Modeling Open Systems with Category Theory

Daniel Sinderson

May 2024

Introduction

Category theory studies composition. By studying and abstracting notions of composition, categories manage to encapsulate a shockingly flexible and wide-reaching language for describing and working with structure of all sorts [5]. Much of modern mathematics, from vector spaces to groups, can be formalized in category theory. Unlike with set theory though, this formalization is from a bird's eye point of view: the details of a particular field go out of focus and only its high level structure remains [9]. This is useful. High levels of abstraction provide high levels of generality, and a general, common language of how things are structured is useful for both organizing thought and sharing it.

Because of this generality, category theory also has some potential for reformulating difficult problems in other fields or even opening a broader class of phenomena to mathematical rigor. This is an active field of research. There are people using the tools of category theory to formalize and investigate everything from quantum mechanics and linguistics to logic and systems theory [2, 4, 6, 8].

It's this last application to systems theory that we will be looking more deeply at in this paper. Though we are merely wandering at the edge of the work being done, we will see how to use category theory to build the specifications for, and simulate the behaviors of, dynamical systems built from the composition of smaller, simpler systems. We will do this by modeling a gene transcription network as the composition of individual differential systems. With even this introductory view of the subject matter it should be easy to see the promise of this approach to ease thinking about and designing models for large, complicated systems.

Category Theory Basics

The first step to understanding category theory is to understand what a category is. If you're new to higher mathematics, it's time to take a deep breath. This will seem like a lot.

Definition 1: Category

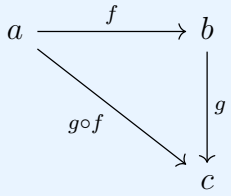
A category $\mathcal{C}$ is defined by the following:

1. $\mathcal{C}$ contains a collection of objects $\mathsf{ob}(\mathcal{C})$. We will denote that an object is in a category using set notation: $c \in \mathcal{C}$.

2. For any two objects $a, b \in \mathcal{C}$ there is a collection of morphisms, or arrows, between those objects $\mathcal{C}(a, b)$ called the homset. This is short for "set of homomorphisms." we will denote an element $f \in \mathcal{C}(a, b)$ using function notation: $f : a \to b$.

3. Every object $a \in \mathcal{C}$ has a morphism to itself $\mathsf{id}_a : a \to a$ called its identity. This morphism doesn't do anything. It's like multiplying a number by 1.

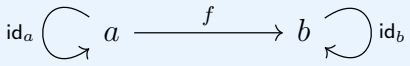
4. For every two morphisms $f : a \to b$ and $g : b \to c$ there's a third morphism $g \circ f : a \to c$ that is their composition. The circle is the symbol for function composition and $g \circ f$ is read "g after f."



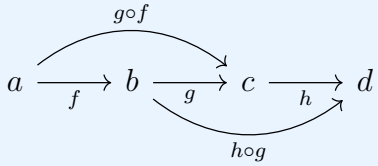
Definition: Category, cont.

These objects and morphisms are further constrained by the following properties:

1. (Unitality) Any morphism $f : a \to b$ can be composed with the identity morphisms of a and b such that $f \circ \mathsf{id}_a = \mathsf{id}_b \circ f = f$.



2. (Associativity) For any morphisms $f : a \to b$, $g : b \to c$, and $h : c \to d$, $h \circ (g \circ f) = (h \circ g) \circ f$. Since it doesn't matter what order we apply the morphisms, we write this $h \circ g \circ f$.



Let's break this down a little bit. We have a collection of objects with connections between them. We know that every object is connected to itself. And we know that if object a is connected to b, and b is connected to c, then a is connected to c through b.[1]

We also know that the objects and connections follow two rules. First, that moving from an object to that same object doesn't do anything (unitality). And second, that no matter how you group moving along connections it always amounts to the same trip (associativity). That's it. Categories are honestly pretty simple. But from these humble beginnings we are going to climb very high.

A Quick Note on Isomorphism

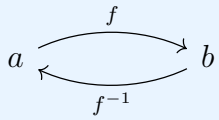
When most of us think of two things being "the same" in mathematics we think of equalities like $1 + 1 = 2$. Equality is not the only useful notion of sameness in mathematics though. When thinking about whether two objects in a category are meaningfully "the same," a more useful notion than equality is that of isomorphism. An isomorphism between objects means

[1] For people with some background in math or logic this might be familiar as the transitive property.

that, while they may not be exactly the same, as in equality, there exists a way to move between the two objects without losing any information.

Definition 2: Isomorphism

Two objects a, b in a category are isomorphic if there exists a morphism $f : a \to b$ for which there is another morphism $f^{-1} : b \to a$ such that $f^{-1} \circ f = \mathsf{id}_a$ and $f \circ f^{-1} = \mathsf{id}_b$. Such a morphism is called an isomorphism.



Another way to think about isomorphism is as a renaming. Two objects are isomorphic if it's possible to represent one as the other by a simple, and reversible, relabeling. For instance, an example of two sets that are isomorphic are the sets $\{a, b, c\}$ and $\{1, 2, 3\}$. By relabeling $a \mapsto 1$, $b \mapsto 2$, and $c \mapsto 3$, we have represented the first set by means of the second set. Specifically, we've labeled the first three letters of the alphabet by the numbers 1, 2, and 3.

Functors and Categories of Categories

Most mathematical structures have some way of mapping one instance of that structure to another instance. Functions map sets to sets. Linear transformations map vector spaces to other vector spaces. And group homomorphisms map groups to other groups. Since categories are a mathematical structure, it makes sense to ask if there exists a way to map between them. Such a map would have to preserve all of the things that make a category a category: identity morphisms and composite morphisms, and the rules of unitality and associativity. We call such a map between categories a functor.

Definition 3: Functor

A functor $F : \mathcal{C} \to \mathcal{D}$ is a map between categories $\mathcal{C}$ and $\mathcal{D}$ such that the following hold:

1. For any object $a \in \mathcal{C}$ there is an object $Fa \in \mathcal{D}$.
2. For any morphism $f : a \to b$ between objects a and b in $\mathcal{C}$ there is a morphism $Ff : Fa \to Fb$ between objects Fa and Fb in $\mathcal{D}$.
3. For all objects $a \in \mathcal{C}$ and $Fa \in \mathcal{D}$, $F\mathsf{id}_a = \mathsf{id}_{Fa}$.
4. For any composition of morphisms $g \circ f$ in $\mathcal{C}$, $F(g \circ f) = Fg \circ Ff$ in $\mathcal{D}$.

There are some things to note here. The first is that, like sets and functions, the mapping of objects from one category to another is unique. Every object in the source category is mapped to exactly one object in the target category. The same goes for morphisms. The second thing to note is that identity morphisms get mapped to identity morphisms and compositions get mapped to compositions.

Let's check that these rules are enough to preserve unitality and associativity. We will check unitality first.

Proof 1: Preservation of Unitality

Let a functor $F : \mathcal{C} \to \mathcal{D}$ be given, with objects $a, b \in \mathcal{C}$ and morphism $f : a \to b$.

$$
\begin{aligned}
F(\mathsf{id}_b \circ f) &= F\mathsf{id}_b \circ Ff && \text{by the functor composition rule} \\
&= \mathsf{id}_{Fb} \circ Ff && \text{by the functor identity rule} \\
&= Ff && \text{by the definition of the identity} \\
&= Ff \circ \mathsf{id}_{Fa} && \text{by the definition of the identity} \\
&= Ff \circ F\mathsf{id}_a && \text{by the functor identity rule} \\
&= F(f \circ \mathsf{id}_a) && \text{by the functor composition rule}
\end{aligned}
$$

Thus the functor preserves unitality, since $F(\mathsf{id}_b \circ f) = F(f \circ \mathsf{id}_a) = Ff$. $\square$

Now let's check that associativity is preserved.

Proof 2: Preservation of Associativity

Let a functor $F : \mathcal{C} \to \mathcal{D}$ be given, with objects $a, b, c, d \in \mathcal{C}$ and morphisms $f : a \to b$, $g : b \to c$, and $h : c \to d$.

$$\begin{aligned} F(h \circ (g \circ f)) &= Fh \circ F(g \circ f) && \text{by the functor composition rule} \\ &= Fh \circ (Fg \circ Ff) && \text{by the functor composition rule} \\ &= (Fh \circ Fg) \circ Ff && \text{by associativity in the category } \mathcal{D} \\ &= F(h \circ g) \circ Ff && \text{by the functor composition rule} \\ &= F((h \circ g) \circ f) && \text{by the functor composition rule} \end{aligned}$$

Thus the functor preserves associativity, since $F(h \circ (g \circ f)) = F((h \circ g) \circ f)$. □

With functors in our toolbox, it's now possible to construct a category where the objects themselves are categories and the morphism between them are functors. We call this category **Cat**. This, along with our next step, will prove to be extremely powerful.

A Quick Note on Avoiding Paradox

For anyone who has taken a course in logic or set theory, that last paragraph might be ringing some alarm bells. Whenever we have recursion in our structure like this, there's the opportunity for paradox to sneak in. The category of categories is no exception.

To avoid the paradox of whether **Cat** contains itself, category theorists employ formal notions of "size" to differentiate distinct universes of categories or by invoking a seperation between sets and classes [6, 9]. These formalisms can get pretty technical though, so we will be ignoring them in this paper. For us it will be enough to assume that whenenever we talk about a category whose objects are also categories, our object categories are in some way "smaller" than the category that they're a part of, and thus of a different type.[2]

Functor Categories and Natural Transformations

For our next, and last, step up into abstraction let's talk about functor categories. Functor categories are categories whose objects are functors and whose morphisms are maps called

[2]In general, this "smallness" will be that the category's collection of objects forms a set, or that the homset between any two objects forms a set.

natural transformations.

