Category Theory for Discrete Mathematics

A Companion to CS251

Supplementary Notes

Abstract

Category theory is sometimes called “abstract nonsense”—and honestly, when you first encounter it, the abstraction can feel unmotivated. Why care about “objects and morphisms” when you could just talk about sets and functions?

Here’s the secret: category theory isn’t about being abstract for its own sake. It’s about noticing that the *same patterns* keep appearing across different areas of mathematics and computer science. Once you see these patterns, you can transfer intuition from one domain to another. You understand *why* certain constructions work, not just *that* they work.

These notes develop category theory alongside the CS251 curriculum, introducing categorical concepts when they illuminate the discrete math you’re already learning. Think of this as a second lens on the same material—one that reveals hidden structure and connections.

Contents

1 Week 1–2: The Category of Sets

Why Categories?

Here's a question that sounds almost too simple: what do we really need to know about a mathematical object?

One answer: we need to know what it *is*—its internal structure. A set is a collection of elements. A group is a set with an operation satisfying certain axioms. And so on.

But there's another answer, one that turns out to be surprisingly powerful: we can understand an object by understanding how it *relates* to other objects. What are the functions from this set to that one? Which functions preserve the structure we care about?

This is the categorical perspective. Instead of asking "what is a set?", we ask "what are the functions between sets?" Instead of asking "what is a group?", we ask "what are the group homomorphisms?" The objects matter, but the *morphisms*—the structure-preserving maps—matter more.

1.1 Categories: Objects and Morphisms

Let's make this precise.

Definition 1.1 (Category)**.** A **category** $\mathcal{C}$ consists of:

1. A collection $\text{ob}(\mathcal{C})$ of **objects**
2. For each pair of objects A, B, a collection $\text{Hom}(A, B)$ of **morphisms** (or **arrows**) from A to B
3. For each object A, an **identity morphism** $\text{id}_A \in \text{Hom}(A, A)$
4. For morphisms $f \in \text{Hom}(A, B)$ and $g \in \text{Hom}(B, C)$, a **composition** $g \circ f \in \text{Hom}(A, C)$

subject to:

- **Associativity:** $(h \circ g) \circ f = h \circ (g \circ f)$
- **Identity:** $f \circ \text{id}_A = f$ and $\text{id}_B \circ f = f$

We write $f : A \to B$ to indicate $f \in \text{Hom}(A, B)$.

That's the formal definition, and it might seem like a lot of abstraction for not much payoff. But here's the thing: this definition captures an enormous range of mathematical structures. Once you prove something about categories in general, you get theorems about sets, groups, topological spaces, programming languages, and more—all for free.

Example 1.1 (The category **Set**)**.** The most fundamental example: the category **Set** has:

- Objects: all sets
- Morphisms: functions between sets
- Composition: function composition
- Identity: the identity function $\text{id}_A(x) = x$

This is the category you've been working in all along—you just didn't call it that.

Example 1.2 (Counting morphisms)**.** In **Set**, how many morphisms are there from $A = \{1, 2\}$ to $B = \{a, b, c\}$?

Each element of A can map to any element of B independently, so there are $|B|^{|A|} = 3^2 = 9$ morphisms. This is the same counting argument from Week 2's material on functions—but now we're calling these functions "morphisms" to emphasize that they're the *relationships* we care about.

1.2 Commutative Diagrams

Category theorists love diagrams. And once you get used to them, you'll love them too.

A diagram is just a picture showing objects as dots (or labels) and morphisms as arrows between them. The magic happens when we say a diagram *commutes*.

Definition 1.2 (Commutative diagram)**.** A diagram of objects and morphisms **commutes** if all paths between any two objects give the same composite morphism.

In other words: if there are two different ways to get from A to C by following arrows, they compose to the same morphism.

Example 1.3. The triangle:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\
 & C
\end{tikzcd}$$

commutes if and only if $g \circ f = h$. There are two paths from A to C: the direct route h, and the two-step route f then g. Commutativity says they're the same function.

Example 1.4. The square:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & B \arrow[d, "g"] \\
C \arrow[r, "k"'] & D
\end{tikzcd}$$

commutes if and only if $g \circ f = k \circ h$. Going across-then-down equals going down-then-across.

