AN INVESTIGATION OF ELEMENTARY CATEGORY THEORY, WITH APPLICATIONS IN PURE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE

MATTHEW DRURY, BEN BROOK, AND OLIVER DIXON

Abstract. In this preliminary investigation of elementary Category Theory, we discuss the foundational concepts of analytical abstraction, coupled with an exploration of the methods by which the study of categories facilitate the discovery of fundamental insights into complex networks of mathematical structure. We exploit this understanding to establish basic parallels between categorical instantiations of structure and modern concepts in Abstract Algebra, Set Theory, and Logic. A commentary of Category Theory in Computer Science and Functional Programming is also included, throughout which we interlace theoretical discourse with concrete examples in the purely functional Haskell programming language.

Contents

1. **[MD]** Theoretical Underpinnings: Axiomatic Constructions 1
2. **[BB]** Category-Theoretic Interpretations of Familiar Structures 5
3. **[OD]** Further Applications: Functional Programming and λ-Calculus 8

Cited Works 12

1. Theoretical Underpinnings: Axiomatic Constructions

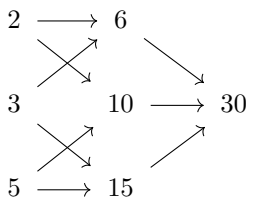
[Written by Matthew Drury]

Prose, Excluding Floats: 3–1/2 pages
Word Count of Prose: Approx. 1839 words

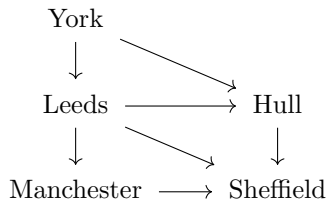
1.1. **The Essence of Categorical Thinking.** A category is a mathematical structure that links a collection of objects with non-symmetric relationships called morphisms. Each morphism can be said to traverse from one object to another. They are used to create abstract models of mathematical theories based on the role each object plays. Categories have the versatility to talk about different areas of mathematics in the same language, which is why it is sometimes said to be the mathematics of mathematics. The objects and morphisms have little restriction as to what they can represent which can be seen in the commutative diagrams illustrated in Figure 1.1 (Figure 1.1a is adapted from [Che22]).

These diagrams are usually used to give a sample of a category to demonstrate these properties. Mathematicians like to generalise with categories and consider infinite objects and morphisms, meaning we want categories to link all types of

Department of Mathematics, University of York, United Kingdom
E-mail addresses: md1499@york.ac.uk, bb1170@york.ac.uk, od641@york.ac.uk.
Date: Spring–Summer Term, 2023.



(A) Factors



(B) Train routes from York to Sheffield

FIGURE 1.1. Intuitive Examples of Categories

objects that we can define. The most valuable mathematical facts tend to be those that apply to the widest range of situations. For instance, uncovering the quadratic formula being more important than solving a particularly difficult quadratic. It makes all non-trivial quadratics easier to solve and uncovers a method to determine whether the solutions are real or complex using the discriminant term "b^2-4ac". By doing this we have found a relationship between all quadratics and their solutions. From there we could generalise even further and think about how this is significant for higher order polynomials.

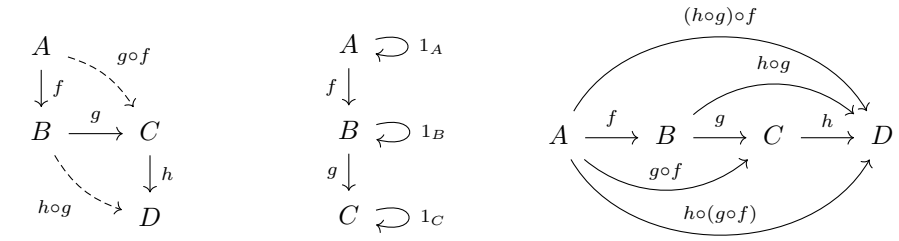
Morphisms are combined together in a process called composition. It means to do one morphism and then another, given that the target object of the first morphism is the same as the source of the second morphism. Above we can see that we can go from York to Leeds to Manchester. This means we have a morphism that goes from York to Manchester. There are four different ways to get to Sheffield, meaning four morphisms. These may not be the same morphisms, as they only share in source and target, not necessarily meaning. When two different composite morphisms are equal, we say they commute, hence the name *commutative diagram*.

There are also identity morphisms. These go from an object to itself and are equivalent to not doing a morphism at all, as when they are composed with another morphism f, the resulting composite morphism is equal to f. The morphism has done nothing, like how multiplying a number by 1(The multiplicative identity) does not change its value. It is similar to stating that something is equal to itself which is a trivial fact but an important feature of the structure of categories. It should be noted that it may not be the only morphism from one object to itself. These are called endomorphisms and the identity is just one of them.