Since natural transformations map functors to other functors, we know that they must preserve the essential "functor-ness" of functors. This "functor-ness" is a bit harder to pin down than the "category-ness" that functors themselves preserve. We need to see what natural transformations do to functors, but also what they do to the objects and arrows of the categories that the functors are acting on.

Definition 4: Natural Transformation

A natural transformation $\alpha : F \Rightarrow G$ is a map between functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{C} \to \mathcal{D}$ such that the following holds:

1. For each object $c \in \mathcal{C}$, there is a morphism $\alpha_c : Fc \to Gc$ in $\mathcal{D}$. These are called the component morphisms of α.

2. For every morphism $f : a \to b$ in $\mathcal{C}$, $\alpha_b \circ Ff = Gf \circ \alpha_a$. This is called the naturality condition, and it preserves functoriality.

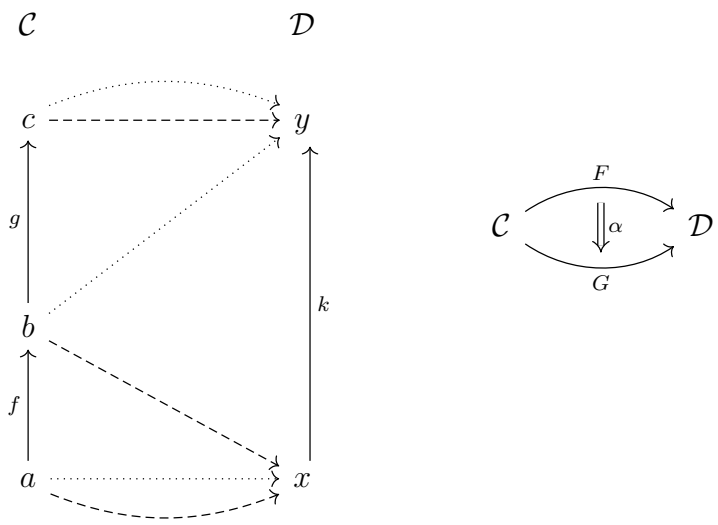
The naturality condition ensures that the following diagram commutes.

$$\begin{array}{ccc} Fa & \xrightarrow{\alpha_a} & Ga \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ Fb & \xrightarrow[\alpha_b]{} & Gb \end{array}$$

We can represent α diagramatically in the following condensed form.

$$\mathcal{C} \;\overset{F}{\underset{G}{\Downarrow \alpha}}\; \mathcal{D}$$

This definition is short but there's a whole extra layer of stuff to keep track of since we added a layer of abstraction. Let's get a better idea of what this definition captures by using a picture.



Here we see two simple categories, $\mathcal{C}$ and $\mathcal{D}$, two functors F (the dashed lines) and G (the dotted lines) between them, and a natural transformation α that deforms F into G. Let's dig into this picture a bit and pick apart the functors and the natural transformation.

First, let's note all the important mappings of the functors and the component morphisms of the natural transformation.

$$\begin{aligned} Fa &= x & Fb &= x & Fc &= y \\ Ff &= \mathsf{id}_x & Fg &= k & F(g \circ f) &= k \end{aligned}$$

$$\begin{aligned} Ga &= x & Gb &= y & Gc &= y \\ Gf &= k & Gg &= \mathsf{id}_y & G(g \circ f) &= k \end{aligned}$$

$$\begin{aligned} \alpha_a &: Fa \to Ga & \alpha_a &= \mathsf{id}_x \\ \alpha_b &: Fb \to Gb & \alpha_b &= k \\ \alpha_c &: Fc \to Gc & \alpha_c &= \mathsf{id}_y \end{aligned}$$

Next, let's check that the functors preserve composition.[3]

$$F(g \circ f) = k = k \circ \mathsf{id}_x = Fg \circ Ff \quad \checkmark$$

[3]You should be able to convince yourself that they preserve identities.

$$G(g \circ f) = k = \mathsf{id}_y \circ k = Gg \circ Gf \quad \checkmark$$

And now let's look at the natural transformation $\alpha : F \Rightarrow G$ and make sure that it satisfies the naturality condition for the morphisms $f, g \in \mathcal{C}$.

$$\alpha_b \circ Ff = k \circ \mathsf{id}_x = Gf \circ \alpha_a \quad \checkmark$$

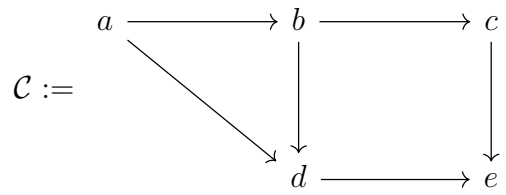
$$\alpha_c \circ Fg = \mathsf{id}_y \circ k = Gg \circ \alpha_b \quad \checkmark$$

Everything is working as expected. Hopefully this helped to unpack some of the complexity behind the natural transformation definition. In a real example, our functors would be defined equationally and we'd prove these properties for arbitrary morphisms instead of proving them for each morphism individually like we did here.

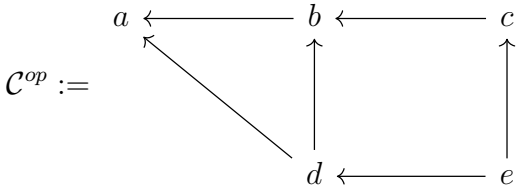
Duality

The last thing we will cover in this section is the very simple, and very powerful, property of duality. Every category has an opposite category that we write as $\mathcal{C}^{op}$. The objects in this opposite category are exactly the same as in the original category, but every arrow is turned around: its source becomes its target and its target becomes its source. Let's look at a simple example.

Let $\mathcal{C}$ be the category below.



It's opposite category, $\mathcal{C}^{op}$, is the following.



And that's it. One reason duality turns out to be so powerful is that it doubles our effectiveness. For every structure in a category, we get the dual structure essentially for free [3]. Duality also plays an important role in representability and the Yoneda lemma which are major features of category theory that, unfortunately, we won't have space for in this paper.

Structure between Categories

A category in isolation, like a set in isolation, doesn't give us much. It's only when we begin adding relations and mappings between categories that the theory comes into its own. These will include categorical versions of some familiar structures, like embeddings, equivalence relations, and isomorphisms. But it will also include some structures that might seem strange and perplexing at first, like adjunctions and monads.

Subcategories

Let's start with the simplest structure first, that of a subcategory relation. Like any of the other "sub-" relations from other fields of mathematics, the subcategory relation encapsulates the notion of one category being in some way contained inside of another. This means that the pattern of objects and morphisms that make up the subcategory are faithfully present in the other category. Our first step will be to define this "faithfulness."

Definition 5: Faithful Functor

A functor $F : \mathcal{C} \to \mathcal{D}$ is faithful if for each $x, y \in \mathcal{C}$, the map $\mathcal{C}(x, y) \to \mathcal{D}(Fx, Fy)$ is injective.

In other words, when a functor between categories is faithful, it means that every morphism between objects x, y in the source category $\mathcal{C}$ is mapped to exactly one morphism between Fx, Fy in the target category $\mathcal{D}$. No morphisms in the source get collapsed into the same morphism in the target. All of them are "there."

However, this doesn't mean that objects can't be collapsed. For that we need another condition on our functor.

Definition 6: Injective on Objects

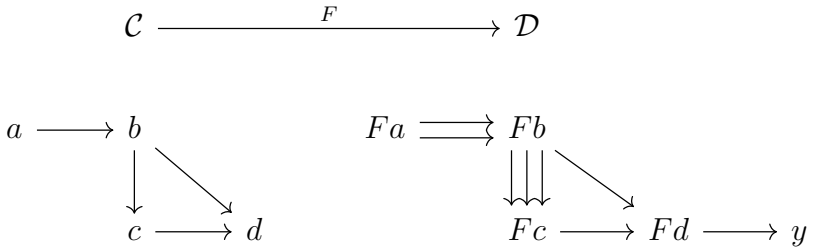
A functor $F : \mathcal{C} \to \mathcal{D}$ is injective on objects if for each $x \in \mathcal{C}$, there is a unique $y \in \mathcal{D}$ such that $y = Fx$.

With these two conditions we can define our subcategory relation.

Definition 7: Subcategory

A category $\mathcal{C}$ with a functor $F : \mathcal{C} \to \mathcal{D}$ that is both faithful and injective on objects is a subcategory of the category $\mathcal{D}$. A functor F of this type is called an **embedding** of $\mathcal{C}$ into $\mathcal{D}$.

Let's take a look at this using the two small categories below with a functor F between them.



You should be able to convince yourself that $\mathcal{C}$ is a subcategory of $\mathcal{D}$ using the definitions of subcategory and functor above. As expected we get injectivity on morphisms and objects. But what if we also have surjectivity on morphisms? To look into this let's define another property that a functor can have.

Definition 8: Full Functor

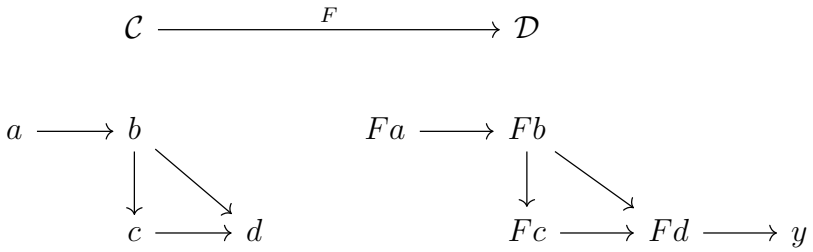
A functor $F : \mathcal{C} \to \mathcal{D}$ is full if for each $x, y \in \mathcal{C}$, the map $\mathcal{C}(x, y) \to \mathcal{D}(Fx, Fy)$ is surjective.

This gives us a stronger version of the subcategory relation called a full subcategory that rejects any extraneous morphisms in the target category.