Why do we draw pictures instead of just writing equations? Two reasons. First, our visual system is remarkably good at tracking paths—much better than parsing nested compositions like $k \circ h \circ g^{-1} \circ f$. Second, the diagrams make structure visible. You can see at a glance how objects relate.

> **Programming Connection**
>
> Commutative diagrams are like type-directed specifications. The diagram says "these two ways of computing a result must agree," without specifying *how* to compute them. This is why diagrams appear in API documentation: they express invariants that implementations must satisfy.

1.3 Monomorphisms, Epimorphisms, and Isomorphisms

In Week 2, you learned about injective, surjective, and bijective functions. Those definitions talked about elements: "every element of B is hit" or "no two elements map to the same thing."

But here's a remarkable fact: we can characterize these properties *without mentioning elements at all.* We just need to talk about how functions compose.

Definition 1.3 (Monomorphism)**.** A morphism $f : A \to B$ is a **monomorphism** (or **mono**) if it is left-cancellable:

$$\text{For all } g, h : X \to A, \quad f \circ g = f \circ h \implies g = h$$

Think about what this says: if two paths into A become equal after applying f, they must have been equal all along. The function f doesn't create any "collisions" that weren't already there.

Definition 1.4 (Epimorphism)**.** A morphism $f : A \to B$ is an **epimorphism** (or **epi**) if it is right-cancellable:

$$\text{For all } g, h : B \to X, \quad g \circ f = h \circ f \implies g = h$$

This says: if two paths out of B agree when preceded by f, they must agree everywhere. The function f "reaches enough" of B that you can't hide disagreements.

Definition 1.5 (Isomorphism)**.** A morphism $f : A \to B$ is an **isomorphism** if there exists $g : B \to A$ such that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$. We write $A \cong B$.

Isomorphisms are the "perfect" morphisms: they have a two-sided inverse, so A and B are interchangeable as far as the category is concerned.

Theorem 1.1. *In* **Set***:*

1. *f is mono $\iff$ f is injective*
2. *f is epi $\iff$ f is surjective*
3. *f is iso $\iff$ f is bijective*

Proof. (Mono $\iff$ injective) Suppose f is mono and $f(a) = f(b)$. Here's the trick: define $g, h : \{*\} \to A$ by $g(*) = a$, $h(*) = b$. These are functions from a one-element set that "pick out" a and b. Then $f \circ g = f \circ h$ (both send $*$ to $f(a) = f(b)$), so by the mono property, $g = h$, hence $a = b$.

Conversely, suppose f is injective and $f \circ g = f \circ h$. For any x, $f(g(x)) = f(h(x))$, so $g(x) = h(x)$ by injectivity. Thus $g = h$. □

So in **Set**, the categorical definitions recover exactly what we already knew. Why bother with the new terminology?

Warning

Because in other categories, mono $\neq$ injective and epi $\neq$ surjective! For example, in the category of rings, the inclusion $\mathbb{Z} \hookrightarrow \mathbb{Q}$ is epic but not surjective. (The rationals aren't "hit" by integers, but any two ring homomorphisms out of $\mathbb{Q}$ that agree on $\mathbb{Z}$ must agree everywhere—because a ring homomorphism is determined by where it sends 1.)

The cancellation definitions are the "right" generalizations. They capture the essential property without relying on elements.

1.4 Sections, Retractions, and Idempotents

Sometimes a morphism doesn't have a full inverse, but it has a one-sided inverse. These partial inverses turn out to be important.

Definition 1.6 (Section and retraction)**.** Given $f : A \to B$:

- A **section** (right inverse) is a map $s : B \to A$ with $f \circ s = \mathrm{id}_B$.
- A **retraction** (left inverse) is a map $r : B \to A$ with $r \circ f = \mathrm{id}_A$.

The terminology might seem arbitrary, but it has geometric origins. Think of $f : A \to B$ as projecting a space onto a smaller one. A section "lifts" points back up; a retraction "pulls back" the projection.

If f has a section, then f must be surjective (in **Set**)—the section provides a way to "choose" a preimage for every element of B. If f has a retraction, then f must be injective—you can "undo" f on its image.

Definition 1.7 (Idempotent)**.** A map $p : A \to A$ is **idempotent** if $p \circ p = p$.

