

# Category Theory & Programming

*for Rivieria Scala Clojure (Note this presentation uses Haskell)*

*by Yann Esposito*

@yogsototh, +yogsototh

ENTER  
FULLSCREEN

HTML presentation: use arrows, space, swipe to navigate.

# Plan

---

- General overview
- Definitions
- Applications

# Not really about: Cat & glory

---



credit to Tokuhiro Kawai (川井徳寛)

# General Overview

## *Recent Math Field*

1942-45, Samuel Eilenberg & Saunders Mac Lane

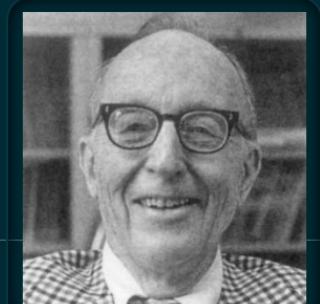
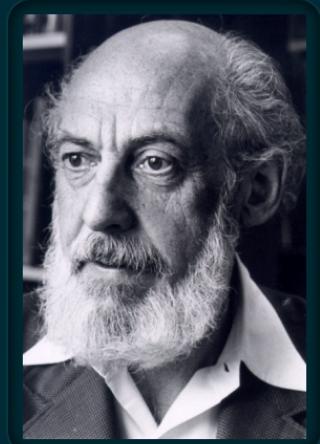
Certainly one of the more abstract branches of math

- *New math foundation*

- formalism abstraction, package entire theory★

- *Bridge between disciplines*

- Physics, Quantum Physics, Topology, Logic, Computer Science★



★: When is one thing equal to some other thing?, Barry Mazur, 2007

★: Physics, Topology, Logic and Computation: A Rosetta Stone, John C. Baez, Mike Stay, 2009

# From a Programmer perspective

---

| *Category Theory is a new language/framework for Math*

- Another way of thinking
- Extremely efficient for generalization

# Math Programming relation

---

Programming *is* doing Math

Strong relations between type theory and category theory.

Not convinced?

Certainly a *vocabulary* problem.

One of the goal of Category Theory is to create a *homogeneous vocabulary* between different disciplines.



# Vocabulary

---

Math vocabulary used in this presentation:

*Category, Morphism, Associativity, Preorder, Functor,  
Endofunctor, Categorial property, Commutative  
diagram, Isomorph, Initial, Dual, Monoid, Natural  
transformation, Monad, Klesli arrows, κata-morphism,*

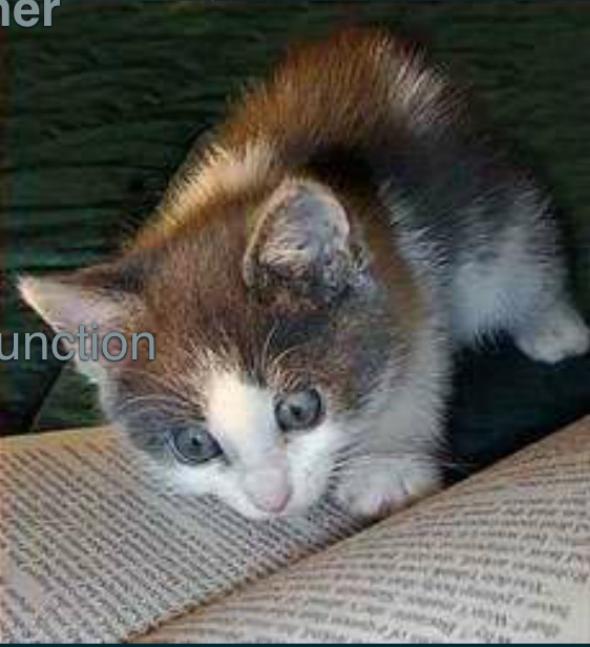
...



1/16

# Programmer Translation

Mathematician	Programmer
Morphism	Arrow
Monoid	String-like
Preorder	Acyclic graph
Isomorph	The same
Natural transformation	rearrangement function
Funny Category	LOLCat



# Plan

---

- General overview

- Definitions

- Applications

- Category

- Intuition

- Examples

- Functor

- Examples

# Category

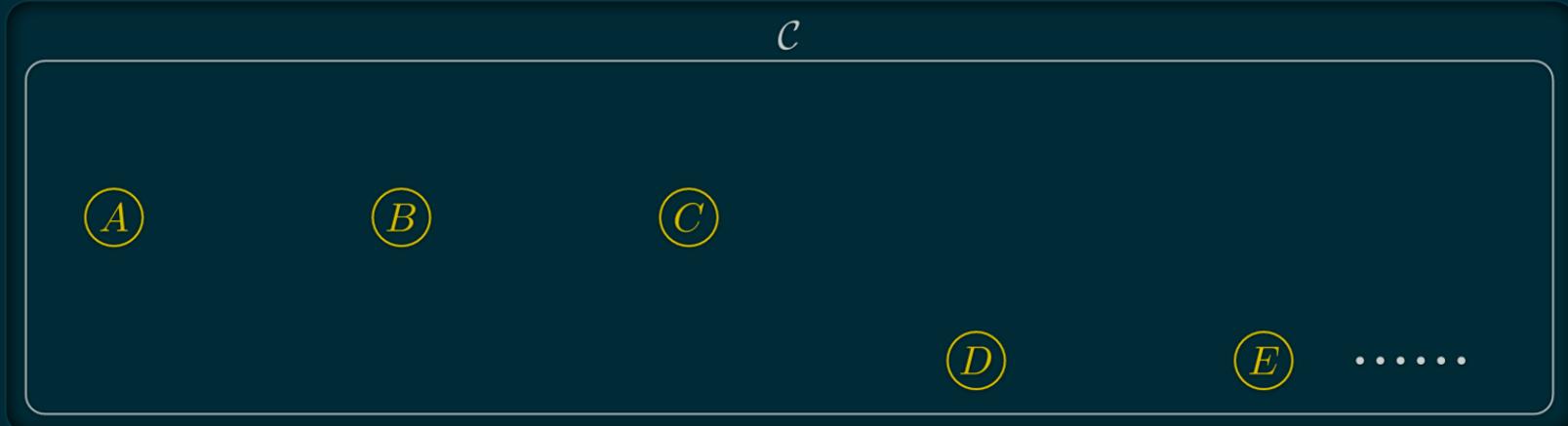
---

A way of representing *things* and *ways to go between things*.

A Category  $\mathcal{C}$  is defined by:

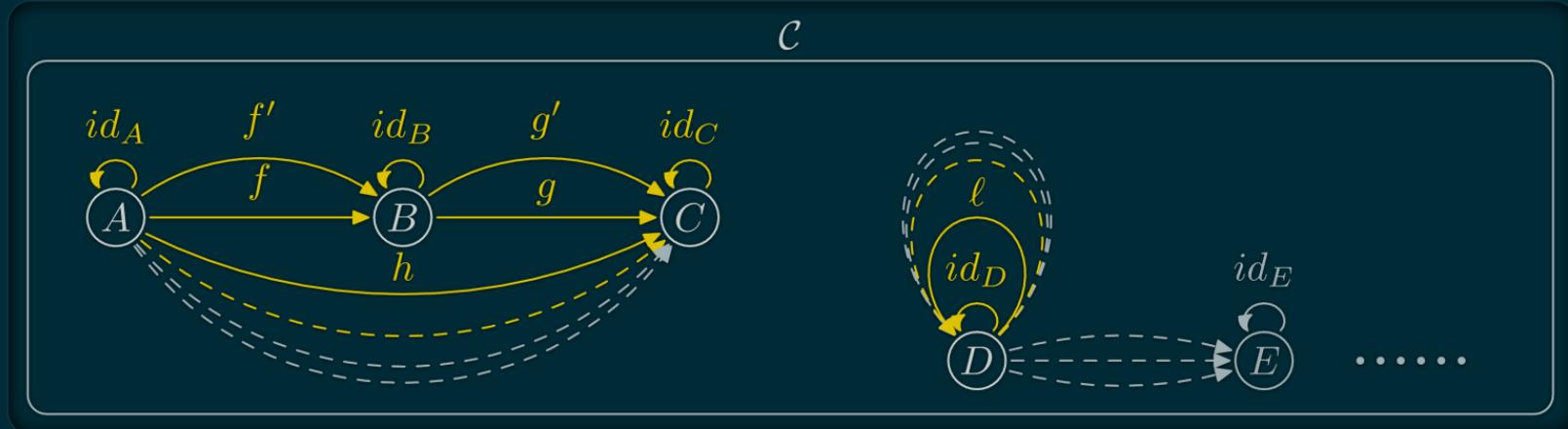
- *Objects*  $\text{ob}(\mathcal{C})$ ,
- *Morphisms*  $\text{hom}(\mathcal{C})$ ,
- a *Composition law* ( $\circ$ )
- obeying some *Properties*.

# Category: Objects



$\text{\textbackslash ob\{\text{\textit{mathcal{C}}}\}}$  is a collection

# Category: Morphisms



$(A)$  and  $(B)$  objects of  $(C)$

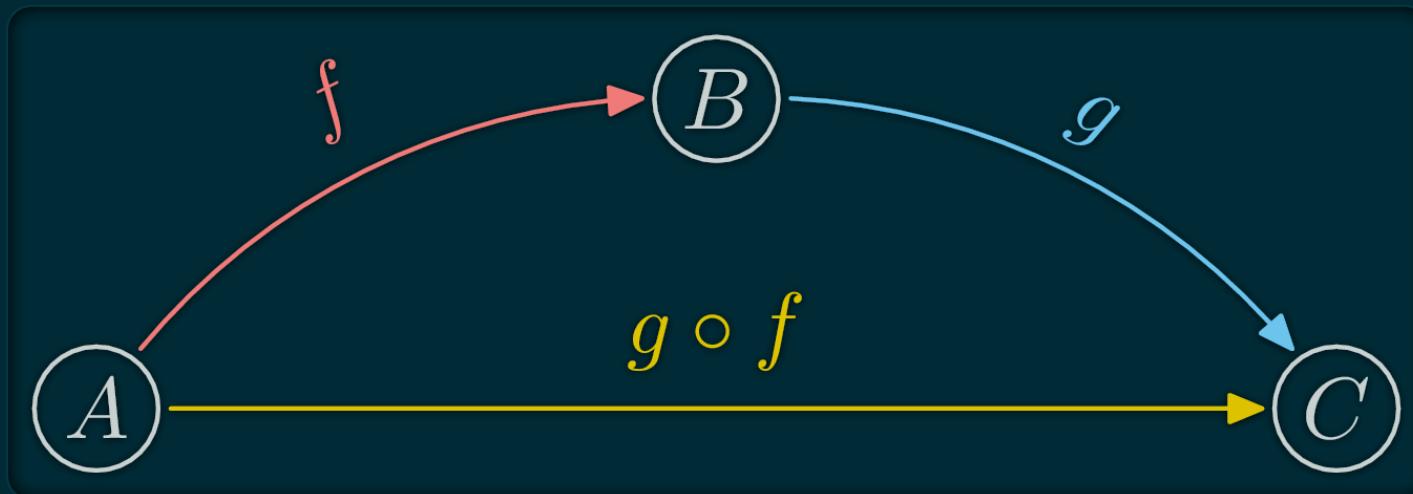
$(\hom{A,B})$  is a collection of morphisms