Definition 9: Full Subcategory

A category $\mathcal{C}$ with a functor $F : \mathcal{C} \to \mathcal{D}$ that is full, faithful, and injective on objects is a full subcategory of the domain $\mathcal{D}$. A functor F of this type is called a **full embedding** of $\mathcal{C}$ into $\mathcal{D}$.

Let's take a look at an example by paring down our category D from the previous example.



Again, convince yourself that this is indeed a full embedding.

Equivalence Relations between Categories

Even more familiar than the subcategory relation, but less simple in the categorical context, are equivalence relations. When comparing the "sameness" of categories there are several relations we can define. In order of strictest to loosest these are equality, isomorphism, and equivalence. Equality and isomorphism are actually quite simple.

Definition 10: Equality of Categories

Two categories are equal if they contain exactly the same objects and morphism.

Definition 11: Isomorphism of Categories – Subcategories

Two categories are isomorphic is they're both subcategories of each other.

This is similar to the situation with sets where mutual subsets define an equality of sets. By ensuring that every object and morphism in the source has a unique object and morphism in the target that it gets mapped to under the embedding $F : \mathcal{C} \to \mathcal{D}$ and vice versa under the embedding $G : \mathcal{D} \to \mathcal{C}$, we ensure that our mappings between sets and objects are bijective and that our categories are isomorphic.

Another way to define an isomorphism between categories is as an isomorphism in the category of categories and functors.

Definition 12: Isomorphism of Categories – Category of Categories

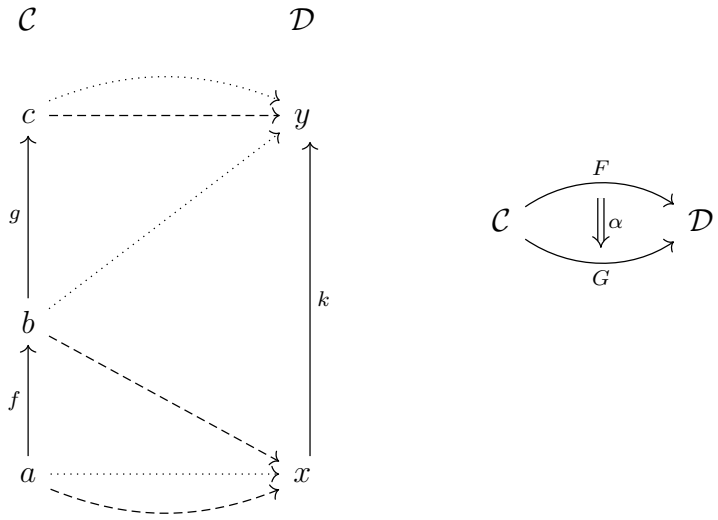
Given two categories $\mathcal{C}, \mathcal{D} \in \mathbf{Cat}$ and a pair of functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$, $\mathcal{C}$ and $\mathcal{D}$ are isomorphic if $G \circ F = \mathsf{id}_{\mathcal{C}}$ and $F \circ G = \mathsf{id}_{\mathcal{D}}$.

More useful than these is the notion of categorical equivalence and the notion of natural isomorphism that it relies on.

Definition 13: Natural Isomorphism

A natural isomorphism is a natural transformation $\alpha : F \Rightarrow G$ where all of the component morphisms of the transformation $\alpha_c : Fc \to Gc$ are isomorphisms.

Let's try to get a handle on what this means by looking at the image we used to explore natural transformations.



As before, the components of α are the following:

$$\alpha_a = \mathsf{id}_x$$

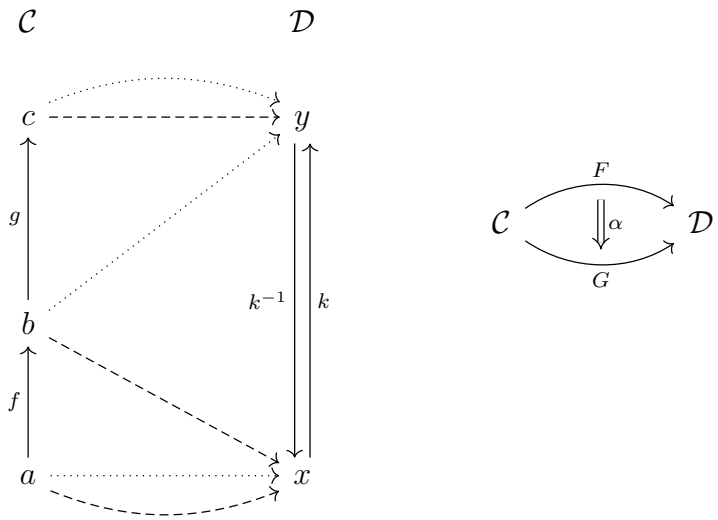
$$\alpha_b = k$$

$$\alpha_c = \mathsf{id}_y$$

Our task now is to figure out which, if any, of these component morphisms are isomorphism. For both identity morphisms the answer is yes: $\mathsf{id}_x \circ \mathsf{id}_x = \mathsf{id}_x$. In general, if you do nothing

and then do nothing again, you've still not done anything.[4] The case is different for k. There simply is no morphism from y to x in $\mathcal{D}$ that could be the inverse of k, so k cannot be an isomorphism. This means our natural transformation α is not a natural isomorphism.

Let's make a small addition and try again.



With this new diagram we've explicitly added an inverse of k, making it an isomorphism. Since every component morphism of α is now an isomorphism, α itself is now a natural isomorphism. With natural isomorphisms in hand we can define an equivalence between categories.

Definition 14: Categorical Equivalence

Two categories $\mathcal{C}$ and $\mathcal{D}$ are equivalent, written $\mathcal{C} \simeq \mathcal{D}$, if there exist two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ with natural isomorphisms η and ϵ such that $\eta : \mathsf{id}_{\mathcal{C}} \cong G \circ F$ and $\epsilon : \mathsf{id}_{\mathcal{D}} \cong F \circ G$.

It might seem difficult to understand what's going on beneath this definition. Let's try again using another property on functors.

Definition 15: Essentially Surjective Functor

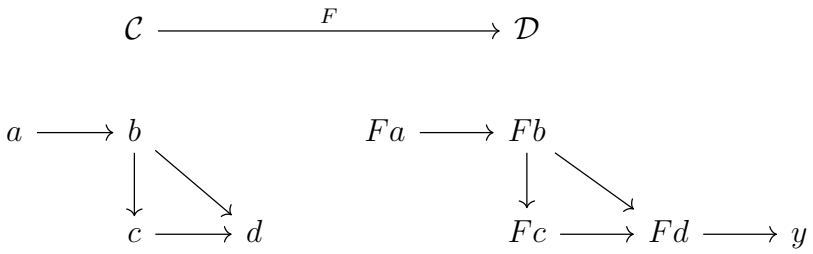
A functor $F : \mathcal{C} \to \mathcal{D}$ is essentially surjective on objects if for every object $y \in \mathcal{D}$ there exists an object $x \in \mathcal{C}$ such that $Fx \cong y$.

[4] This is the kind of high-octane thinking that math opens up to us. Breathe it in.

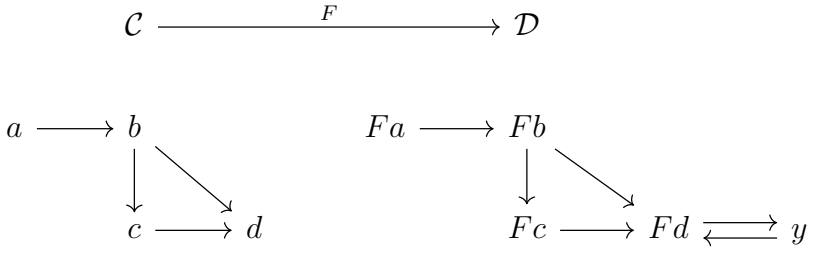
Definition 16: Categorical Equivalence, revisited

Two categories $\mathcal{C}$ and $\mathcal{D}$ are equivalent if there exists a functor F between them that is faithful, full, and essentially surjective on objects.

Let's look more deeply at this functor and what each of its properties guarantees. Being essentially surjective on objects means that every object in our target category has some object from the source category that maps to it (up to isomorphism). The combination of a full and faithful functor guarantees that for every set of morphisms between objects in our source category will be isomorphic to the set of morphisms between the image of those objects in the target category. In other words, we have an isomorphism between morphisms and a surjection between objects (up to isomorphism). If we look at our full subcategory example from above, are $\mathcal{C}$ and $\mathcal{D}$ equivalent?

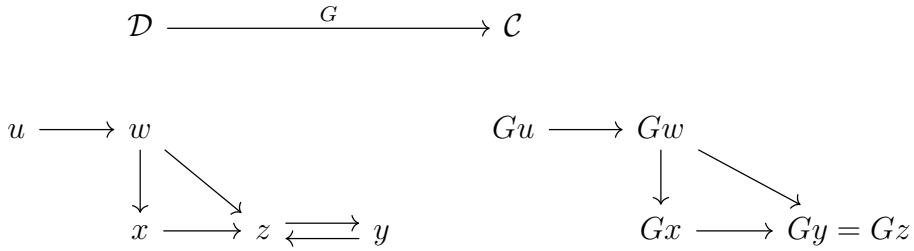


We know that F is full and faithful by definition of a full subcategory. But it's not surjective on objects since there's no object in $\mathcal{C}$ that maps to y, or anything isomorphic to y, in $\mathcal{D}$. Let's make a quick modification so that our two categories can be equivalent.



Our collections of objects are now surjective up to isomorphism, so our functor is full, faithful, and essentially surjective and our two categories are equivalent. Since categorical equivalence is an equivalence relation we know that the relation must be reflexive, symmetric, and transitive. It's trivial to see that the relation is reflexive since the identity functor on a category meets all of our requirements, so we know a category is always equivalent to itself. But how about symmetry? Let's look at this using our example categories above and

introduce a new functor G.