Idempotents are "projection-like" maps. Once you've applied p, applying it again does nothing—you've already projected onto the image. In **Set**, every idempotent $p : A \to A$ corresponds to a subset $B = \{p(a) : a \in A\}$ with inclusion $i : B \to A$ and retraction $r : A \to B$ such that $p = i \circ r$.

This connection between idempotents and subobjects generalizes to other categories, and it's surprisingly useful in programming (think: caching, memoization, normalization).

1.5 Terminal and Initial Objects

Some objects are special because of how *uniquely* they relate to everything else.

Definition 1.8 (Terminal and initial objects)**.** An object 1 is **terminal** if for every A there exists a unique map $A \to 1$. An object 0 is **initial** if for every A there exists a unique map $0 \to A$.

In **Set**, any singleton $\{*\}$ is terminal—there's exactly one function from any set to a one-element set (send everything to $*$). The empty set is initial—there's exactly one function from $\emptyset$ to any set (the empty function, which vacuously satisfies the definition of a function).

Here's a lovely perspective: elements of a set A correspond bijectively to morphisms $1 \to A$. A map from a singleton "picks out" a point. So category theory lets us talk about "elements" without ever mentioning elements—we just talk about maps from terminal objects. This is called the **generalized element** perspective.

1.6 Universal Properties: Products and Coproducts

Now we come to one of the deepest ideas in category theory: **universal properties**.

The usual way to define Cartesian products is: $A \times B = \{(a, b) : a \in A, b \in B\}$. That's a fine definition, but it's very specific to sets. What if we want products in other categories?

The categorical approach is different. Instead of saying what a product *is*, we say what a product *does*. We characterize it by its relationship to everything else.

Definition 1.9 (Product)**.** A **product** of objects A and B is an object $A \times B$ together with morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that:

For any object X with morphisms $f : X \to A$ and $g : X \to B$, there exists a *unique* morphism $\langle f, g\rangle : X \to A \times B$ making this diagram commute:

$$\begin{tikzcd}
& X \arrow[ld, "f"'] \arrow[d, dashed, "{\langle f,g\rangle}"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

Read this carefully: the product is “the best way to map into A and B simultaneously.” Any other way to do it factors uniquely through the product. The dashed arrow is determined by f and g—that’s the universal property.

In **Set**, the Cartesian product $A \times B = \{(a,b) : a \in A, b \in B\}$ with projections $\pi_1(a,b) = a$ and $\pi_2(a,b) = b$ satisfies this. The unique map is $\langle f, g\rangle(x) = (f(x), g(x))$.

Definition 1.10 (Coproduct)**.** A **coproduct** of objects A and B is an object $A+B$ together with morphisms $\iota_1 : A \to A+B$ and $\iota_2 : B \to A+B$ such that:

For any object X with morphisms $f : A \to X$ and $g : B \to X$, there exists a unique morphism $[f,g] : A+B \to X$ making this diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f"'] & A+B \arrow[d, dashed, "{[f,g]}"] & B \arrow[l, "\iota_2"'] \arrow[dl, "g"] \\
 & X &
\end{tikzcd}$$

Notice the arrows are reversed! Coproducts are “dual” to products. The coproduct is “the best way to map out of A or B.” In **Set**, this is the disjoint union.

Programming Connection

Products correspond to `pair` types, `struct`s, or records. Coproducts correspond to `Either` types, tagged unions, or `enum`s. The unique morphism $[f,g]$ is pattern matching:

```
case x of
  Left a  -> f(a)
  Right b -> g(b)
```

The universal property says this is *the* way to consume a sum type: you must handle both cases, and that determines a unique function.

1.7 Exercises

Exercise 1.1. Let $A = \{1,2,3\}$ and $B = \{a,b\}$.

(a) How many morphisms are there in $\mathrm{Hom}(A,B)$?

(b) How many morphisms are there in $\mathrm{Hom}(B,A)$?

(c) How many of the morphisms $A \to B$ are epimorphisms?

(d) How many of the morphisms $B \to A$ are monomorphisms?

Exercise 1.2. Prove that every isomorphism is both a monomorphism and an epimorphism.

Exercise 1.3. Let $f : A \to B$ and $g : B \to C$. Prove:

(a) If $g \circ f$ is mono, then f is mono.