$(f:A \rightarrow B)$  denote the fact  $(f)$  belongs to  $(\hom{A,B})$

$(\hom{C})$  the collection of all morphisms of  $(C)$

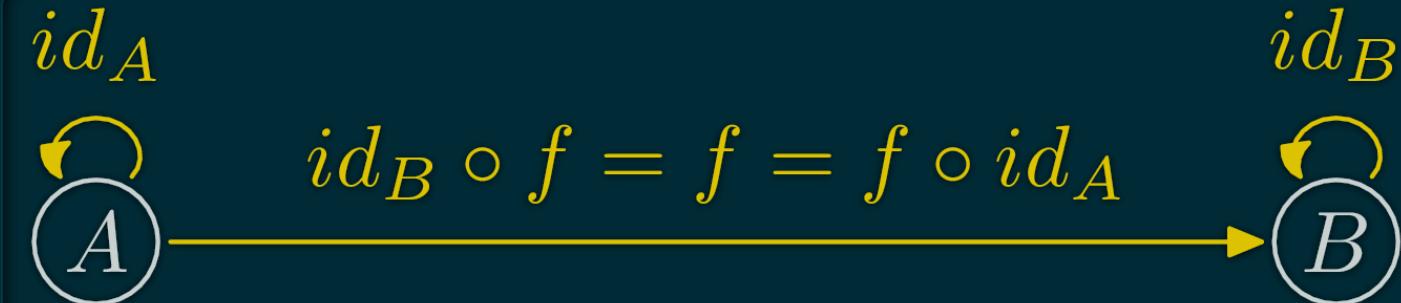
# Category: Composition

Composition ( $\circ$ ): associate to each couple  $(f:A \rightarrow B, g:B \rightarrow C)$   $g \circ f:A \rightarrow C$



# Category laws: neutral element

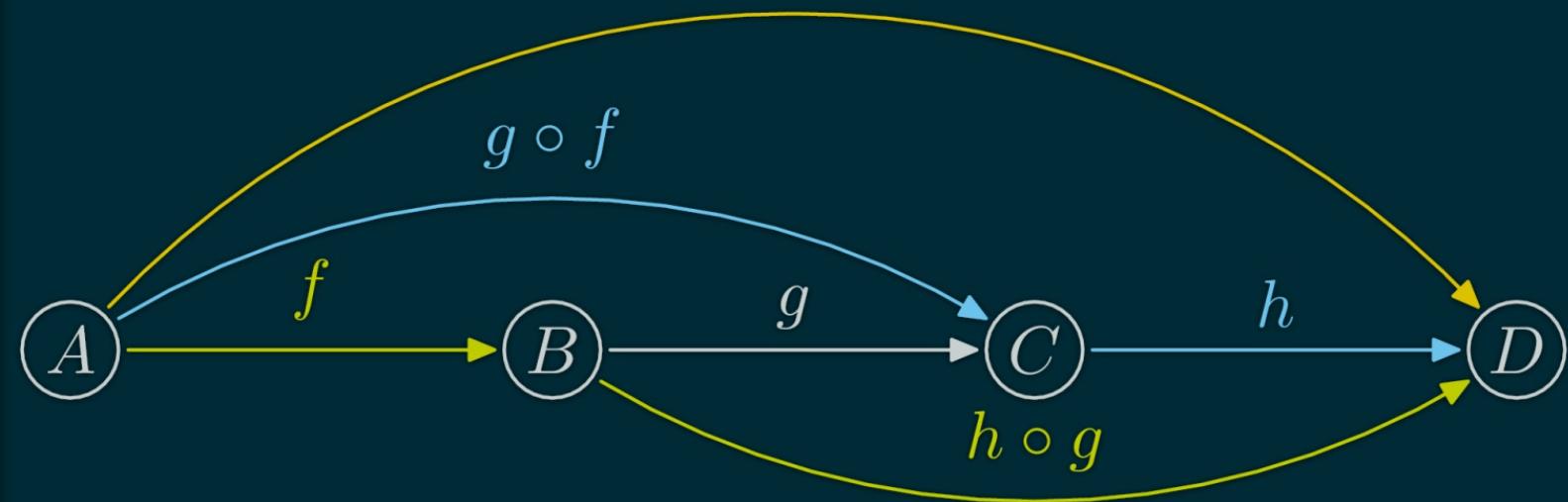
for each object  $\langle X \rangle$ , there is an  $\langle \text{id}_X : X \rightarrow X \rangle$ ,  
such that for each  $\langle f : A \rightarrow B \rangle$ :



# Category laws: Associativity

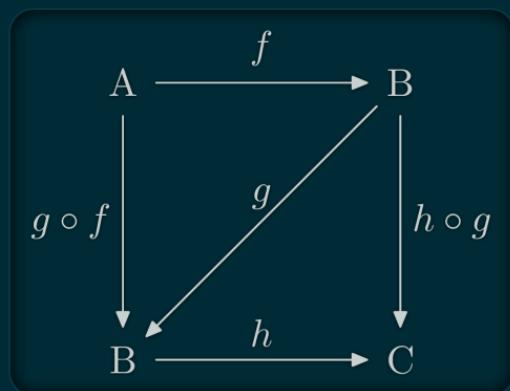
Composition is associative:

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

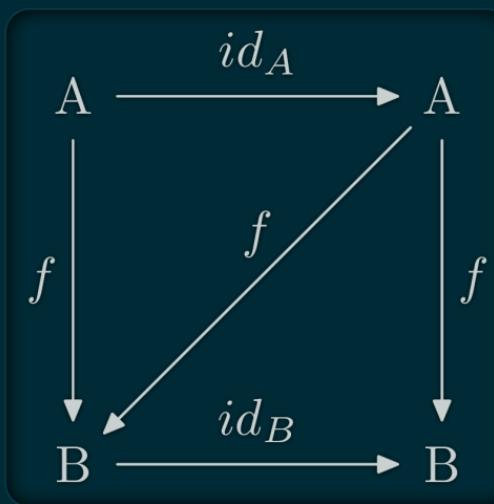


# Commutative diagrams

Two path with the same source and destination are equal.



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



$$\backslash(id_B \circ f = f = f \circ id_A \backslash)$$

# Question Time!

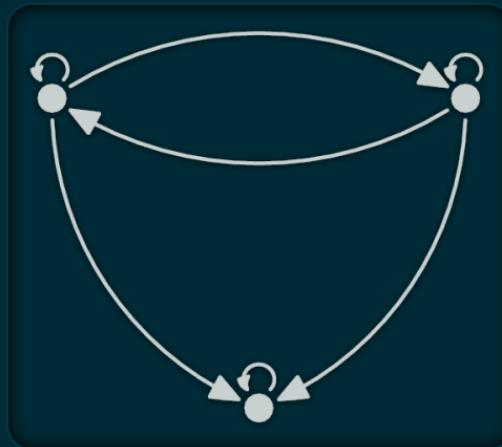
---



- French-only joke -

# Can this be a category?

$(\text{ob}(C), \text{hom}(C))$  fixed, is there a valid  $\circ$ ?

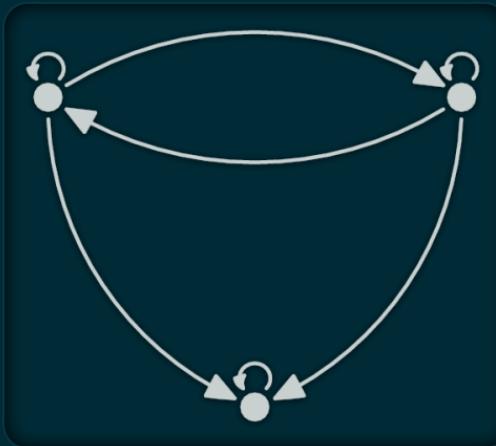


# Can this be a category?

$(\text{ob}(C), \text{hom}(C))$  fixed, is there a valid  $\circ$ ?



YES



# Can this be a category?

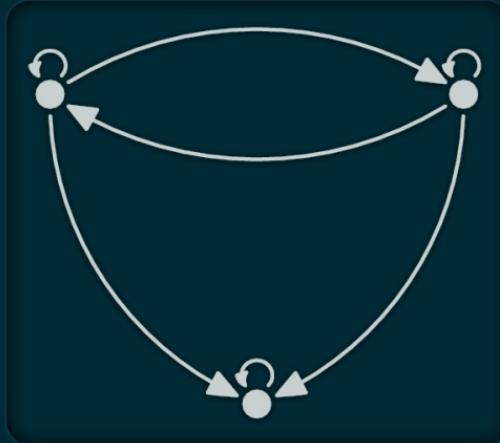
$(\text{ob}(C), \text{hom}(C))$  fixed, is there a valid  $\circ$ ?



YES



no candidate for  $(g \circ f)$   
NO



# Can this be a category?

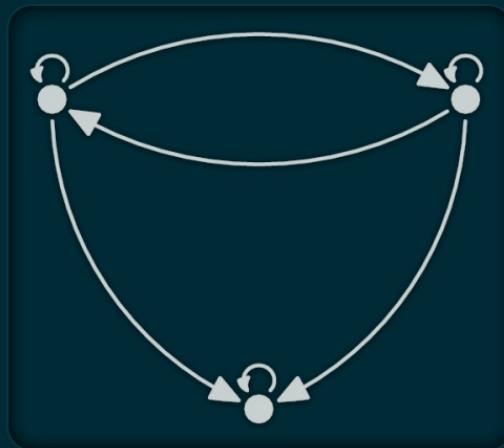
$(\text{ob}(C), \text{hom}(C))$  fixed, is there a valid  $\circ$ ?