Categories illuminate subtle similarities. By creating categories we abstract things to their roles and by comparing categories or spotting patterns and structures within them, we can see properties shared by things that would initially seem very different. Lines of thought in one context can be more easily translated into another by seeing equivalent structural features. Mathematicians can then try new methods to solve problems and prove new facts or describe multiple problems as one that is more general.

1.2. **Axioms and Notation.** We will now formally define a category. The following axioms are necessary for a structure to be considered a category.

(1) **Objects:** In a category $\mathcal{C}$ there is $\mathrm{ob}(\mathcal{C})$, which is a collection of all the objects of $\mathcal{C}$.



(A) Composition (B) Identities (C) Associativity

FIGURE 1.2. Essential Properties of Categorical Structures

(2) **Morphisms:** For each pair ordered pair of objects (A, B) in a category $\mathcal{C}$ we have $\hom_{\mathcal{C}}(A, B)$, also written as $\mathcal{C}(A, B)$. This is the collection of all morphisms from A to B, short for "homomorphism set". Homomorphisms are those that preserve the structure of objects and for many useful categories they are necessary for the axioms to hold. However, this property does not need to hold to use this notation. A morphism f is notated to map A to B by $f\colon A \to B$.

(3) **Composition:** For objects $A, B, C \in \text{ob}(\mathcal{C})$ if there exists morphisms $(f\colon A \to B) \in \hom_{\mathcal{C}}(A, B)$ and $(g\colon B \to C) \in \hom_{\mathcal{C}}(B, C)$, then there exists $(g \circ f\colon A \to C) \in \hom_{\mathcal{C}}(A, C)$

(4) **Identities:** Every object A in a category $\mathcal{C}$ has an identity morphism which maps from A to itself. It is notated $1_A\colon A \to A$ for $A \in ob(\mathcal{C}), 1_A \in \hom_{\mathcal{C}}(A, A)$. For $f \in \hom_{\mathcal{C}}(A, B), g \in \hom_{\mathcal{C}}(B, A)$, the identity has the property that $f \circ 1_A = f, 1_A \circ g = g$. Each identity is unique to its object. If there were two identities $I_1, I_2 \in A$ then $(f \circ I_1 = f = f \circ I_2) \implies I_1 = I_2$

(5) **Associativity:** A morphism $h \circ (g \circ f) = (h \circ g) \circ f$. This means that two morphisms are equal if they are composed from the same morphisms in the same order regardless of the order in which we compute the individual composite pairs.

1.3. **Size of Categories.** The collection $\text{ob}(\mathcal{C})$ and the collections $\hom_{\mathcal{C}}(A, B)$ do not have to be finite. The idea of category size means to place categories in a hierarchy of containment. This is to accommodate how $\text{ob}(\mathcal{C})$ and $\hom_{\mathcal{C}}(A, B)$ do not have to be sets either.

When defining an infinite collection of objects by their properties, it can create contradictions. Famously in set theory there is Russell's paradox [Rus03]. It states that if you have a set S that contains every set which doesn't contain itself, then suppose that S does not contain S, it would imply that S is in fact in S as it does not contain itself and vice versa. Formally,

$$S = \{x \text{ is a set} \mid x \notin x\}, (S \in S) \iff (S \notin S) \tag{1.1}$$

There were many examples of these paradoxes that were uncovered in the early 20th century. Mathematicians are too rigorous to allow such a thing, so to solve the issue they devised axiomatic systems to limit the properties that members of sets can be said to follow. This makes some sets very convoluted to define and

means there are well defined collections of objects that cannot be put into a set. Categories address this issue in a more elegant way. Firstly, categories do not claim to contain elements; the core concept is instead relationships. Secondly, categories have a hierarchy of containment such that they only contain categories "smaller" than themselves. It follows then that categories are not defined in a self referential manner like in Russell's paradox.

A small category is where $\mathrm{ob}(\mathcal{C})$ and each $\hom_{\mathcal{C}}(A, B)$ can be described as a set. A locally small category is where $\mathrm{ob}(\mathcal{C})$ does not form a set but each $\hom_{\mathcal{C}}(A, B)$ does. A large category is where nether $\mathrm{ob}(\mathcal{C})$ nor $\hom_{\mathcal{C}}(A, B)$ are a set. The hierarchy means that if one wanted a category where small categories are objects, it would have to be a large category. Then, if you wanted a category of large categories, you would need a super-large category and so on. This can be used to formally think about multiple layers of generalization and abstraction.