(b) If $g \circ f$ is epi, then g is epi.

Exercise 1.4 (Universal property verification)**.** Let $A = \{1,2\}$, $B = \{a,b,c\}$, $X = \{x,y\}$. Define $f : X \to A$ by $f(x) = 1, f(y) = 2$ and $g : X \to B$ by $g(x) = a, g(y) = c$.

(a) Write out $A \times B$ explicitly.

(b) Construct the unique $\langle f, g \rangle : X \to A \times B$.

(c) Verify that $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$.

Exercise 1.5. The **exponential** B^A in **Set** is the set of all functions from A to B, with evaluation $\varepsilon : B^A \times A \to B$ defined by $\varepsilon(f, a) = f(a)$.

The universal property says: for any $g : X \times A \to B$, there exists a unique $\tilde{g} : X \to B^A$ such that $\varepsilon \circ (\tilde{g} \times \mathrm{id}_A) = g$.

Interpret this in programming: what is $\tilde{g}$ doing? (Hint: currying.)

Exercise 1.6. Show that elements of a set A correspond bijectively to morphisms $1 \to A$ in **Set**, where 1 is a singleton.

Exercise 1.7. Let $f : A \to B$.

(a) If there exists $s : B \to A$ with $f \circ s = \mathrm{id}_B$, prove that f is surjective.

(b) If there exists $r : B \to A$ with $r \circ f = \mathrm{id}_A$, prove that f is injective.

Exercise 1.8. Let $p : A \to A$ be idempotent in **Set**, and let $B = \{p(a) : a \in A\}$. Construct maps $r : A \to B$ and $i : B \to A$ such that $p = i \circ r$ and $r \circ i = \mathrm{id}_B$.

2 Week 3: Preorders as Categories and Functors

Why View Preorders as Categories?

Here's something that might seem like overkill: taking a simple concept (preorders) and dressing it up in categorical language. But this perspective reveals that preorders and categories are the *same idea at different levels of richness.*

A category has objects and morphisms, with possibly many morphisms between any two objects. A preorder has elements and relationships, with at most one relationship between any two elements (either $a \leq b$ or not). Preorders are "thin" categories—and once you see this, concepts from one world transfer to the other.

2.1 Preorders Are Categories

Definition 2.1 (Preorder)**.** A **preorder** $(P, \leq)$ is a set P with a relation $\leq$ that is reflexive ($a \leq a$) and transitive ($a \leq b$ and $b \leq c$ imply $a \leq c$).

Theorem 2.1. *Every preorder $(P, \leq)$ corresponds to a category:*

- *Objects: elements of P*
- *Morphisms: there is exactly one morphism $a \to b$ iff $a \leq b$ (and none otherwise)*
- *Identity: provided by reflexivity ($a \leq a$)*
- *Composition: provided by transitivity*

Such a category is called ***thin****—at most one morphism between any two objects.*

The category axioms (identity laws, associativity) are automatically satisfied because there's at most one morphism between any pair of objects. Any two composites with the same domain and codomain must be equal—there's only one morphism they could be!

Example 2.1. Consider divisibility on $\{1, 2, 3, 6\}$, where $a \leq b$ means $a \mid b$. As a category:

$$\begin{tikzcd}
 & 6 & \\
2 \arrow[ur] & & 3 \arrow[ul] \\
 & 1 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

The arrow $1 \to 6$ exists (since $1 \mid 6$) and equals the composition $1 \to 2 \to 6$. We don't draw it separately because it's the unique morphism—no extra information.

Example 2.2. The power set $(\mathcal{P}(X), \subseteq)$ is a preorder category. What are products and coproducts here?

- The product of A and B is their intersection $A \cap B$—the greatest lower bound.
- The coproduct of A and B is their union $A \cup B$—the least upper bound.

The universal properties of products/coproducts *become* the definitions of greatest lower bound and least upper bound. Category theory unifies these concepts.

2.2 Functors

If categories are the "stages" on which mathematics plays out, functors are the "translations" between stages. A functor takes objects to objects and morphisms to morphisms, while respecting the categorical structure.