We can map y and z onto the same object G and keep the functor essentially surjective. But is it still full and faithful? Naively we might think they aren't, since id_y, id_z, and both parts of the isomorphism between y and z all get mapped to id_{Gy}, which certainly doesn't seem injective. But G is full and faithful. Double checking that this is the case is a great opportunity to see why full and faithful functors are defined in terms of homsets.

Proof 3: F is Full and Faithful

Given a functor $G : \mathcal{D} \to \mathcal{C}$ as defined above, for G to be full and faithful there must be a bijection between the homset of any two objects in $\mathcal{D}$ and the homset of their images in $\mathcal{C}$. Since most of the objects and morphisms we are concerned with above obviously meet this requirement by inspection, let's focus on $y, z \in \mathcal{D}$ and the isomorphism between them, which we will call $f : y \to z$ and $f^{-1} : z \to y$, and their images. Listing them out, we have the following homsets.

$$\mathcal{D}(y, y) = \{\mathsf{id}_y\} \cong \{\mathsf{id}_{Gy}\} = \mathcal{C}(Gy, Gy)$$

$$\mathcal{D}(z, z) = \{\mathsf{id}_z\} \cong \{\mathsf{id}_{Gz}\} = \mathcal{C}(Gz, Gz)$$

$$\mathcal{D}(y, z) = \{f\} \cong \{\mathsf{id}_{Gy}\} = \mathcal{C}(Gy, Gz)$$

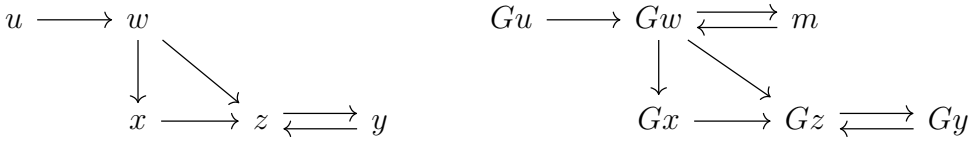
$$\mathcal{D}(z, y) = \{f^{-1}\} \cong \{\mathsf{id}_{Gz}\} = \mathcal{C}(Gz, Gy)$$

So even though we have four morphisms that are all getting mapped to a single identity, each of the homsets individually is a singleton set in both the source and the target categories, meaning that they're all isomorphic, so our functor G is indeed full and faithful. □

So we now know that categorical equivalence is reflexive and symmetric, but is it transi-

tive? To test this we will need another small category and another functor.

$$\mathcal{D} \xrightarrow{\quad H \quad} \mathcal{E}$$



You should be able to convince yourself that $\mathcal{D} \simeq \mathcal{E}$ and that $\mathcal{C} \simeq \mathcal{E}$ using the same reasoning we used for our initial equivalence $\mathcal{C} \simeq \mathcal{D}$.

Since categorical equivalence is reflexive, symmetric, and transitive we know that it does define an equivalence relation.

Adjunctions

Another important relationship that can exist between categories is an adjunction. Adjunctions are a generalization of categorical equivalence. Where an equivalence requires natural isomorphisms, an adjunction gets by with natural transformations. What this means intuitively is that an adjunction is a way to encapsulate the notion of an imperfect inverse [5]. In an isomorphism or equivalence it's possible to move from one category to the other and then back and return to where you started (up to isomorphism of objects). In an adjunction the guarantee is weaker: with each traversal an error accrues, but there is structure to it.

Let's look at this with an example using the two functions below.

$$\mathbb{Z} \xrightarrow{\quad f \quad} \mathbb{R} \qquad\qquad \mathbb{R} \xrightarrow{\quad g \quad} \mathbb{Z}$$

$$x \longmapsto 2x \qquad\qquad x \longmapsto \left\lceil \tfrac{1}{2}x \right\rceil$$

Here f multiplies by 2 and g divides by two and then rounds up to the nearest integer. Let's throw in some values of x and track it through our functions.

$$g(f(3)) = g(6) = 3 \qquad\qquad f(g(3)) = f(2) = 4$$
$$g(f(4)) = g(8) = 4 \qquad\qquad f(g(4)) = f(2) = 4$$
$$g(f(5)) = g(10) = 5 \qquad\qquad f(g(5)) = f(3) = 6$$

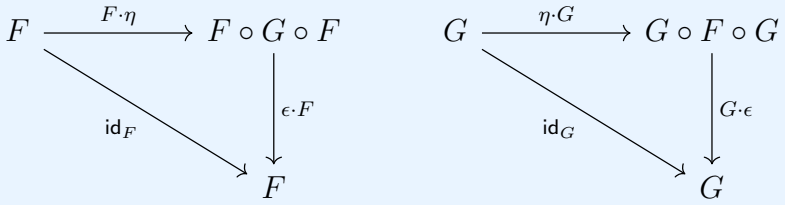
Clearly these aren't isomorphisms: $f(g(x))$ isn't always equal to x or to $g(f(x))$. But the error is small and, more importantly, it's structured. For any $x \in \mathbb{Z}$ and any $y \in \mathbb{R}$, $f(x) \leq y$

if and only if $x \leq g(y)$. The specific structure here isn't important to us, only that there is one.

Let's make this precise now.

Definition 17: Adjunction

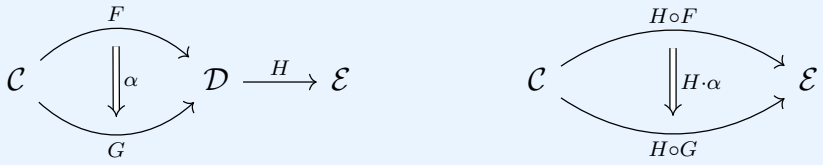
An adjunction between two categories $\mathcal{C}$ and $\mathcal{D}$ exists, written $\mathcal{C} \dashv \mathcal{D}$, if there exist two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ with natural transformations $\eta : \mathsf{id}_{\mathcal{C}} \Rightarrow G \circ F$ and $\epsilon : F \circ G \Rightarrow \mathsf{id}_{\mathcal{D}}$ such that the following diagrams commute.



You may have noticed there's something strange going on in the diagrams above. Each of the objects is a functor and each of the morphisms is a natural transformation, but many of the morphisms appear to be the result of some kind of composition of a functor and a natural transformation. This operation is called whiskering. Whiskering is important for understanding the adjunction definition above and for understanding monads in the next section, so we will spend some time here to understand it.

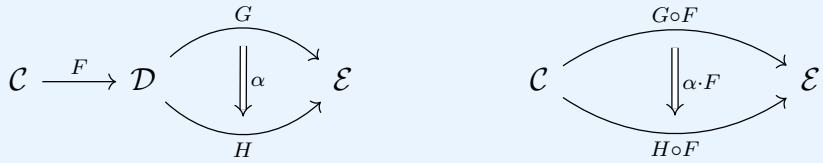
Definition 18: Whiskering on the Right

Given functors $F, G : \mathcal{C} \to \mathcal{D}$ and $H : \mathcal{D} \to \mathcal{E}$ along with a natural transformation $\alpha : F \Rightarrow G$, if we whisker on the right we get a new natural transformation $H \cdot \alpha : (H \circ F) \Rightarrow (H \circ G)$ with components $(H \cdot \alpha)_a = H\alpha_a$.

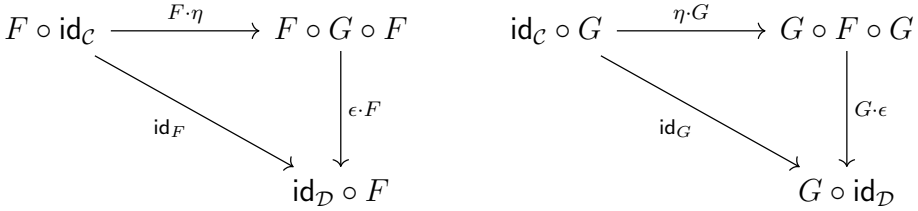


Definition 19: Whiskering on the Left

Given functors $F : \mathcal{C} \to \mathcal{D}$ and $G, H : \mathcal{D} \to \mathcal{E}$ along with a natural transformation $\alpha : G \Rightarrow H$, if we whisker on the left we get a new natural transformation $\alpha \cdot F : (G \circ F) \Rightarrow (H \circ F)$ with components $(\alpha \cdot F)_a = \alpha_{Fa}$.



The results of this operation are easier to see if we look at the triangle diagrams above with the identity functors put in.



Let's contrast our definition of adjunction to our definition of an isomorphism to help us get some intuition about how they work. Given functors $F : \mathcal{C} \to \mathcal{D}$ and $G, H : \mathcal{D} \to \mathcal{E}$, we would have an isomorphism if $G \circ F = \mathsf{id}_{\mathcal{C}}$ and $F \circ G = \mathsf{id}_{\mathcal{D}}$. But we don't have this in an adjunction. Instead we have the natural transformations $\eta : \mathsf{id}_{\mathcal{C}} \Rightarrow G \circ F$ and $\epsilon : F \circ G \Rightarrow \mathsf{id}_{\mathcal{D}}$ and the triangle identities above. At a high level, these tell us that–while the compositions don't equal the identities–there is still a way to travel from the identities on our categories to our compositions of functors. This traveling accrues an error now though: it's no longer reversible.

Monads

Our last relationship between categories, the monad, is explicitly a functor from a category to itself[5] rather than a relationship between arbitrary categories. Like an adjunction though, a monad mutates the objects of the category it acts on in a non-reversible way: there's some loss of information that occurs that prevents an output from being uniquely identified with an input. The connection between adjunctions and monads is actually quite deep. But before we look into that, let's see the standard definition.