YES

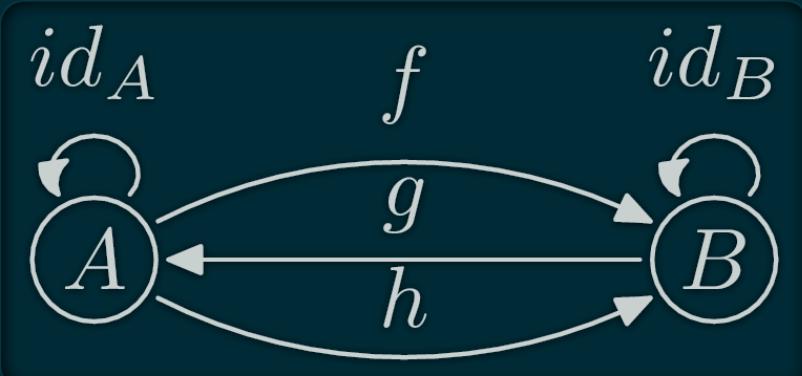
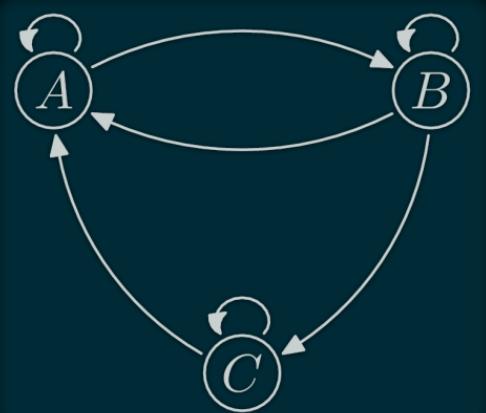


no candidate for  $(g \circ f)$   
NO

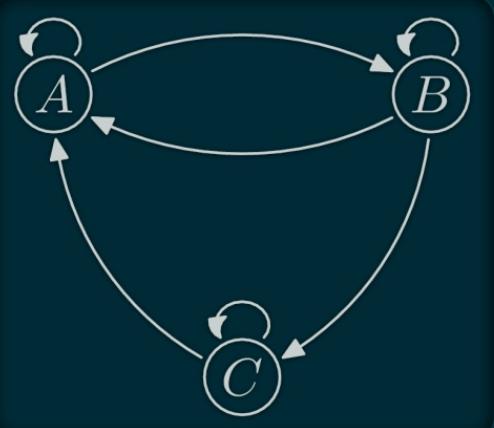


YES

# Can this be a category?

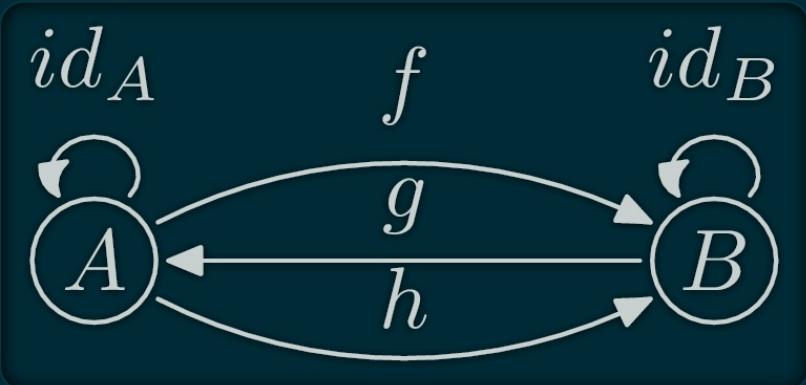


# Can this be a category?

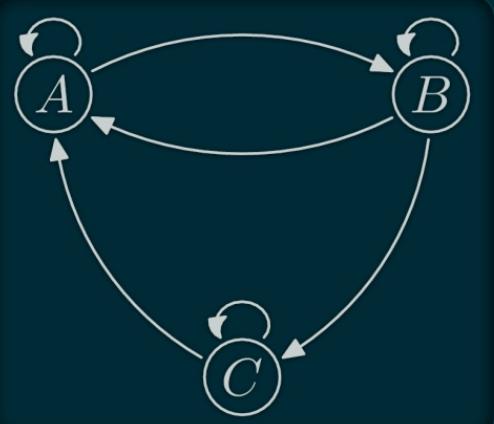


no candidate for  $\backslash(f:C \rightarrow B\backslash)$

NO

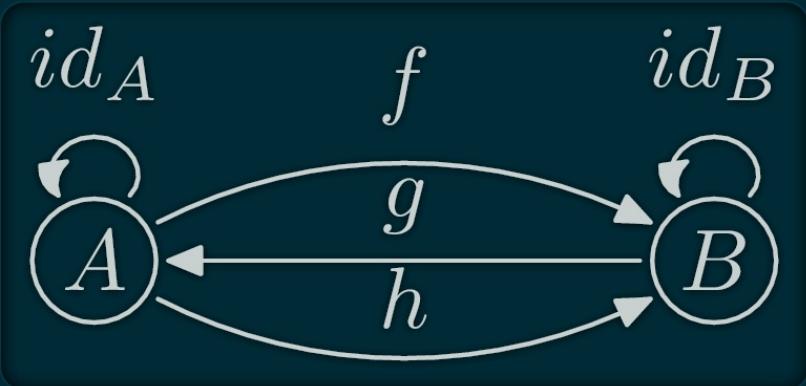


# Can this be a category?



no candidate for  $(f:C \rightarrow B)$

NO



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

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

but  $(h \neq f)$

NO

# Categories Examples

---



*- Basket of Cats -*

# Category $\mathbf{\mathbb{Set}}$

---

- $\mathbf{ob}\{\mathbf{\mathbb{Set}}\}$  are *all* the sets
- $\mathbf{hom}\{E, F\}$  are *all* functions from  $(E)$  to  $(F)$
- $\circ$  is functions composition

# Category $\mathbf{\mathbb{Set}}$

---

- $\mathbf{ob}(\mathbf{\mathbb{Set}})$  are *all* the sets
- $\mathbf{hom}(E, F)$  are *all* functions from  $(E)$  to  $(F)$
- $\circ$  is functions composition
- $\mathbf{ob}(\mathbf{\mathbb{Set}})$  is a proper class ; not a set
- $\mathbf{hom}(E, F)$  is a set
- $\mathbf{\mathbb{Set}}$  is then a *locally small category*

# Categories Everywhere?

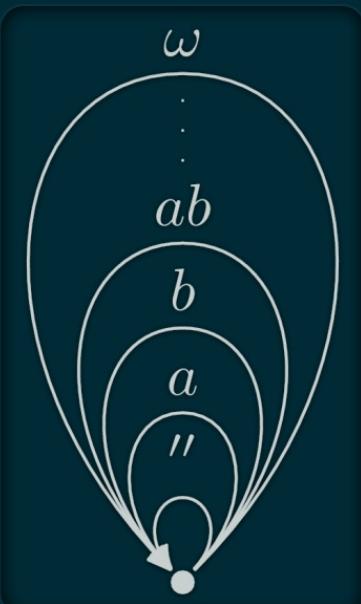
- $\backslash(\text{Mon}\backslash)$ : (monoids, monoid morphisms,  $\circ$ )
- $\backslash(\text{Vec}\backslash)$ : (Vectorial spaces, linear functions,  $\circ$ )
- $\backslash(\text{Grp}\backslash)$ : (groups, group morphisms,  $\circ$ )
- $\backslash(\text{Rng}\backslash)$ : (rings, ring morphisms,  $\circ$ )
- Any deductive system  $T$ : (theorems, proofs, proof concatenation)
- $\backslash(\text{Hask}\backslash)$ : (Haskell types, functions,  $(.)$  )
- ...



# Smaller Examples

## Strings

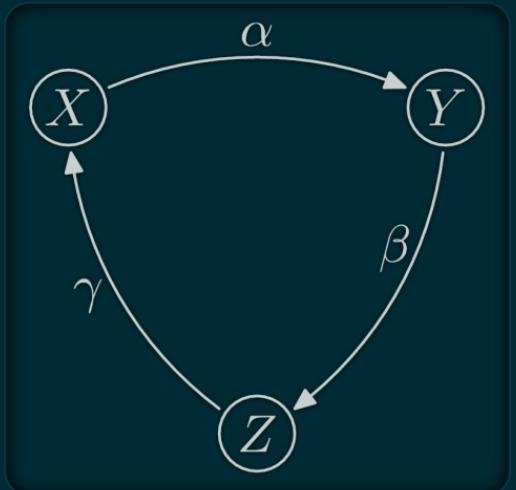
- $\text{\ob{Str}}$  is a singleton
- $\text{\hom{Str}}$  each string
- $\circ$  is concatenation  $(++)$
- $"" ++ u = u = u ++ ""$
- $(u ++ v) ++ w = u ++ (v ++ w)$



# Finite Example?

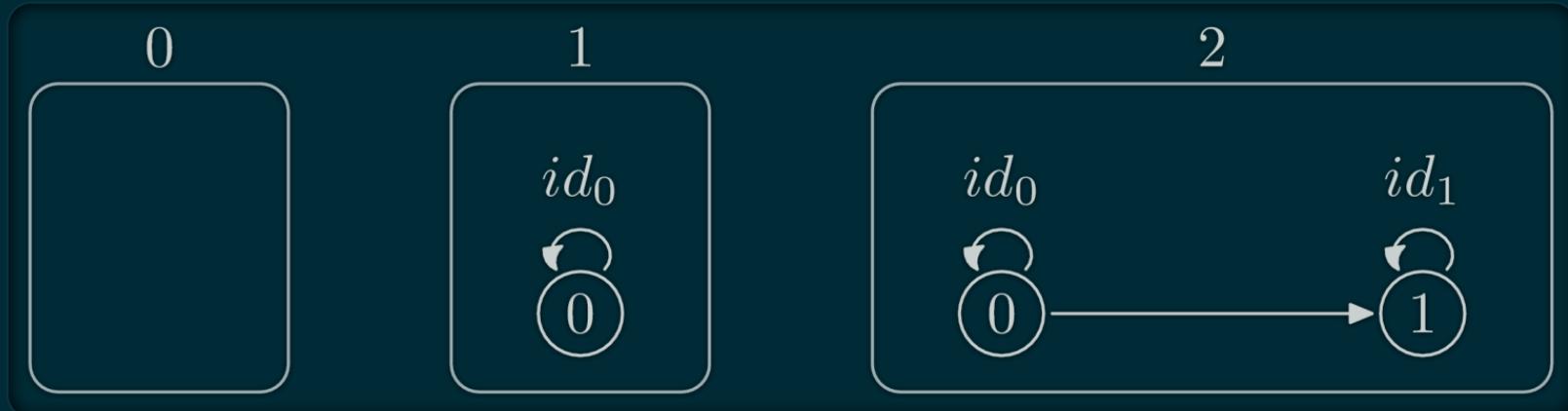
## Graph

- $\text{ob}(G)$  are vertices
- $\text{hom}(G)$  each path
- $\circ$  is path concatenation
- $\text{ob}(G) = \{X, Y, Z\}$ ,
- $\text{hom}(G) = \{\varepsilon, \alpha, \beta, \gamma, \alpha\beta, \beta\gamma, \dots\}$
- $(\alpha\beta \circ \gamma = \alpha\beta\gamma)$



# Number construction

Each Numbers as a whole category



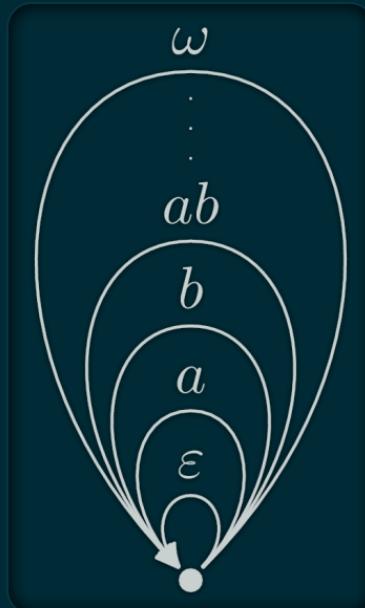
# Degenerated Categories: Monoids

Each Monoid  $((M, e, \odot) : \text{ob}(M) = \{\cdot\}, \text{hom}(M) = M, \circ = \odot)$

Only one object.