We can easily avoid Russell's paradox and create categories of categories. This is a sign that categories are a robust idea, as we can think about a collection of categories using the same language as within individual categories.

1.4. **Functors.** A functor maps between the objects of two categories, similarly to how a function maps between two sets. Functors preserve the structure of the category, so we want equivalent morphisms between the equivalent objects that have been chosen. This relationship is illustrated in Figure 1.3.

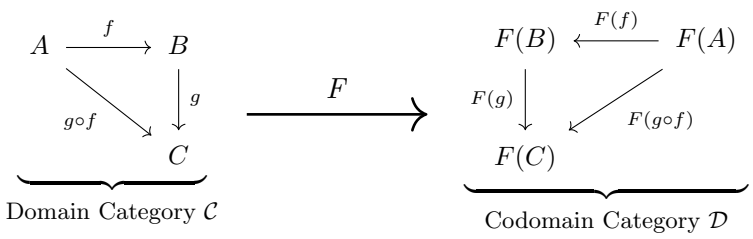


Figure 1.3. A functor F between simple categories $\mathcal{C}$ and $\mathcal{D}$

For categories $\mathcal{C}$, $\mathcal{D}$, if we have a functor $F\colon \mathcal{C} \to \mathcal{D}$ between them:

(1) **Objects:** For every $A \in \mathrm{ob}(\mathcal{C})$, $F_A \in \mathrm{ob}(\mathcal{D})$. Thus, A is mapped to $F(A)$.
(2) **Morphisms:** For every $f \in \hom_{\mathcal{C}}(A, B)$, we have $F(f) \in \hom_{\mathcal{D}}(F_A, F_B)$. Thus, f is mapped to $F(f)$.
(3) **Identities:** For every $A \in \mathrm{ob}\,\mathcal{C}$, we have 1_A. For every 1_A, there is corresponding morphism $F(1_A) \in \hom_{\mathcal{D}}(F(A), F(A))$. Thus, the identity of A is mapped to the identity of $F(A)$.
(4) **Composites:** For every $f \in \hom_{\mathcal{C}}(A, B)$, $g \in \hom_{\mathcal{C}}(B, C)$, we have that $F(g \circ f) = F_g \circ F_f$. This means that the order in which compositions and functors are applied is unimportant. This implies that composition in each category has an equivalent effect on its morphisms.

The commutative diagram above shows how from A we can traverse $g \circ f\colon A \to C$ and then apply F, or we can apply F, then traverse $F(g \circ f)\colon F(A) \to F(C)$.

In general, a functor is a way of finding the structure of one category in another. For instance, a category with two objects and one morphism between them $f\colon A \to B$ can have a functor to any category by selecting any morphism in it, which could

even be the identity of an object. This principle extends to categories of infinite objects.

1.5. **Non-Small Categories.** The large category **Cat** has objects which are all small categories and homomorphisms which are all functors between them. Each functor $F\colon \mathcal{C} \to \mathcal{D}$ can be seen as a function between sets $\mathrm{ob}(\mathcal{C})$ and $\mathrm{ob}(\mathcal{D})$. It could be injective, meaning that each object of $\mathcal{C}$ maps to a different object of $\mathcal{D}$. Otherwise, the functor is showing that one object in $\mathcal{D}$ has the same categorical properties as a structure of objects in $\mathcal{C}$.

If it is bijective, then the functor is an isomorphism, a morphism that has an inverse which composes to form the identity. This implies that $\mathcal{C}$ and $\mathcal{D}$ can reach the same categories with functors and be reached by the same categories with functors. In general, isomorphisms are a way of saying that objects indistinguishable from the perspective of the category they are in. However, it is not a suitable notion of equivalence between categories.

The identities in **Cat** are identity functors which map a category to a copy of itself with the exact same objects, morphisms, compositions. More generally it is one of the endofunctors which map categories to themselves.

Composition is simply doing one functor after another. For functors $F\colon \mathcal{C} \to \mathcal{D}$, $G\colon \mathcal{D} \to \mathcal{E}$, we can describe a single functor $G \circ F\colon \mathcal{C} \to \mathcal{E}$. When a lot of functors have been composed together we can get an increasingly subtle structural similarity between categories that we would not be able to notice by laying out diagrams next to each other and seeing a pattern.

---

2. Category-Theoretic Interpretations of Familiar Structures

[Written by Ben Brook]

Prose, Excluding Floats: 2–1/2 pages
Word Count of Prose: Approx. 1294 words

2.1. **Introductory Examples.** Now that we have established an axiomatic definition of the category, the natural next step involves the exploration of some simple applications of categories in a familiar context.

