GEOMETRIC DEEP LEARNING (**L65**)

Pietro Liò *University of Cambridge*
Petar Veličković *Google DeepMind / University of Cambridge*

Lent Term 2025

CST Part III / MPhil ACS / MPhil MLMI

This lecture will be potentially intense…

Please help yourself
to a chocolate ☺

A reminder on L65 Practical submissions

Deadline for Practical 1 is **this Wednesday** (5 March), 4pm

Do **not** leave this to the last minute!
(And thank you to those who already submitted.)

Also, as a reminder, *you are supposed to submit a .zip file* with two files:

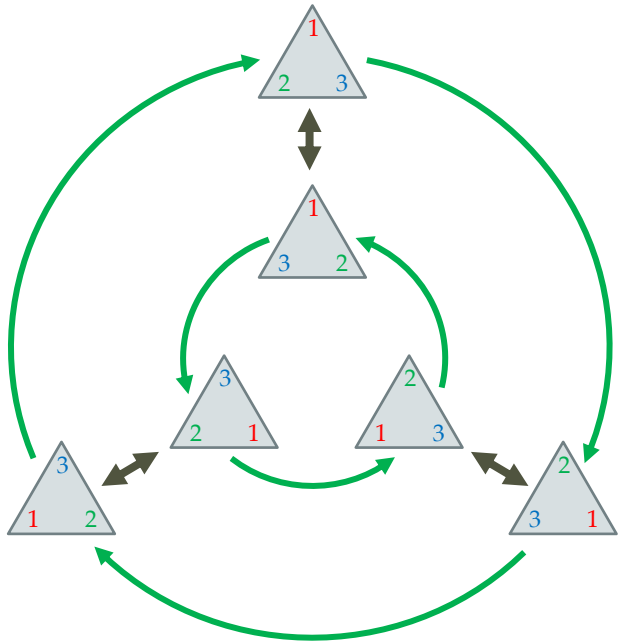
- *An .ipynb file* with your Colab solutions
- *A four-page .pdf file (+refs)* with Paper discussions in two perspectives

This is **important** to be considered for the full range of 30 marks

Please email Pietro and I **asap** if you have any questions or concerns

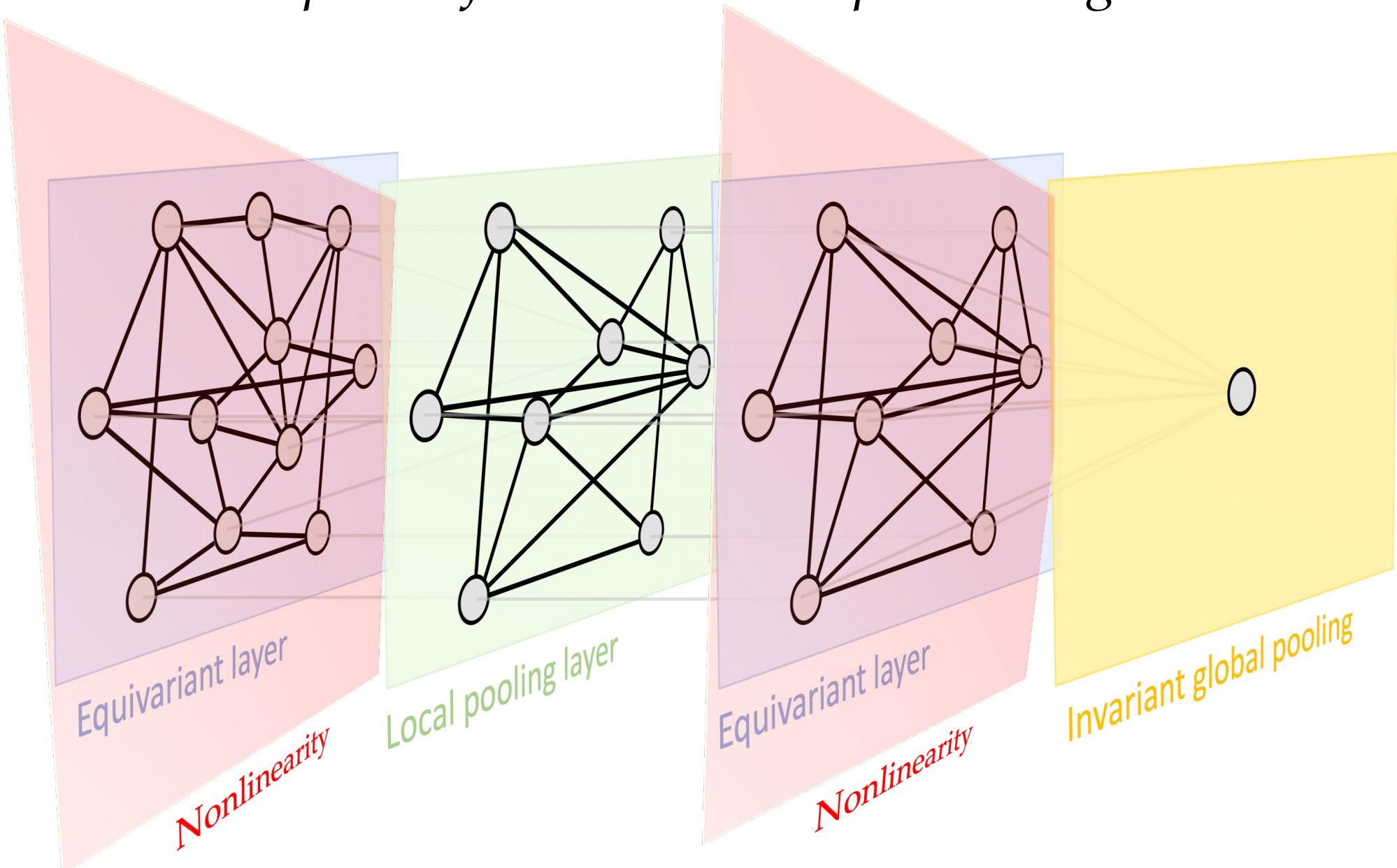
Recap: Symmetries

A **symmetry** of an object is a transformation of that object that leaves it unchanged



The symmetries of a triangle,
as generated by 120-degree ***rotations*** *R and* ***flips*** *F.*

Recall: The blueprint of Geometric Deep Learning



Recall: Popular architectures as instances of GDL blueprint

Architecture	**Domain Ω**	**Symmetry Group $\mathfrak{G}$**
CNN	Grid	Translation
Spherical CNN	Sphere / $\mathrm{SO}(3)$	Rotation $\mathrm{SO}(3)$
Mesh CNN	Manifold	Isometry $\mathrm{Iso}(\Omega)$ / Gauge Symmetry $\mathrm{SO}(2)$
GNN	Graph	Permutation Σ_n
Deep Sets	Set	Permutation Σ_n
Transformer	Complete Graph	Permutation Σ_n
E(3) GNN	Geometric Graph	Permutation $\Sigma_n \times$ Euclidean $E(3)$
LSTM	1D Grid	Time warping

Recall: Popular architectures as instances of GDL blueprint

Architecture	**Domain Ω**	**Symmetry Group $\mathfrak{G}$**
CNN (L3)	Grid	Translation
Spherical CNN (L3)	Sphere / $\mathrm{SO}(3)$	Rotation $\mathrm{SO}(3)$
Mesh CNN (L8)	Manifold	Isometry $\mathrm{Iso}(\Omega)$ / Gauge Symmetry $\mathrm{SO}(2)$
GNN (L4)	Graph	Permutation Σ_n
Deep Sets (L3)	Set	Permutation Σ_n
Transformer (L5)	Complete Graph	Permutation Σ_n
E(3) GNN (DD1)	Geometric Graph	Permutation $\Sigma_n \times$ Euclidean $E(3)$
LSTM (see book ☺)	1D Grid	Time warping

9. INTO THE REALM CATEGORICAL

Going beyond symmetries

Petar Veličković

*Now: going **beyond** symmetries*

We designed neural networks that are *fully resistant* to ***symmetries***

- *Invariance*: Symmetry on input *does not affect* output
- *Equivariance*: Symmetry on input *affects* output *the same way*

This turned out sufficient to re-derive “all the fan-favourites”!

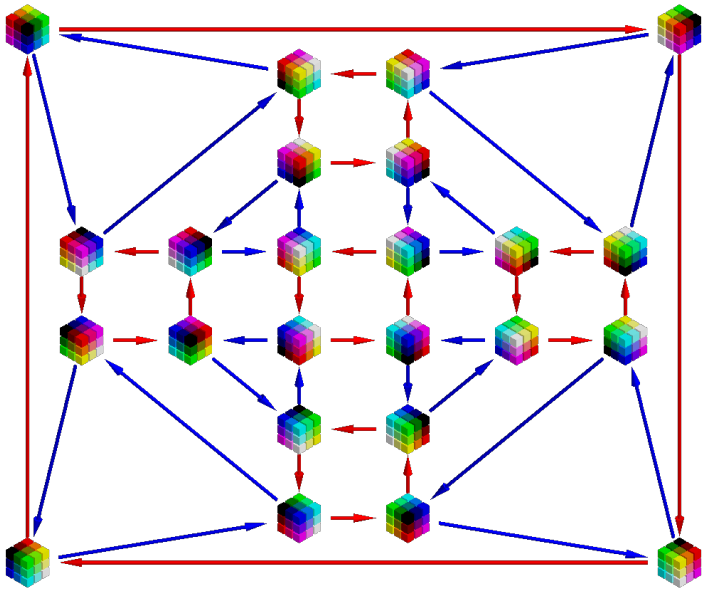
But: are symmetries the **whole** story?

Let’s recall the symmetry group *axioms*…

Abstract Groups

A *group* is a set $\mathfrak{G}$ with a binary operation denoted $\mathfrak{gh}$ satisfying the following properties:

- *Associativity:* $(\mathfrak{gh})\mathfrak{k} = \mathfrak{g}(\mathfrak{hk})$ for all $\mathfrak{g}, \mathfrak{h}, \mathfrak{k} \in \mathfrak{G}$
- *Identity*: there exists a unique $\mathfrak{e} \in \mathfrak{G}$ satisfying
$$\mathfrak{ge} = \mathfrak{eg} = \mathfrak{g}$$
- *Inverse:* for each $\mathfrak{g} \in \mathfrak{G}$ there is a unique inverse $\mathfrak{g}^{-1} \in \mathfrak{G}$,
such that $\mathfrak{g}\mathfrak{g}^{-1} = \mathfrak{g}^{-1}\mathfrak{g} = \mathfrak{e}$
- *Closure*: for every $\mathfrak{g}, \mathfrak{h} \in \mathfrak{G}$, we have $\mathfrak{gh} \in \mathfrak{G}$



Rotational symmetries of the cube (group O_h)

Abstract Groups

A *group* is a set $\mathfrak{G}$ with a binary operation denoted $\mathfrak{gh}$ satisfying the following properties:

- *Associativity:* $(\mathfrak{gh})\mathfrak{k} = \mathfrak{g}(\mathfrak{hk})$ for all $\mathfrak{g}, \mathfrak{h}, \mathfrak{k} \in \mathfrak{G}$
- *Identity*: there exists a unique $\mathfrak{e} \in \mathfrak{G}$ satisfying

$$\mathfrak{ge} = \mathfrak{eg} = \mathfrak{g}$$

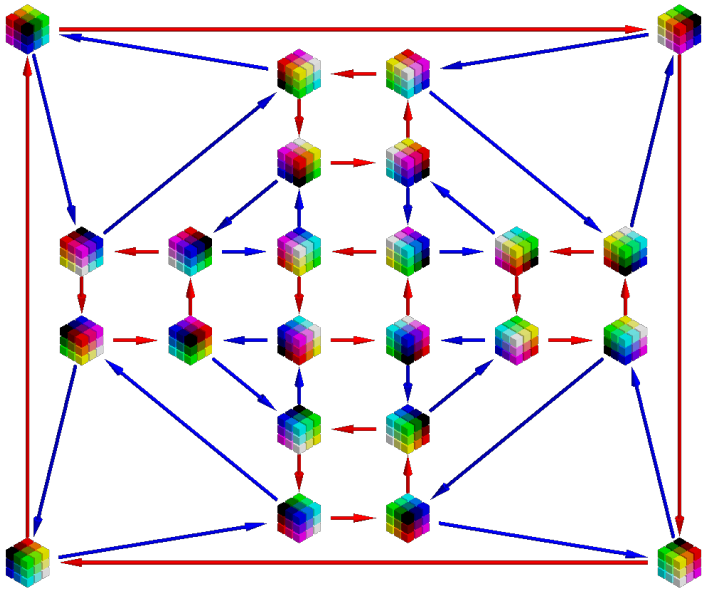
- *Inverse:* for each $\mathfrak{g} \in \mathfrak{G}$ there is a unique inverse $\mathfrak{g}^{-1} \in \mathfrak{G}$,

such that $\mathfrak{gg}^{-1} = \mathfrak{g}^{-1}\mathfrak{g} = \mathfrak{e}$

No information loss

- *Closure*: for every $\mathfrak{g}, \mathfrak{h} \in \mathfrak{G}$, we have $\mathfrak{gh} \in \mathfrak{G}$

Transformations must *all* compose



Rotational symmetries of the cube (group O_h)

In reality…

Not all transformations we care about are ***invertible*** (lossless)

Not all pairs of transformations we care about will be ***composable***

…hence, we may want to go **beyond** what symmetries can explain

It seems we need a more general language than group theory…
A perspective to reason about the structure of *general computation*

Category theory offers such a perspective!

Into the realm categorical

Transitioning into the domain of *categories* will allow us to **generalise** many of the concepts we investigated with *groups*

Groups	→	Categories
Group actions	→	Functors
Equivariance	→	Commutative diagrams

Critically, these objects will *relax* some of the *constraints* of groups
This will allow us to build *better-aligned* neural networks!

Beware: Many intros on CT are geared towards mathematicians
Here be dragons!



What will we do today?

Today's lecture block is imagined as a mix of:

- *Philosophy*: **why** should we bother to study CT?
- *Key concepts*: **categories**, **objects**, **morphisms**, **functors**
- *Mini-tutorial:* get us used to manipulating these & derive **monads**
- *Grounding*: Re-derive GDL using **monad algebra homomorphisms**
- *Generalising*: Go beyond, by switching monads with **endofunctors**
- *Syntax*: How to leverage algebra to describe **implementations**?

Expect many *take-home* or *on-the-spot* **exercises**

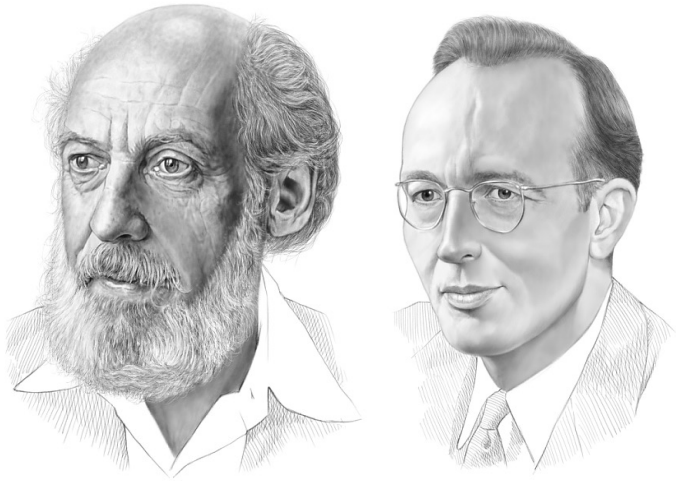
A lot of what will be covered is *foundational* CT material
(apologies to the category theorists in the audience---this will be somewhat slow!)

GENERAL THEORY OF NATURAL EQUIVALENCES

BY

SAMUEL EILENBERG AND SAUNDERS MacLANE

This may be regarded as a continuation of the Klein Erlanger Programm, in the sense that a geometrical space with its group of transformations is generalized to a category with its algebra of mappings.



Eilenberg & Mac Lane

1945



Category Theory, in a nutshell

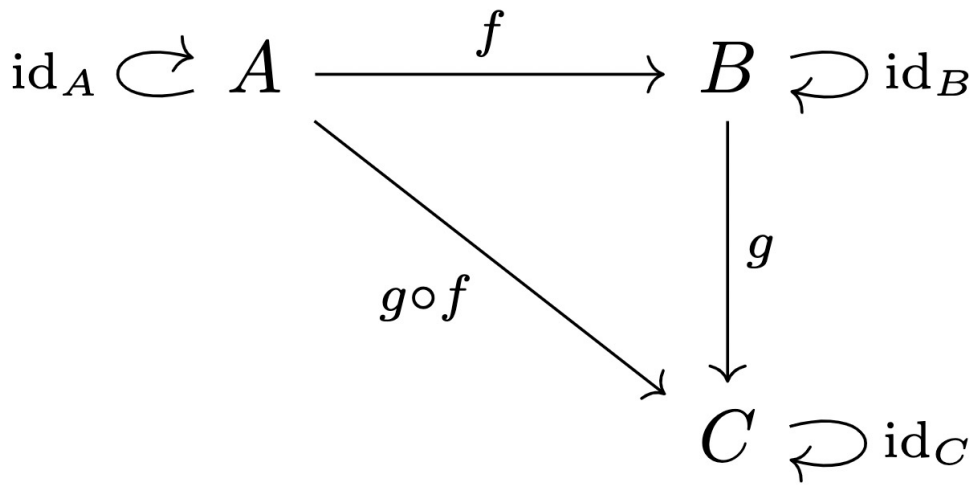
A category: a *universe* of **objects**, and **morphisms** between them, s.t.:

(The word *"universe"* is used here deliberately instead of *"set"*, to avoid *paradoxes*)

- For each object A, there is a unique **identity** morphism $\mathrm{id}_A : A \to A$
- For $f : A \to B$ and $g : B \to C$, there is a **composition**, $g \circ f : A \to C$
- For any morphism $f : A \to B$, it holds that $\mathrm{id}_B \circ f = f \circ \mathrm{id}_A = f$
- For any composable f, g, h, we have $h \circ (g \circ f) = (h \circ g) \circ f$

Studied by using *commutative diagrams*:

(Any pair of paths from A to B yield identical results)



Category Theory, in a nutshell

A category: a *universe* of **objects**, and **morphisms** between them, s.t.:

- For each object A, there is a unique **identity** morphism $\mathrm{id}_A : A \to A$
- For $f : A \to B$ and $g : B \to C$, there is a **composition**, $g \circ f : A \to C$
- For any morphism $f : A \to B$, it holds that $\mathrm{id}_B \circ f = f \circ \mathrm{id}_A = f$
- For any composable f, g, h, we have $h \circ (g \circ f) = (h \circ g) \circ f$

Note that a category *generalises* a group!