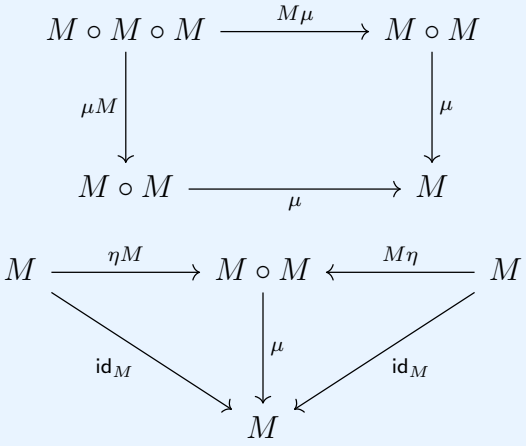
[5]When the domain and codomain of a functor are the same category we often call it an endofunctor.

Definition 20: Monad

Given a category $\mathcal{C}$, a monad on $\mathcal{C}$ is a functor $M : \mathcal{C} \to \mathcal{C}$ with the following two natural transformations:

1. $\eta : \mathsf{id}_{\mathcal{C}} \Rightarrow M$ called the unit.
2. $\mu : M \circ M \Rightarrow M$ called the multiplication.

These natural transformation are required to make the following diagrams commute.



For those with some experience with algebraic structures, this creates a monoidal structure on the collection of endofunctors $\{M, M \circ M, M \circ M \circ M, \dots\}$ where the monoidal unit is η and the monoidal product is μ.[a]

[a] This is the reason for the oft maligned phrase that "monads are just monoids in the category of endofunctors."

As we mentioned before there's a deep connection between adjunctions and monads. Specifically, every adjunction gives rise to a monad $G \circ F : \mathcal{C} \to \mathcal{C}$ where G is the right adjoint functor and F is the left adjoint functor. We can think of this induced monad as the impression or shadow left by the adjunction [9].[6] Looking at this more rigorously we can construct an alternate definition.

[6] Specifically the impression left on the codomain of the right adjoint.

Definition 21: Monad Induced by an Adjunction

Every adjunction $F \dashv G$ induces a monad $G \circ F : \mathcal{C} \to \mathcal{C}$ where the unit is the natural transformation $\eta : \mathsf{id}_{\mathcal{C}} \Rightarrow G \circ F$ and the multiplication is the whiskered natural transformation $G \cdot \epsilon \cdot F : G \circ F \circ G \circ F \Rightarrow G \circ F$.

Monads often show up in applied settings when the notion of a side-effect or additional structure on outputs is required. In the context of categorical systems theory, monads make it possible to compose nondeterministic sytems.

Structure within Categories

We will now look at the kinds of structures that we can find within a category. We call these structures universal properties. The general theory of universal properties is rich, but it's also very abstract and, unfortunately, outside of the scope of this paper [9]. Instead, we will focus on two specific examples of a universal property that are important to the theory we explore later on: terminal objects and products.

Terminal Objects

A terminal object in a category is an object that has exactly one morphism to it from every other object in the category. It's the Rome of a category: all roads lead to it.

Definition 22: Terminal Object

Let $t \in \mathcal{C}$ be given. Then t is a terminal object in $\mathcal{C}$ if the following hold:

1. For all $x \in \mathcal{C}$, there is exactly one morphism $f : x \to t$.
2. For any other object t' in $\mathcal{C}$ that meets condition (1), there exists a unique isomorphism $g : t \to t'$.

Note here that a terminal object in a category is unique up to unique isomorphism and not strictly unique: as long as there's a unique isomorphism between them, there can be an infinite number of terminal objects in the category. This is in fact the case in the category **Set** of sets and functions where all singleton sets $\{*\}$ are terminal. If we think about this it

should make sense. For any set S there is exactly one function $f : S \to \{*\}$: the function that sends all elements in S to $*$. This includes any other singleton set $\{-\}$. And since both singletons have the same cardinality and since f is surjective we know that this one function between them is an isomorphism.

Also note that not every category has a terminal object. For instance the category below has no object that has a morphism to it from every other object.

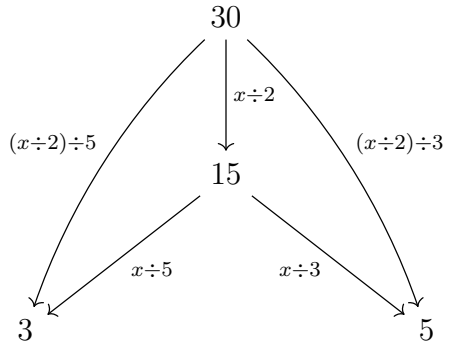
$$a \longrightarrow b$$

$$c \longrightarrow d$$

The existence of a terminal object is not a given in a category, but is structure that some categories have that tells us something about them. The same is true for products.

Products

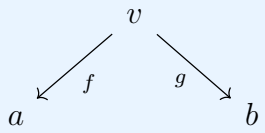
Products in category theory are different than the products of two numbers that we are used to, though it is related in a very generalized way. A product in a category is an object with morphisms from it to two other objects that is in some precise way the "simplest" or "best" such object. This is vague, but take the product of integers $3 * 5 = 15$ as an example. In this case, 15 is the simplest number such that both 3 and 5 are among its divisors. Contrast this to 30 which does have 3 and 5 as divisors, but also 2. This extra divisor is superfluous and makes 30 a worse candidate for $3 * 5$ than 15. Let's see this in diagram form.



Now let's make this precise in the general case.

Definition 23: Product

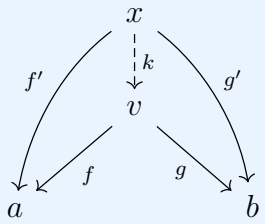
Let $a, b, v \in \mathcal{C}$ and morphisms $f : v \to a$ and $g : v \to b$ be given.



The object v is a product of a and b if the following holds:

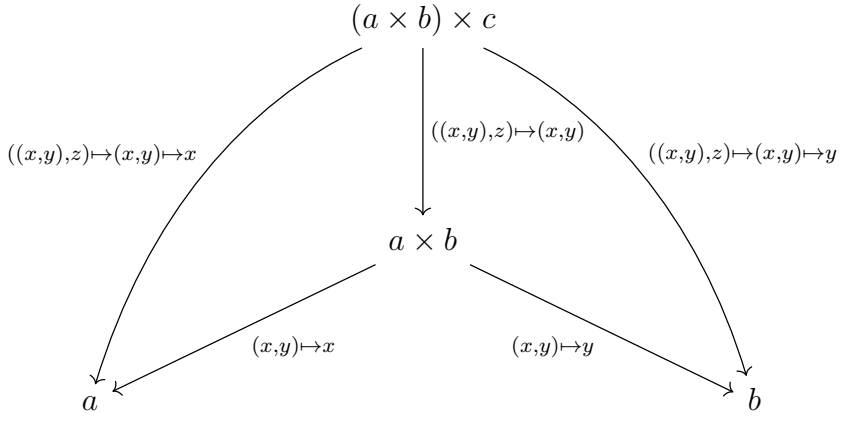
- For any other object x with morphisms $f' : x \to a$ and $g' : x \to b$, there is a unique morphism $k : x \to v$ such that $f \circ k = f'$ and $g \circ k = g'$.

This means the following diagram commutes:



This condition ensures that the object v is the universal or canonical object satisfying the pattern, with the morphisms of any other such object being able to be written as a sort of embellishment of the morphisms from v.

In **Set** the categorical product is exactly the Cartesian product of sets $a \times b = \{(x, y) \mid x \in a \text{ and } y \in b\}$, and f and g are the projections $f : (x, y) \mapsto x$ and $g : (x, y) \mapsto y$. Like in our integer example above there are other candidates, say the object $(a \times b) \times c$, but all of these other candidates will have a unique factorization to the Cartesian product: they all have extra stuff. Again we can see the example of $(a \times b) \times c$ in a diagram.



It turns out that the category of categories also has products, and they are very similar to the cartesian product of sets or the external direct product of groups.

Definition 24: The Product of Categories

Given two categories $\mathcal{C}$ and $\mathcal{D}$, their product is a new category $\mathcal{C} \times \mathcal{D}$ defined as follows.

1. A collection of objects (c, d) where $c \in \mathcal{C}$ and $d \in \mathcal{D}$.
2. For any two objects $(c, d), (c', d') \in \mathcal{C} \times \mathcal{D}$, a collection of morphisms $(f, g) : (c, d) \to (c', d')$ where $f : c \to c'$ is a morphism in $\mathcal{C}$ and $g : d \to d'$ is a morphism in $\mathcal{D}$.
3. Identity morphisms $\mathsf{id}_{(c,d)} = (\mathsf{id}_c, \mathsf{id}_d)$.
4. And composition of morphisms done by composing the morphisms component-wise, so $(h, j) \circ (f, g) = (h \circ f, j \circ g)$.

We can see how this is similar to the external direct product in the way that the structures are redefined component-wise, and thus how the product category inherits its structure from its underlying categories. This product of categories is central to our next section.

Adding Structure to Categories

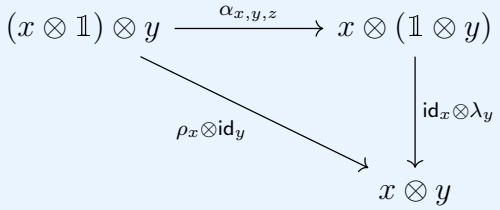
The last piece of category theory we need before moving on to how this all relates to the theory of systems is the notion of a monoidal category. Much like we can add additional structure on a set, we can do the same with a category. For instance, if we take a set S and add a binary operation $* : S \times S \to S$ that's associative and has a unit we get a structure called a monoid. If we do something similar and slightly more complicated to a category, we get a monoidal category.