For the first three of these examples, our objects will be nothing but sets and morphisms nothing but functions (both with some additional restrictions). Categories of this type are sometimes called *concrete* [Awo10]. However, it's important to remember that categories need not contain functions nor sets. In fact, we will see this in our final introductory example, where we do away with both entirely!

All the exemplary categories that we are going to touch on will be infinite. This means that we won't be using any diagrams for this section of the report.

2.2. **Category of Sets.** To begin, an easily digestible example is the category of sets, whose objects are sets, and morphisms are total functions between sets. These are the set-theoretic functions with which we are familiar and have been using throughout the Autumn Term. We will follow a common denotation for this category: **Set** [Lei14]. We can easily verify that this category satisfies the category axioms seen in Section 1.2.

- There are objects, which are sets.

- There are morphisms between those objects, which are functions. Remember, however: morphisms are not necessarily functions! In fact, there are an abundance of examples where this is not the case.
- The composition of two functions is also a function, with its domain and codomain coming from the functions involved in the composition operation. Hence, the axiom of composition is satisfied.
- Let $A \in \mathrm{ob}\,(\mathbf{Set})$. We have that the identity morphism for A is $(1_A \colon A \to A) \in \hom(A, A)$, which maps each element of A to itself.
- The morphisms of **Set** are associative due to the native associativity of function composition. Let $A, B, C, D \in \mathrm{ob}\,(\mathbf{Set})$ be distinct. Also, let $f \in \hom(A, B)$, $g \in \hom(B, C)$, and $h \in \hom(C, D)$. Thus,

$$(f \circ g) \circ h = f \circ (g \circ h). \tag{2.1}$$

Hence, all the category axioms are satisfied. Going forwards, we will avoid the pedantry of listing that our category contains morphisms and objects.

2.3. **Category of Partially Ordered Sets.** A nice path to advance down is the exploration of a category whose objects are sets that satisfy certain properties. In other words, the objects are sets that have additional structure imposed upon them. We will also assign functions that preserve this structure to be the morphisms of our category. In this case, let's think about partially ordered sets, or *posets*. We must first establish some definitions:

- A poset is a set, A, with a relation, $\sim_A$, that is reflexive, transitive, and antisymmetric. We covered these properties in the Autumn Term.
- Let A and B be posets. A monotone map $m \colon A \to B$ is an order-preserving function such that

$$\forall\, a, b \in A \qquad a \sim_A b \implies m(a) \sim_B m(b). \tag{2.2}$$

We have a category **Pos** whose objects are posets and morphisms are monotone maps between those posets. Once more, we can verify the satisfaction of the relevant axioms.

- Let $A, B, C \in \mathrm{ob}\,(\mathbf{Pos})$ be distinct. Also, let $f \in \hom(A, B)$ and $g \in \hom(B, C)$. Since monotone maps are functions, we can compose them to get $g \circ f \colon A \to C$. Is this new function also a monotone map? Well, for all $a_1, a_2 \in A$, we have that

$$a_1 \sim_A a_2 \implies f(a_1) \sim_B f(a_2) \tag{2.3}$$

$$\implies g(f(a_1)) \sim_C g(f(a_2)). \tag{2.4}$$

  Thus, $g \circ f$ is a monotone map, and the composition axiom is satisfied.
- The identity morphism for any fixed object, A, is the monotone map $1_A \colon A \to A$, such that $a \mapsto a$.
- The associativity axiom is satisfied since our morphisms are functions, and function composition is associative.

2.4. **Category of Finite-Dimensional Vector Spaces.** We can look to Linear Algebra to provide another simple infinite category. The category of finite-dimensional $\mathbf{R}$-*vector spaces* is often denoted $\mathbf{FinVect}_{\mathbf{R}}$ [HHP08].

- A finite-dimensional $\mathbf{R}$-vector space is an element of $\{\mathbf{R}^n \mid n \in \mathbf{N}\}$.

- A linear map is a function, f, between two vector spaces, such that vector addition and scalar multiplication is preserved. That is,

$$f(\vec{u}+\vec{v}) = f(\vec{u}) + f(\vec{v}); \qquad [\textbf{Additivity}] \tag{2.5}$$

$$f(\lambda\vec{u}) = \lambda f(\vec{u}), \qquad [\textbf{Homogeneity}] \tag{2.6}$$

  with $\lambda \in \mathbf{R}$ and $\vec{u}, \vec{v} \in \mathbf{R}^n$ for some $n \in \mathbf{N}$.