Examples:

- $(\text{Integer}, 0, +)$ ,  $(\text{Integer}, 1, *)$ ,
- $(\text{Strings}, "", ++)$ , for each  $a$ ,  $([a], [], ++)$



# Degenerated Categories: Preorders $\mathcal{P}(\mathbf{P}, \leq)$

- $\text{ob}(\mathbf{P}) = \{\mathbf{P}\}$ ,
- $\text{hom}(x, y) = \{x \leq y\} \Leftrightarrow x \leq y$ ,
- $(y \leq z) \circ (x \leq y) = (x \leq z)$

*At most one morphism between two objects.*

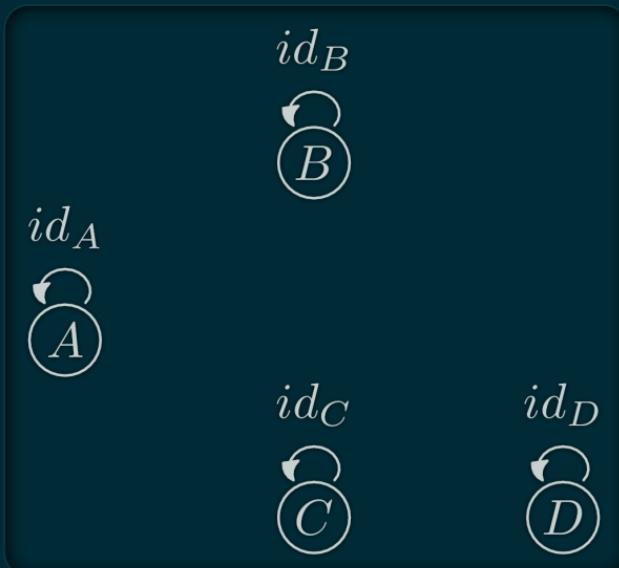


# Degenerated Categories: Discrete Categories

## Any Set

Any set  $\langle E : \text{ob}(E)=E, \text{hom}(x,y)=\{x\} \Leftrightarrow x=y \rangle$

Only identities



# Choice

---

The same object can be seen in many different way as a category.

You can choose what are object, morphisms and composition.

ex: **Str** and  $\text{discrete}(\Sigma^*)$

# Categorical Properties

---

Any property which can be expressed in term of category, objects, morphism and composition.

- **Dual**:  $(\mathcal{D})$  is  $(\mathcal{C})$  with reversed morphisms.
- **Initial**:  $(Z \in \text{ob}(\mathcal{C}))$  s.t.  $(\forall Y \in \text{ob}(\mathcal{C}), |\text{hom}(Z, Y)| = 1)$   
Unique ("up to isomorphism")
- **Terminal**:  $(T \in \text{ob}(\mathcal{C}))$  s.t.  $(T)$  is initial in the dual of  $(\mathcal{C})$
- **Functor**: structure preserving mapping between categories
- ...

# Isomorph

---

*isomorphism*:  $\backslash(f:A \rightarrow B\}$  which can be "undone" i.e.

$\backslash(\exists g:B \rightarrow A\}, \backslash(g \circ f = id_A\} \ \& \ \backslash(f \circ g = id_B\}$

in this case,  $\backslash(A\}$  &  $\backslash(B\}$  are *isomorphic*.

$A \cong B$  means A and B are essentially the same.

In Category Theory,  $=$  is in fact mostly  $\cong$ .

For example in commutative diagrams.



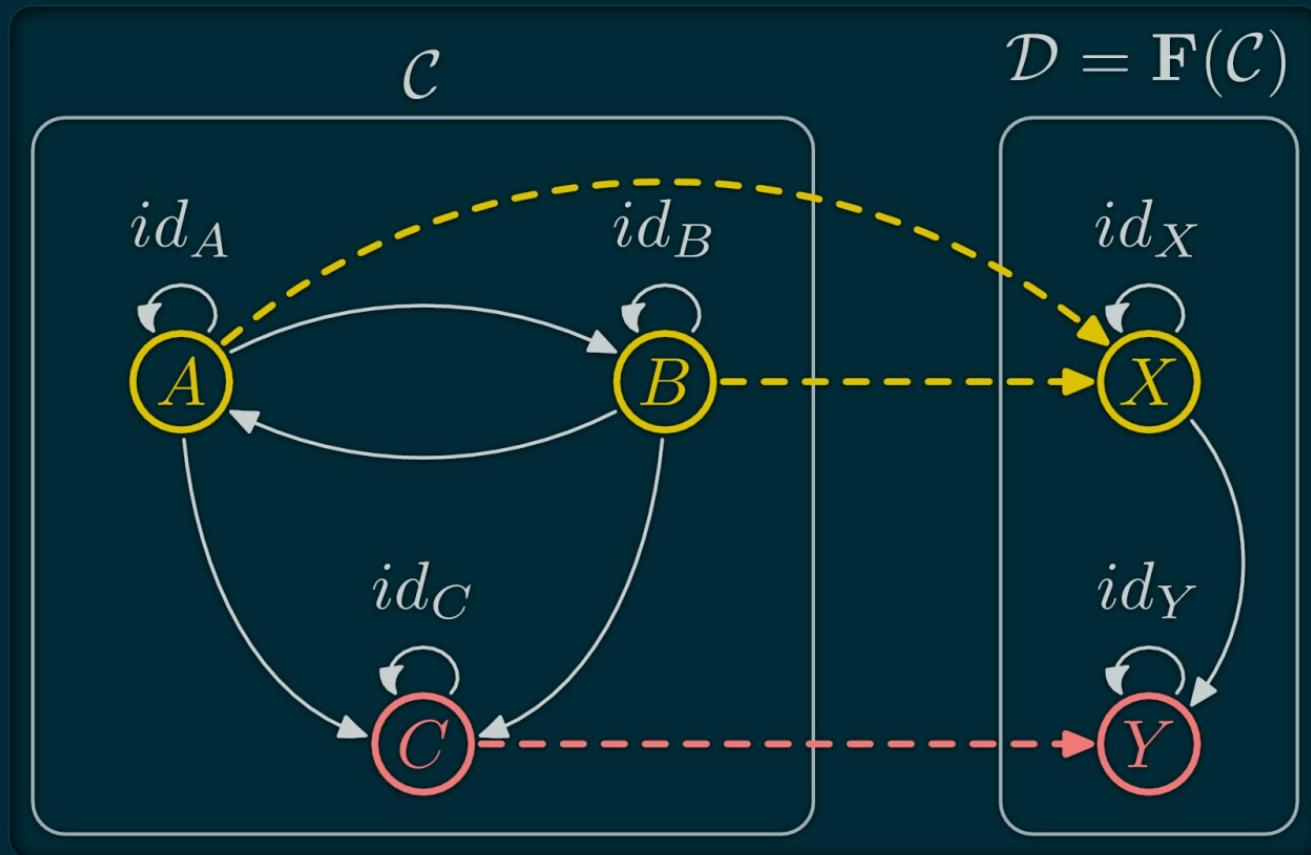
# Functor

---

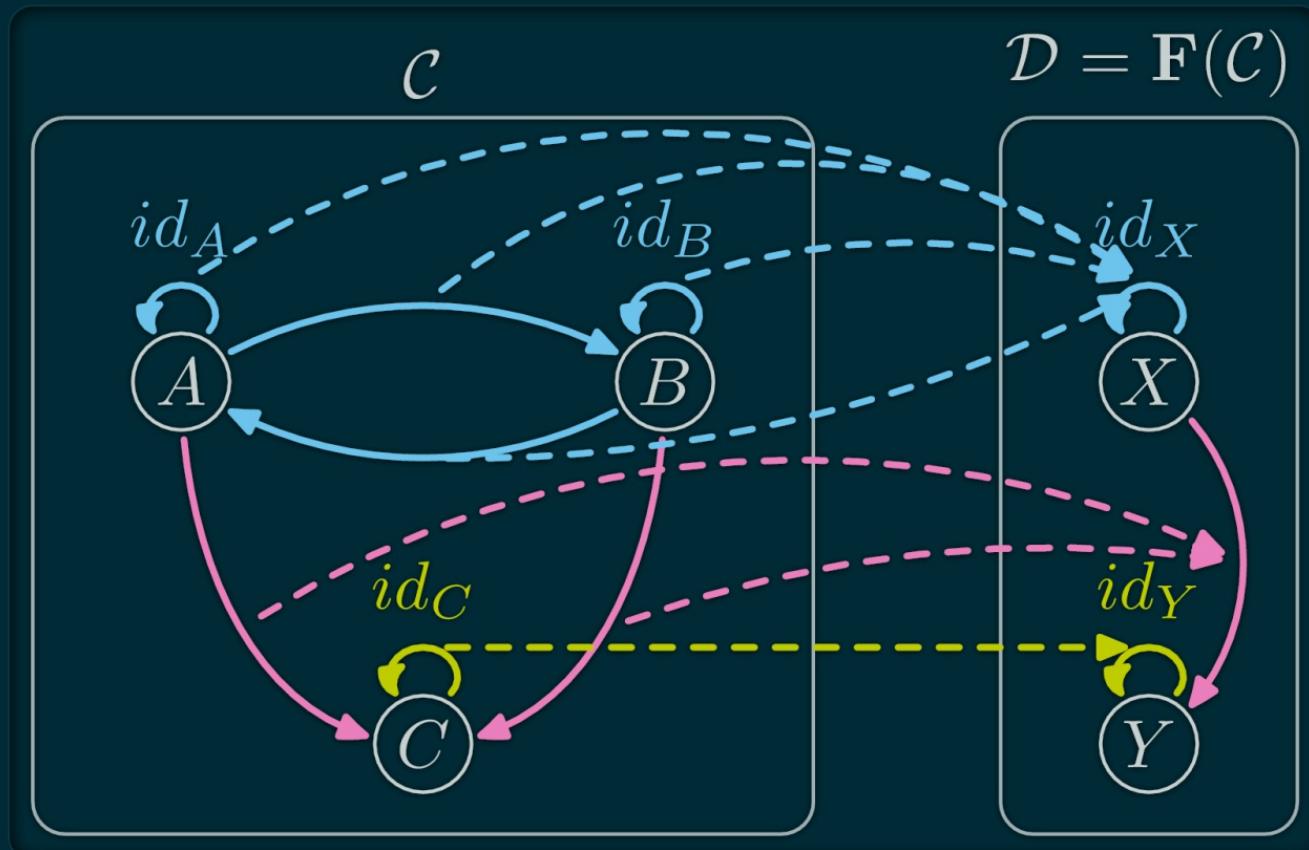
A functor is a mapping between two categories. Let  $\mathcal{C}$  and  $\mathcal{D}$  be two categories. A *functor*  $F$  from  $\mathcal{C}$  to  $\mathcal{D}$ :

- Associate objects:  $(A \in \text{ob}(\mathcal{C}))$  to  $(F(A) \in \text{ob}(\mathcal{D}))$
- Associate morphisms:  $(f:A \rightarrow B)$  to  $(F(f) : F(A) \rightarrow F(B))$  such that
  - $(F(\text{id}_X)) = (\text{id})$
  - $(F(g \circ f)) = (F(g) \circ F(f))$