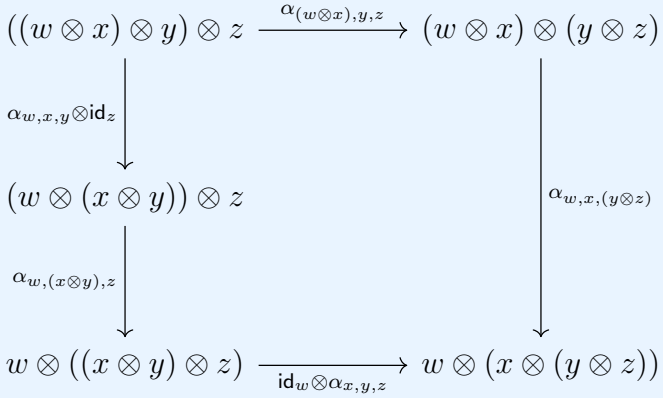
Definition 25: Monoidal Categories

A category $\mathcal{C}$ is monoidal if the following exist.

1. A functor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ called the monoidal product.
2. An object $\mathbb{1} \in \mathcal{C}$ called the unit.
3. A natural isomorphism $\alpha : (a \otimes b) \otimes c \Rightarrow a \otimes (b \otimes c)$ called the associator, with components $\alpha_{x,y,z} : (x \otimes y) \otimes z \to x \otimes (y \otimes z)$.
4. A natural isomorphism $\lambda : \mathbb{1} \otimes a \Rightarrow a$ called the left unitor with components $\lambda_x : \mathbb{1} \otimes x \to x$.
5. A natural isomorphism $\rho : a \otimes \mathbb{1} \Rightarrow a$ called the right unitor with components $\lambda_x : x \otimes \mathbb{1} \to x$.

All of the above must exist such that the following two diagrams, called the triangle identity and the pentagon identity, commute.





We can create a special kind of monoidal category, called a cartesian category, with two additional properties.

Definition 26: Cartesian Monoidal Category

A monoidal category $\mathcal{C}$ is cartesian when its monoidal product is its categorical product and its monoidal unit is its terminal object. This obviously means that $\mathcal{C}$ must contain all products and have a terminal object.

An example of this would be **Set**, where the monoidal product is the Cartesian product and the monoidal unit is the singleton set.

Categorical System Theory

Systems are everywhere in science and engineering. Whether discrete or continuous, deterministic or stochastic, all such systems have two things in common: states that they can be in, and rules for how the system's current state changes [8]. These two features alone describe what are called closed systems. Closed systems are systems that don't interact with others or with an outside environment that they're a part of. This is a hobbling limitation. In order to open these systems up, we need to give them an interface. We need them to be able to accept inputs that shape the way they evolve, and we need them to be able to expose some part of their current state to their surrounding environment. This is the gift that category theory will give us. By opening our systems up, we open them to the power of composition. We can connect them.

The Category $\mathbf{Lens}_{\mathcal{C}}$

Given a cartesian category, it's possible to construct a new category of systems whose states are drawn from the objects of your base category and whose rules for updating their state are drawn from the morphism of your base category. We call this category $\mathbf{Lens}_{\mathcal{C}}$, where $\mathcal{C}$ is the base category. The objects in this category are called arenas and the morphisms between them are called lenses [8].

Definition 27: Lenses

Given a cartesian category $\mathcal{C}$ and objects $A^-, A^+, B^-, B^+ \in \mathcal{C}$, a lens consists of a passforward map $f : A^+ \to B^+$ and a passback map $f^\# : A^+ \times B^- \to A^-$ between two arenas as follows:

$$\begin{pmatrix} f^\# \\ f \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} B^- \\ B^+ \end{pmatrix}$$

For our purposes we will be sticking to one such cartesian category: the category of lenses over the category of Euclidean spaces and smooth functions, $\mathbf{Lens}_{\mathbf{Euc}}$.

Definition 28: The Category $\mathbf{Lens}_{\mathcal{C}}$

Given the cartesian category $\mathcal{C}$, the category $\mathbf{Lens}_{\mathcal{C}}$ has the following properties.

1. A collection of objects called arenas. An arena $\begin{pmatrix} A^- \\ A^+ \end{pmatrix}$ is a pair of objects in $\mathcal{C}$.
2. For each pair of arenas a collection of morphisms $\begin{pmatrix} f^\# \\ f \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} B^- \\ B^+ \end{pmatrix}$ called lenses.
3. For each arena an identity lens $\begin{pmatrix} \pi_2 \\ \mathsf{id}_{A^+} \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} A^- \\ A^+ \end{pmatrix}$ where the passback map π_2 is the projection $\pi_2 : A^+ \times A^- \to A^-$.

Definition: The Category $\mathbf{Lens}_{\mathcal{C}}$, cont.

4. For any two compatible lenses a composite lens as follows:

$$\begin{pmatrix} f^{\#} \\ f \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} B^- \\ B^+ \end{pmatrix}$$

$$\begin{pmatrix} g^{\#} \\ g \end{pmatrix} : \begin{pmatrix} B^- \\ B^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} C^- \\ C^+ \end{pmatrix}$$

$$\begin{pmatrix} g^{\#} \\ g \end{pmatrix} \circ \begin{pmatrix} f^{\#} \\ f \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} C^- \\ C^+ \end{pmatrix}$$

   such that the passforward map is defined as $a^+ \mapsto g(f(a^+))$, and the passback map is defined as $(a^+, c^-) \mapsto f^{\#}(a^+, g^{\#}(f(a^+), c^-))$.

$\mathbf{Lens}_{\mathcal{C}}$ is also a monoidal category with the monoidal product being defined as follows.

1. The monoidal unit is the arena $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

2. Given lenses $\begin{pmatrix} f^{\#} \\ f \end{pmatrix} : \begin{pmatrix} A^- \\ A^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} B^- \\ B^+ \end{pmatrix}$ and $\begin{pmatrix} g^{\#} \\ g \end{pmatrix} : \begin{pmatrix} C^- \\ C^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} D^- \\ D^+ \end{pmatrix}$, the monoidal product

$$\begin{pmatrix} f^{\#} \\ f \end{pmatrix} \otimes \begin{pmatrix} g^{\#} \\ g \end{pmatrix} : \begin{pmatrix} A^- \times C^- \\ A^+ \times C^+ \end{pmatrix} \leftrightarrows \begin{pmatrix} B^- \times D^- \\ B^+ \times D^+ \end{pmatrix}$$

   such that the passforward map is defined as $(a^+, c^+) \mapsto (f(a^+), g(c^+))$ and the passback map is defined as $((a^+, c^+), (b^-, d^-)) \mapsto ((f^{\#}(a^+, b^-)), g^{\#}(c^+, d^-))$.

Note that since $\mathcal{C}$ is cartesian, this means that 1 is the terminal object in $\mathcal{C}$ and $A^- \times C^-$ is the categorical product of objects A^- and C^- in $\mathcal{C}$.

Case Study: Modeling Transcription Networks

To test out this formalism with a real world example, we will be modeling a small portion of the transcription network of the *e. coli* bacterium. Specifically we will be modeling the

multi-output feed forward loop that's responsible for the production of proteins that create a nanometer-scale motor and flagella propeller that the bacteria produce to move themselves away from stressful environments. Before we dig into that though, let's look at the basics of how gene transcription works.

Gene Transcription

Gene transcription is the process by which genes in a cell's DNA produce proteins. At a high-level, the process consists of environmental signals, transcription factors, promoters and genes, mRNA, and response proteins. When an environmental signal activates a related transcription factor, this activated transcription factor interacts with the promoter region of a gene as either an activator or a repressor for the gene. This activation or repression then leads to the increased or decreased production of mRNA from that gene and, therefore, an increase or decrease in the production of the gene's related proteins.

The activation or repression curve of the interaction of a transcription factor and the promoter region of a gene can be modeled by either a Hill function (sigmoidal curve ranging from β_0 to β) or, more simply, as a logic step function. This is how we will model them in this case study.

$$\text{Activator:} \quad f(X^*) = \frac{\beta X^{*n}}{K^n + X^{*n}} \quad \text{or} \quad \beta * P(X^* > K)$$

$$\text{Repressor:} \quad f(X^*) = \frac{\beta}{1 + \left(\frac{X^*}{K}\right)^n} \quad \text{or} \quad \beta * P(X^* < K)$$

Activation functions can also be multivariate, with a promoter responding to multiple transcription factors. These multivariate functions can take many forms, with some acting similar to logic gates or behaving additively.

$$f(X^*, Y^*) = \beta * P(X^* > K_x \wedge Y^* > K_y)$$

$$f(X^*, Y^*) = \beta * P(X^* > K_x \vee Y^* > K_y)$$

$$f(X^*, Y^*) = \beta_x X^* + \beta_y Y^*$$

The dynamics of transcription networks, the way that the system responds to input signals, are well modeled by the following differential equation, where Y is the resulting

protein concentration, β is the maximal activity, and $\alpha = \alpha_{\text{deg}} + \alpha_{\text{dil}}$ is the combined protein degradation and dilution rates [1].

$$\frac{dY}{dt} = \beta f(X^*) - \alpha Y$$

It's common for the product of a gene to be a transcription factor for other genes. By connecting all of these relations together into a graph we get a gene transcription network.

The *e. coli* Motor Flagella Network

The transcription network that we will be modeling is the motor flagella network of *e. coli* bacteria. This network is a graph with 14 nodes and 25 edges arranged in a common subgraph called a multi-output feed forward loop shown in figure 1 below [1].

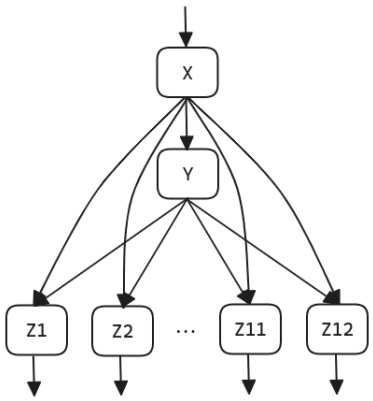


Figure 1: multi-output feed forward loop

In this multi-output FFL every edge represents an activator relationship, and all of the activation functions of the output genes $Z1$, $Z2$, ..., $Z12$ mix their input signals as a logical OR gate.[7]

This subgraph architecture has several features that are very important to the purpose it serves. First, the feed forward relation between genes X and Y and the OR gate mixing in the output genes' activation functions create a sign-sensitive filter to short fluctuations in the input signal. This ensures that the process is robust to signal noise and is unlikely to stop prematurely. Second, the activation thresholds for X and Y on the output genes can be

[7]In reality they're combined as a weighted sum. We are using OR gates since they simplify the computations while retaining the underlying behavior.

tuned to ensure that the output genes activate in the proper order: the first output genes to activate are also the first to deactivate.[8] This ordering is impossible without the mediating effect of gene Y [1].