The objects of $\mathbf{FinVect}_{\mathbf{R}}$ are finite-dimensional real-valued vector spaces, and its morphisms are linear maps between them. We will verify that $\mathbf{FinVect}_{\mathbf{R}}$ satisfies the category axioms:

- Let $\mathbf{R}^a, \mathbf{R}^b, \mathbf{R}^c \in \mathrm{ob}\,(\mathbf{FinVect}_{\mathbf{R}})$ be distinct. Additionally, suppose that $f \in \hom(\mathbf{R}^a, \mathbf{R}^b)$ and $g \in \hom(\mathbf{R}^b, \mathbf{R}^c)$. Trivially, $g \circ f \colon \mathbf{R}^a \to \mathbf{R}^c$ is a map. Now, let $\vec{u}, \vec{v} \in \mathbf{R}^a$ and $\lambda \in \mathbf{R}$. We have that

$$\begin{aligned}(g \circ f)(\vec{u}+\vec{v}) &= g(f(\vec{u})) + g(f(\vec{v})) \\ &= (g \circ f)(\vec{u}) + (g \circ f)(\vec{v}) \qquad [\text{By Eqn. 2.5}]\end{aligned} \tag{2.7}$$

$$\begin{aligned}(g \circ f)(\lambda\vec{u}) &= \lambda g(f(\vec{u})) \\ &= \lambda(g \circ f)(\vec{u}), \qquad [\text{By Eqn. 2.6}]\end{aligned} \tag{2.8}$$

  so the axiom of composition is met.
- The identity morphism for any object is the linear map such that $\vec{u} \mapsto \vec{u}$.
- The associativity axiom is satisfied due to functional nature of the maps.

2.5. **Category of Propositions.** As mentioned at the start of this section, the final exemplary category that we cover is not concrete. In this regard, it is similar to the categories we will see later in this report. Here we leverage how objects and morphisms can be anything such that the category axioms are met.

In this case, we are going to explore a category whose objects are propositions and morphisms are proofs, which we will denote **Prop**. We will consider this category informally, by employing our well-established intuition of proofs. Suppose $P, Q \in \mathrm{ob}\,(\mathbf{Prop})$. If there is a proof which, under P, gives Q – or a proof leading from P to Q – then this gives a morphism $(f \colon P \to Q) \in \hom(P, Q)$ (which is not a function!) We will use $P \vdash Q$ to denote this. We can confirm the categorical nature of this structure:

- Let $P, Q, R \in \mathrm{ob}\,(\mathbf{Prop})$ be distinct and let $f \in \hom(P, Q), g \in \hom(Q, R)$. Then $P \vdash Q \wedge Q \vdash R$ so, intuitively, $P \vdash R$ and $g \circ f$ belongs to the category.
- We can also intuitively reason that there is a proof leading from any proposition to itself, so each object has an identity morphism.
- Finally, we know that morphism composition is associative, again through our intuition. Let $P, Q, R, S \in \mathrm{ob}\,(\mathbf{Prop})$ be distinct. Let $f \in \hom(P, Q)$, $g \in \hom(Q, R)$, and $h \in \hom(R, S)$. That $(h \circ g) \circ f$ is a morphism asserts both that there is a proof leading from Q to S, and that P leads to Q. Likewise, that $h \circ (g \circ f)$ is a morphism asserts both the existence of a proof leading from P to R, and that R leads to S. These two assertions are identical, thus morphism composition is associative.

We covered this category informally in that some facts stated above do not hold for *every* conceivable system of logic, but just some particularly well-behaved constructions[1]. Our intuition was hopefully enough to take value from the example. A further, and more rigorous, exploration of this concept is detailed by [BS09].

[1] In the scope of this report, the distinctions between varying systems of logic is unimportant.

3. Further Applications: Functional Programming and λ-Calculus

[Written by Oliver Dixon]

Prose, Excluding Floats: 3–1/2 pages
Word Count of Prose: Approx. 1072 words

3.1. **Functional Programming and Haskell.** In purely functional languages, there is no allowance for context, or mutable variables of any kind. Each function must accept some data, perform some strict transformation upon the data—as defined by the algorithm—and return the result. Whilst this robust paradigm does open a wide range of mathematical avenues involving proof, safety, and reproducibility, the prohibition of stateful computation renders many common tasks, such as I/O or socket communication, largely impossible, as these imperatively defined operations inherently contravene the purity principles of functional programming.

Haskell is a commonly used purely functional programming language, and suffers, as do all languages in the same class, from this blaring issue. Indeed, early versions of Haskell did not support the chaining of stateful computation in any sense, due to the obligatory absence of a fixed execution order in functional paradigms; programmers were forced to resort to breaking the purity of the language through aesthetically unpleasant techniques, ultimately obviating the mathematical essence of the Haskell formal type system.