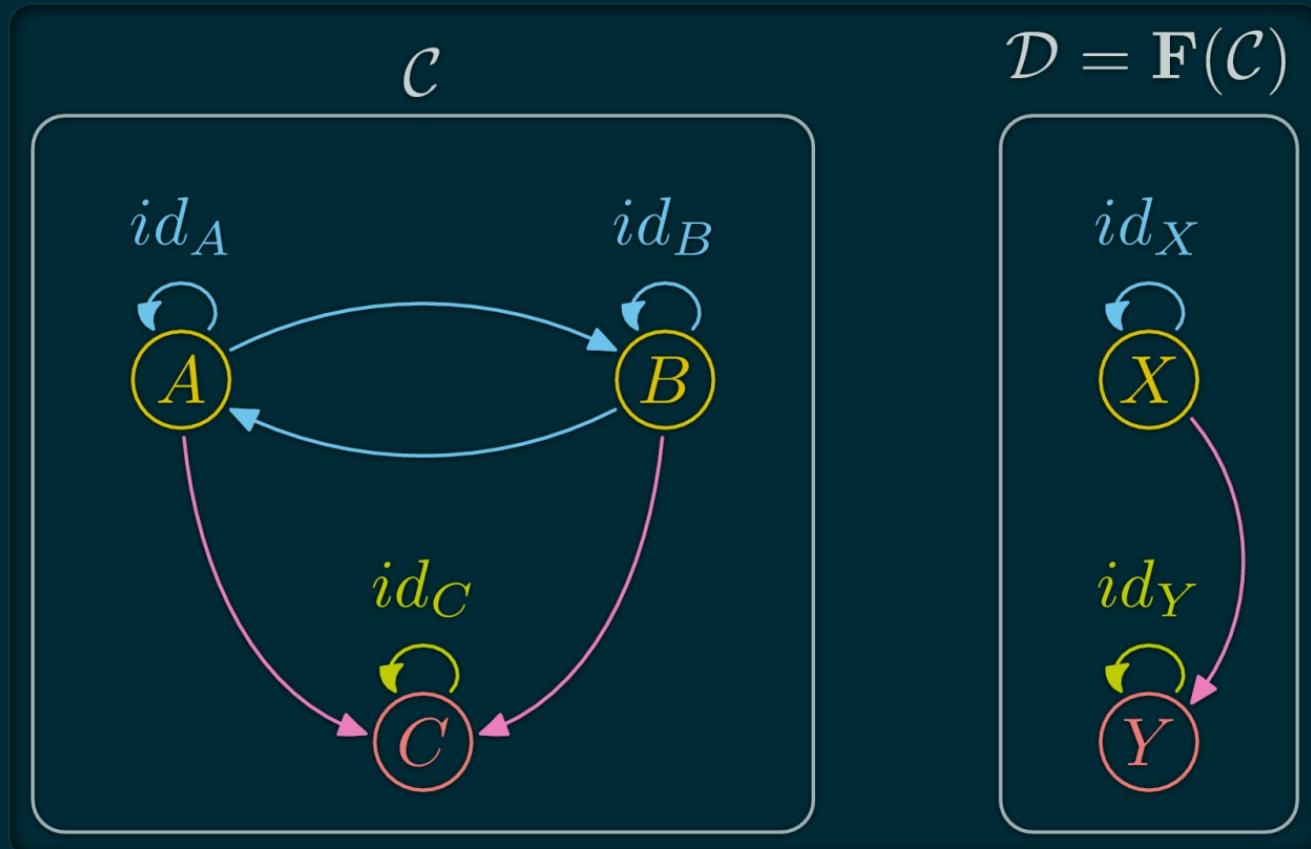
# Functor Example ( $\text{ob} \rightarrow \text{ob}$ )



# Functor Example ( $\text{hom} \rightarrow \text{hom}$ )

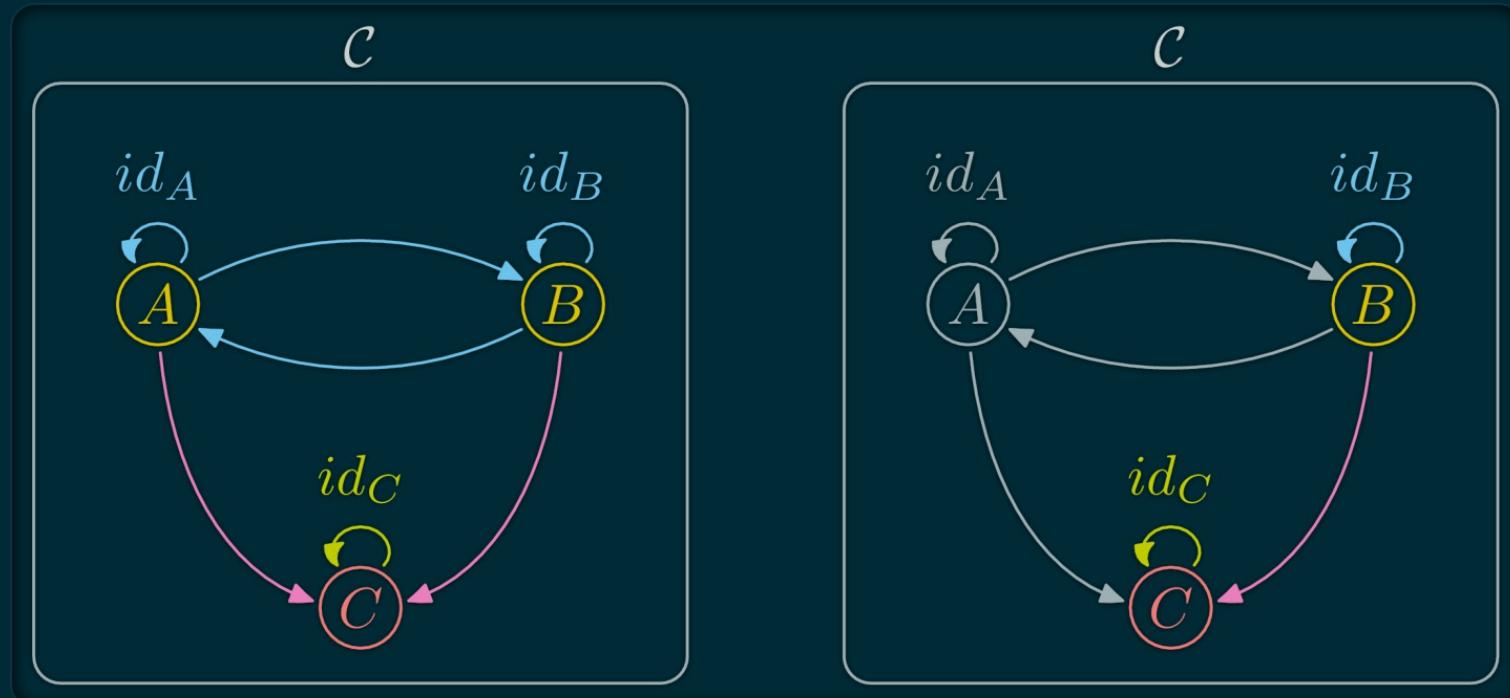


# Functor Example



# Endofunctors

An *endofunctor* for  $(\mathcal{C})$  is a functor  $(F:\mathcal{C} \rightarrow \mathcal{C})$ .



# Category of Categories

---

Categories and functors form a category:  $\mathbf{Cat}$

- $\mathbf{ob}(\mathbf{Cat})$  are categories
- $\mathbf{hom}(\mathbf{Cat})$  are functors
- $\circ$  is functor composition



# Plan

---

- General overview

- Definitions

- Applications

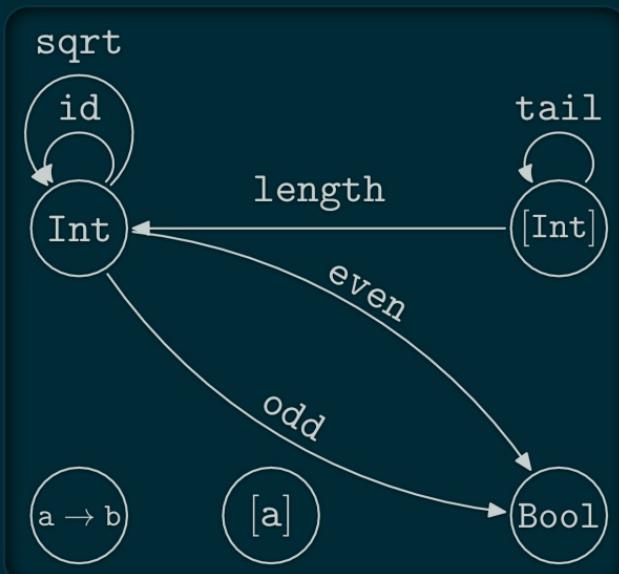
- $\text{\textbackslash}(\text{Hask})$  category
- Functors
- Natural transformations
- Monads
- κata-morphisms

# Hask

Category  $\mathcal{Hask}$ :

- $\mathcal{O}b(\mathcal{Hask}) = \mathcal{H}$  Haskell types
- $\mathcal{H}om(\mathcal{Hask}) = \mathcal{H}$  Haskell functions
- $\circ = (.)$  Haskell function composition

Forget glitches because of `undefined`.



# Haskell Kinds

---

In Haskell some types can take type variable(s). Typically: `[a]`.

Types have *kinds*; The kind is to type what type is to function. Kind are the types for types (so meta).

```
Int, Char :: *
[], Maybe :: * -> *
(,), (->) :: * -> * -> *
[Int], Maybe Char, Maybe [Int] :: *
```

# Haskell Types

Sometimes, the type determine a lot about the function★:

`fst :: (a,b) -> a` -- *Only one choice*

`snd :: (a,b) -> b` -- *Only one choice*

`f :: a -> [a]` -- *Many choices*

-- *Possibilities: f x=[], or [x], or [x,x] or [x,...,x]*

`? :: [a] -> [a]` -- *Many choices*

-- *can only rearrange: duplicate/remove/reorder elements*

-- *for example: the type of addOne isn't [a] -> [a]*

`addOne l = map (+1) l`

-- *The (+1) force 'a' to be a Num.*

★:Theorems for free!, Philip Wadler, 1989

# Haskell Functor vs \(\mathbf{Hask}\) Functor

---

A Haskell Functor is a type  $F :: * \rightarrow *$  which belong to the type class  $\text{Functor}$  ; thus instantiate  $fmap :: (a \rightarrow b) \rightarrow (F a \rightarrow F b)$ .

$F : \text{ob}(\mathbf{Hask}) \rightarrow \text{ob}(\mathbf{Hask})$

&  $fmap : \text{hom}(\mathbf{Hask}) \rightarrow \text{hom}(\mathbf{Hask})$

The couple  $(F, fmap)$  is a  $\mathbf{Hask}$ 's functor if for any  $x :: F a$ :

-  $fmap id x = x$

-  $fmap (f.g) x = (fmap f . fmap g) x$

# Haskell Functors Example: Maybe

```
data Maybe a = Just a | Nothing  
instance Functor Maybe where  
  fmap :: (a -> b) -> (Maybe a -> Maybe b)  
  fmap f (Just a) = Just (f a)  
  fmap f Nothing = Nothing
```

```
fmap (+1) (Just 1) == Just 2  
fmap (+1) Nothing == Nothing  
fmap head (Just [1,2,3]) == Just 1
```

# Haskell Functors Example: List

```
instance Functor ([]) where  
  fmap :: (a -> b) -> [a] -> [b]  
  fmap = map
```

```
fmap (+1) [1,2,3]      == [2,3,4]  
fmap (+1) []           == []  
fmap head [[1,2,3],[4,5,6]] == [1,4]
```

# Haskell Functors for the programmer

---

Functor is a type class used for types that can be mapped over.