Identities still exist, but are now defined *separately for each object*
Not all morphisms need to have an inverse
Not all pairs of morphisms can compose with each other

Category Theory, in a nutshell

A category: a *universe* of **objects**, and **morphisms** between them, s.t.:

- For each object A, there is a unique **identity** morphism $\mathrm{id}_A : A \to A$
- For $f : A \to B$ and $g : B \to C$, there is a **composition**, $g \circ f : A \to C$
- For any morphism $f : A \to B$, it holds that $\mathrm{id}_B \circ f = f \circ \mathrm{id}_A = f$
- For any composable f, g, h, we have $h \circ (g \circ f) = (h \circ g) \circ f$

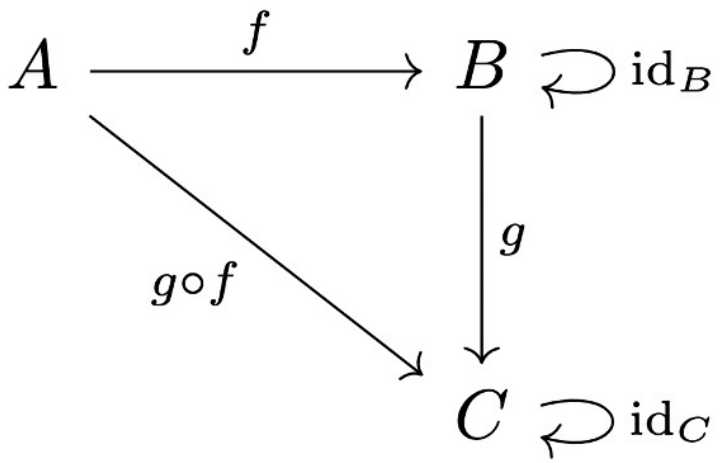
Critically, we **do not** assume *knowledge* of what's in the objects!

Allows us to focus on the *structure,* and not get bogged down in *details*

Excellent recipe for discovering **connections**!

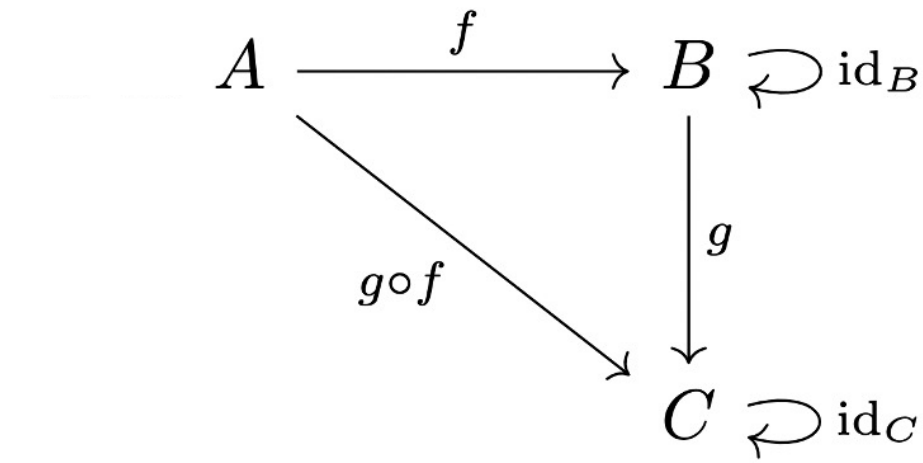
*What is **not** a category?*

***All** identities* must exist!
This is **not** a category:

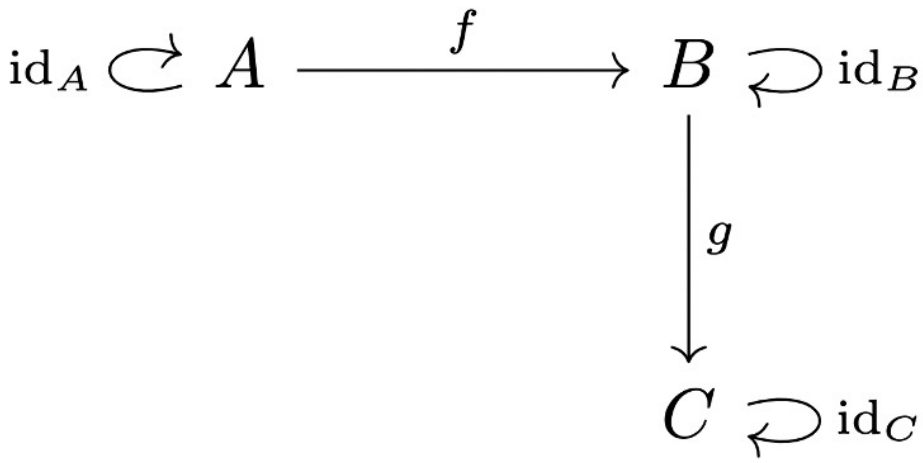


*What is **not** a category?*

***All** identities* must exist!
This is **not** a category:

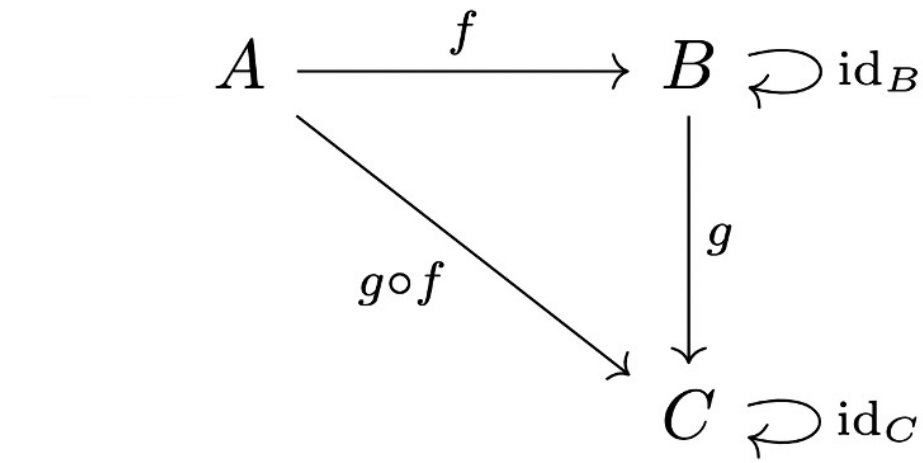


***All** compositions* must exist!
This is also **not** a category:

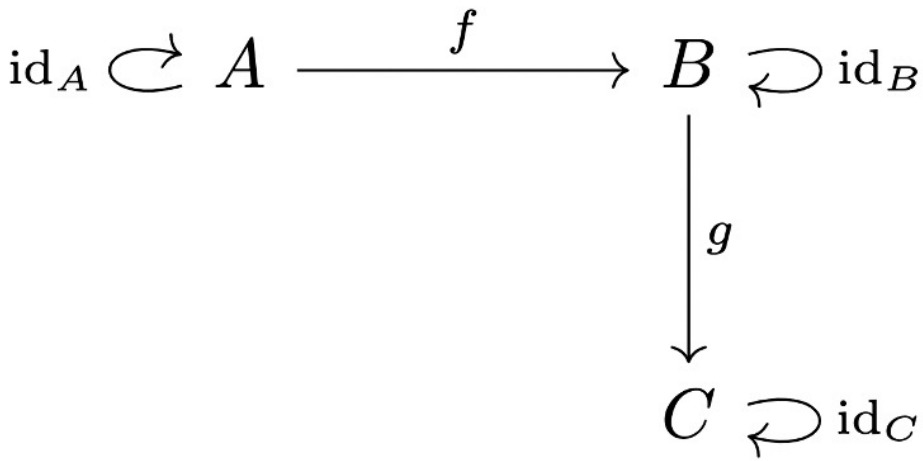


*What is **not** a category?*

***All** identities* must exist!
This is **not** a category:



***All** compositions* must exist!
This is also **not** a category:



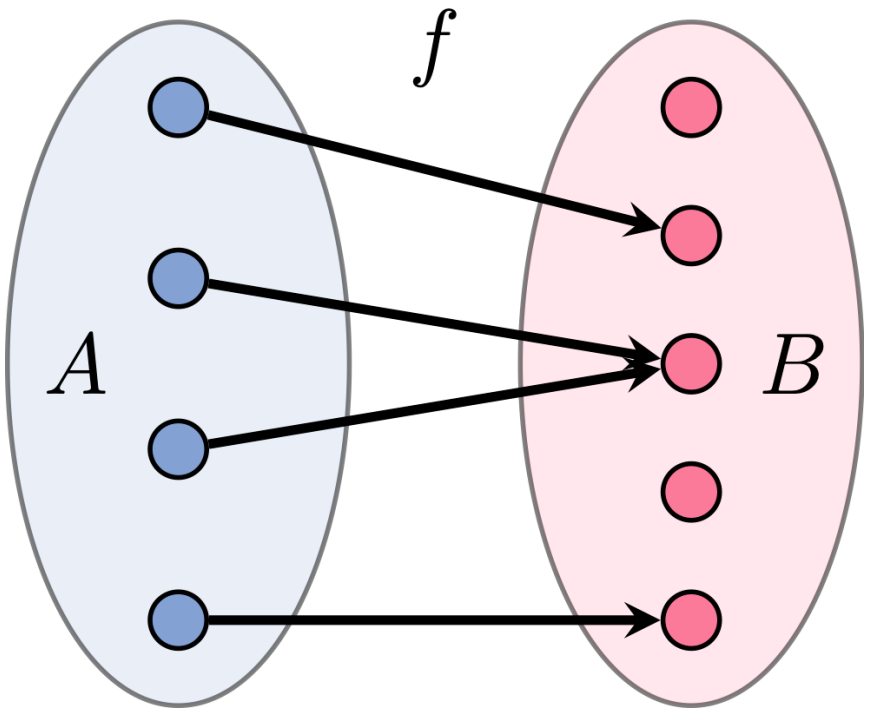
We will often *not* draw all of these, as we focus on what's *important*
But know that they must exist!

*The **Set** category*

Typical standard example is the **Set** category

- Objects are *sets*
- Morphisms are *functions* between them

Note why objects are in a “universe” and not a “set”
Defining a ‘set of sets’ is tricky
It can lead to Russell’s paradox: *“does a set of all sets contain itself”*?
The notion of a **universe** allows us to handle this issue



*The **Set** category*

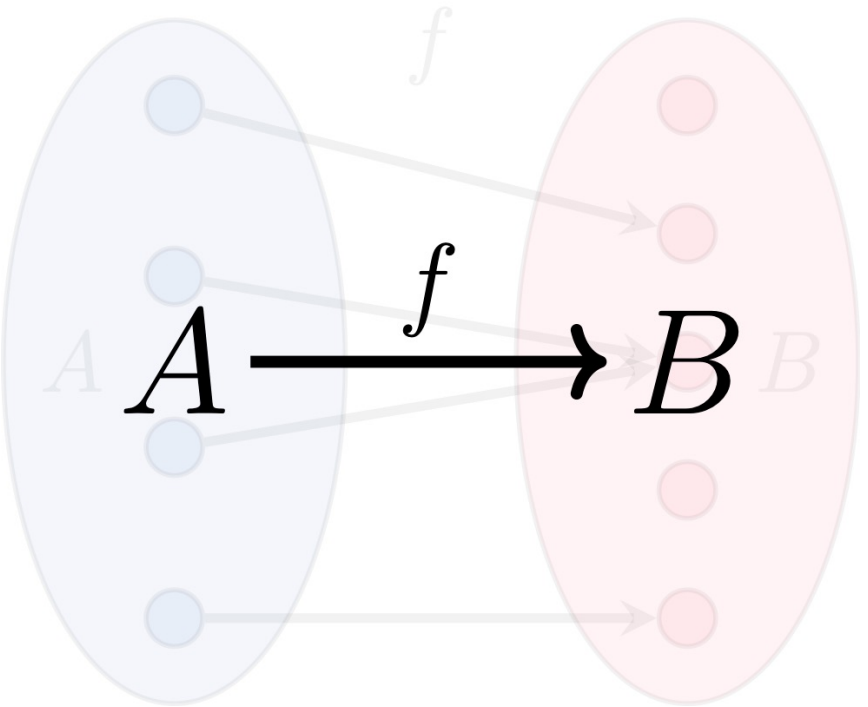
Typical standard example is the **Set** category

- Objects are *sets*
- Morphisms are *functions* between them

Exercise:
Show **Set** satisfies all the category axioms

Our category is **oblivious** to what's in the sets!

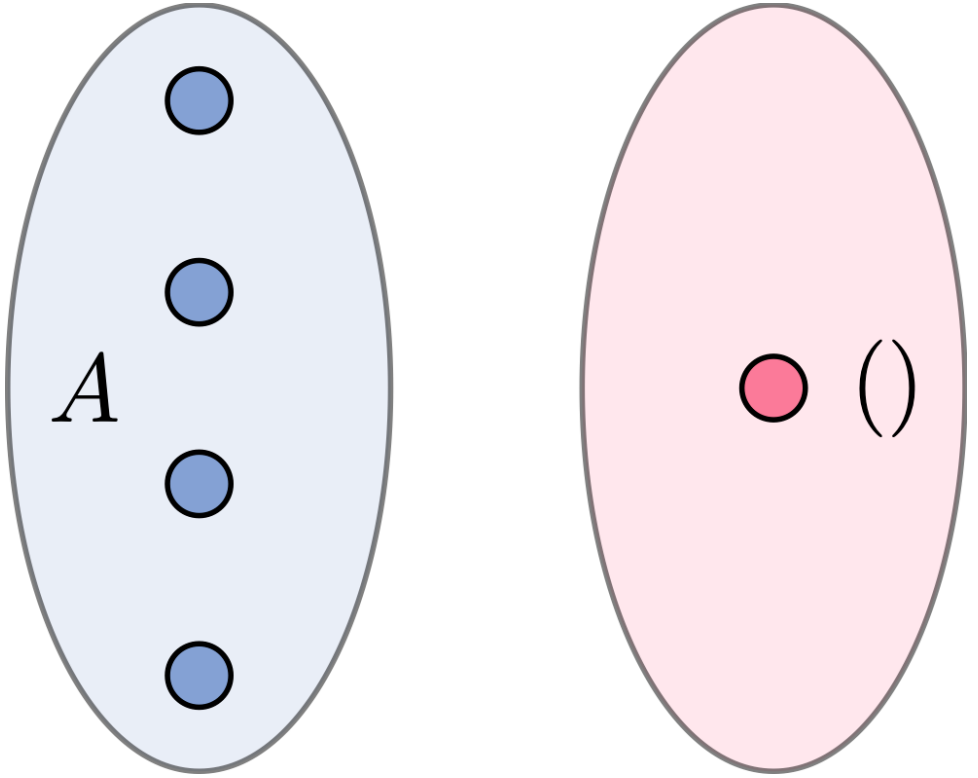
Wondering whether this *drops* a lot of data?
…not quite!
I will now show you a *mind-blowing result* 💥



Identifying the unit set

Let us discuss the one-element set (sometimes denoted (), or `unit`)

What can you tell me about morphisms from *any* set A to ()?

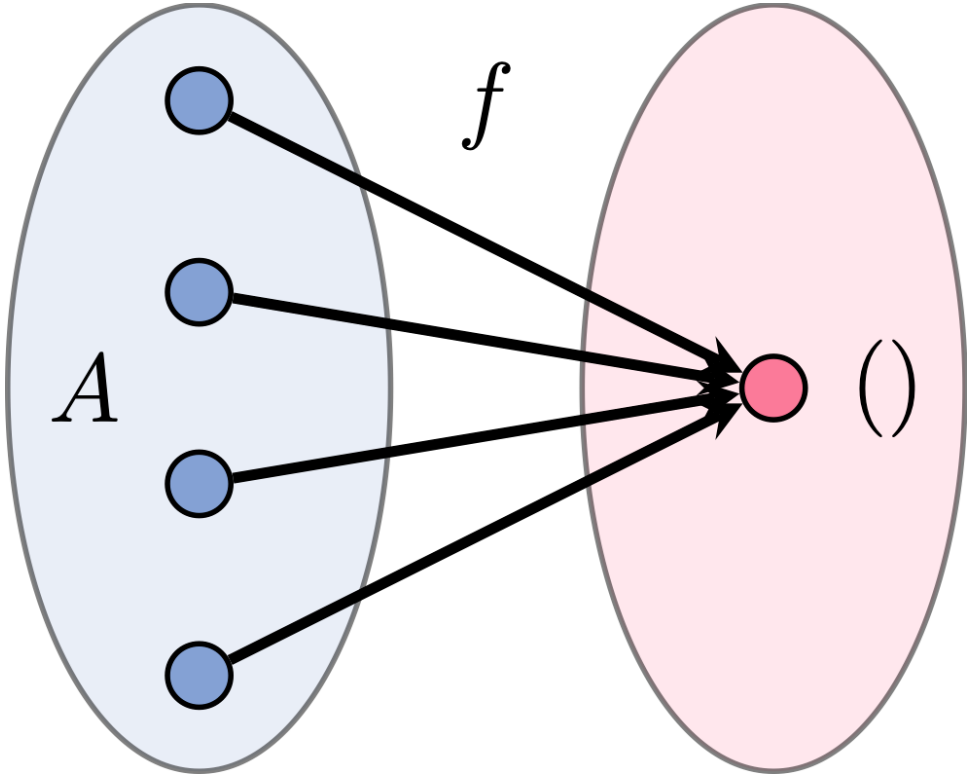


Identifying the unit set

Let us discuss the one-element set (sometimes denoted (), or `unit`)

What can you tell me about morphisms from *any* set A to ()?