Due to the strength of the Haskell type system and function interface, we may define a corresponding category, **Hask**, within which the objects are Haskell types, and the morphisms are functions[2]. By analysing the structure of **Hask**, its endofunctors, and the categories formed by taking product, we can draw a swathe of parallels between generic purely functional paradigms; these connexions will eventually reveal the powerful concept of *monads*, allowing stateful computation, control flow, and error-handling.

3.2. **Monoidal Categories and Monoids.** Before fully exploiting the structure of **Hask**, we must develop the theory of *monoidal categories* and their corresponding *monoids*. Monoidal categories can be regarded as a six-tuple $(\mathcal{C}_0, \otimes, I, \alpha, \lambda, \rho)$, containing various components [Kel82]:

- A base category, $\mathcal{C}_0$;
- A bifunctor $\otimes\colon \mathcal{C}_0 \times \mathcal{C}_0 \to \mathcal{C}_0$;
- An identity object $I \in \mathrm{ob}\,\mathcal{C}_0$;
- An associativity natural transformation $\alpha_{A,B,C}\colon (A \otimes B) \otimes C \to A \otimes (B \otimes C)$;
- A left-identity natural transformation $\lambda_A\colon I \otimes A \to A$;
- A right-identity natural transformation $\rho_A\colon A \otimes I \to A$.

To maintain brevity, the natural transformations are often omitted from the tuple-descriptions of monoidal categories: $(\mathcal{C}_0, \otimes, I)$. In this context, the natural transformations α, λ, and ρ are used to induce certain properties on $\otimes$—*up to isomorphism*—which it may not possess natively; the effects of these transformations can be illustrated as morphisms on a pair of abstracted commutative diagrams (c.f. Figure 3.1).

[2]Due to the λ-Calculus concept of *currying*, named after Haskell Curry, functions taking multiple arguments may be decomposed into a chain of function compositions, in which each function strictly accepts and returns a single argument. This is made explicit in Haskell, where the type signature of a function `f` may be defined as `f :: a -> b -> c`, invoked as `f a b`, and expected to return a value of type `c`. This function signature is trivially equivalent to the *uncurried* form of `f`, defined as `f :: (a -> b) -> c`.

Then, monoids can be considered as three-tuples, consisting of an object in a monoidal category $(\mathcal{C}, \otimes, I)$, coupled with two transformations:

- A base object $M \in \operatorname{ob}\mathcal{C}_0$;
- A *multiplication* transformation $\mu \in \hom_{\mathcal{C}}(M \otimes M, M)$;
- A *unit* transformation $\eta \in \hom_{\mathcal{C}}(I, M)$.

Once more, the natural transformations from the parent monoidal category $\mathcal{C}$ can be used to induce properties on μ, as in Figure 3.2.

$$\begin{array}{ccccc}
[(A \otimes B) \otimes C] \otimes D & \xrightarrow{\alpha} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\alpha} & A \otimes [B \otimes (C \otimes D)] \\
{\scriptstyle \alpha \otimes \mathrm{id}\, D}\downarrow & & & & \uparrow{\scriptstyle \mathrm{id}\, A \otimes \alpha} \\
[A \otimes (B \otimes C)] \otimes D & & \xrightarrow[\alpha]{} & & A \otimes [(B \otimes C) \otimes D]
\end{array}$$

(A) Associativity on $\otimes$ induced by α

$$\begin{array}{ccc}
(B \otimes I) \otimes C & \xrightarrow{\alpha} & B \otimes (I \otimes C) \\
& {\scriptstyle \rho \otimes \mathrm{id}\, C} \searrow \quad \swarrow {\scriptstyle \mathrm{id}\, B \otimes \lambda} & \\
& B \otimes C &
\end{array}$$

(B) Left- and right-identities of objects A, B, C, D induced by λ and ρ

FIGURE 3.1. **(Monoidal Categories)** The natural transformations invoke commutativity on the bifunctor $\otimes$.

$$\begin{array}{ccccc}
(M \otimes M) \otimes M & \xrightarrow{\alpha} & M \otimes (M \otimes M) & \xrightarrow{\mathrm{id}\, M \otimes \mu} & M \otimes M \\
{\scriptstyle \mu \otimes \mathrm{id}\, M}\downarrow & & & & \downarrow{\scriptstyle \mu} \\
M \otimes M & & \xrightarrow[\mu]{} & & M
\end{array}$$

(A) Associativity on μ induced by α

$$\begin{array}{ccccc}
I \otimes M & \xrightarrow{\eta \otimes \mathrm{id}\, M} & M \otimes M & \xleftarrow{\mathrm{id}\, M \otimes \eta} & M \otimes I \\
& {\scriptstyle \lambda} \searrow & {\scriptstyle \mu}\downarrow & \swarrow {\scriptstyle \rho} & \\
& & M & &
\end{array}$$

(B) Left- and right-identities of M induced by λ and ρ

FIGURE 3.2. **(Monoids)** The natural transformations from the parent monoidal category also apply within the monoid.