- Containers: [] , Trees, Map, HashMap...

- "Feature Type":

- Maybe a: help to handle absence of a.

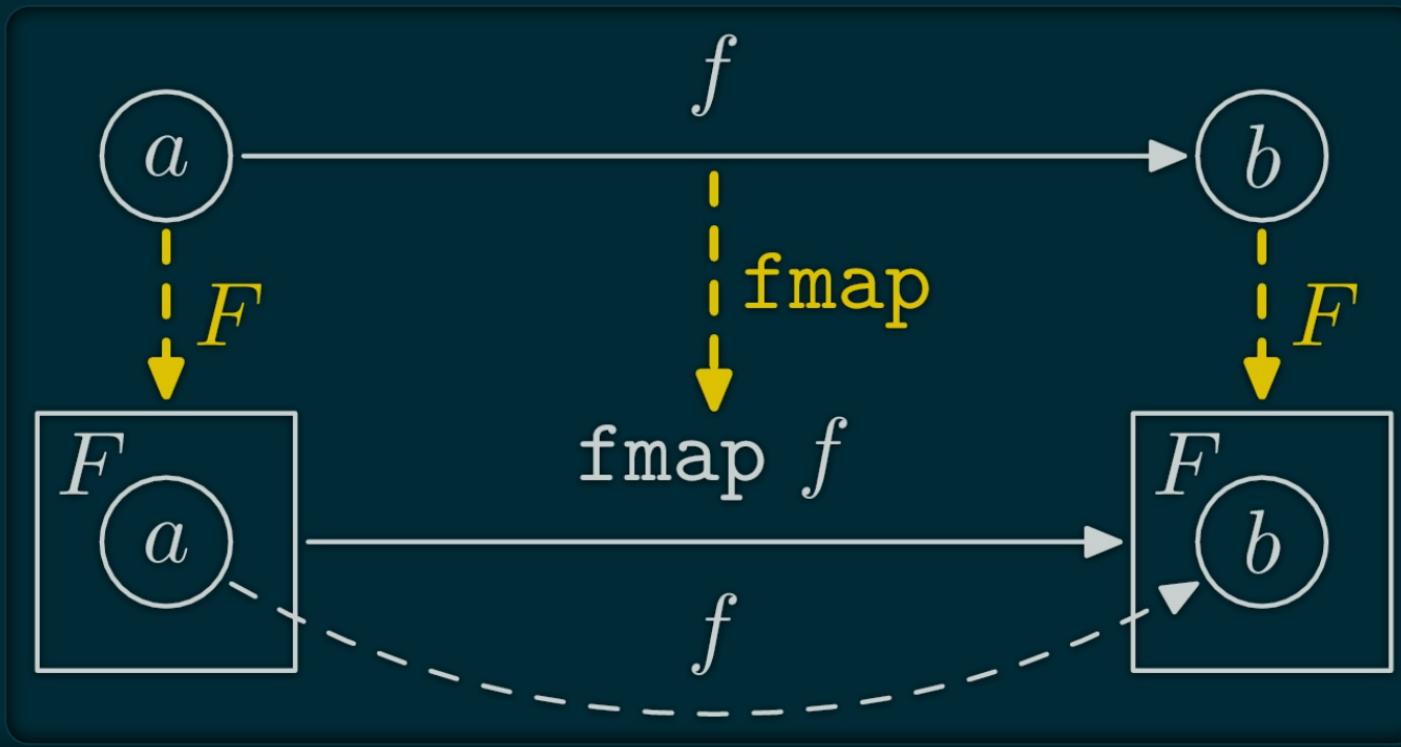
Ex: safeDiv x 0 ⇒ Nothing

- Either String a: help to handle errors

Ex: reportDiv x 0 ⇒ Left "Division by 0!"

# Haskell Functor intuition

Put normal function inside a container. Ex: list, trees...



# Haskell Functor properties

---

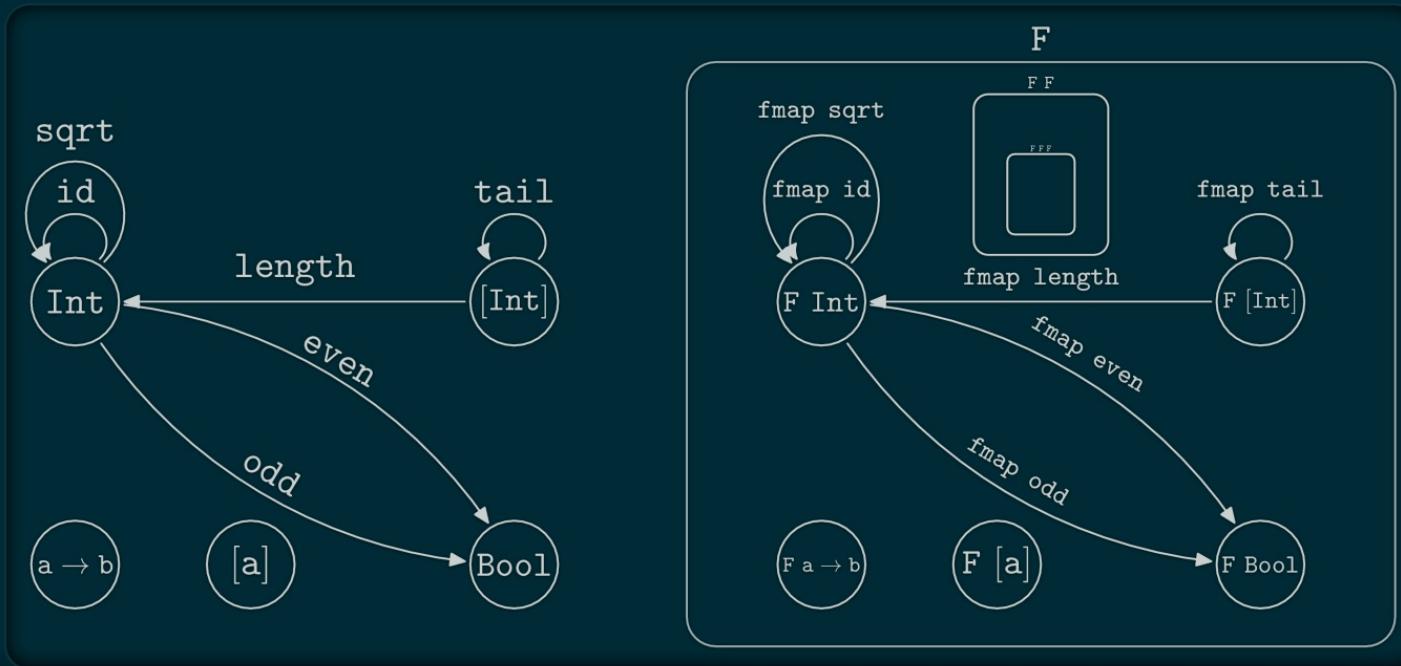
Haskell Functors are:

- *endofunctors* ;  $F : \mathcal{C} \rightarrow \mathcal{C}$  here  $(\mathcal{C} = \text{Hask})$ ,
- a couple **(Object,Morphism)** in  $(\text{Hask})$ .

# Functor as boxes

Haskell functor can be seen as boxes containing all Haskell types and functions.

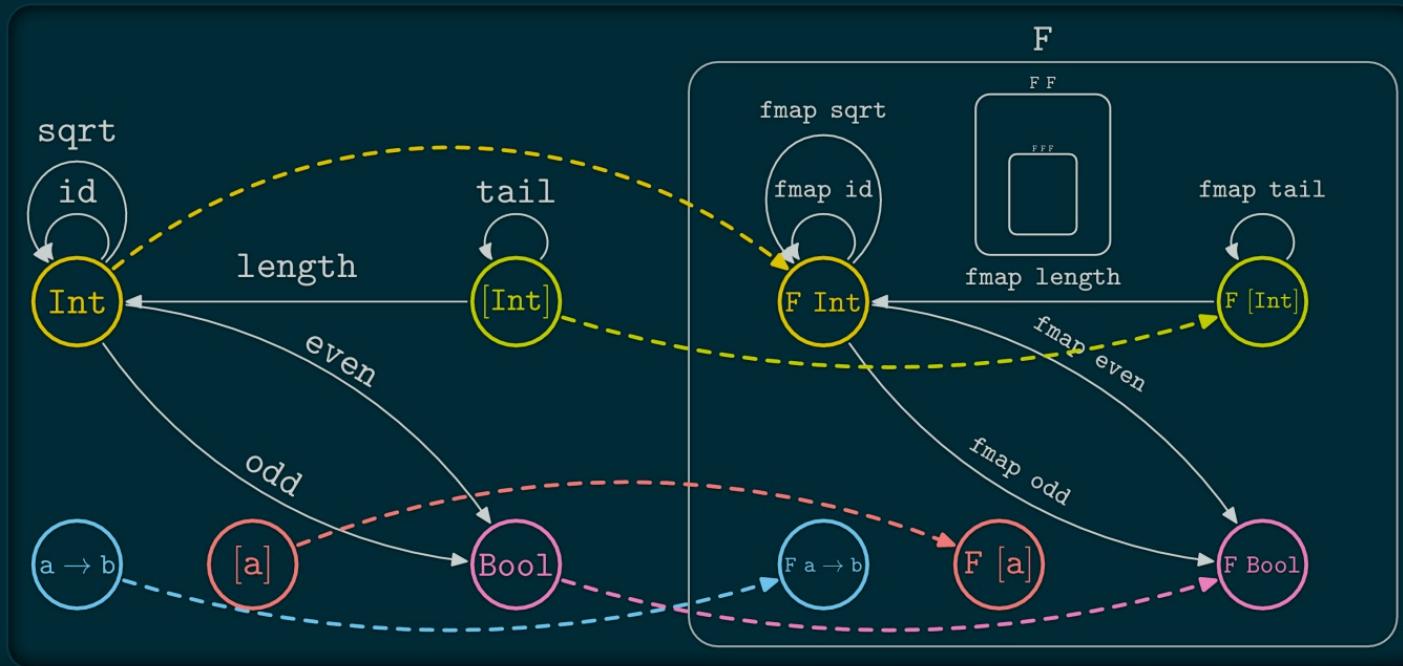
Haskell types is fractal:



# Functor as boxes

Haskell functor can be seen as boxes containing all Haskell types and functions.