Definition 2.2 (Functor)**.** A **functor** $F : \mathcal{C} \to \mathcal{D}$ between categories consists of:

1. A mapping $F : \mathrm{ob}(\mathcal{C}) \to \mathrm{ob}(\mathcal{D})$ on objects
2. For each $f : A \to B$ in $\mathcal{C}$, a morphism $F(f) : F(A) \to F(B)$ in $\mathcal{D}$

such that:

- $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ (preserves identities)
- $F(g \circ f) = F(g) \circ F(f)$ (preserves composition)

The key insight: a functor preserves the *structure* of how things compose. If you have a commutative diagram in $\mathcal{C}$, applying F gives a commutative diagram in $\mathcal{D}$.

Theorem 2.2. *A functor between preorder categories is exactly a* ***monotone function****:* $a \leq b \implies F(a) \leq F(b)$.

This is satisfying: monotone functions are the "structure-preserving maps" for preorders, and functors are the "structure-preserving maps" for categories. When we view preorders as thin categories, these concepts coincide.

Example 2.3 (Power set functor)**.** Here's a functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ that you've been using all along:

- On objects: $\mathcal{P}(A) = \{S : S \subseteq A\}$ (power set)
- On morphisms: $\mathcal{P}(f)(S) = \{f(x) : x \in S\}$ (direct image)

Given a function $f : A \to B$, we get a function $\mathcal{P}(f) : \mathcal{P}(A) \to \mathcal{P}(B)$ that pushes subsets forward.

Check the functor laws: $\mathcal{P}(\mathrm{id}_A)(S) = \{\mathrm{id}_A(x) : x \in S\} = S$, so $\mathcal{P}(\mathrm{id}_A) = \mathrm{id}_{\mathcal{P}(A)}$. And $\mathcal{P}(g \circ f)(S) = \{(g \circ f)(x) : x \in S\} = \{g(f(x)) : x \in S\} = \mathcal{P}(g)(\mathcal{P}(f)(S))$.

Example 2.4 (Forgetful functors)**.** Some functors "forget" structure:

- $U : \mathbf{Grp} \to \mathbf{Set}$ takes a group to its underlying set, forgetting the operation
- $U : \mathbf{Top} \to \mathbf{Set}$ takes a topological space to its underlying set, forgetting which sets are open

These are called *forgetful functors.* They're useful because they let us compare structured objects via their underlying sets.

2.3 The Category Rel

Here's a category that generalizes **Set** in an interesting direction.

Definition 2.3 (The category **Rel**)**.** The category **Rel** has:

- Objects: sets
- Morphisms $R : A \to B$: relations $R \subseteq A \times B$
- Composition: $S \circ R = \{(a, c) : \exists b.\, (a, b) \in R \wedge (b, c) \in S\}$
- Identity: $\Delta_A = \{(a, a) : a \in A\}$ (the diagonal relation)

In **Set**, a morphism $A \to B$ relates each element of A to *exactly one* element of B. In **Rel**, a morphism can relate each element of A to *zero, one, or many* elements of B. Functions are the "deterministic" special case.

There's a functor **Set** → **Rel** that sends each function $f : A \to B$ to its graph $\{(a, f(a)) : a \in A\}$. So **Set** embeds inside **Rel**.

Example 2.5. Let $R = \{(1, a), (1, b), (2, c)\} \subseteq \{1, 2\} \times \{a, b, c\}$ and $S = \{(a, x), (b, x), (c, y)\} \subseteq \{a, b, c\} \times \{x, y\}$.

To compute $S \circ R$: for each pair (i, z), check if there's an intermediate element j with $(i, j) \in R$ and $(j, z) \in S$.

- $(1, x)$: yes, via $j = a$ or $j = b$
- $(1, y)$: no intermediate works
- $(2, x)$: no, because $(2, a)$ and $(2, b)$ aren't in R
- $(2, y)$: yes, via $j = c$

So $S \circ R = \{(1, x), (2, y)\}$.

2.4 Galois Connections (Adjunctions for Preorders)

Galois connections are one of those ideas that, once you see them, you start finding them everywhere.

Definition 2.4 (Galois connection)**.** Let $(P, \leq)$ and $(Q, \leq)$ be preorders. A **Galois connection** is a pair of monotone functions $F : P \to Q$ and $G : Q \to P$ such that:

$$F(p) \leq q \iff p \leq G(q)$$

We write $F \dashv G$ ("F is left adjoint to G").