It is important that the domain of the μ transformation is a *product combination* of the monoidal object; this will allow the imperative-like threading of state in functional paradigms.

3.2.1. *Examples of Monoidal Categories: Set Theory.* The canonical example of a monoidal category whose bifunctor/tensor product is not associative is **Set**, with the cross product; this is not *naturally associative*, but can be made associative up to isomorphism with a suitable choice of the natural transformation α. The details are established in [FS18], using an associativity transform as described in Equation 3.1, providing a monoidal category of the form $(\mathbf{Set}, \times, I)$, where I represents some fixed singleton.

$$\alpha_{A,B,C} \colon (A \times B) \times C \to A \times (B \times C) \tag{3.1}$$

3.2.2. *Examples of Monoidal Categories: Haskell.* In Haskell, the simplest practical implementation of a monoidal category is outlined in Listing 3.1, where the base category is **Hask**, the identity is the empty tuple, and the bifunctor is the tuple-building native function.

```haskell
type (,) :: * -> * -> * -- The type signature of the tuple-builder
cross :: a -> b -> (a, b) -- Uncurried function signature
cross = (,) -- A trivial application of the binary packing function
```

Listing 3.1. A binary Haskell function `cross` that encodes its arguments into a tuple. In category-theoretic language, the corresponding monoidal category could be expressed as the three-tuple (**Hask**, `cross`, `()`).

3.3. **The Category of Endofunctors.** For the purposes of Functional Programming, and the wider formal treatment of functional type systems, a particularly useful monoidal category concerns the *category of endofunctors* over some fixed base category $\mathcal{C}_0$. Denoted as $\mathbf{Endo}(\mathcal{C}_0)$, this forms a monoidal category with the associated bifunctor being the standard operation of endofunctor composition; the identity element is the obligatory identity endofunctor. Objects in $\mathbf{Endo}(\mathcal{C}_0)$ are the endofunctors over $\mathcal{C}_0$, and morphisms are the natural transformations between these objects. Monoids in the category of endofunctors are sometimes called monads. Thus, endofunctors over $\mathcal{C}_0$ with appropriately selected morphisms μ and η are henceforth termed as monads [Mac98].

3.3.1. *Monads in Haskell.* An abbreivated definition of a Haskell `Monad` is given in Listing 3.2; here we see the semi-curried form of the *bind operator*. The importance of `>>=` is reflected by its inclusion in the Haskell logo!

```haskell
type Monad :: (* -> *) -> Constraint
class Applicative m => Monad m where
        (>>=) :: m a -> (a -> m b) -> m b -- The fabled "bind" operator!
        return :: a -> m a -- Inject unstructured data into a monad
```

Listing 3.2. The Haskell `>>=` and `return` functions allow programmers to interact with the `Monad` class in the categorical sense.

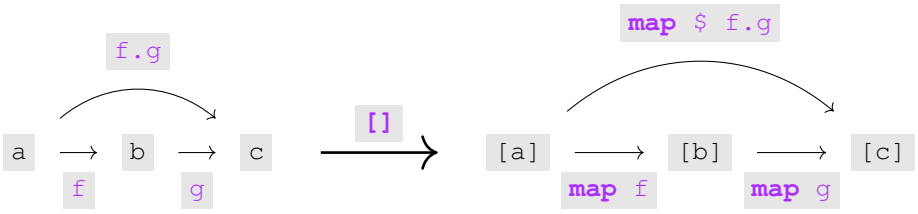


FIGURE 3.3. The action of the `[]` functor on **Hask**. The list constructor functor *lifts* the Haskell types `a`, `b`, and `c` into the list context.

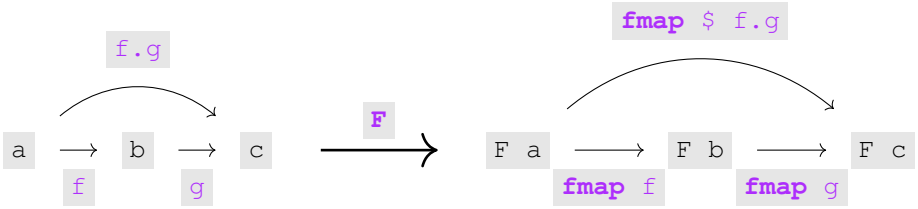


FIGURE 3.4. The action of the generic functor `F` on some **Hask**. The mappings of the objects are implicit in the mappings of the morphisms, as domains and codomains must be preserved between isomorphic categories.