Simulation Results

To simulate this network we used the excellent AlgebraicDynamics library for the Julia programming language. Full code for the simulation can be found in the appendix. Using the formalism above for wiring open systems together with lenses, we can represent our composed system as the following wiring diagram.

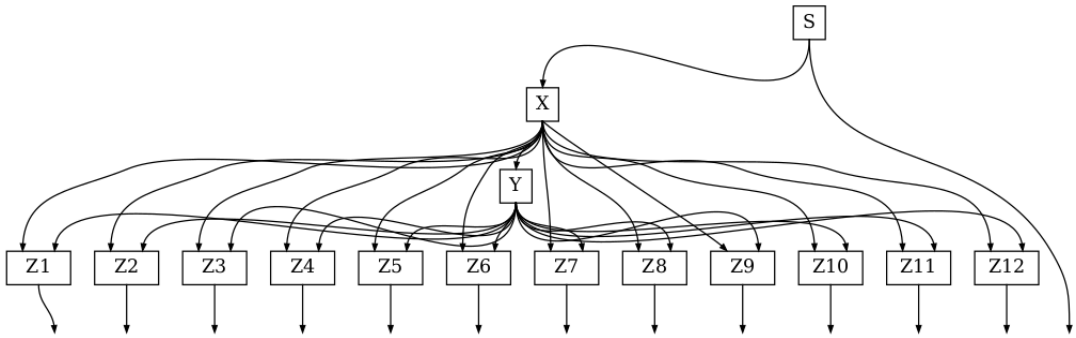


Figure 2: flagella network wiring diagram

Here each node is a "continuous machine"—i.e. a differential system—based on the transcription network dynamics below.

$$\text{Gene X:} \quad \frac{dX}{dt} = [\beta * P(S > \kappa_S) - \alpha * X]$$

$$\text{Gene Y:} \quad \frac{dY}{dt} = [\beta * P(X > \kappa_X) - \alpha * Y]$$

$$\text{Output Genes } Z1 \ldots Z12\text{:} \quad \frac{dZ_i}{dt} = [\beta * P(X > \kappa_X \ \mathsf{OR}\ Y > \kappa_Y) - \alpha * Z_i]$$

The signal node, S, is a contrived signal used to test the dynamics of the model. It's output is included in figure 3 below of the simulation's results. For a larger image, see appendix B.

The simulation stays true to almost all of the expected dynamics of the network. It's robust to signal noise and, other than a few exceptions, it activates and deactivates the genes

[8] This is called first-in-first-out, or FIFO, order.

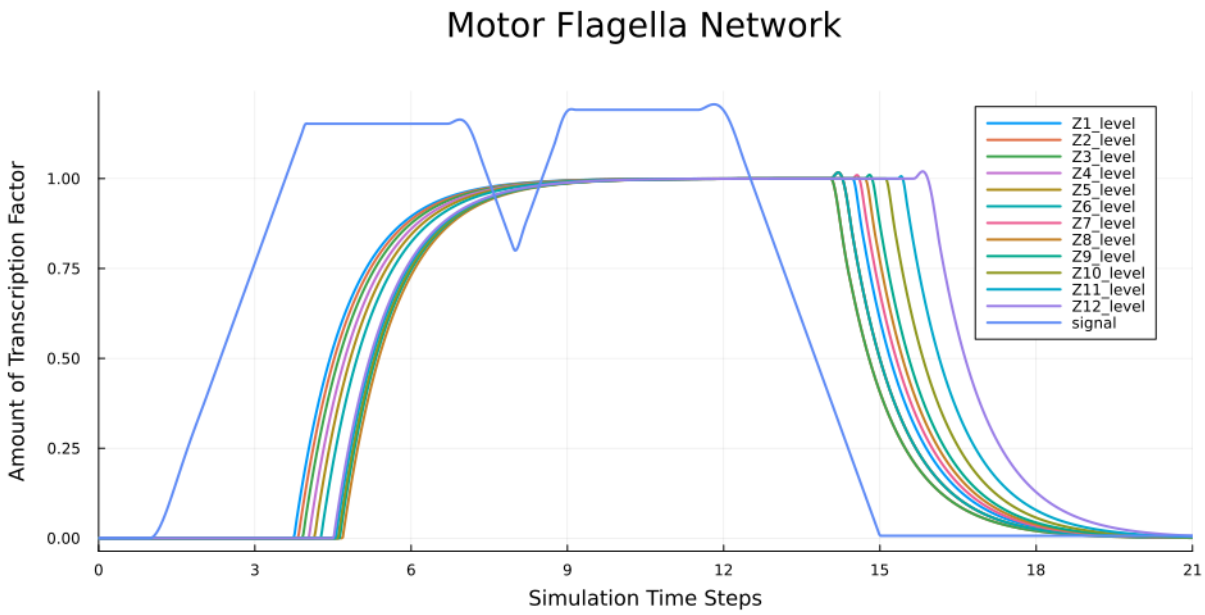


Figure 3: Results from the motor flagella system simulation

in the correct order. The reason for the errors in the ordering is the lack of real activation thresholds for the genes. All activation thresholds in the simulation were tuned by hand using only the crude heuristic that $\kappa_{z_n} < \kappa_{z_{n+1}}$ for the X activation and $\kappa_{z_n} > \kappa_{z_{n+1}}$ for the Y activation.

In future modeling work it should be possible to find real-world data with more time and journal access for secondary research. Or, if none are found, it should also be possible to define a loss function based on the expected dynamics and tune the model's parameters algorithmically using genetic programming or some other form of heuristic search. Lastly, it might also improve the accuracy of the model if we removed the logical approximations in it. We could replace the step functions with hill functions and the OR gates in the output genes' activation functions for a weighted sum.

Conclusion and Future Work

You may have noticed that there wasn't actually much category theory in the case study section. We didn't even explicitly use any lenses or arenas but kept them hidden inside the wiring diagrams. This was on purpose.

This pattern repeats in much of applied category theory. The category theory itself is used

to find the right abstraction over the domain and define what it means to compose whatever the atomic objects end up being (and often what it means to multiply them as well). Then on top of this theory is created a visual and diagrammatic language that encapsulates the abstraction [4, 5, 8]. This visual language is what we work with.

The benefits of using the diagrams is that they are often easier to work with and, crucially, are easier to understand; especially for stakeholders that don't have mathematical training. The diagram is simultaneously "the math" and the visual aid.

The ability to model a fairly complicated system in the time alotted for this project is evidence of how effective it is. This was done without background in modeling or gene transcription. And because the method is so general thanks to category theory, it's possible to use this same language across a variety of system types.

Had time and paper length allowed, we could have used this same language to model a multitude of other systems. We could've changed the underlying Cartesian category of our lenses from the category of Euclidean spaces and smooth maps to the category of sets and functions. With this we could compose and multiply finite state machines and design deterministic, mission critical software. Then we could change the lenses to monadic lenses and start composing and multiplying nondeterministic systems like Markov decision processes to model and design decision-making processes under uncertainty [8].

The ability to capture such disparate processes within a single formalism is the gift of abstraction. Learning this general, common language has helped us organize our thoughts about what systems are and how we use them to model phenomena in the world. We hope it has also helped us share them.

References

[1] Alon, U. *An introduction to systems biology: design principles of biological circuits*. Chapman and Hall/CRC, 2019.

[2] Asudeh, A., and Giorgolo, G. *Enriched meanings: Natural language semantics with category theory*, vol. 13. Oxford University Press, 2020.

[3] Cheng, E. *The joy of abstraction: an exploration of math, category theory, and life*. Cambridge University Press, 2022.

[4] Coecke, B., and Kissinger, A. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[5] Fong, B., and Spivak, D. I. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.

[6] Goldblatt, R. *Topoi: the categorial analysis of logic*. Elsevier, 2014.

[7] Milewski, B. *Category theory for programmers*. Blurb, 2018.

[8] Myers, D. J. Categorical systems theory. *Manuscript in preparation* (2022).

[9] Riehl, E. *Category theory in context*. Courier Dover Publications, 2017.

