Bunch


compose


compose

etc…

Using composable abstractions means your code can grow without getting more complex

Categories and Monoids capture the essence of composition in software!

Look for Monoids and Categories in your domain where you can

You can even bludgeon non-composable things into *free monoids* and *free categories*

VI. Abstraction

Spanner

AbstractSpanner

Spanner

AbstractToolThing

AbstractSpanner

Spanner

GenerallyUsefulThing

AbstractToolThing

AbstractSpanner

Spanner

AbstractGenerallyUsefulThingFactory

GenerallyUsefulThing

AbstractToolThing

AbstractSpanner

Spanner

WhateverFactoryBuilder

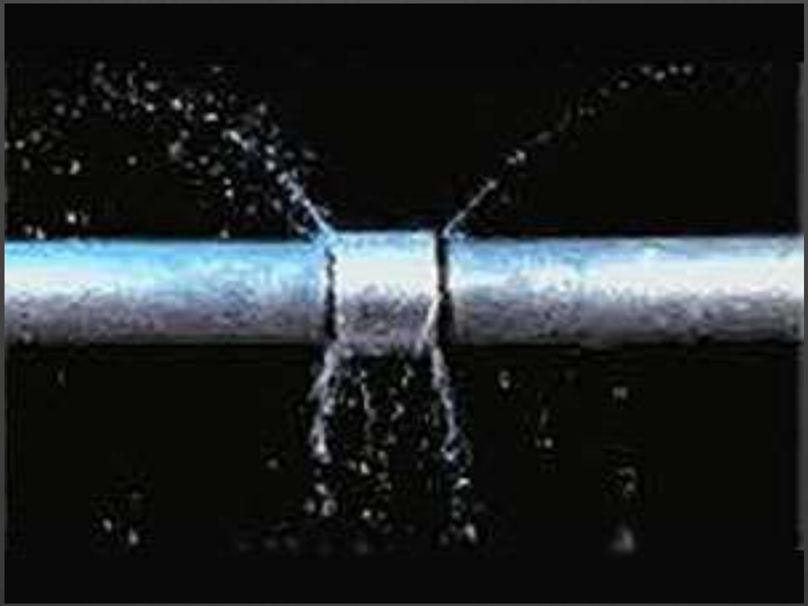
AbstractGenerallyUsefulThingFactory

GenerallyUsefulThing

AbstractToolThing

AbstractSpanner

Spanner



That’s not what abstraction means.

Code shouldn’t know things that aren’t needed.

```
def getNames(users: List[User]):
                                List[Name] = {
  users.map(_.name)
}
```

```scala
def getNames(users: List[User]):
                                List[Name] = {
  println(users.length)
  users.map(_.name)
}
```

Over time…

```
def getNames(users: List[User]):
                    List[Name] = {
  println(users.length)
  if (users.length == 1) {
    s”${users.head.name} the one and only"

  } else {
    users.map(_.name)
  }
}
```

“Oh, now we need the roster of names! A simple list won’t do.”

```
def getRosterNames(users: Roster[User]):
                          Roster[Name] = {
  users.map(_.name)
}
```

```
def getRosterNames(users: Roster[User]):
                          Roster[Name] = {
  LogFactory.getLogger.info(s"When you party with ${users.rosterTitle}, you must party hard!")
  users.map(_.name)
}
```

Over time…

```
def getRosterNames(users: Roster[User]):
                          Roster[Name] = {
  LogFactory.getLogger.info(s"When you
party with ${users.rosterTitle}, you must
party hard!")
  if (users.isFull) EmptyRoster("(everyone)
")
  else users.map(_.name)
}
```

When code knows too much, soon new things will appear that actually require the other stuff.

Coupling has increased. The mixed concerns will tangle and snarl.

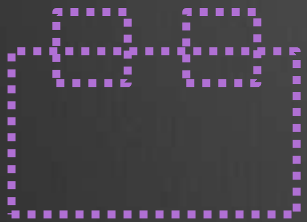
Code is rewritten each time for trivially different requirements

```
def getNames[F: Functor](users: F[User]):
                                F[Name] = {
  Functor[F].map(users)(_.name)
}
```

```
getNames(List(alice, bob, carol))

getNames(Roster(alice, bob, carol))
```

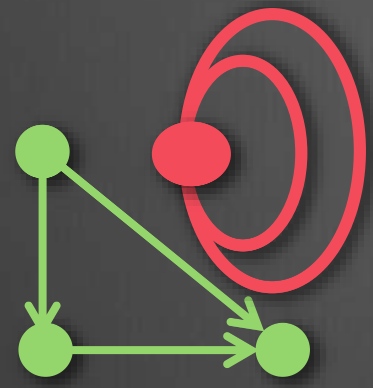
Reusable out of the box!

Not only is the abstract code not weighed down with useless junk, it *can’t* be!



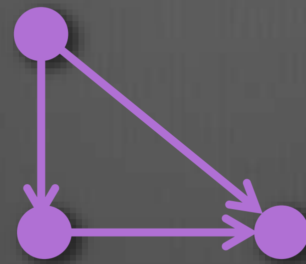
Abstraction is about hiding unnecessary information. This a good thing.

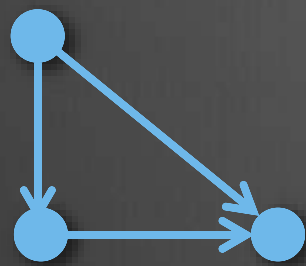
We actually know *more* about what the code does, because we have stronger guarantees!

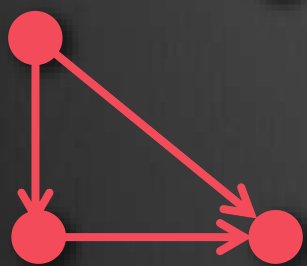


We’ve seen deep underlying patterns beneath superficially different things

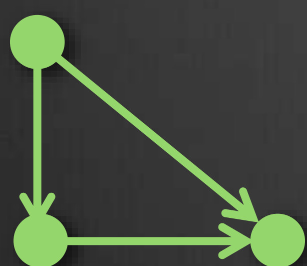
$A \times B \leftrightarrow A+B$







Just about everything ended up being in a category, or being one.



There is no better way to understand the patterns underlying software than studying Category Theory.

Further reading

- Awodey, “Category Theory”
- Lawvere & Schanuel, “Conceptual Mathematics: an introduction to categories”
- Jeremy Kun, “Math ∩ Programming” **at** http://jeremykun.com/
- Gabriel Gonzalez “Haskell for all”
  - http://www.haskellforall.com/2012/08/the-category-design-pattern.html
  - http://www.haskellforall.com/2014/04/scalable-program-architectures.html