3.4. **Functors in Haskell.** This interdisciplinary review of Category Theory and Functional Programming becomes useful when considering the category of endofunctors over the **Hask** category, **Endo** (**Hask**), thus forming a monoidal category with monads as endofunctors over **Hask** [MT19]. In Haskell, these are simply typed as `Functor`, defined as a typeclass providing appropriate mappings from **Hask** to **Hask** for types and functions, as shown in Listing 3.3. The Haskell function `fmap` is used to lift a function `a -> b`, embedded in **Hask** as the domain category, to the functorial context `f a -> f b`, embedded in **Hask** as the codomain category.

Haskell Functors can be solidified with the most trivial example: the *list constructor*, which takes types $A, B, C \in \mathrm{ob}\,\mathbf{Hask}$ and lifts them into the list structure with the `fmap` endofunctor[3]. This process is illustrated in case of lists in Figure 3.3, and in the general case in Figure 3.4.

3.5. **Controlling State with Haskell Monads.** To address our original problem of stateless computation, how might a *specific* usage of `Monad` allow the threading of state through pseudo-imperative function calls? By recalling the domain of the *multiplication* transformation on the monoid, as defined in Section 3.2, we have an immensely useful functor transformation **Endo** (**Hask**), such that $\mu\colon M \times M \to M$,

[3]In the Haskell `[]` instantiation of `Functor`, the `fmap` field is set to the stricter `map` function; this is an unimportant implementation detail in this case.

```haskell
type Functor :: (* -> *) -> Constraint
class Functor (f :: * -> *) where
        fmap :: (a -> b) -> ( f a -> f b ) -- The curried form of fmap.
        (<$) :: a -> f b -> f a
        {-# MINIMAL fmap #-}

instance Functor [] -- The list constructor is an exemplary Functor!
instance Functor Maybe -- Maybe is used for controlling uncertainty.
```

Listing 3.3. The Haskell `Functor` type signature, of which the list type constructor `[]` is an instance.

where M is an object in $\mathcal{C}_0 \coloneqq \mathbf{Hask}$. This can be implemented as a Haskell function which allows the merging of two **Hask** instances into a single *combined instance*. By exploiting the lazy evaluation of Haskell, such that functions are only executed when directly invoked, programmers can enforce an execution order by chaining evaluations of some transformation μ. Context is achieved by *applying* a **Hask**, as the standard function parameter, to a given context; the function must then return the transformed context according to the prescribed algorithm or process.

Cited Works

[Che22] Eugenia Cheng. *The Joy of Abstraction*. Cambridge, UK: Cambridge University Press, 2022, p. 114. ISBN: 978-1-108769389.

[Rus03] Bertrand Russell. *The Principles of Mathematics*. Cambridge, UK: Cambridge University Press, 1903. ISBN: 978-1-282284036.

[Awo10] Steve Awodey. *Category Theory*. 2nd Ed. Oxford Logic Guides. Oxford, UK: Oxford University Press, 2010, p. 7. ISBN: 978-0-199587360.

[Lei14] Tom Leinster. *Basic Category Theory*. 1st Ed. Cambridge, UK: Cambridge University Press, 2014, p. 11. ISBN: 978-1-107044241.

[HHP08] Masahito Hasegawa, Martin Hofmann, and Gordon Plotkin. "Finite Dimensional Vector Spaces Are Complete for Traced Symmetric Monoidal Categories". In: *Lecture Notes in Computer Science* 4800 (2008), p. 367.

[BS09] John Baez and Mike Stay. "Physics, Topology, Logic and Computation: A Rosetta Stone". In: *New Structures for Physics*. Heidelberg, Germany: Springer, 2009, pp. 39–43. DOI: 10.1007/978-3-642-12821-9_2.

[Kel82] Gregory M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge, UK: Cambridge University Press, 1982. ISBN: 978-0-521287029.

[FS18] Brendan Fong and David Spivak. *Seven Sketches in Compositionality*. *arXiv* pre-print, 2018, p. 136. DOI: 10.48550/ARXIV.1803.05316.

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. 2nd Ed. Graduate Texts in Mathematics. London, UK: Springer, 1998, p. 138. ISBN: 0-387984038.

[MT19] Bartosz Milewski and Igal Tabachnik. *Category Theory for Programmers*. Blurb, 2019, pp. 229–230. ISBN: 978-0-464243878.

[Section 3 is wholly dedicated to *MQ*.]