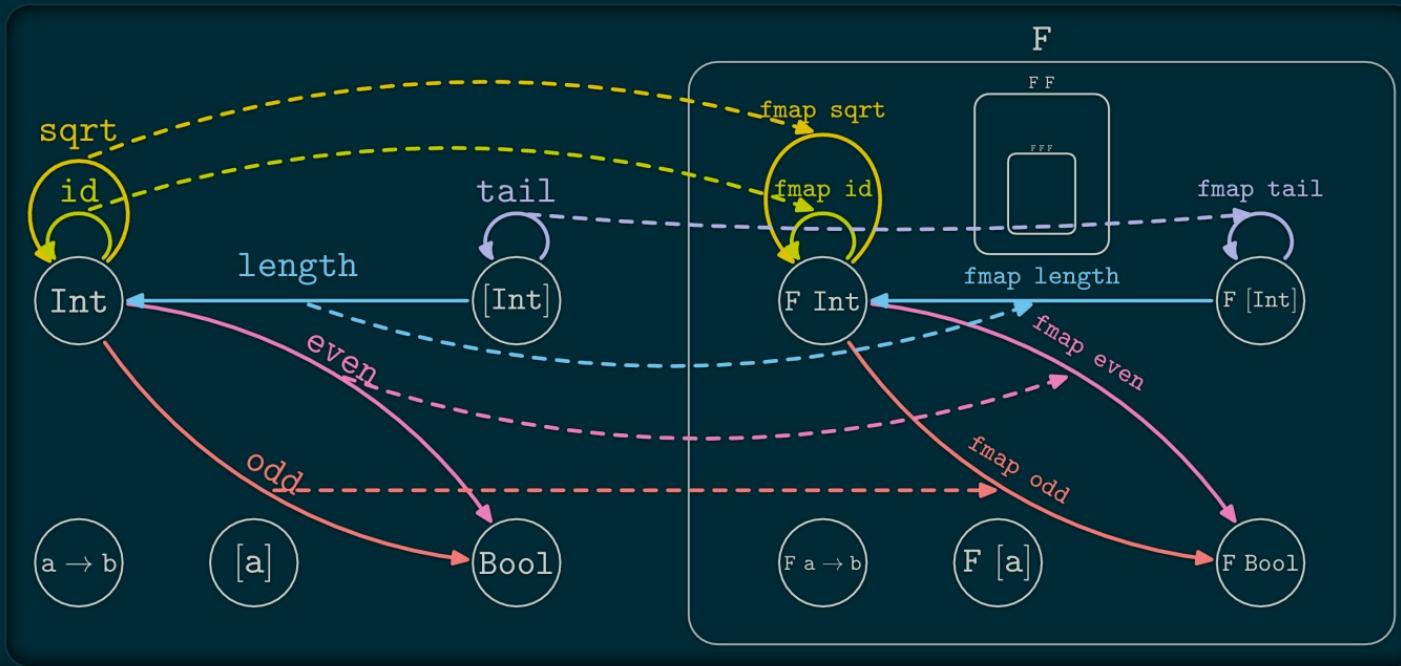
Haskell types is fractal:



# Functor as boxes

Haskell functor can be seen as boxes containing all Haskell types and functions.

Haskell types is fractal:



# "Non Haskell" Hask's Functors

---

A simple basic example is the  $\text{id}_\text{Hask}$  functor. It simply cannot be expressed as a couple  $(F, \text{fmap})$  where

- $F : * \rightarrow *$
- $\text{fmap} : (a \rightarrow b) \rightarrow (F a) \rightarrow (F b)$

Another example:

- $F(T) = \text{Int}$
- $F(f) = \underline{\_} \rightarrow 0$

# Also Functor inside $\mathbf{Hask}$

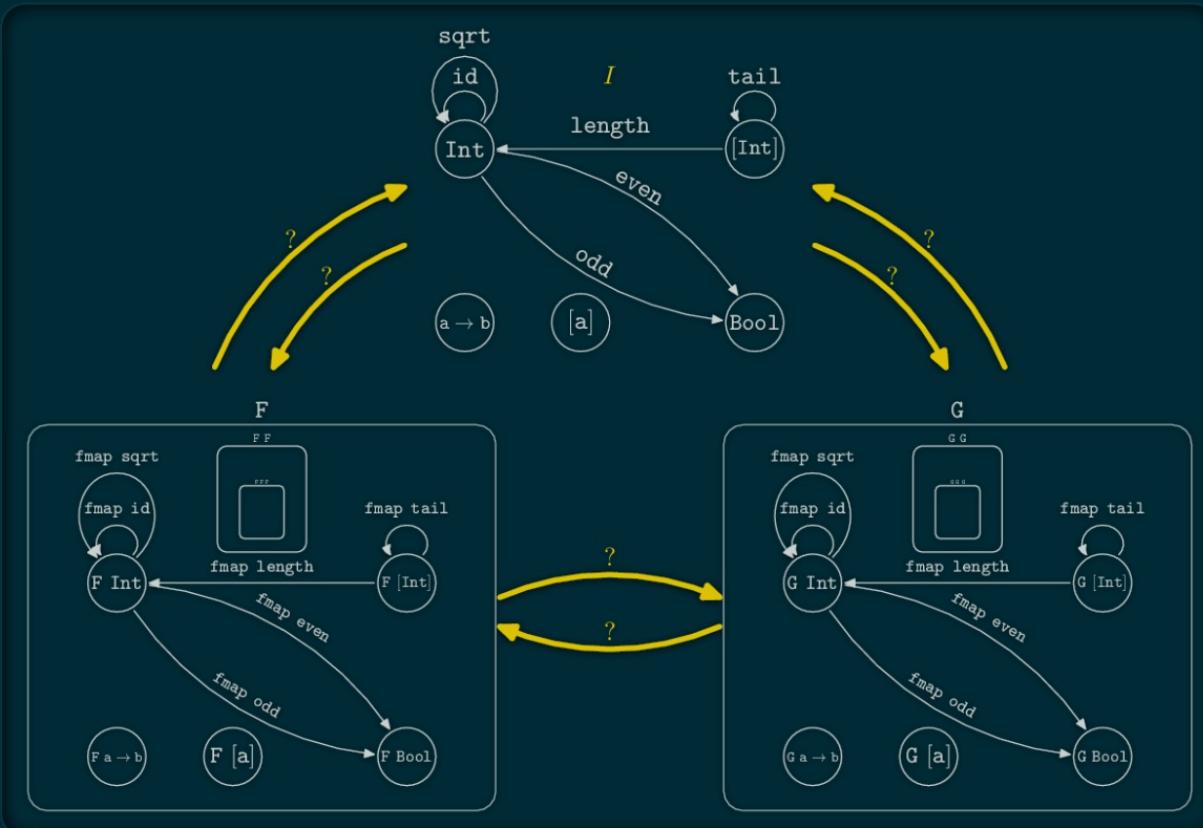
---

$\mathbf{[a]}$  is a category. Idem for  $\mathbf{Int}$ .

$\mathbf{length}$  is a Functor from the category  $\mathbf{[a]}$  to the category  $\mathbf{Int}$ :

- $\mathbf{\text{ob}[\mathbf{[a]}] = \{ \cdot \}}$
- $\mathbf{\text{hom}[\mathbf{[a]}] = \mathbf{[a]}}$   $\Rightarrow$   $\mathbf{\text{hom}[\mathbf{Int}] = \mathbf{[Int]}}$
- $\mathbf{( . = \mathbf{[++]})}$
- $\mathbf{( . = \mathbf{[+]})}$
- id:  $\mathbf{length [] = 0}$
- comp:  $\mathbf{length (l ++ l') = (length l) + (length l')}$

# Category of $\text{\textbackslash}\text{\textbackslash} \text{Hask} \text{\textbackslash}\text{\textbackslash}$ Endofunctors



# Category of Functors

---

If  $\mathcal{C}$  is *small* ( $\hom{\mathcal{C}}$  is a set). All functors from  $\mathcal{C}$  to some category  $\mathcal{D}$  form the category  $\mathrm{Func}(\mathcal{C}, \mathcal{D})$ .

- $\mathrm{ob}(\mathrm{Func}(\mathcal{C}, \mathcal{D}))$ : Functors  $F: \mathcal{C} \rightarrow \mathcal{D}$
- $\hom{\mathrm{Func}(\mathcal{C}, \mathcal{D})}$ : *natural transformations*
- $\circ$ : Functor composition

$\mathrm{Func}(\mathcal{C}, \mathcal{C})$  is the category of endofunctors of  $\mathcal{C}$ .

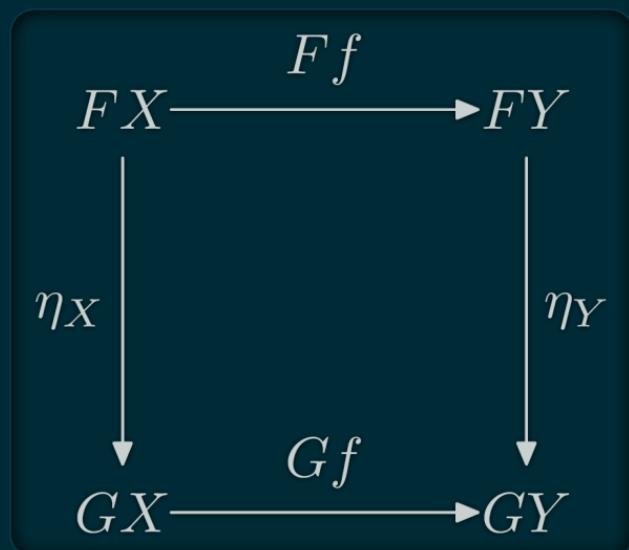
# Natural Transformations

Let  $F$  and  $G$  be two functors from  $C$  to  $D$ .

A *natural transformation*: family  $\eta : \{\eta_X\}_{X \in \text{ob}(C)}$  for  $X \in \text{ob}(C)$  s.t.

ex: between Haskell functors;  $F a \rightarrow G a$

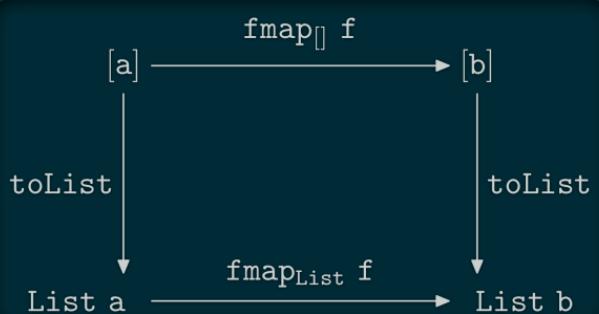
Rearrangement functions only.



# Natural Transformation Examples (1/4)

```
data List a = Nil | Cons a (List a)
toList :: [a] -> Tree a
toList [] = Nil
toList (x:xs) = Cons x (toList xs)
```

`toList` is a natural transformation. It is also a morphism from `[]` to `List` in the Category of \(\mathbf{Hask}\) endofunctors.



# Natural Transformation Examples (2/4)

```
data List a = Nil | Cons a (List a)
toHList :: List a -> [a]
toHList Nil = []
toHList (Cons x xs) = x:toHList xs
```

`toHList` is a natural transformation. It is also a morphism from `List` to `[]` in the Category of `\(Hask\)` endofunctors.

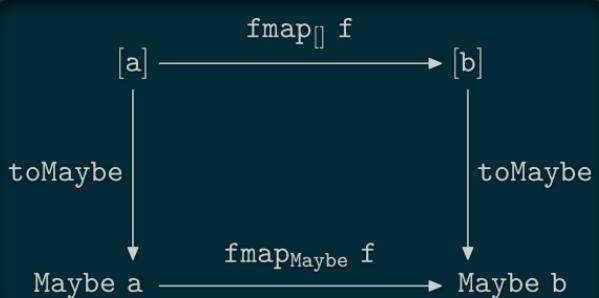


`toList . toHList = id` & `toHList . toList = id`  
therefore `[]` & `Tree` are isomorph.