There is **exactly one!**
This *uniquely* defines () (up to isomorphism)



Identifying the unit set

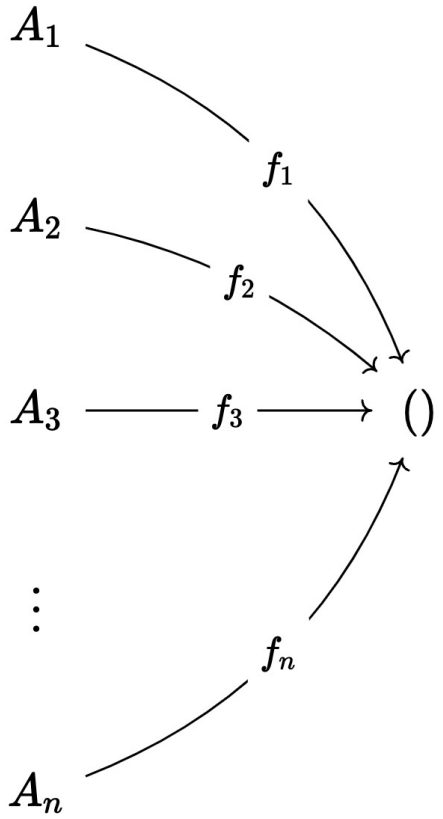
Let us discuss the one-element set (sometimes denoted (), or `unit`)

What can you tell me about morphisms from *any* set A to ()?

We can *uniquely* identify the `unit` like this!
We *only* need to look at the *morphisms*
No need to look at the sets themselves

Categorically, () is called a *terminal object*

We can also define a notion of an *initial object*
(exactly *one* morphism coming out of it!)
What would this be for **Set**?



Identifying the unit set

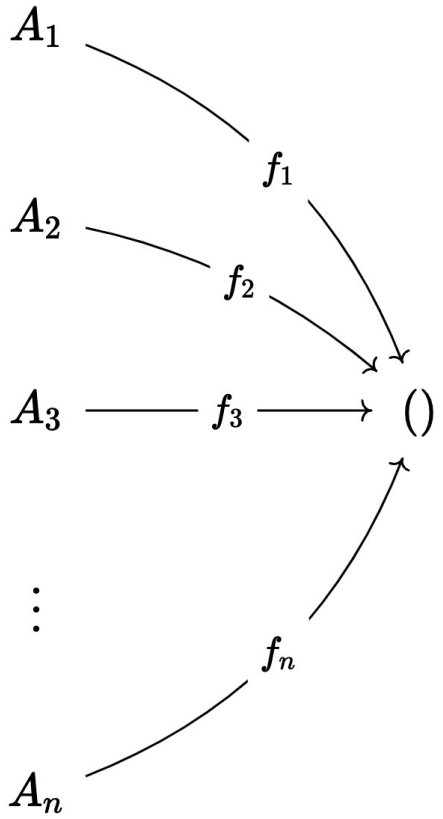
Let us discuss the one-element set (sometimes denoted (), or `unit`)

What can you tell me about morphisms from *any* set A to ()?

We can *uniquely* identify the `unit` like this!
We *only* need to look at the *morphisms*
No need to look at the sets themselves

Categorically, () is called a *terminal object*

We can also define a notion of an *initial object*
(exactly *one* morphism coming out of it!)
What would this be for **Set**?
The *empty set*, $\emptyset$, with the *absurd* morphism

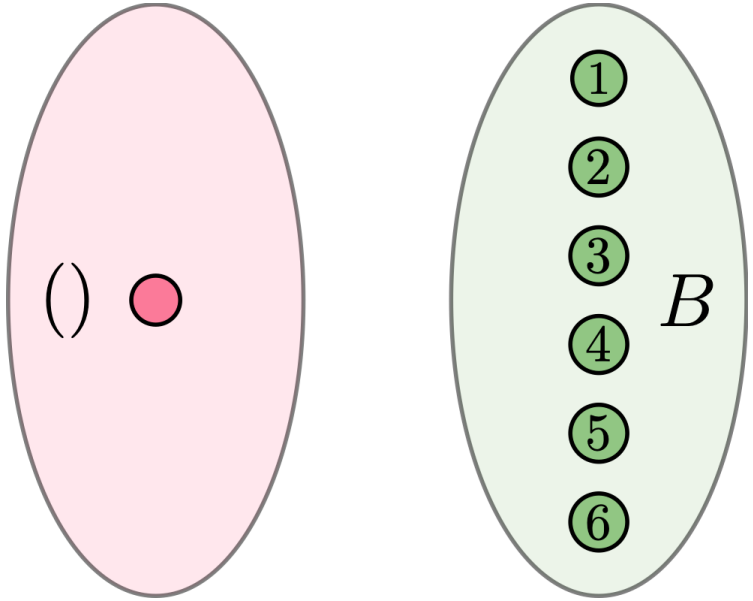


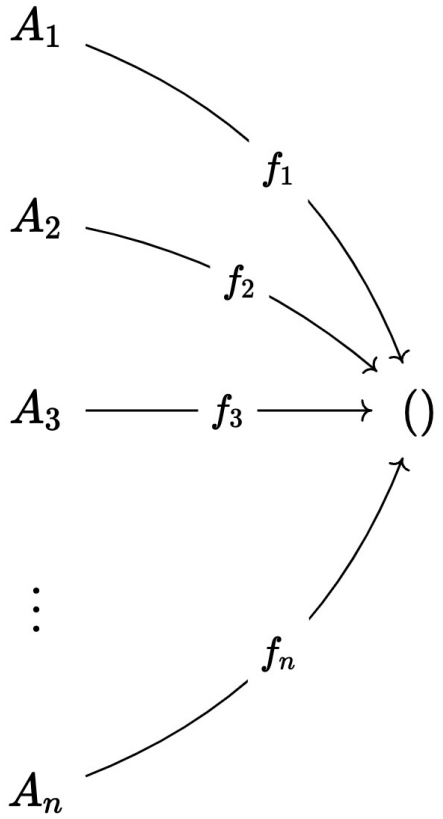
What can we do with the unit?

We have successfully identified the `unit` (up to isomorphism)

What can you tell me about morphisms from () to *any* set B?

(elements of B are numbered purely to illustrate the point)



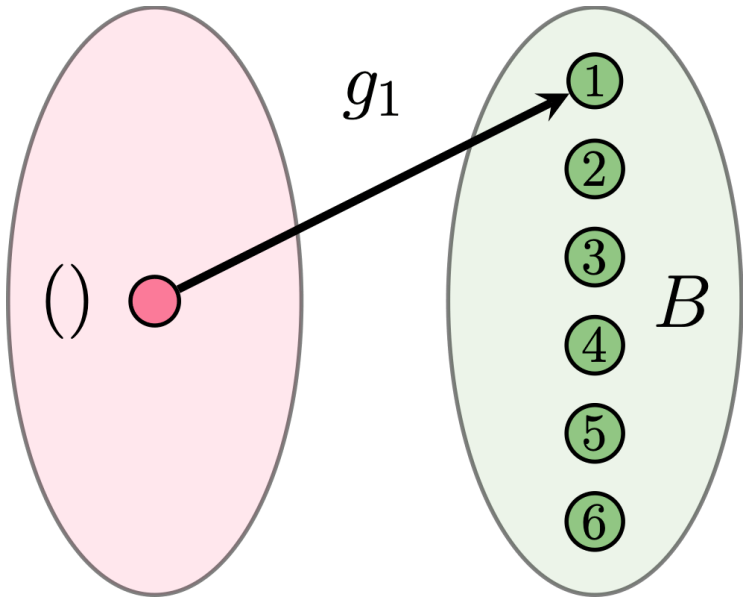


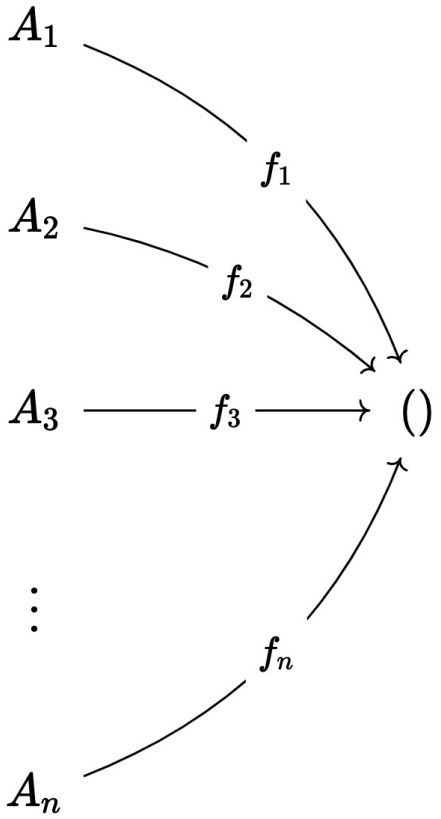
What can we do with the unit?

We have successfully identified the `unit` (up to isomorphism)

What can you tell me about morphisms from () to *any* set B?

They are exactly the *constant functions* on B!
(one per element)





What can we do with the unit?

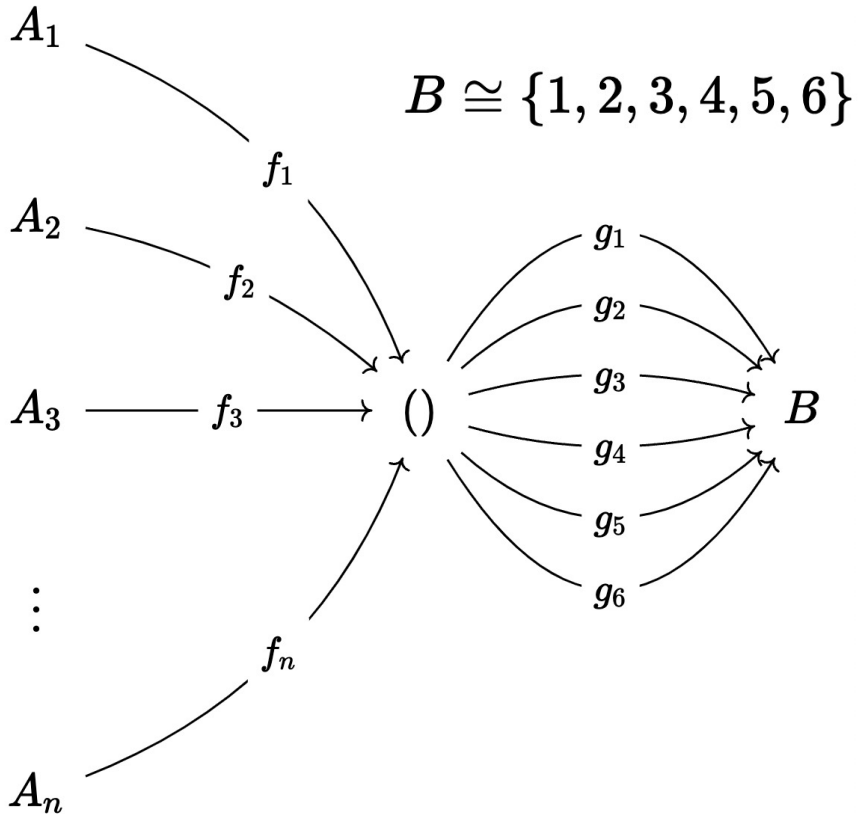
We have successfully identified the `unit` (up to isomorphism)

What can you tell me about morphisms from () to *any* set B?

They are exactly the *constant functions* on B!
(one per element)

Together, they specify *exactly* B's **elements**!

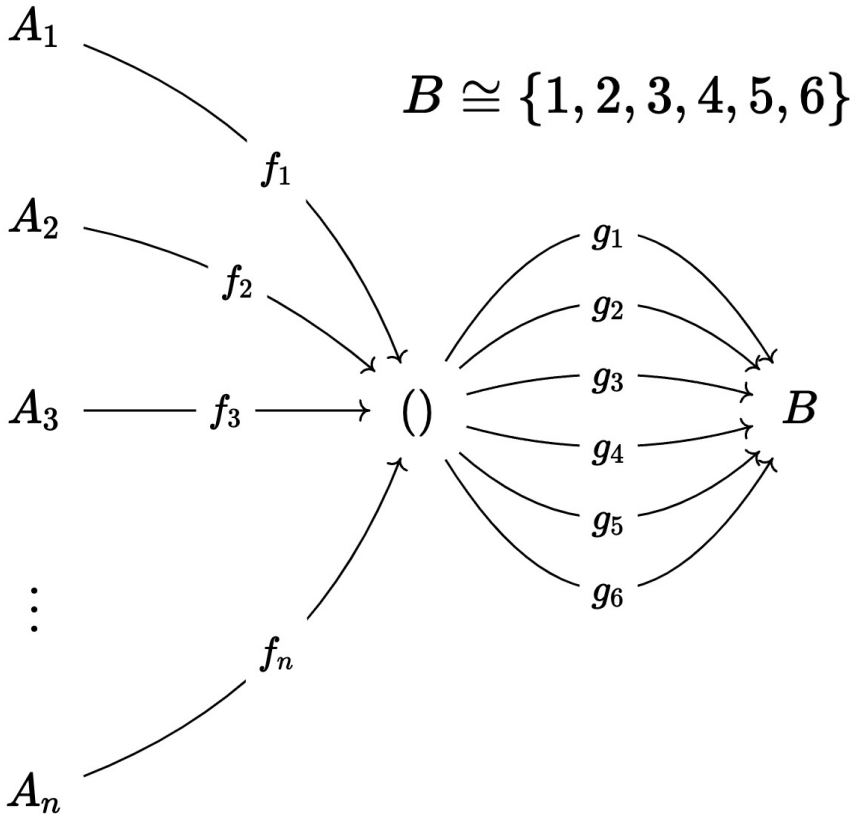
Even though we only looked at morphisms…
We were still able to reconstruct *any* set!
(up to isomorphism)



In summary:

Category theory allows to detect *patterns* in *computation* (morphisms)
Without focussing on the *specifics* of the *data* being computed over
And all this is done *without drastic loss of information*!

Hence, CT is **perfect** for our purposes ☺
(And, generally, a wonderful tool to connect disparate areas of math)



Before we go further

We are now about to dive into a variety of categories of interest

Before we do that, an important disclaimer!
Results from CT are under the assumption of the *category* we work in!
It is often an important first step in *any* CT derivation to establish this.

Results in one category need not naturally transfer to another!

Consider the category **Rel**:

- Objects are *sets* (same as in **Set**!)
- Morphisms are *relations* between them

What are the *initial and terminal objects* in **Rel**?

Before we go further

We are now about to dive into a variety of categories of interest

Before we do that, an important disclaimer!
Results from CT are under the assumption of the *category* we work in!
It is often an important first step in *any* CT derivation to establish this.

Results in one category need not naturally transfer to another!

Consider the category **Rel**:
- Objects are *sets* (same as in **Set**!)
- Morphisms are *relations* between them

What are the *initial and terminal objects* in **Rel**?

In **Rel**, the *empty set*, $\emptyset$, is *both* the initial and terminal object!

Vector spaces as categories

We would also want to reason about *vectors* and how they *transform*
This will be quite important for neural network design!

A standard choice is to use the **Vect** category

Objects in **Vect** are *vector spaces, V*
V is a set of vectors supporting two operations:

- Addition ($\mathbf{v} + \mathbf{w}$) and scalar multiplication ($\lambda\mathbf{v}$)
- Usual axioms (associativity, distributivity…) apply

Morphisms in **Vect** are *linear functions* $f : V \to V'$ between two spaces

The study of **Vect** is thus *exactly* the study of *linear algebra*

Groups as single-object categories

How can we recover *groups* within the categorical realm?

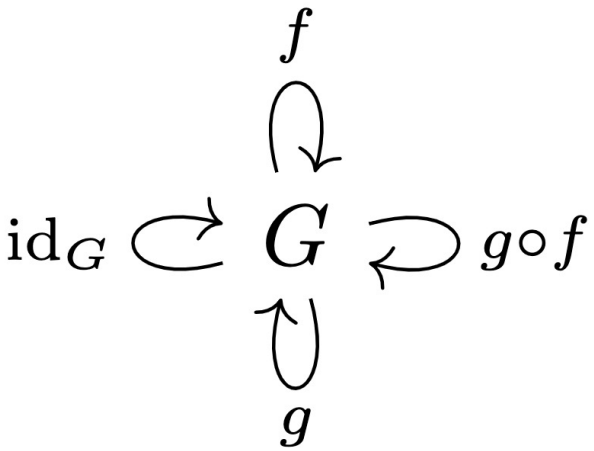
We need to:

- Make *all identities equal* *(to obtain unique* **e***)*
- Make *all* pairs of morphisms *composable* *(to satisfy closure)*
- Make *all* morphisms have an *inverse*

Groups as single-object categories

How can we recover *groups* within the categorical realm?

We need to:
- Make *all identities equal* *(to obtain unique* e*)*
- Make *all* pairs of morphisms *composable* *(to satisfy closure)*
- Make *all* morphisms have an *inverse*



The first two conditions result in the *same* prescription:
A group must be a single-object category!