Appendix A: Simulation Code

```
using AlgebraicDynamics
using AlgebraicDynamics.DWDDynam
using Catlab.WiringDiagrams, Catlab.Programs

using LabelledArrays
using OrdinaryDiffEq, Plots, Plots.PlotMeasures

# Define the primitive systems
signal(u, x, p, t) = [0.4 * (2 < t < 5) - 0.4 * (8 < t < 9) + 0.4 * (9 < t < 10
    .1) - 0.4 * (11 < t < 12) + 0.4 * (12 < t < 13.05) - 0.4 * (14 < t < 17)]
flhDC(u, x, p, t) = [p.β1 * (x[1] > p.κ1) - p.α1 * u[1]]
fliA(u, x, p, t) = [p.β2 * (x[1] > p.κ2) - p.α2 * u[1]]
fliL(u, x, p, t) = [p.β3 * (x[1] > p.κ3 || x[2] > p.κ4) - p.α3 * u[1]]
fliE(u, x, p, t) = [p.β4 * (x[1] > p.κ5 || x[2] > p.κ6) - p.α4 * u[1]]
fliF(u, x, p, t) = [p.β5 * (x[1] > p.κ7 || x[2] > p.κ8) - p.α5 * u[1]]
flgA(u, x, p, t) = [p.β6 * (x[1] > p.κ9 || x[2] > p.κ10) - p.α6 * u[1]]
flgB(u, x, p, t) = [p.β7 * (x[1] > p.κ11 || x[2] > p.κ12) - p.α7 * u[1]]
flhB(u, x, p, t) = [p.β8 * (x[1] > p.κ13 || x[2] > p.κ14) - p.α8 * u[1]]
fliD(u, x, p, t) = [p.β9 * (x[1] > p.κ15 || x[2] > p.κ16) - p.α9 * u[1]]
flgK(u, x, p, t) = [p.β10 * (x[1] > p.κ17 || x[2] > p.κ18) - p.α10 * u[1]]
fliC(u, x, p, t) = [p.β11 * (x[1] > p.κ19 || x[2] > p.κ20) - p.α11 * u[1]]
meche(u, x, p, t) = [p.β12 * (x[1] > p.κ21 || x[2] > p.κ22) - p.α12 * u[1]]
mocha(u, x, p, t) = [p.β13 * (x[1] > p.κ23 || x[2] > p.κ24) - p.α13 * u[1]]
flgM(u, x, p, t) = [p.β14 * (x[1] > p.κ25 || x[2] > p.κ26) - p.α14 * u[1]]

# ContinuousMachine{T}(ninputs, nstates, noutputs, f, r)
s = ContinuousMachine{Float64}(0, 1, 1, signal, (u, p, t) -> u)
X = ContinuousMachine{Float64}(1, 1, 1, flhDC, (u, p, t) -> u)
Y = ContinuousMachine{Float64}(1, 1, 1, fliA, (u, p, t) -> u)
Z1 = ContinuousMachine{Float64}(2, 1, 1, fliL, (u, p, t) -> u)
Z2 = ContinuousMachine{Float64}(2, 1, 1, fliE, (u, p, t) -> u)
Z3 = ContinuousMachine{Float64}(2, 1, 1, fliF, (u, p, t) -> u)
Z4 = ContinuousMachine{Float64}(2, 1, 1, flgA, (u, p, t) -> u)
Z5 = ContinuousMachine{Float64}(2, 1, 1, flgB, (u, p, t) -> u)
Z6 = ContinuousMachine{Float64}(2, 1, 1, flhB, (u, p, t) -> u)
Z7 = ContinuousMachine{Float64}(2, 1, 1, fliD, (u, p, t) -> u)
Z8 = ContinuousMachine{Float64}(2, 1, 1, flgK, (u, p, t) -> u)
Z9 = ContinuousMachine{Float64}(2, 1, 1, fliC, (u, p, t) -> u)
Z10 = ContinuousMachine{Float64}(2, 1, 1, meche, (u, p, t) -> u)
Z11 = ContinuousMachine{Float64}(2, 1, 1, mocha, (u, p, t) -> u)
Z12 = ContinuousMachine{Float64}(2, 1, 1, flgM, (u, p, t) -> u)


# WiringDiagram([external inports], [external outports])
```

```
# Box(name, [inports], [outports])
# add_box!(wiring_diagram, box)
motor_diagram = WiringDiagram([], [
    :Z1_level,
    :Z2_level,
    :Z3_level,
    :Z4_level,
    :Z5_level,
    :Z6_level,
    :Z7_level,
    :Z8_level,
    :Z9_level,
    :Z10_level,
    :Z11_level,
    :Z12_level,
    :signal,
    #    :X_level,
    #    :Y_level
])
signal_generator = add_box!(motor_diagram, Box(:S, [], [:signal_level]))
geneX = add_box!(motor_diagram, Box(:X, [:X_signal], [:X_level]))
geneY = add_box!(motor_diagram, Box(:Y, [:X_level], [:Y_level]))
geneZ1 = add_box!(motor_diagram, Box(:Z1, [:X_level, :Y_level], [:Z1_level]))
geneZ2 = add_box!(motor_diagram, Box(:Z2, [:X_level, :Y_level], [:Z2_level]))
geneZ3 = add_box!(motor_diagram, Box(:Z3, [:X_level, :Y_level], [:Z3_level]))
geneZ4 = add_box!(motor_diagram, Box(:Z4, [:X_level, :Y_level], [:Z4_level]))
geneZ5 = add_box!(motor_diagram, Box(:Z5, [:X_level, :Y_level], [:Z5_level]))
geneZ6 = add_box!(motor_diagram, Box(:Z6, [:X_level, :Y_level], [:Z6_level]))
geneZ7 = add_box!(motor_diagram, Box(:Z7, [:X_level, :Y_level], [:Z7_level]))
geneZ8 = add_box!(motor_diagram, Box(:Z8, [:X_level, :Y_level], [:Z8_level]))
geneZ9 = add_box!(motor_diagram, Box(:Z9, [:X_level, :Y_level], [:Z9_level]))
geneZ10 = add_box!(motor_diagram, Box(:Z10, [:X_level, :Y_level], [:Z10_level]))
geneZ11 = add_box!(motor_diagram, Box(:Z11, [:X_level, :Y_level], [:Z11_level]))
geneZ12 = add_box!(motor_diagram, Box(:Z12, [:X_level, :Y_level], [:Z12_level]))


#add_wires(wiring_diagram, [
#    (box, output number) => (box, input_number),
#    (box, output number) => (box, input_number),
#    etc.
#])
add_wires!(motor_diagram, [
    (signal_generator, 1) => (geneX, 1),
    (geneX, 1) => (geneY, 1),
    (geneX, 1) => (geneZ1, 1),
    (geneX, 1) => (geneZ2, 1),
    (geneX, 1) => (geneZ3, 1),
    (geneX, 1) => (geneZ4, 1),
    (geneX, 1) => (geneZ5, 1),
```

```
    (geneX, 1) => (geneZ6, 1),
    (geneX, 1) => (geneZ7, 1),
    (geneX, 1) => (geneZ8, 1),
    (geneX, 1) => (geneZ9, 1),
    (geneX, 1) => (geneZ10, 1),
    (geneX, 1) => (geneZ11, 1),
    (geneX, 1) => (geneZ12, 1),
    (geneY, 1) => (geneZ1, 2),
    (geneY, 1) => (geneZ2, 2),
    (geneY, 1) => (geneZ3, 2),
    (geneY, 1) => (geneZ4, 2),
    (geneY, 1) => (geneZ5, 2),
    (geneY, 1) => (geneZ6, 2),
    (geneY, 1) => (geneZ7, 2),
    (geneY, 1) => (geneZ8, 2),
    (geneY, 1) => (geneZ9, 2),
    (geneY, 1) => (geneZ10, 2),
    (geneY, 1) => (geneZ11, 2),
    (geneY, 1) => (geneZ12, 2),
    (signal_generator, 1) => (output_id(motor_diagram), 13),
    #(geneX, 1) => (output_id(motor_diagram), 14),
    #(geneY, 1) => (output_id(motor_diagram), 15),
    (geneZ1, 1) => (output_id(motor_diagram), 1),
    (geneZ2, 1) => (output_id(motor_diagram), 2),
    (geneZ3, 1) => (output_id(motor_diagram), 3),
    (geneZ4, 1) => (output_id(motor_diagram), 4),
    (geneZ5, 1) => (output_id(motor_diagram), 5),
    (geneZ6, 1) => (output_id(motor_diagram), 6),
    (geneZ7, 1) => (output_id(motor_diagram), 7),
    (geneZ8, 1) => (output_id(motor_diagram), 8),
    (geneZ9, 1) => (output_id(motor_diagram), 9),
    (geneZ10, 1) => (output_id(motor_diagram), 10),
    (geneZ11, 1) => (output_id(motor_diagram), 11),
    (geneZ12, 1) => (output_id(motor_diagram), 12)
])


# final system = oapply(d::WiringDiagram, ms::Vector{M}) where {M<:
    AbstractMachine}
system = oapply(motor_diagram, [s, X, Y, Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, Z9, Z10
    , Z11, Z12])


# Solve and plot
# x0 = LVector(X_signal=0)
u0 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
a = 10
b = 10
params = LVector(
```

```
    β1=a * 13 + b + 1, κ1=1, α1=1,
    β2=2 * a * 13 + b, κ2=a * 7 + b, α2=1,
    β3=1, κ3=a * 1 + b, κ4=a * 12 + b, α3=1,
    β4=1, κ5=a * 2 + b, κ6=a * 11 + b, α4=1,
    β5=1, κ7=a * 3 + b, κ8=a * 10 + b, α5=1,
    β6=1, κ9=a * 4 + b, κ10=a * 9 + b, α6=1,
    β7=1, κ11=a * 5 + b, κ12=a * 8 + b, α7=1,
    β8=1, κ13=a * 6 + b, κ14=a * 7 + b, α8=1,
    β9=1, κ15=a * 8 + b, κ16=a * 6 + b, α9=1,
    β10=1, κ17=a * 9 + b, κ18=a * 5 + b, α10=1,
    β11=1, κ19=a * 10 + b, κ20=a * 4 + b, α11=1,
    β12=1, κ21=a * 11 + b, κ22=a * 3 + b, α12=1,
    β13=1, κ23=a * 12 + b, κ24=a * 2 + b, α13=1,
    β14=1, κ25=a * 13 + b, κ26=a * 1 + b, α14=1
)
tspan = (0.0, 25.0)

# problem = ODEProblem(final system, u0, tspan, params)
# solution = solve(problem, Tsit5())
problem = ODEProblem(system, u0, tspan, params)
solution = solve(problem, Tsit5())

#plot(sol, rabbitfox_system, params,
#    lw=2, title="Lotka-Volterra Predator-Prey Model",
#    xlabel="time", ylabel="population size"
#)

plot(solution, system, params,
    lw=2, title="Motor Flagella Network",
    xlabel="time", ylabel="Presence of Transcription Factors Z",
    size=(2000, 1000)
)
```

Appendix B: Simulation Results