# Natural Transformation Examples (3/4)

```
toMaybe :: [a] -> Maybe a  
toMaybe [] = Nothing  
toMaybe (x:xs) = Just x
```

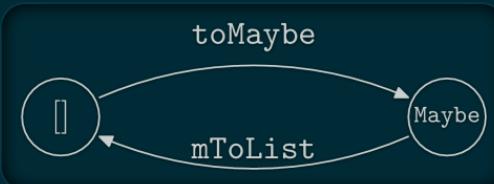
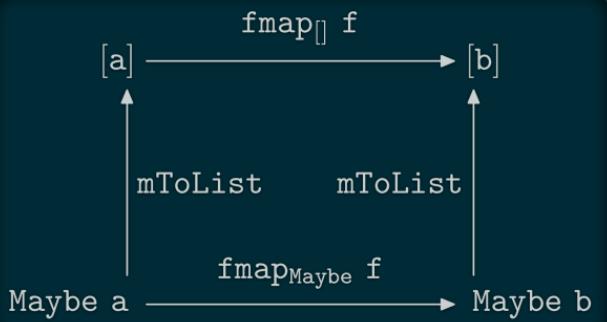
`toMaybe` is a natural transformation. It is also a morphism from `[]` to `Maybe` in the Category of `\(\mathbf{Hask}\)` endofunctors.



# Natural Transformation Examples (4/4)

```
mToList :: Maybe a -> [a]
mToList Nothing = []
mToList Just x = [x]
```

`toMaybe` is a natural transformation. It is also a morphism from `[]` to `Maybe` in the Category of `\(Hask\)` endofunctors.



There is **no isomorphism**.  
Hint: Bool lists longer than 1.

# Composition problem

The Problem; example with lists:

```
f x = [x]    ⇒ f 1 = [1]    ⇒ (f.f) 1 = [[1]] x
g x = [x+1]  ⇒ g 1 = [2]    ⇒ (g.g) 1 = ERROR [2]+1 x
h x = [x+1,x*3] ⇒ h 1 = [2,3] ⇒ (h.h) 1 = ERROR [2,3]+1 x
```

The same problem with most  $f :: a \rightarrow F a$  functions and functor  $F$ .

# Composition Fixable?

---

How to fix that? We want to construct an operator which is able to compose:

$f :: a \rightarrow F b$  &  $g :: b \rightarrow F c$ .

More specifically we want to create an operator  $\circledcirc$  of type

$\circledcirc :: (b \rightarrow F c) \rightarrow (a \rightarrow F b) \rightarrow (a \rightarrow F c)$

Note: if  $F = I$ ,  $\circledcirc = (.)$ .

# Fix Composition (1/2)

---

Goal, find:  $\bigcirc :: (b \rightarrow F c) \rightarrow (a \rightarrow F b) \rightarrow (a \rightarrow F c)$

$f :: a \rightarrow F b$ ,  $g :: b \rightarrow F c$ :

- $(g \bigcirc f) x$  ???
- First apply  $f$  to  $x \Rightarrow f x :: F b$
- Then how to apply  $g$  properly to an element of type  $F b$ ?

# Fix Composition (2/2)

---

Goal, find:  $\bigcirc :: (b \rightarrow F c) \rightarrow (a \rightarrow F b) \rightarrow (a \rightarrow F c)$

$f :: a \rightarrow F b$ ,  $g :: b \rightarrow F c$ ,  $f x :: F b$ :

- Use  $fmap :: (t \rightarrow u) \rightarrow (F t \rightarrow F u)$ !
  - $(fmap g) :: F b \rightarrow F (F c)$  ; ( $t=b$ ,  $u=F c$ )
  - $(fmap g) (f x) :: F (F c)$  it almost WORKS!
  - We lack an important component,  $join :: F (F c) \rightarrow F c$
  - $(g \bigcirc f) x = join ((fmap g) (f x)) \odot$
- $\bigcirc$  is the Kleisli composition; in Haskell: `<=<` (in `Control.Monad`).

# Necessary laws

---

For  $\odot$  to work like composition, we need join to hold the following properties:

- $\text{join}(\text{join}(F(F(F a)))) = \text{join}(F(\text{join}(F(F a))))$
- abusing notations denoting  $\text{join}$  by  $\odot$ ; this is equivalent to  
$$(F \odot F) \odot F = F \odot (F \odot F)$$

- There exists  $\eta :: a \rightarrow F a$  s.t.

$$\eta \odot F = F = F \odot \eta$$

# Klesli composition

Now the composition works as expected. In Haskell  $\circledcirc$  is `<=<` in `Control.Monad`.

```
g <=< f = \x -> join ((fmap g) (f x))
```

```
f x = [x]      ⇒ f 1 = [1]      ⇒ (f <=< f) 1 = [1] ✓  
g x = [x+1]    ⇒ g 1 = [2]      ⇒ (g <=< g) 1 = [3] ✓  
h x = [x+1, x*3] ⇒ h 1 = [2,3] ⇒ (h <=< h) 1 = [3,6,4,9] ✓
```

# We reinvented Monads!

---

A monad is a triplet  $(M, \odot, \eta)$  where

- $\lambda(M)$  an Endofunctor (to type  $a$  associate  $M a$ )
- $\lambda(\odot : M \times M \rightarrow M)$  a nat. trans. (i.e.  $\odot :: M(M a) \rightarrow M a$  ; join)
- $\lambda(\eta : I \rightarrow M)$  a nat. trans. ( $\lambda(I)$  identity functor ;  $\eta : a \rightarrow M a$ )

Satisfying

- $\lambda(M \odot (M \odot M)) = (M \odot M) \odot M$
- $\lambda(\eta \odot M = M = M \odot \eta)$

# Compare with Monoid

---

A Monoid is a triplet  $\langle (E, \cdot, e) \rangle$  s.t.

- $\langle E \rangle$  a set
- $\langle \cdot : E \times E \rightarrow E \rangle$
- $\langle e : 1 \rightarrow E \rangle$

Satisfying

- $\langle x \cdot (y \cdot z) = (x \cdot y) \cdot z, \forall x, y, z \in E \rangle$
- $\langle e \cdot x = x = x \cdot e, \forall x \in E \rangle$

# Monads are just Monoids

---

| *A Monad is just a monoid in the category of endofunctors, what's the problem?*

The real sentence was:

| *All told, a monad in  $X$  is just a monoid in the category of endofunctors of  $X$ , with product  $\times$  replaced by composition of endofunctors and unit set by the identity endofunctor.*

# Example: List

---

- $[] :: * \rightarrow *$  an Endofunctor
- $\backslash(\odot : M \times M \rightarrow M)$  a nat. trans. ( $\text{join} :: M(M a) \rightarrow M a$ )
- $\backslash(\eta : I \rightarrow M)$  a nat. trans.

-- In Haskell  $\odot$  is "join" in "Control.Monad"

```
join :: [[a]] -> [a]
join = concat
```

-- In Haskell the "return" function (unfortunate name)

```
 $\eta :: a \rightarrow [a]$ 
 $\eta x = [x]$ 
```

# Example: List (law verification)

Example: `List` is a functor (`join` is  $\odot$ )

- $\backslash(M \odot (M \odot M) = (M \odot M) \odot M)\backslash$
- $\backslash(\eta \odot M = M = M \odot \eta)\backslash$

$$\begin{aligned} \text{join} [ \text{join} [[x,y,\dots,z]] ] &= \text{join} [[x,y,\dots,z]] \\ &= \text{join} (\text{join} [[[x,y,\dots,z]]]) \\ \text{join} (\eta [x]) &= [x] = \text{join} [\eta x] \end{aligned}$$

Therefore `([],join,η)` is a monad.

# Monads useful?

A *LOT* of monad tutorial on the net. Just one example; the State Monad

DrawScene to State Screen DrawScene ; still **pure**.

```
main = drawImage (width,height)  
  
drawImage :: Screen -> DrawScene  
drawImage screen = do  
    drawPoint p screen  
    drawCircle c screen  
    drawRectangle r screen  
  
drawPoint point screen = ...  
drawCircle circle screen = ...  
drawRectangle rectangle screen = ...
```

```
main = do  
    put (Screen 1024 768)  
    drawImage  
  
drawImage :: State Screen DrawScene  
drawImage = do  
    drawPoint p  
    drawCircle c  
    drawRectangle r  
  
drawPoint :: Point -> State Screen DrawScene  
drawPoint p = do  
    Screen width height <- get  
    ...
```



# κατα-morphism

---



# cata-morphism: fold generalization

acc type of the "accumulator":

fold :: (acc -> a -> acc) -> acc -> [a] -> acc

Idea: put the accumulated value inside the type.

```
-- Equivalent to fold (+1) 0 "cata"  
(Cons 'c' (Cons 'a' (Cons 't' (Cons 'a' Nil))))  
(Cons 'c' (Cons 'a' (Cons 't' (Cons 'a' 0))))  
(Cons 'c' (Cons 'a' (Cons 't' 1)))  
(Cons 'c' (Cons 'a' 2))  
(Cons 'c' 3)
```

4

But where are all the informations? (+1) and 0?

# κata-morphism: Missing Information

---

Where is the missing information?

- Functor operator `fmap`
- Algebra representing the `(+1)` and also knowing about the `0`.

First example, make `length` on `[Char]`

# κata-morphism: Type work

```
data StrF a = Cons Char a | Nil  
data Str' = StrF Str'
```

-- generalize the construction of Str to other datatype

-- Mu: type fixed point

-- Mu :: (\* -> \*) -> \*

```
data Mu f = InF { outF :: f (Mu f) }  
data Str = Mu StrF
```

-- Example

```
foo=InF { outF = Cons 'f'  
        (InF { outF = Cons 'o'  
                (InF { outF = Cons 'o'  
                        (InF { outF = Nil })}))})}
```

# cata-morphism: missing information retrieved

```
type Algebra f a = f a -> a
instance Functor (StrF a) =
  fmap f (Cons c x) = Cons c (f x)
  fmap _ Nil = Nil
```

```
cata :: Functor f => Algebra f a -> Mu f -> a
cata f = f . fmap (cata f) . outF
```

# κata-morphism: Finally length

All needed information for making length.

```
instance Functor (StrF a) =  
  fmap f (Cons c x) = Cons c (f x)  
  fmap _ Nil = Nil
```

```
length' :: Str -> Int  
length' = cata phi where  
  phi :: Algebra StrF Int -- StrF Int -> Int  
  phi (Cons a b) = 1 + b  
  phi Nil = 0
```

```
main = do  
  l <- length' $ stringToStr "Toto"  
  ...
```

# cata-morphism: extension to Trees

Once you get the trick, it is easy to extend to most Functor.

```
type Tree = Mu TreeF
data TreeF x = Node Int [x]

instance Functor TreeF where
  fmap f (Node e xs) = Node e (fmap f xs)

depth = cata phi where
  phi :: Algebra TreeF Int -- TreeF Int -> Int
  phi (Node x sons) = 1 + foldr max 0 sons
```

# Conclusion

---

Category Theory oriented Programming:

- Focus on the type and operators
- Extreme generalisation
- Better modularity
- Better control through properties of types

No cat were harmed in the making of this presentation.