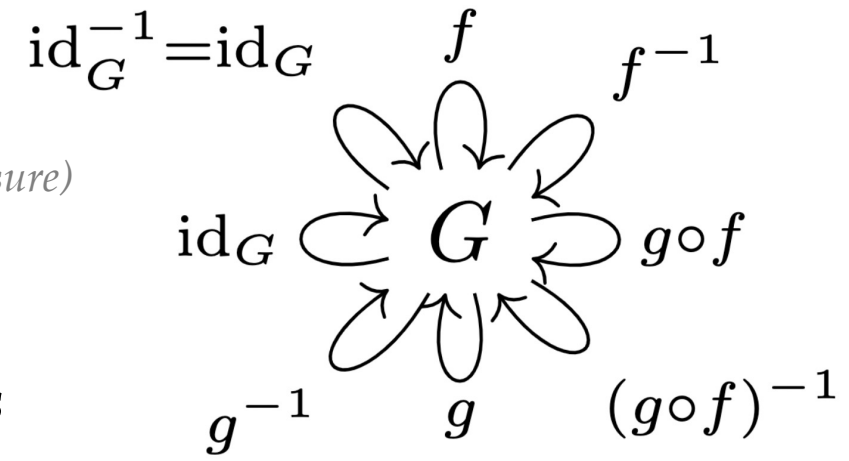
We often denote this category by $\mathcal{B}G$, and its sole object by G
- This means we will have exactly *one* identity, id_G
- And all morphisms $f : G \to G$ will always be compatible to compose

Groups as single-object categories

How can we recover *groups* within the categorical realm?

We need to:

- Make *all identities equal* *(to obtain unique* e*)*
- Make *all* pairs of morphisms *composable* *(to satisfy closure)*
- Make *all* morphisms have an *inverse*



To recover inverses, we need to generalise *bijections*

Morphism $f : A \to B$ is an *isomorphism*, if there exists $g : B \to A$, s.t.:

$$g \circ f = \mathrm{id}_A \qquad f \circ g = \mathrm{id}_B$$

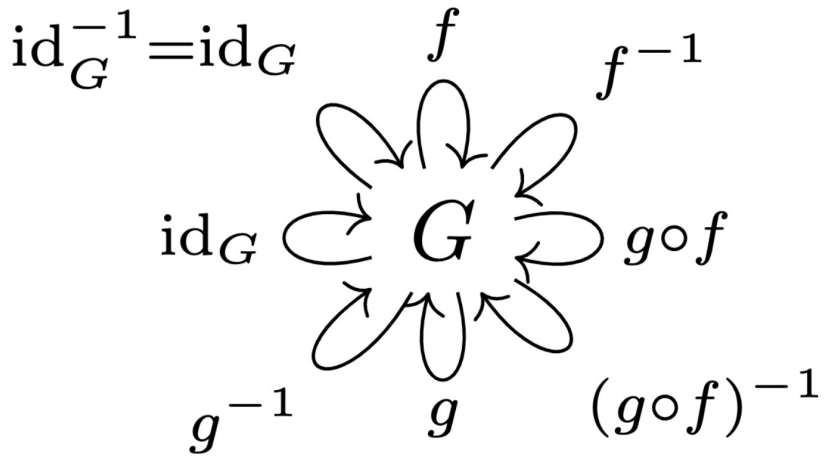
We can now treat g as the "inverse" of f, and we can denote it as f^{-1}

Groups as single-object categories

With inverses added, we can complete our definition:

A group is a category $\mathcal{B}G$ with:

- A single object, G
- All morphisms being *isomorphisms*



Clearly, a very special case of what CT has to offer!

Exercise:
Prove that $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$, using only definitions given so far!

Monoids as single-object categories

What happens if we omit the requirement for all isomorphisms?

We recover a generalisation of groups which are known as *monoids*:

- Operations in a monoid need not be *invertible*
- But they must all *compose*

Often, we will denote such a category $\mathcal{B}M$, and its sole object as M
In summary, a monoid is any category with a single object
This is where the term "monoid" comes from!

(We can also instead omit the full *compositionality* requirement from groups while preserving the rule that all morphisms must be *isomorphisms*, which would give us a *groupoid*)

Functors

The power of CT emerges when we let multiple categories *interact*

Just like we can define *functions* between sets, we can define *functors* between categories: they affect both the objects and morphisms!

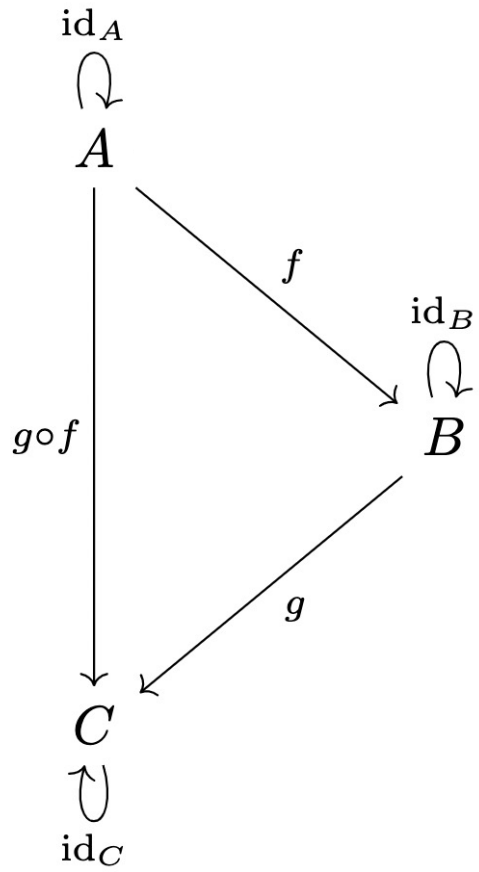
Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Then we can define a functor $F : \mathcal{C} \to \mathcal{D}$
It maps objects in $\mathcal{C}$ to ones in $\mathcal{D}$, and morphisms in $\mathcal{C}$ to ones in $\mathcal{D}$

This must be done in a way that *respects structure* of $\mathcal{C}$!

- $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ for any object A in $\mathcal{C}$
- $F(g \circ f) = F(g) \circ F(f)$ for any composable morphisms f and g in $\mathcal{C}$

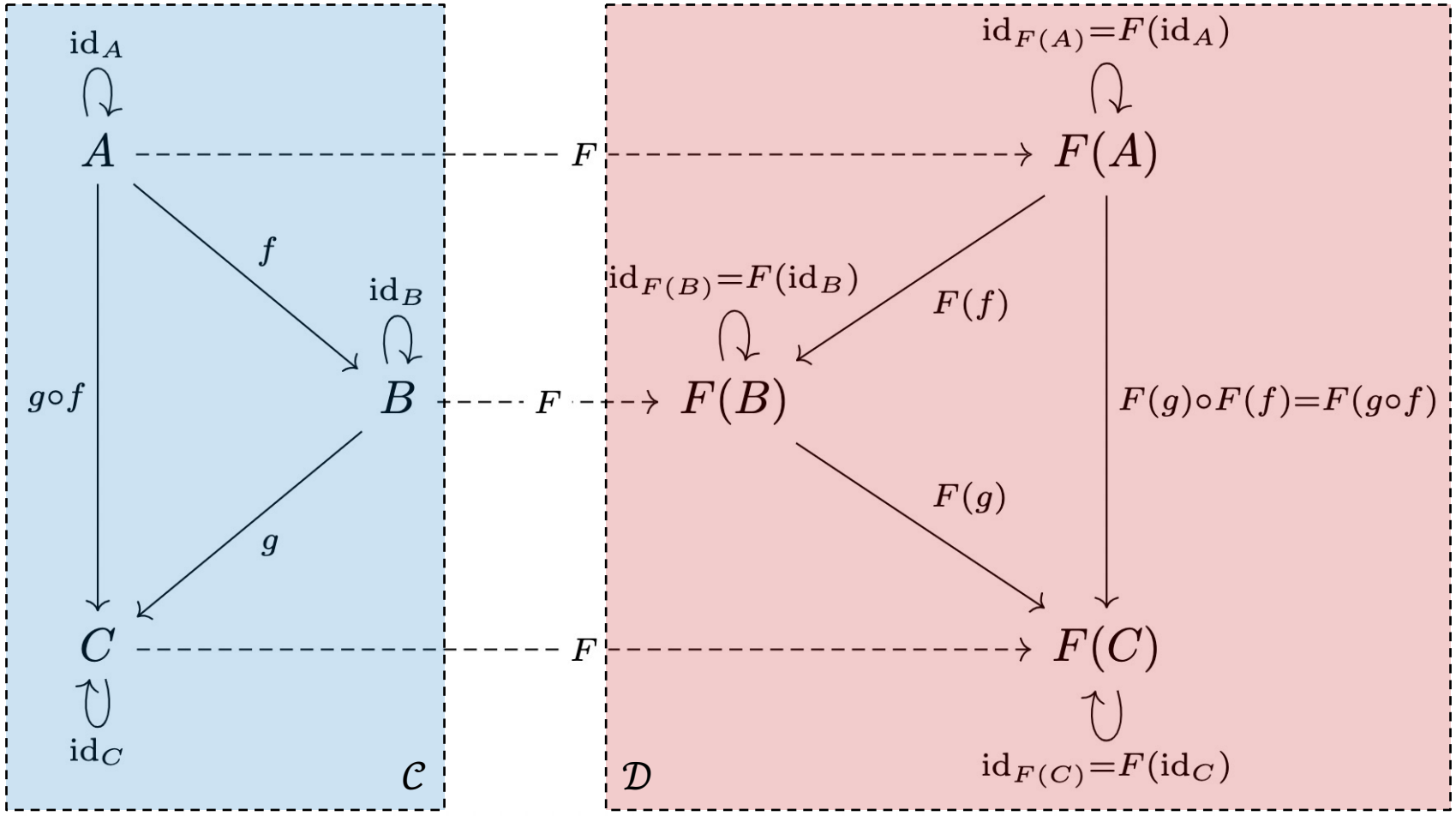
If a functor $F : \mathcal{C} \to \mathcal{C}$ maps a category to itself, it is an *endofunctor*

Functors



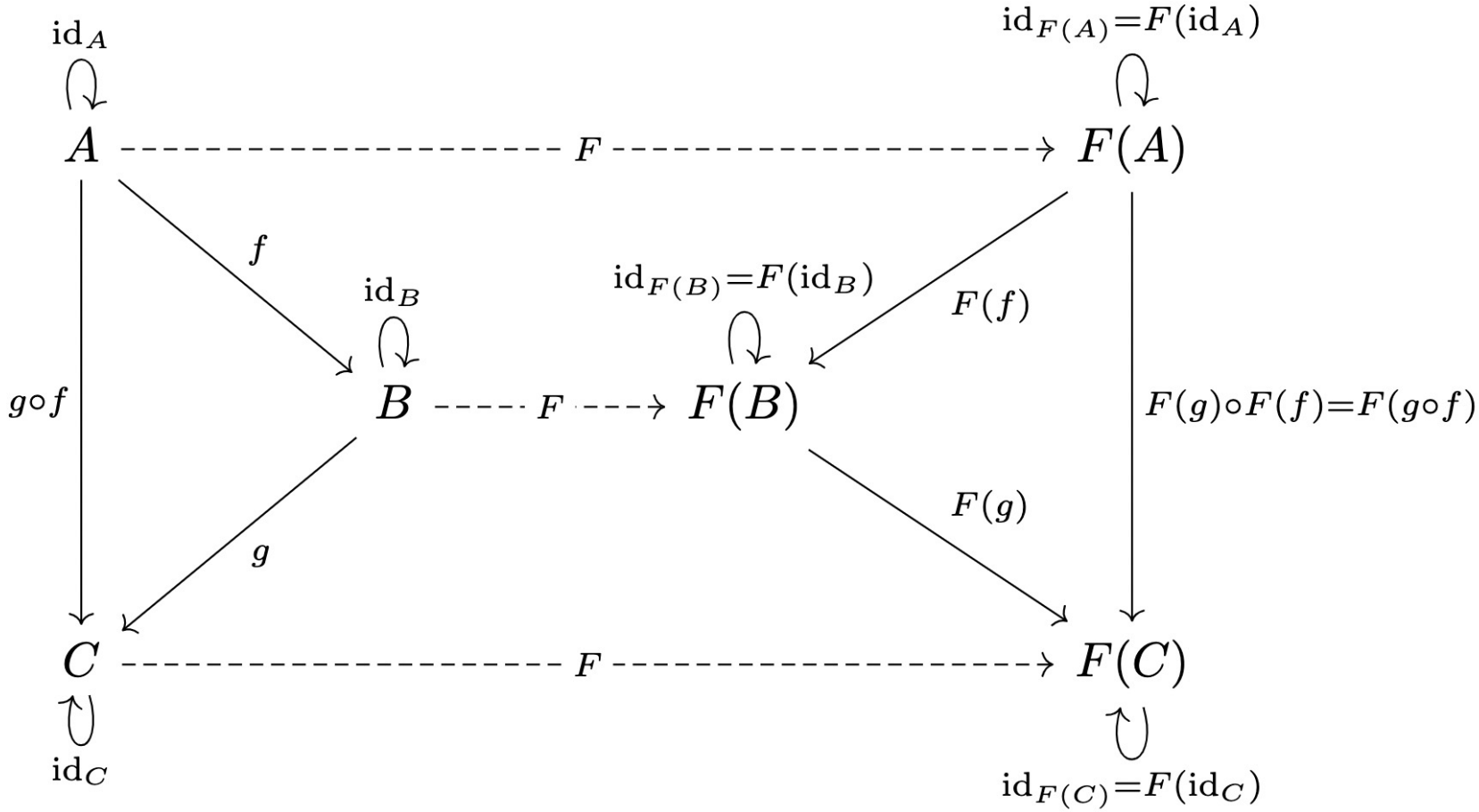
Functors

- $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ for any object A in $\mathcal{C}$
- $F(g \circ f) = F(g) \circ F(f)$ for any composable morphisms f and g in $\mathcal{C}$



Functors

"Every sufficiently good **analogy** is yearning to become a *functor*."
---John Baez



Natural transformations

Much like functors allow us to connect two categories, we can define a concept that that "connects" *two functors*

This is a ***natural transformation***, one of the most powerful CT concepts (Eilenberg and Mac Lane invented functors in order to define naturality!)

Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{C} \to \mathcal{D}$ be two functors
Then, we define a natural transformation, $\alpha : F \Rightarrow G$, as follows:
For every object $X \in \mathcal{C}$, it specifies a morphism $\alpha_X : F(X) \to G(X)$ in $\mathcal{D}$, which commutes with the two functors, like so:

$$\alpha_Y \circ F(f) = G(f) \circ \alpha_X$$

for all morphisms $f : X \to Y$ in $\mathcal{C}$

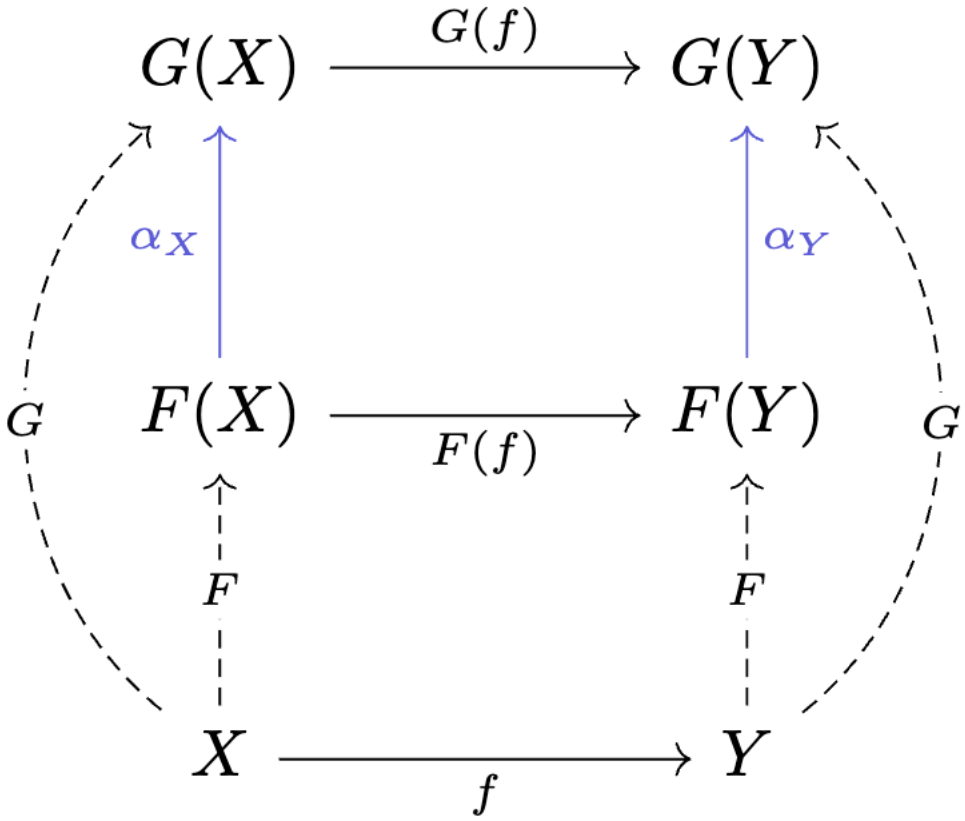
Natural transformations

$$\alpha_Y \circ F(f) = G(f) \circ \alpha_X$$

Importantly, a **natural transformation** defines a separate morphism α_X in $\mathcal{D}$ for **every** object X in $\mathcal{C}$

This morphism is known as the *component* of α in X

(one often just draws the *"naturality square"*)



*A quick recap on **specification***

To specify a *category*, you need to describe:

- The *objects*
- The *morphisms* (+ if not trivial, how they *compose* and what the *identities* are)

To specify a *functor*, you need to describe:

- The two *categories* being related by the functor
- The *mapping* of *objects* between the two categories
- The *mapping* of *morphisms* between the two categories

To specify a *natural transformation*, you need to describe:

- The two *categories* being related by the two functors
- The two *functors* being related by the natural transformation
- The *component morphisms* in the *second* category for objects in the *first*

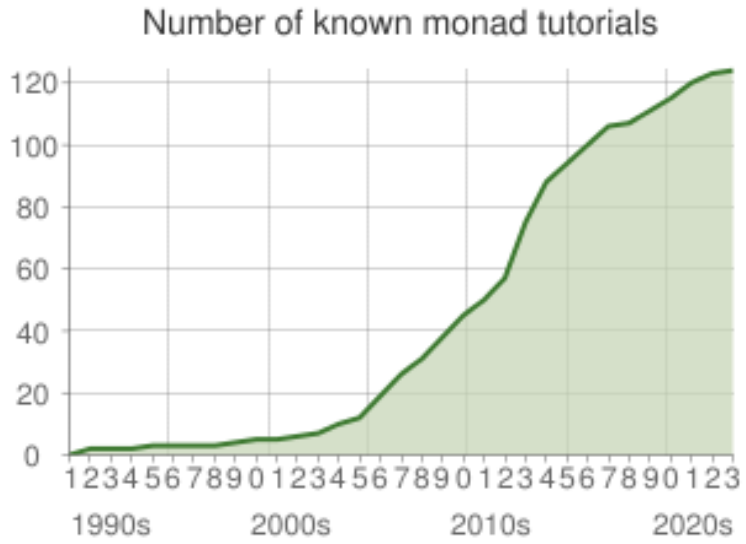
From group actions to monads

To build geometric DL models, we need to reason about *group actions*

We can also make a CT version of group actions, by using a special kind of endofunctor known as a *monad*

A very universal (but tricky) mechanism of expressing *contained computation* in FP

```
class Monad m where
   (>>=)    :: m a -> ( a -> m b) -> m b
   (>>)     :: m a -> m b         -> m b
   return   ::   a                -> m a
```



"All told, a monad in $\mathcal{X}$ is just a *monoid in the category of endofunctors* of $\mathcal{X}$"

Defining monads

A monad in a category $\mathcal{C}$ is a tuple (M, η, μ) comprising:

- an *endofunctor* $M : \mathcal{C} \to \mathcal{C}$
- a *natural transformation* ("`return`") $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow M$
- a *natural transformation* ("`join`") $\mu : M \circ M \Rightarrow M$

Here, $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is an *identity endofunctor*
and $G \circ F : \mathcal{C} \to \mathcal{E}$ is *functor composition* of $F : \mathcal{C} \to \mathcal{D}$ with $G : \mathcal{D} \to \mathcal{E}$

Defining monads

A monad in a category $\mathcal{C}$ is a tuple (M, η, μ) comprising:

- an *endofunctor* $M : \mathcal{C} \to \mathcal{C}$
- a *natural transformation* ("`return`") $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow M$
- a *natural transformation* ("`join`") $\mu : M \circ M \Rightarrow M$

Here, $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is an *identity endofunctor*
and $G \circ F : \mathcal{C} \to \mathcal{E}$ is *functor composition* of $F : \mathcal{C} \to \mathcal{D}$ with $G : \mathcal{D} \to \mathcal{E}$

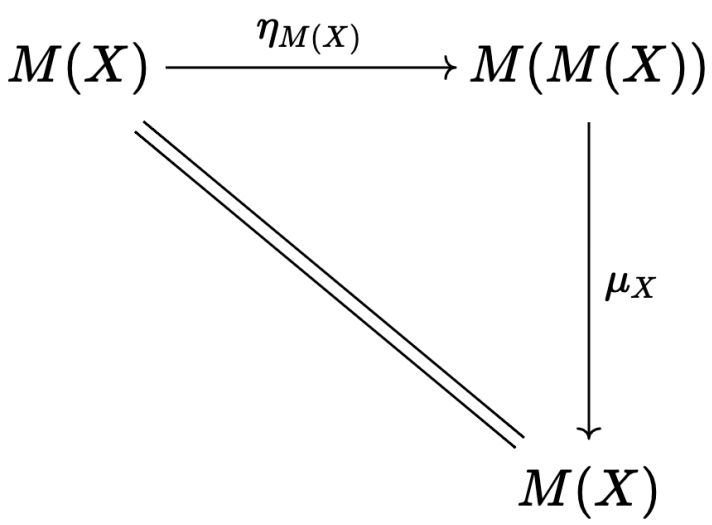
Can interpret M as a *container* specifying *computation* to be carried out
$\eta_X : X \to M(X)$ allows us to *create* a new container with X
$\mu_X : M\big(M(X)\big) \to M(X)$ *executes* one level of nested computation

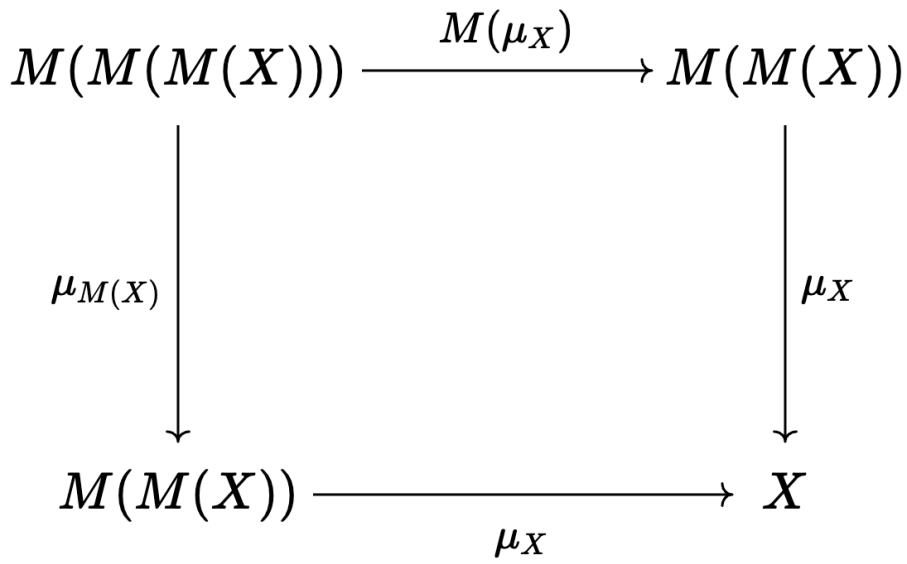
Defining monads

A monad in a category $\mathcal{C}$ is a tuple (M, η, μ) comprising:

- an *endofunctor* $M : \mathcal{C} \to \mathcal{C}$
- a *natural transformation* ("`return`") $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow M$
- a *natural transformation* ("`join`") $\mu : M \circ M \Rightarrow M$

For this to be a monad, the following diagrams in $\mathcal{C}$ must commute:





Example: Power set monad

A monad in a category $\mathcal{C}$ is a tuple (M, η, μ) comprising:

- an *endofunctor* $M : \mathcal{C} \to \mathcal{C}$
- a *natural transformation* ("`return`") $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow M$
- a *natural transformation* ("`join`") $\mu : M \circ M \Rightarrow M$

A standard example is the *power set monad* $\mathcal{P}$ on **Set**

$\mathcal{P}(X)$ maps sets X to their power sets, $\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$
$\mathcal{P}(X)$ maps functions $f : X \to Y$ to functions between their power sets, $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y)$, by applying f to each relevant subset

$\eta_X : X \to \mathcal{P}(X)$ wraps the power set computation around the set X
$\mu_X : \mathcal{P}\big(\mathcal{P}(X)\big) \to \mathcal{P}(X)$ computes the *union* of a set of sets

Group actions as monads

Now we describe the *group action monad* on **Set**
Formally, it is a tuple $(\mathfrak{G}\times -, \eta, \mu)$, where:

Group actions as monads

Now we describe the *group action monad* on **Set**
Formally, it is a tuple $(\mathfrak{G}\times -, \eta, \mu)$, where:

$\mathfrak{G}\times -$: **Set** $\to$ **Set** is an *endofunctor* mapping sets X to products $\mathfrak{G}\times X$ and functions $f : X \to Y$ to $\mathfrak{G}\times f : \mathfrak{G}\times X \to \mathfrak{G}\times Y$, mapping $(\mathfrak{g}, x) \to \big(\mathfrak{g}, f(x)\big)$

Group actions as monads

Now we describe the *group action monad* on **Set**
Formally, it is a tuple $(\mathfrak{G}\times -, \eta, \mu)$, where:

$\mathfrak{G}\times -$: **Set** $\to$ **Set** is an *endofunctor* mapping sets X to products $\mathfrak{G}\times X$ and functions $f : X \to Y$ to $\mathfrak{G}\times f : \mathfrak{G}\times X \to \mathfrak{G}\times Y$, mapping $(\mathfrak{g}, x) \to \big(\mathfrak{g}, f(x)\big)$

$\eta : \mathrm{id}_{\mathbf{Set}} \Rightarrow \mathfrak{G}\times -$ is a *natural transformation*, whose components $\eta_X : X \to \mathfrak{G}\times X$ map elements $x \in X$ to $(\mathfrak{e}, X)$, where $\mathfrak{e} \in \mathfrak{G}$ is the identity

Group actions as monads

Now we describe the *group action monad* on **Set**
Formally, it is a tuple $(\mathfrak{G}\times -, \eta, \mu)$, where:

$\mathfrak{G}\times -$: **Set** $\to$ **Set** is an *endofunctor* mapping sets X to products $\mathfrak{G}\times X$ and functions $f : X \to Y$ to $\mathfrak{G}\times f : \mathfrak{G}\times X \to \mathfrak{G}\times Y$, mapping $(\mathfrak{g}, x) \to \big(\mathfrak{g}, f(x)\big)$

$\eta : \mathrm{id}_{\mathbf{Set}} \Rightarrow \mathfrak{G}\times -$ is a *natural transformation*, whose components $\eta_X : X \to \mathfrak{G}\times X$ map elements $x \in X$ to $(\mathfrak{e}, X)$, where $\mathfrak{e} \in \mathfrak{G}$ is the identity

$\mu : \mathfrak{G}\times\mathfrak{G}\times - \Rightarrow \mathfrak{G}\times -$ is a *natural transformation*, whose components $\mu_X : \mathfrak{G}\times\mathfrak{G}\times X \to \mathfrak{G}\times X$ map $\big(\mathfrak{g}, (\mathfrak{h}, x)\big) \to (\mathfrak{g}\mathfrak{h}, x)$

Group actions as monads

Now we describe the *group action monad* on **Set**
Formally, it is a tuple $(\mathfrak{G}\times -, \eta, \mu)$, where:

$\mathfrak{G}\times -$: **Set** $\to$ **Set** is an *endofunctor* mapping sets X to products $\mathfrak{G}\times X$ and functions $f : X \to Y$ to $\mathfrak{G}\times f : \mathfrak{G}\times X \to \mathfrak{G}\times Y$, mapping $(\mathfrak{g}, x) \to \big(\mathfrak{g}, f(x)\big)$

$\eta : \mathrm{id}_{\mathbf{Set}} \Rightarrow \mathfrak{G}\times -$ is a *natural transformation*, whose components $\eta_X : X \to \mathfrak{G}\times X$ map elements $x \in X$ to $(\mathfrak{e}, X)$, where $\mathfrak{e} \in \mathfrak{G}$ is the identity

$\mu : \mathfrak{G}\times\mathfrak{G}\times - \Rightarrow \mathfrak{G}\times -$ is a *natural transformation*, whose components $\mu_X : \mathfrak{G}\times\mathfrak{G}\times X \to \mathfrak{G}\times X$ map $\big(\mathfrak{g}, (\mathfrak{h}, x)\big) \to (\mathfrak{g}\mathfrak{h}, x)$

Monads tell *how* group actions compose, but do not let us *execute* them
To do this, we will need some more machinery ☺

10. CATEGORICAL DEEP LEARNING

An algebraic theory of architectures

Petar Veličković

*Algebras as a model for **execution***

Objects allowing us to *execute* computation stored inside objects are generally known as *algebras*

Computations may execute *differently* based on the underlying *object* e.g. *permutation actions* transform *node features* and *adjacency matrices* differently -- $\mathbf{PX}$ vs. $\mathbf{PAP}^\top$!

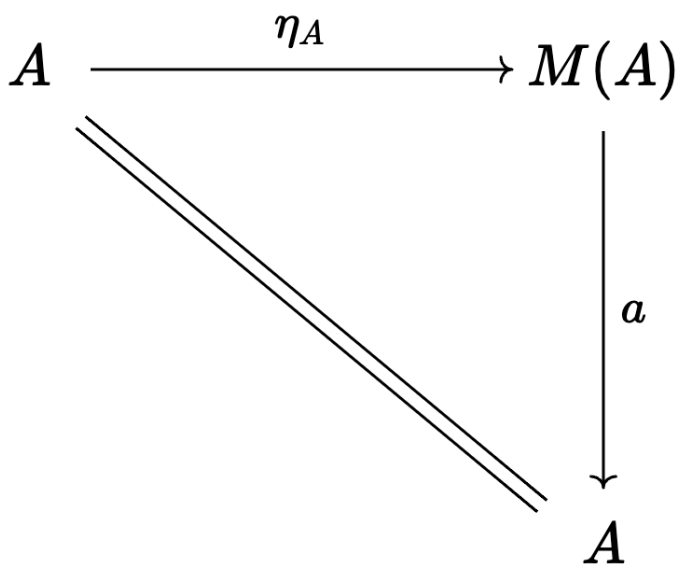
As a result, we typically define algebra *relative to an object*

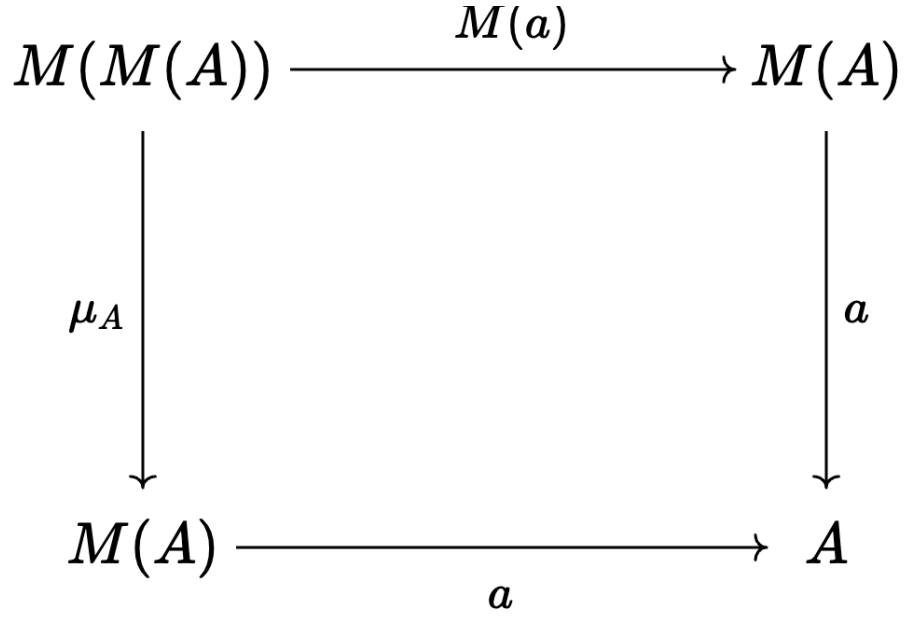
Monad algebras

An *algebra* for a *monad* (M, η, μ) in $\mathcal{C}$ is a pair (A, a) comprising:

- A *carrier object*, $A \in \mathcal{C}$
- A *structure map*, $a : M(A) \to A$, which is a morphism in $\mathcal{C}$

such that the following diagrams in $\mathcal{C}$ commute:





*Group actions as **algebras** for a monad*

Now we can use this construct to *execute* group actions:
Group actions of $\mathfrak{G}$ are *algebras* for the *group action monad* $\mathfrak{G}\times-$

Simple example: the *translation group,* $\mathfrak{G} = \mathbb{Z}_w \times \mathbb{Z}_h$ acting on a $w \times h$ grid of pixels – the carrier object for its algebra will be $\mathbb{R}^{w \times h}$ in this case

The structure map of the algebra for these translations is specified as:

$$\triangleright : \mathbb{Z}_w \times \mathbb{Z}_h \times \mathbb{R}^{w \times h} \to \mathbb{R}^{w \times h}$$

and defined by

$$\big((i', j') \triangleright \mathbf{X}\big)_{i,j} = x_{i-i', j-j'}$$

Equivariant layers as monad algebra ***homomorphisms***

Monad algebras (and their *carriers*) allow us to describe the sets of possible (group-transformed) *inputs* and *outputs* for a neural network