The condition says: asking "is $F(p)$ below q?" is the same as asking "is p below $G(q)$?" The functions F and G are "mates"—they translate questions between the two preorders.

Example 2.6 (Floor and ceiling)**.** Consider $(\mathbb{R}, \leq)$ and $(\mathbb{Z}, \leq)$, with the inclusion $\iota : \mathbb{Z} \to \mathbb{R}$ viewing integers as real numbers.

- $\iota \dashv \lfloor \cdot \rfloor$: For integer n and real x: $n \leq x \iff n \leq \lfloor x \rfloor$. (If n is at most x, then n is at most the floor of x, and vice versa.)
- $\lceil \cdot \rceil \dashv \iota$: For real x and integer n: $x \leq n \iff \lceil x \rceil \leq n$. (If x is at most n, then the ceiling of x is at most n.)

Floor and ceiling are "best approximations" from different directions—that's what the adjunction captures.

Key Result

Galois connections are the "preorder version" of adjunctions, which are arguably the most important concept in category theory. Left adjoints preserve colimits (like unions); right adjoints preserve limits (like intersections). This pattern appears everywhere: in abstract interpretation (approximating program behavior), type inference (finding most general types), and database queries (optimizing joins).

2.5 Exercises

Exercise 2.1. Show that the divisibility preorder on $\{1, 2, 4, 8\}$ forms a total order when viewed as a category.

Exercise 2.2. Let $P = (\{1, 2, 3, 4\}, \leq)$ and $Q = (\{a, b, c\}, \leq)$ where $a \leq b \leq c$.

(a) How many functors (monotone functions) are there from P to Q?

(b) Give an example of a non-monotone function $\{1, 2, 3, 4\} \to \{a, b, c\}$.

Exercise 2.3. Verify that relational composition in **Rel** is associative.

Exercise 2.4 (Application: Type systems)**.** In a programming language with subtyping, we have a preorder on types. The function type constructor is:

- Contravariant in the argument: if $A' \leq A$, then $(A \to B) \leq (A' \to B)$
- Covariant in the result: if $B \leq B'$, then $(A \to B) \leq (A \to B')$

Explain why the argument is contravariant using the substitution principle.

3 Weeks 4–5: Polynomial Functors and Counting

Why Connect Types and Counting?

Here's something that might seem like a coincidence but isn't.

When you count the elements of a product type $A \times B$, you multiply: $|A \times B| = |A| \times |B|$. When you count the elements of a sum type $A + B$, you add: $|A + B| = |A| + |B|$. And when you count functions $A \to B$, you exponentiate: $|A \to B| = |B|^{|A|}$.

The *notation* for types follows the same rules as the *arithmetic* of counting. This isn't a coincidence—it reflects a deep connection between algebra (polynomials, generating functions) and type theory (data structures, recursive types).

3.1 Types Have Sizes

Let's make this precise:

Type	**Notation**	**Inhabitants**
Void (empty)	0	0
Unit	1	1
Bool	2	2
Sum $A + B$	$A + B$	$\|A\| + \|B\|$
Product $A \times B$	$A \times B$	$\|A\| \times \|B\|$
Function $A \to B$	B^A	$\|B\|^{\|A\|}$

The notation B^A for function types makes sense: there are $|B|^{|A|}$ ways to assign an output in B to each input in A. The algebraic notation reflects the combinatorics.

3.2 Polynomial Functors

Now we can describe data types using polynomial expressions.

Definition 3.1 (Polynomial functor)**.** A **polynomial functor** $F : \mathbf{Set} \to \mathbf{Set}$ has the form:

$$F(X) = A_0 + A_1 \times X + A_2 \times X^2 + A_3 \times X^3 + \cdots$$

where the A_i are fixed sets ("coefficient sets") and $X^n = X \times X \times \cdots \times X$ (n times).

Think of X as a "slot" for data. The polynomial describes what shapes of containers you can build, with A_n specifying the "labels" for containers holding exactly n elements.

The **generating function** of F is $\sum_i |A_i| x^i$—the ordinary polynomial you get by replacing sets with their sizes.

Example 3.1 (Maybe/Option)**.** The simplest interesting polynomial functor: $\text{Maybe}(X) = 1 + X$.