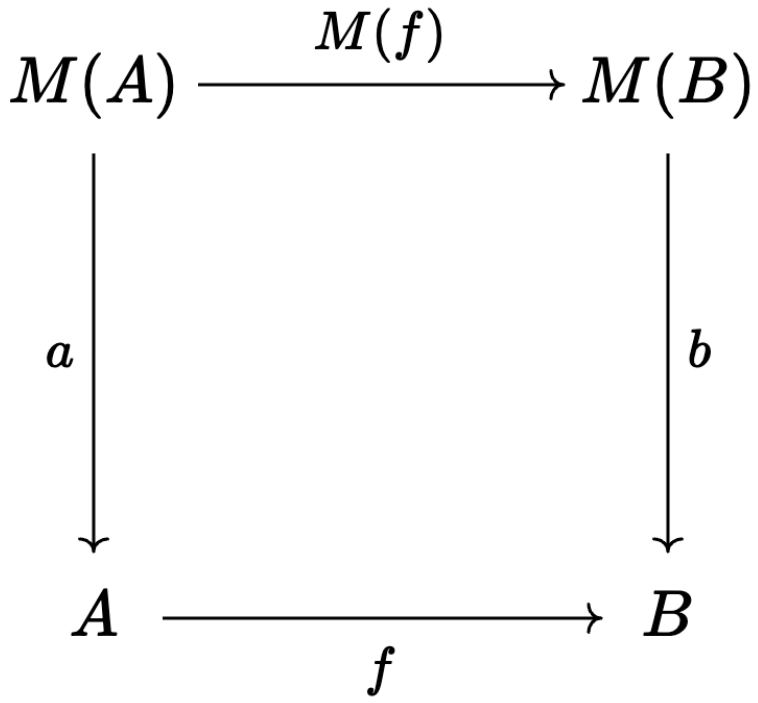
To actually specify an equivariant neural network layer, we need to talk about *transformations* between two carrier objects

This is a *monad algebra* ***homomorphism,*** which we describe next

Monad algebra ***homomorphisms***

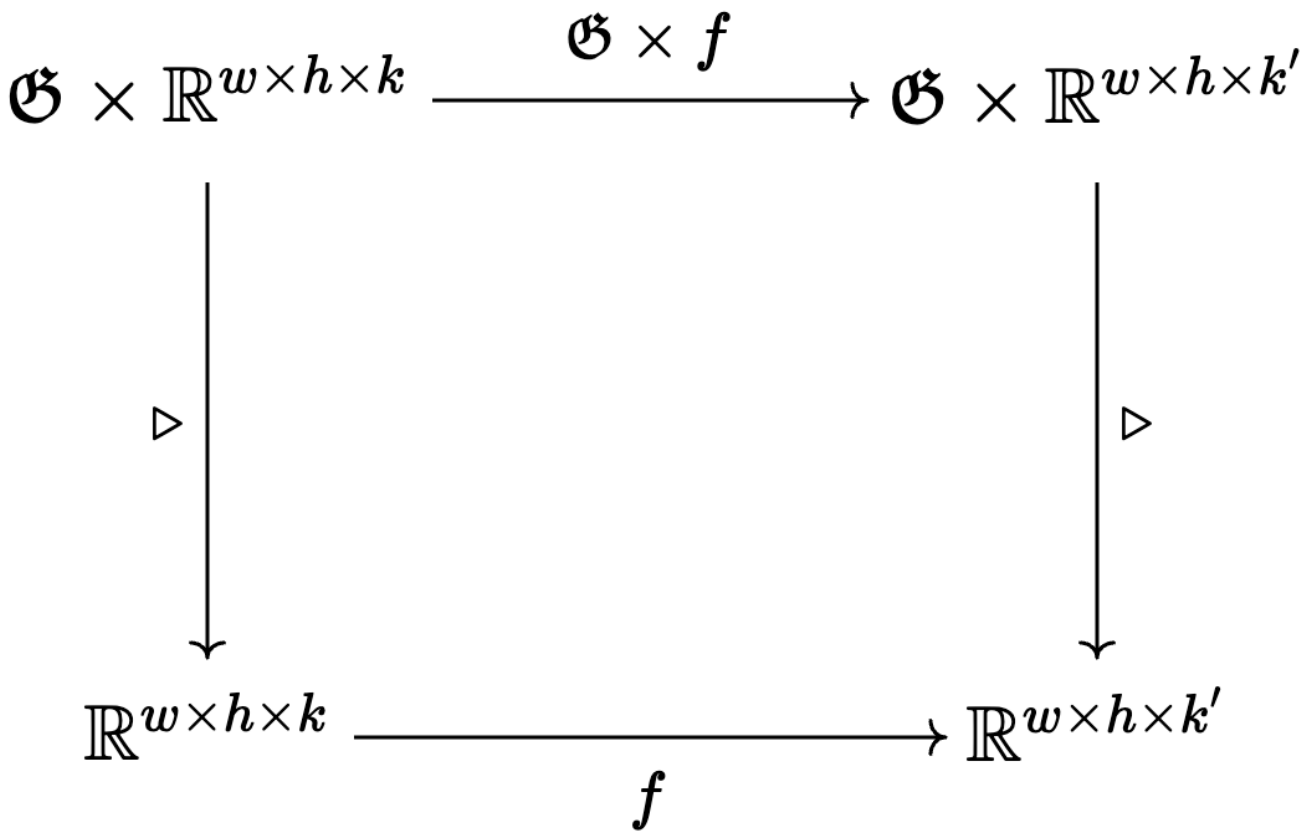
Let (M, η, μ) be a *monad* in $\mathcal{C}$, and let $(A, a), (B, b)$ be two of its *algebras*.

A *monad algebra homomorphism* is a morphism $f : A \to B$ in $\mathcal{C}$
which makes the following diagram in $\mathcal{C}$ commutes:



Monad algebra homomorphisms are equivariant maps

As a simple illustration, let's take the *translation group algebra* with $\triangleright$, over two *carriers* (different dimensional features stored in each pixel)



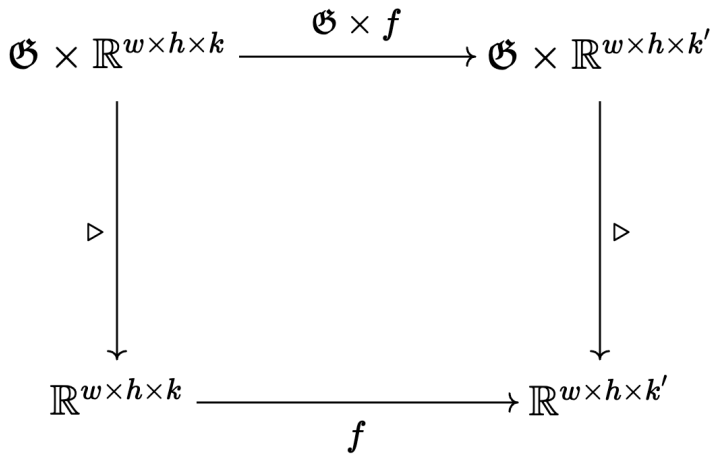
Monad algebra homomorphisms are equivariant maps

As a simple illustration, let's take the *translation group algebra* with $\triangleright$, over two *carriers* (different dimensional features stored in each pixel)

The commutativity of this diagram induces a constraint (for all $\mathfrak{g}, \mathbf{X}$):

$$f(\mathfrak{g} \triangleright \mathbf{X}) = \mathfrak{g} \triangleright f(\mathbf{X})$$

which is exactly *shift equivariance*!



Monad algebra homomorphisms are equivariant maps

As a simple illustration, let's take the *translation group algebra* with $\triangleright$, over two *carriers* (different dimensional features stored in each pixel)

The commutativity of this diagram induces a constraint (for all $\mathfrak{g}, \mathbf{X}$):

$$f(\mathfrak{g} \triangleright \mathbf{X}) = \mathfrak{g} \triangleright f(\mathbf{X})$$

which is exactly *shift equivariance*!

The idea generalises to *any* pair of algebras $(A, a), (B, b)$ for $\mathfrak{G}\times-$:

$$f\big(a(\mathfrak{g}, x)\big) = b\big(\mathfrak{g}, f(x)\big)$$

(successfully deriving *all* of geometric deep learning ☺)

Equivariant maps

are

group action monad

algebra homomorphisms

Equivariant maps

are

group action monad

algebra homomorphisms

Exercise: How to specify **invariant** maps?

Resolving the monad algebra homomorphism constraints

Equations such as

$$f(\mathfrak{g} \triangleright \mathbf{X}) = \mathfrak{g} \triangleright f(\mathbf{X})$$

only give us *constraints*; to solve them, we need to explicitly materialise our diagram in a category such as **Vect**

(generally, for neural network layers we might prefer the category **Aff** of *affine* spaces)

Resolving the monad algebra homomorphism constraints

Equations such as

$$f(\mathfrak{g} \triangleright \mathbf{X}) = \mathfrak{g} \triangleright f(\mathbf{X})$$

only give us *constraints*; to solve them, we need to explicitly materialise our diagram in a category such as **Vect**

For example, if we assume $f_{\mathbf{W}} : \mathbb{R}^{w \times h \times k} \to \mathbb{R}^{w \times h \times k'}$ is a morphism in **Vect**, it must be expressible as $f_{\mathbf{W}}(\mathbf{X}) = \mathbf{X}\mathbf{W}$ for some $\mathbf{W} \in \mathbb{R}^{k \times k'}$

The constraint then gives us *weight sharing instructions* in $\mathbf{W}$, which lead to the usual *circulant matrix* solution for translation
Exercise: Derive this for $w = 2, h = 1$.

Endofunctor *algebras*

We have successfully rediscovered *all of geometric deep learning*

But, *nowhere* in the definition of monad algebra homomorphism did we need to use the monad axioms!

Hence, we can *generalise* this concept to arbitrary *endofunctors*!
This will be our first step *beyond* group-equivariant models

Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor on $\mathcal{C}$. An ***endofunctor*** *algebra* for F is a pair (A, a) where $A \in \mathcal{C}$ and $a : F(A) \to A$ is a *structure map* in $\mathcal{C}$

NB. There are *no* structural constraints to be satisfied this time!
Examples of endofunctor algebras are *abundant* in *computer science*

Example: ***Lists*** *as endofunctor algebras*

Let A be a set, and $1 + A\times - : \mathbf{Set} \to \mathbf{Set}$ be an endofunctor

Then, let $\mathrm{List}(A)$ be the set of all *lists* of elements coming from A
And $[\mathrm{Nil}, \mathrm{Cons}] : 1 + A\times\mathrm{List}(A) \to \mathrm{List}(A)$ be a pair of maps specified as

```
data List a = Nil
            | Cons a (List a)
```

Then $(\mathrm{List}(A), [\mathrm{Nil}, \mathrm{Cons}])$ is an *algebra* for this endofunctor!

It describes lists *inductively*, as either an *empty list* or a *concatenation* of an element of type A with another list---typical in *functional languages*

Example: ***Binary trees*** *as endofunctor algebras*

Let A be a set, and $A + (-)^2 : \mathbf{Set} \to \mathbf{Set}$ be an endofunctor

Then, let $\mathrm{Tree}(A)$ be the set of all *binary trees* with A-labelled leaves
And $[\mathrm{Leaf}, \mathrm{Node}] : A + \mathrm{Tree}(A)^2 \to \mathrm{Tree}(A)$ be a pair of maps as in:

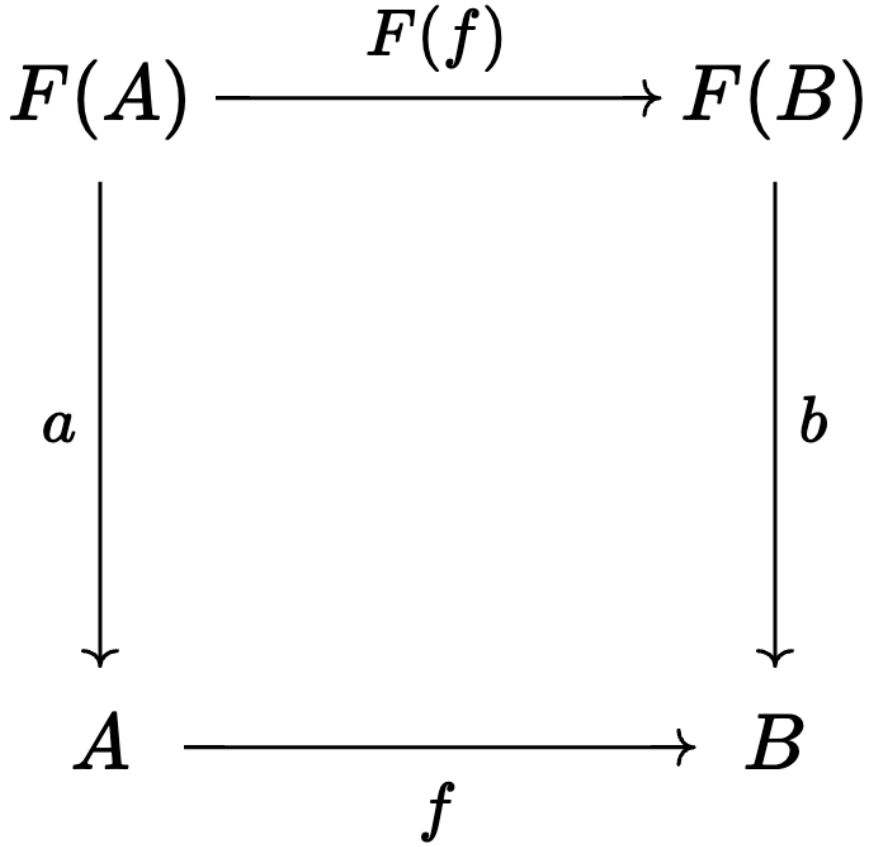
```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Then $(\mathrm{Tree}(A), [\mathrm{Leaf}, \mathrm{Node}])$ is an *algebra* for this endofunctor!

It describes trees *inductively*, as either a labelled *leaf node* or two *subtrees*
Exercise: How would you specify trees with labels in *all* nodes?
Exercise: How about trees with a *variable branching factor*?

Endofunctor algebra ***homomorphisms***

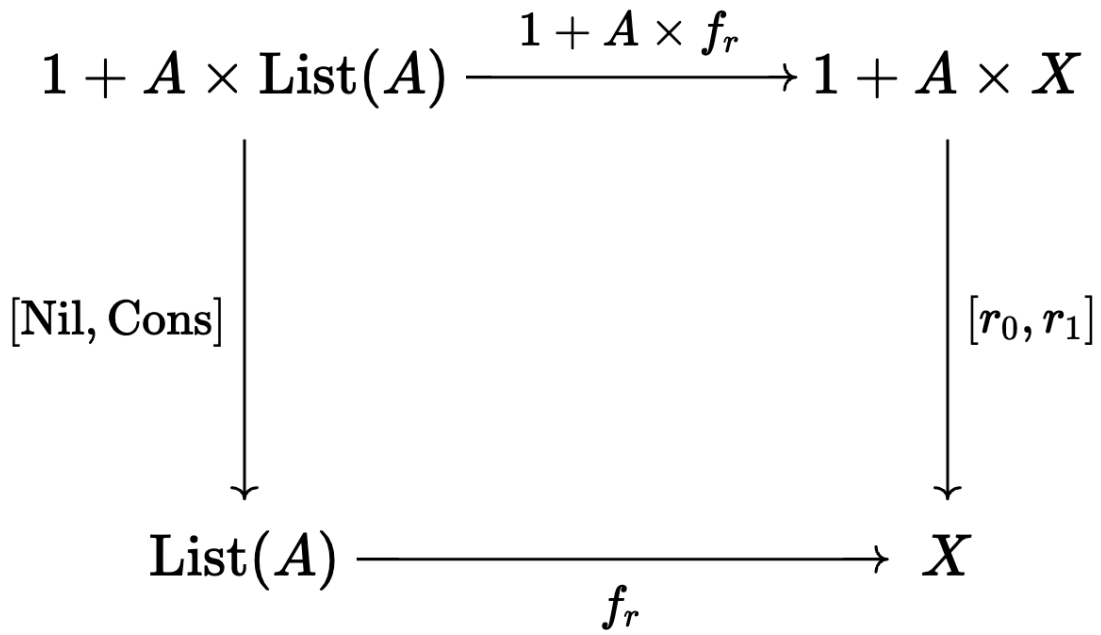
Now we can describe *homomorphisms* between two *endofunctor algebras*
The diagram is the same as for *monad* algebra homomorphisms!



Endofunctor algebra ***homomorphisms***

Now we can describe *homomorphisms* between two *endofunctor algebras*

A classic example are *folds* over a list
We are mapping from $(\text{List}(A), [\text{Nil}, \text{Cons}])$ to some $(X, [r_0, r_1])$
Both are algebras for the endofunctor $1 + A \times - : \mathbf{Set} \to \mathbf{Set}$



Endofunctor algebra ***homomorphisms***

Now we can describe *homomorphisms* between two *endofunctor algebras*

A classic example are *folds* over a list
We are mapping from $(\text{List}(A), [\text{Nil}, \text{Cons}])$ to some $(X, [r_0, r_1])$
Both are algebras for the endofunctor $1 + A\times - : \mathbf{Set} \to \mathbf{Set}$

We recover *list equivariance constraints!*

$$f_r(\text{Nil}) = r_0(\cdot)$$
$$f_r\big(\text{Cons}(h, t)\big) = r_1\big(h, f_r(t)\big)$$

which intuitively specify how a neural network layer f_r aligning to a *fold* operation specified by $[r_0, r_1]$ should look like

Endofunctor algebra ***homomorphisms***

Now we can describe *homomorphisms* between two *endofunctor algebras*

A classic example are *folds* over a list
We are mapping from $(\mathrm{List}(A), [\mathrm{Nil}, \mathrm{Cons}])$ to some $(X, [r_0, r_1])$
Both are algebras for the endofunctor $1 + A\times - : \mathbf{Set} \to \mathbf{Set}$

We recover *list equivariance constraints!*

$$f_r(\mathrm{Nil}) = r_0(\cdot)$$
$$f_r\big(\mathrm{Cons}(h, t)\big) = r_1\big(h, f_r(t)\big)$$

Note, this is similar in spirit to group equivariance, but the operations on the LHS are **not** symmetries! Cons does *not* leave a list *unchanged*.

Endofunctor algebra ***homomorphisms***

Now we can describe *homomorphisms* between two *endofunctor algebras*

A classic example are *folds* over a list
We are mapping from $(\text{List}(A), [\text{Nil}, \text{Cons}])$ to some $(X, [r_0, r_1])$
Both are algebras for the endofunctor $1 + A\times - : \mathbf{Set} \to \mathbf{Set}$

We recover *list equivariance constraints!*

$$f_r(\text{Nil}) = r_0(\cdot)$$
$$f_r\big(\text{Cons}(h, t)\big) = r_1\big(h, f_r(t)\big)$$

Exercise: what would these constraints look like for *binary tree* folds?

Endofunctor ***coalgebras***

Our generalisation also allows us to study the *dual* concept of an endofunctor *coalgebra* (where the morphisms are reversed)

Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$, a *coalgebra* for this endofunctor is a pair (A, a) with $A \in \mathcal{C}$ and $a : A \to F(A)$ its structure map in $\mathcal{C}$

Intuitively, while the execution of algebras guarantees *termination*, the execution of coalgebras guarantees *productivity* – they may perform potentially *infinite* computations, and are great to represent *automata*

Example: ***Mealy machines*** *as endofunctor coalgebras*

Let O, I be sets of possible *outputs* and *inputs*
We consider the endofunctor $(I \to O \times -) : \mathbf{Set} \to \mathbf{Set}$

Then the set $\text{Mealy}_{O,I}$ of *Mealy machines* with O-outputs and I-inputs together with the map $\text{next} : \text{Mealy}_{O,I} \to \left(I \to O \times \text{Mealy}_{O,I}\right)$ specified as

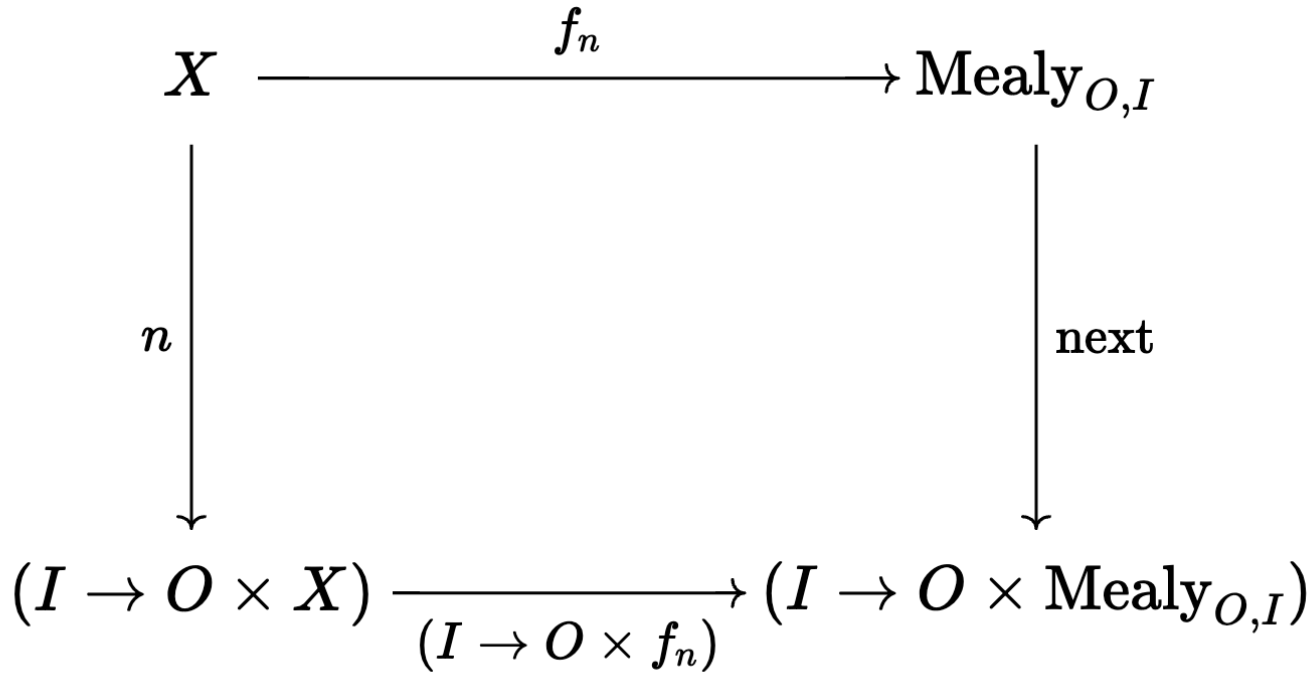
```
data Mealy o i = MkMealy {
  next :: i -> (o, Mealy o i)
}
```

is a *coalgebra* for this endofunctor.
This describes Mealy machines *coinductively*, as a system that, given an input, produces an output and another (transitioned) Mealy machine

*Endofunctor **coalgebra** homomorphisms*

Much like with lists, we can now describe *unfolds* of Mealy machines as *homomorphisms* between two *endofunctor coalgebras*

We map from $\left(\text{Mealy}_{O,I}, \text{next}\right)$ to some coalgebra (X, n)
Both are *coalgebras* for the endofunctor $(I \to O \times -) : \mathbf{Set} \to \mathbf{Set}$



*Endofunctor **coalgebra** homomorphisms*

Much like with lists, we can now describe *unfolds* of Mealy machines as *homomorphisms* between two *endofunctor coalgebras*

We map from $\left(\text{Mealy}_{O,I}, \text{next}\right)$ to some coalgebra (X, n)
Both are *coalgebras* for the endofunctor $(I \to O \times -) : \mathbf{Set} \to \mathbf{Set}$

This yields *equivariance constraints* for a *Mealy machine*:

$$n(x)(i)_1 = \text{next}\Big(f_n(x)\Big)(i)_1$$
$$f_n(n(x)(i)_2) = \text{next}\Big(f_n(x)\Big)(i)_2$$

This also *generalises* group equivariance, with the transformations corresponding to a *possibly infinite automaton* that need *not* be *invertible*!

*Endofunctor **coalgebra** homomorphisms*

Much like with lists, we can now describe *unfolds* of Mealy machines as *homomorphisms* between two *endofunctor coalgebras*

We map from $\left(\text{Mealy}_{O,I}, \text{next}\right)$ to some coalgebra (X, n)
Both are *coalgebras* for the endofunctor $(I \to O \times -) : \mathbf{Set} \to \mathbf{Set}$

This yields *equivariance constraints* for a *Mealy machine*:

$$n(x)(i)_1 = \text{next}\Big(f_n(x)\Big)(i)_1$$
$$f_n(n(x)(i)_2) = \text{next}\Big(f_n(x)\Big)(i)_2$$

Exercise: What would the constraints look like for a **Moore** machine?

Latest work: `map` / `filter` *invariants*

In recent lectures, we've noticed how modern *neural networks*, without any steering, learn how to specialise in specific directions (e.g. *positional* and *semantic* heads in the p-RoPE study…)

If a model *tends* towards such behaviours it means they are **important**
Yet, we know such learned behaviours won't easily stay *robust*

The answer? Could we seek out functions that are "by design" positional/semantic/… for **all** sizes (and eventually use them?)

Latest work: general-purpose regularities

In recent lectures, we've noticed how modern *neural networks*, without any steering, learn how to specialise in specific directions (e.g. *positional* and *semantic* heads in the p-RoPE study…)

If a model *tends* towards such behaviours it means they are **important**
Yet, we know such learned behaviours won't easily stay *robust*

The answer? Could we seek out functions that are "by design" positional/semantic/… for **all** sizes (and eventually use them?)

`map` / `filter` *invariants*

Two standard FP functions we can leverage:

```
map     ::    (a -> b) -> [a] -> [b]
filter  ::    (a -> Bool) -> [a] -> [a]
```

Functions that are `map` invariant are independent of the *data contents* and are well understood (see e.g. Wadler's "Theorems for Free")

What about `filter` invariants?

They offer us a particular form of *length generalisation*:
"This function will still work even if I remove some of the elements"

Yet, until recently, much less well understood ☺

(N)FIs

Definition 1. *A monomorphic function* $f : [a] \to [a]$ *is filter invariant (FI) iff*

$$(\forall p : a \to \mathsf{Bool}) \;\; f \circ \mathsf{filter}\ p = \mathsf{filter}\ p \circ f$$

A polymorphic function $f : (\forall a)[a] \to [a]$ *is naturally filter invariant (NFI) iff it is filter invariant at every type and is a natural transformation, ie*

$$(\forall h : a \to b) \;\; f \circ \mathsf{map}\ h = \mathsf{map}\ h \circ f$$

Here are some examples: reverse is NFI. Any sorting function is FI. Filter is FI for any predicate. The identity is NFI. The composition of FIs (resp NFIs) is FI (resp NFI). The following are NFI:

$$\begin{aligned}
\mathsf{rpt} &:: \mathsf{Int} \to a \to [a] \\
\mathsf{rpt}\ n\ x &= [x \mid i \leftarrow [0..n-1]] \\
\mathsf{inflate} &:: \mathsf{Int} \to [a] \to [a] \\
\mathsf{inflate}\ n &= \mathsf{fold}\ (\lambda x\ xs \to \mathsf{rpt}\ n\ x\ +\!+\ xs)\ []
\end{aligned}$$

NFIs

If a function `f xs` is NFI:

- It contains only elements in `xs`
- There exists a *k* such that, if *x* occurs *n* times in `xs`, it occurs *kn* times in `f xs`
- This means that the output of NFIs are are *permutations of copies* of the input elements *(simplicial permutations)*

FIs

Cannot be inductively enumerated like this (at least not yet!), but there are specific cases worth studying:

We can also classify folds which are FI. If $\alpha : a \to [a] \to [a]$ is such that

$$\begin{aligned} \mathsf{filter}\ p\ (\alpha(x, xs)) &= \mathsf{filter}\ p\ xs & px = \mathsf{false} \\ \mathsf{filter}\ p\ (\alpha(x, xs)) &= \alpha(x, \mathsf{filter}\ p\ xs) & px = \mathsf{true} \end{aligned}$$

then $\mathsf{foldr}\ \alpha\ [] \in \mathsf{FI}$. The class functions defined this way includes NFIs like reverse, id and the empty list function, as well as FIs which are *not* NFI like sort, nub, and filter.

NFIs are fully specified by ***doubletons***

Back to length generalization: A notable property of NFIs is that they are determined by a *single* input-output example of length two, where the input elements are unique. In machine learning language, NFIs are learnable from a single length-two training example. Briefly, the algorithm for recovering $f\ xs$ from the example $[x, y] \mapsto f\ [x, y]$ is:

1. WLOG, assume the elements of xs are unique. For a pair of elements p, q occurring in order xs, define $g : p \mapsto x$, $g : q \mapsto y$, and apply g^{-1} to $f\ [x, y]$. By map and filter symmetry, this gives $f\ [p, q]$.

2. Since filter preserves order, the elements of $f\ [p, q]$ occur as a not-necessarily-contiguous ordered subsequence in $f\ xs$. Repeating Step 1 for each pair of elements and merging the results in order then recovers $f\ xs$.

For the subclass of FIs that satisfy the length-scaling property above [1], we have the analogous result that any function f is determined by the complete set of doubleton examples defined by pairs of input elements. The merging of sublist results in Step 2 above suggests a sheaf-based analysis that will be developed in future work, and which we hope will apply to more diverse instances of length generalization.

Issues with the picture so far

While we went beyond group equivariance and into *arbitrary structured computation*, our approach so far leaves a few things to be desired

First, to actually *find* neural networks that exactly satisfy these constraints, we need to map our diagrams into a category like **Vect**, which only treats morphisms as *linear transformations*

Therefore, both our approach and GDL can only give us *linear layers*
We cannot reason about *entire architectures*!
(beyond tricks like custom *loss functions* / *data augmentation*)

Further, we currently cannot explicitly reason about *parameters*
And they are central when modelling the forward / backward passes!

Into the realm ***2-categorical***

To be able to model whole architectures and parameters, we step from (1-)categories into *2-categories*

In 2-categories, we have *objects*, *morphisms* and *2-morphisms* (morphisms between morphisms)

In fact, we have already secretly been working in a 2-category…

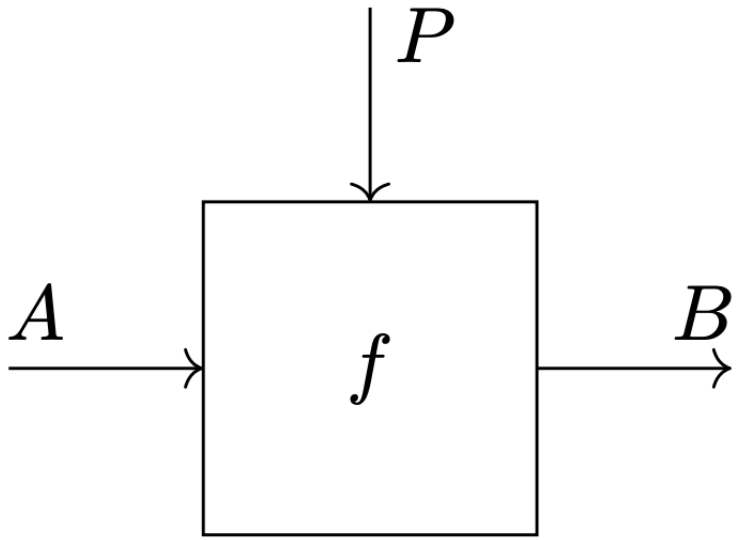
The category **Cat** has
objects as *categories* $\mathcal{C}$ and
morphisms as *functors* $F : \mathcal{C} \to \mathcal{D}$ and
2-morphisms as *natural transformations* $\alpha : F \Rightarrow G$

The 2-category ***Para***

To be able to explicitly describe parameters inside neural networks, we need a new construction – the 2-category **Para** (Cruttwell *et al.*, PLS'22)

Within **Para**, *objects* are still sets, but *morphisms* become *parametric*: $(P, f) : A \to B$ is a morphism in **Para** for all $P \in \mathbf{Set}$ and $f : P \times A \to B$

Morphisms in **Para** admit a convenient graphical representation, with *parameters* explicitly drawn:

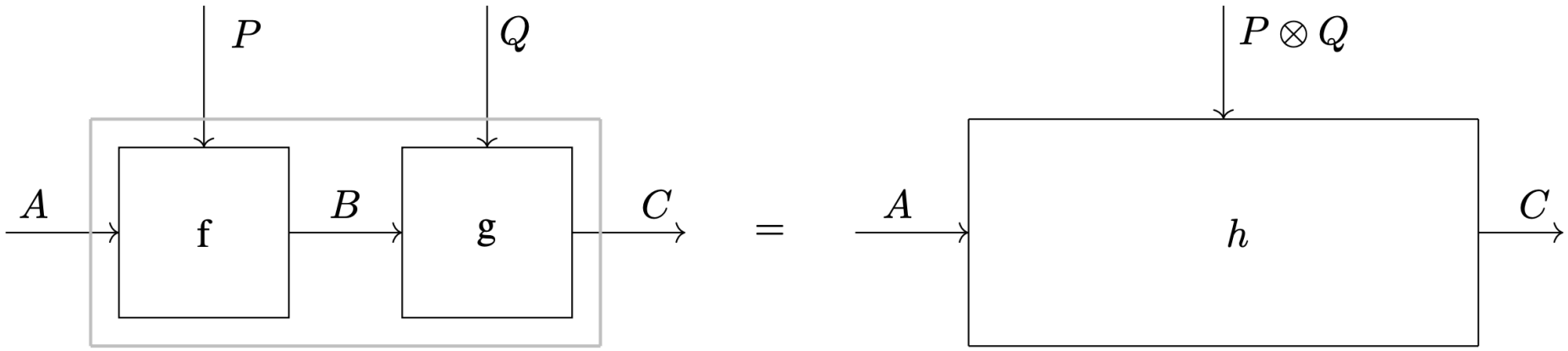


The 2-category ***Para***

To be able to explicitly describe parameters inside neural networks, we need a new construction – the 2-category **Para** (Cruttwell *et al.*, PLS'22)

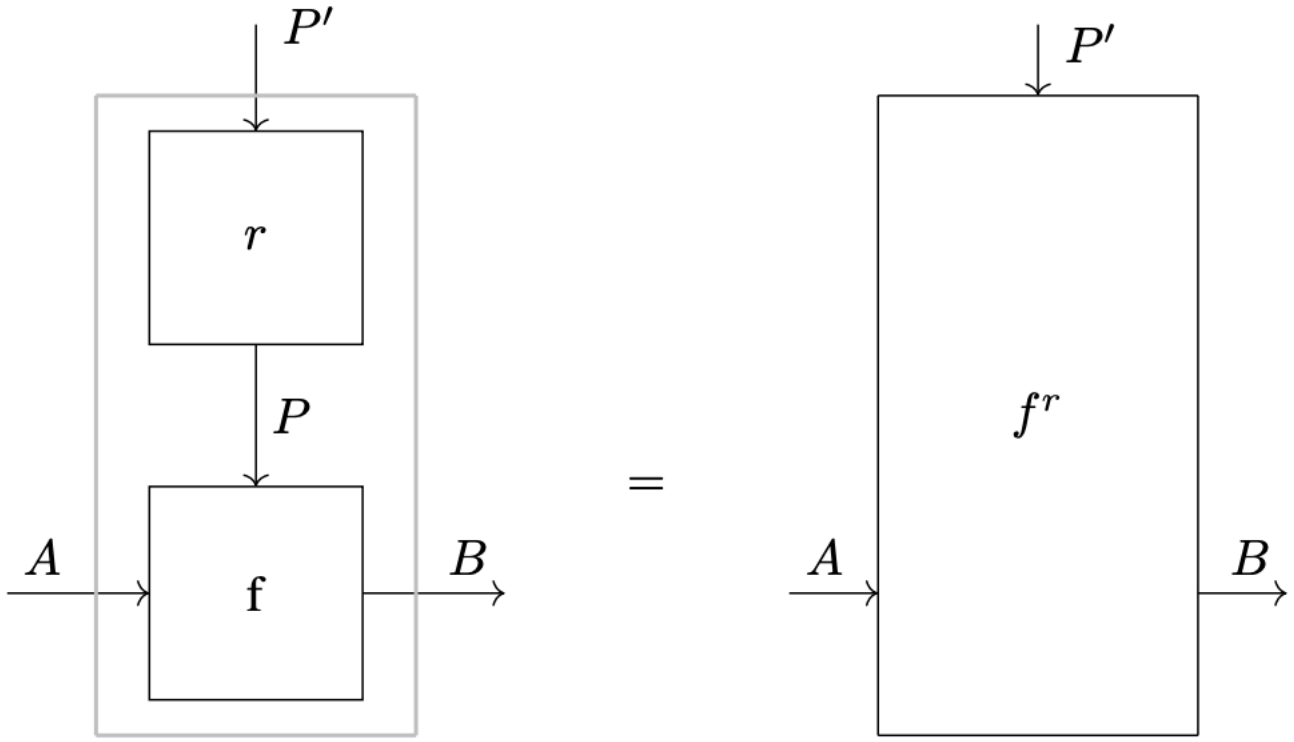
Within **Para**, *objects* are still sets, but *morphisms* become *parametric*: $(P, f) : A \to B$ is a morphism in **Para** for all $P \in \mathbf{Set}$ and $f : P \times A \to B$

Morphisms in **Para** compose by chaining their computations; for $(P, f) : A \to B$ and $(Q, g) : B \to C$ we compose into $(P \otimes Q, h)$:



The 2-category ***Para***

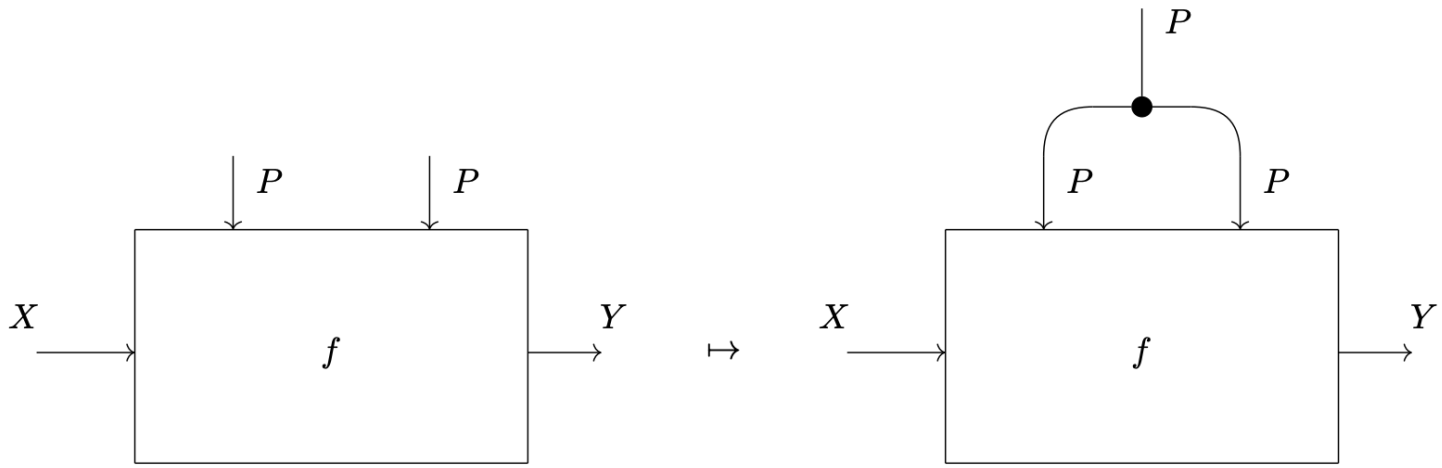
Further, **Para** admits *2-morphisms* $(P, f) \Rightarrow (P', f^r)$ which specify *reparametrisations* using a *reparametrisation map* $r : P' \to P$



The 2-category ***Para***

Further, **Para** admits *2-morphisms* $(P, f) \Rightarrow (P', f^r)$ which specify *reparametrisations* using a *reparametrisation map* $r : P' \to P$

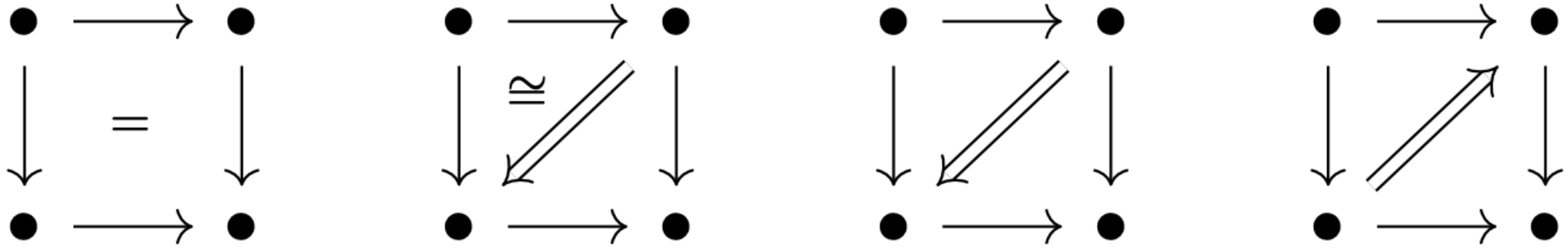
Using this construct, we can naturally express *weight tying*, e.g., in a two-layer network specified by $(P \times P, f)$, we can tie the weights of the two layers into (P, f^r) by composing with the *copy map*, $\Delta_P : P \to P \times P$



Commutative diagrams in 2-categories?

Now that we have a 2-category like **Para**, we can express *entire* neural networks in a way that explicitly tracks *all* their *parameters*!
This allows for constraint resolution beyond individual (linear) layers.

You may be tempted to rebuild the theory of endofunctor algebra homomorphisms in **Para**… and it is definitely possible!
But, the picture is not so clear anymore ☺

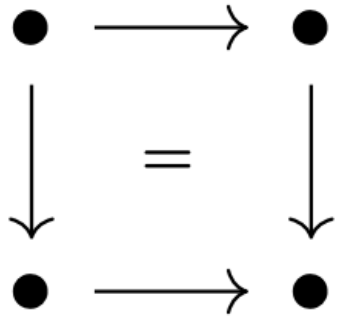


Diagrams in 2-categories

Because we now allow for *2-morphisms*, it is possible to construct *2-categorical diagrams* where 2-morphisms *bridge* the relevant paths

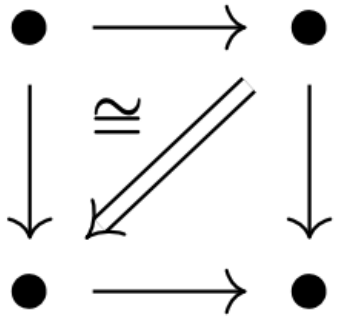
There are *four* possible outcomes, depending on type of 2-morphism:

Commutative



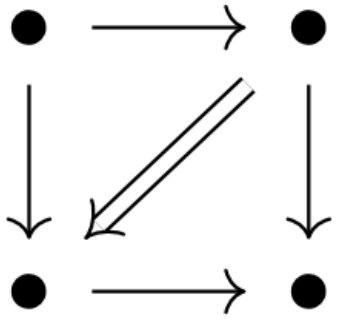
Equal

Pseudo-commutative



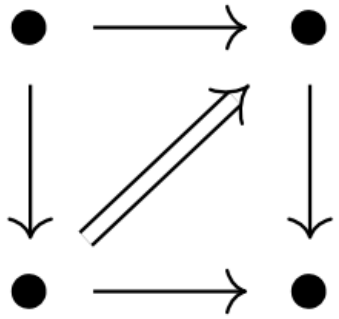
Isomorphic

Lax commutative



2-morphism ⇒

Colax commutative



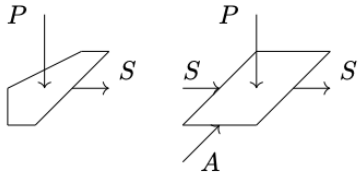
2-morphism ⇐

Lax algebras of parametric endofunctors

Likely all of these variants will yield interesting neural networks
But already focusing only on *lax* 2-algebras is sufficient to recover *all* variants of weight tying in *recursive* and *recurrent* neural networks

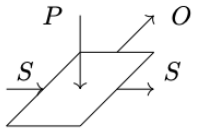
Folding recurrent neural network

$$1 + A \times S \xrightarrow{(P,\mathsf{cell}^{\mathsf{rcnt}})} S$$



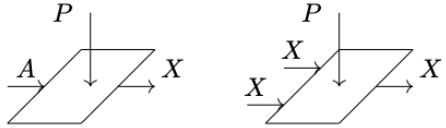
Unfolding recurrent neural network

$$S \xrightarrow{(P,\langle \mathsf{cell}_o,\mathsf{cell}_n \rangle)} O \times S$$



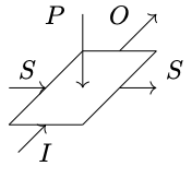
Recursive neural network

$$A + S^2 \xrightarrow{(P,\mathsf{cell}^{\mathsf{rcsv}})} S$$



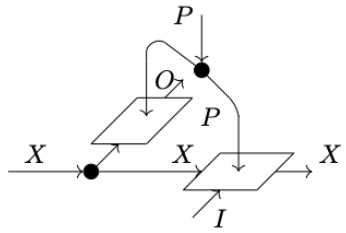
Full recurrent neural network

$$S \xrightarrow{(P,\mathsf{cell}^{\mathsf{Mealy}})} (I \to O \times S)$$



"Moore machine" neural network

$$S \xrightarrow{(P,\mathsf{cell}^{\mathsf{Moore}})} O \times (I \to S)$$



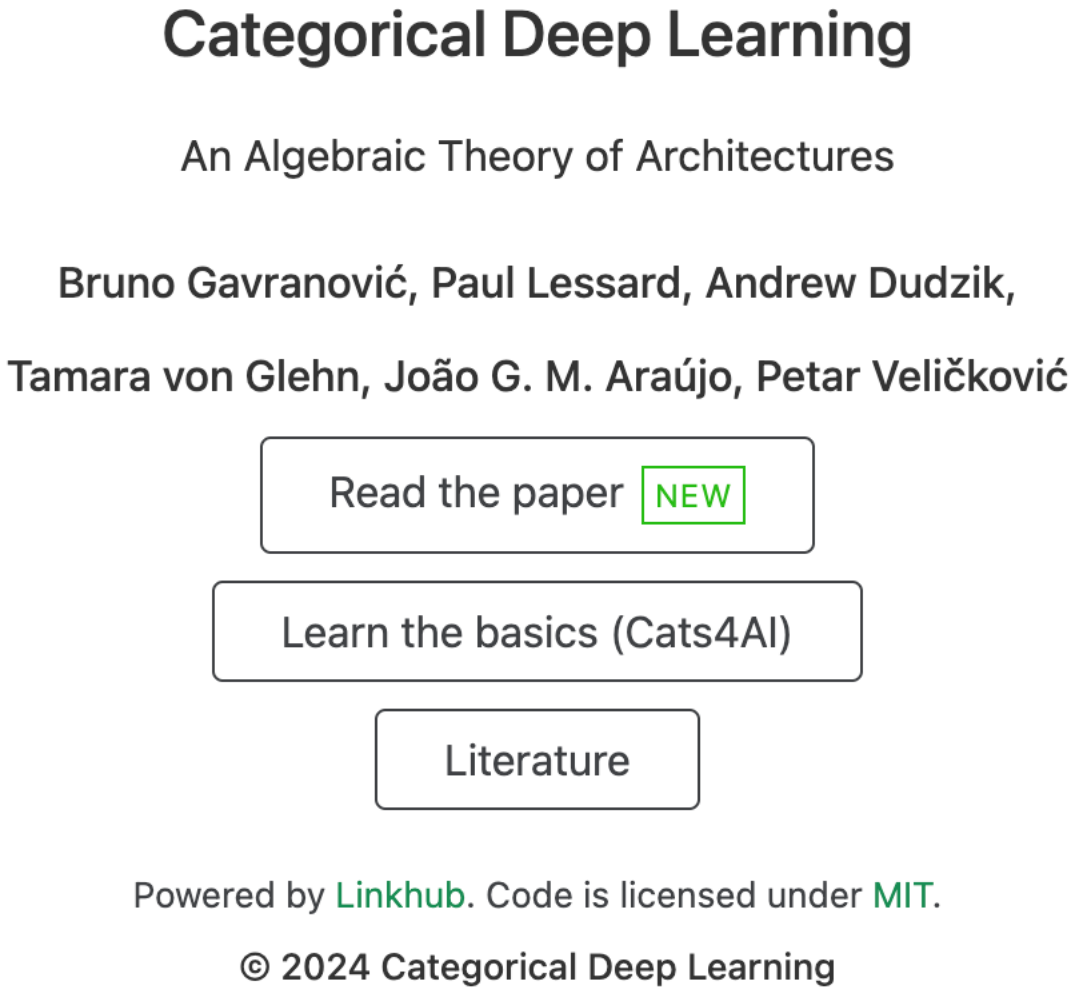
The future is Categorical

The categorical deep learning picture is very *rich*, and only formed recently

We invite you to contribute!

categoricaldeeplearning.com

Work by Bruno Gavranović, Paul Lessard, Andrew Dudzik, Tamara von Glehn, João G. M. Araújo and Petar Veličković (ICML'24)

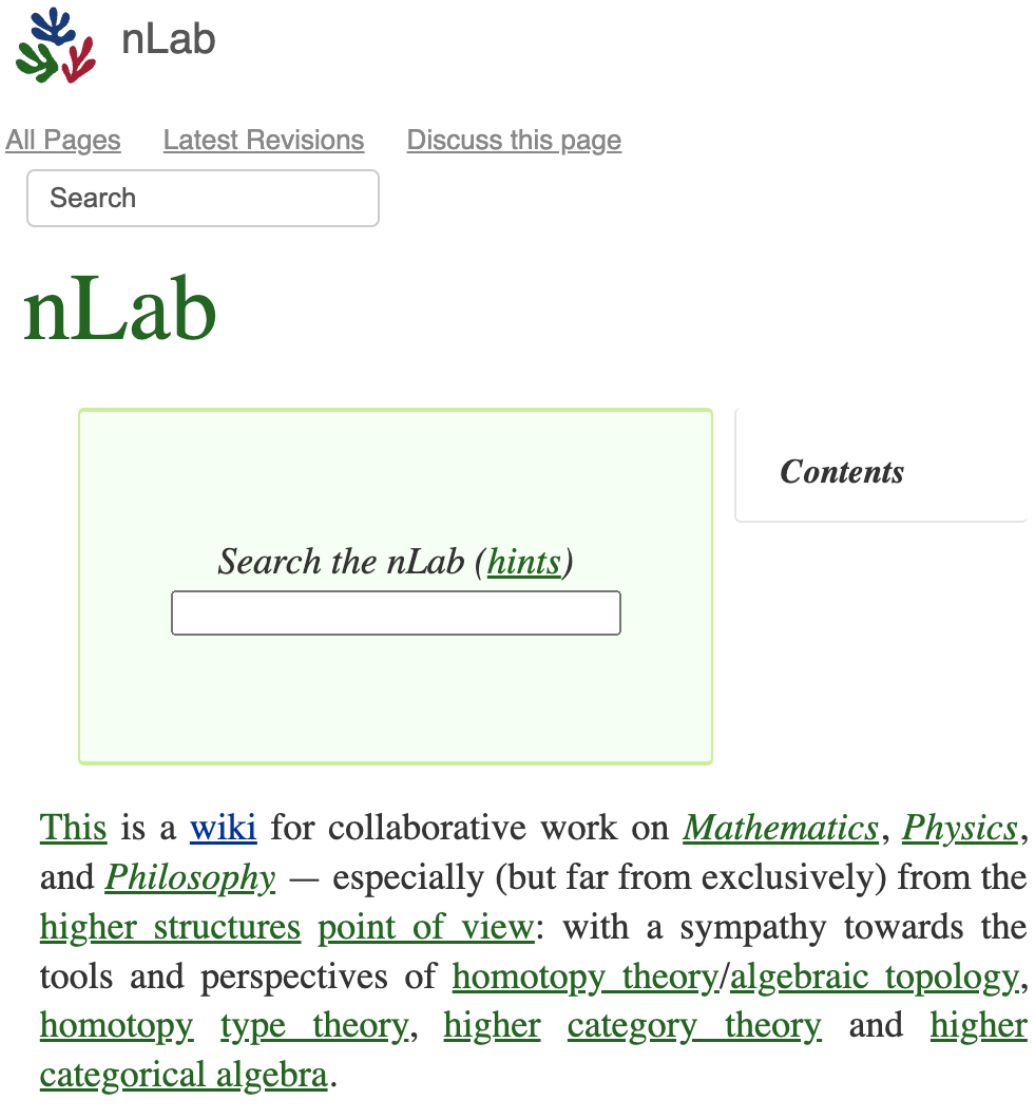


Not for the faint-hearted: ***nLab***

If, during your journey, you ever get stuck on a term and would like to understand it better…

nLab is arguably the best resource ☺
Think: *Wikipedia for categories*

Just bear in mind…
nLab was mostly written *by* mathematicians, *for* mathematicians.
Most articles written under the assumption you *understand other terms*
Beware of rabbit holes!!!



Signing off for now…

I'm off to Chile for Khipu!

Therefore, I am *signing off* from L65 (until Easter term presentations ☺)

We would **highly appreciate** any feedback you might have, at any point.
What worked well? What didn't?
What can we do better next year?

I hope you have enjoyed the course! ☺
(…and, if the course piqued your interest, I hope to collaborate with you at some point!)