This says: a $\text{Maybe}(X)$ is either an element of 1 (one choice—call it Nothing) or an element of X (call it Just x).

- $\text{Maybe}(\emptyset) = 1 + \emptyset = 1 = \{\text{Nothing}\}$
- $\text{Maybe}(\{a\}) = 1 + \{a\} = \{\text{Nothing}, \text{Just } a\}$
- $\text{Maybe}(\{a, b\}) = 1 + \{a, b\} = \{\text{Nothing}, \text{Just } a, \text{Just } b\}$

The generating function is $1 + x$, confirming: $|\text{Maybe}(X)| = 1 + |X|$.

Example 3.2 (Lists up to length 2)**.** $\text{List}_2(X) = 1 + X + X^2$ describes lists of length at most 2.

- The 1 term: the empty list $[]$
- The X term: one-element lists $[x]$
- The X^2 term: two-element lists $[x_1, x_2]$

For $X = \{a, b\}$: $|\text{List}_2(\{a, b\})| = 1 + 2 + 4 = 7$.
Explicitly: $[]$, $[a]$, $[b]$, $[a, a]$, $[a, b]$, $[b, a]$, $[b, b]$.

Example 3.3 (Full lists)**.** What about lists of arbitrary length? We need infinitely many terms:

$$\text{List}(X) = 1 + X + X^2 + X^3 + \cdots$$

As a formal power series, this is $\frac{1}{1-X}$. (Don't worry about convergence—we're doing algebra, not analysis.)

Here's the beautiful part: the recursive definition $\text{List}(X) = 1 + X \times \text{List}(X)$ captures the same idea. A list is either empty (1) or a head element (X) followed by a tail ($\text{List}(X)$). Solving this equation gives back the infinite sum.

3.3 Functors on Morphisms

We said polynomial functors are functors. That means they act not just on objects (sets) but also on morphisms (functions). Given $f : A \to B$, what is $F(f) : F(A) \to F(B)$?

Example 3.4 (Maybe as a functor)**.** Given $f : A \to B$, define $\text{Maybe}(f) : \text{Maybe}(A) \to \text{Maybe}(B)$ by:

$$\text{Maybe}(f)(\text{Nothing}) = \text{Nothing} \qquad \text{Maybe}(f)(\text{Just } a) = \text{Just } f(a)$$

In words: Nothing stays Nothing, and Just a becomes Just $f(a)$.

Programming Connection

If you know Haskell (or any language with functors), you'll recognize this as `fmap`! The functor laws $F(\text{id}) = \text{id}$ and $F(g \circ f) = F(g) \circ F(f)$ are exactly the laws that `fmap` must satisfy:

```
fmap id = id
fmap (g . f) = fmap g . fmap f
```

This isn't a coincidence. Haskell's `Functor` typeclass is literally the category-theoretic concept.

3.4 Map Objects and Currying

Here's a question: can we treat "the set of all functions from A to B" as a first-class object? Yes, and this leads to one of the most useful ideas in functional programming.

Definition 3.2 (Exponential / map object)**.** In **Set**, the **map object** (or **exponential**) B^A is the set of all functions $A \to B$. It comes with an evaluation map:

$$\text{ev} : B^A \times A \to B, \quad \text{ev}(f, a) = f(a)$$

The universal property says: any function $g : X \times A \to B$ (taking a pair) corresponds to a unique function $\tilde{g} : X \to B^A$ (taking X and returning a function). The correspondence is:

$$\tilde{g}(x) = \lambda a.\, g(x, a)$$

This is **currying**: converting a two-argument function into a function that takes one argument and returns a function of the other. The universal property says currying is a bijection—every way of consuming a pair corresponds to exactly one way of returning a function.

3.5 Combinatorial Species (Advanced)

Species are a categorified version of generating functions. Instead of tracking just the *count* of structures, they track the structures themselves, along with how relabeling affects them.

Definition 3.3 (Species)**.** A **species** is a functor $F : \mathcal{B} \to \mathbf{Set}$ where $\mathcal{B}$ is the category of finite sets and bijections.

- $F(A)$ = the set of "F-structures on the set A"
- $F(\sigma)$ = "relabeling by the bijection σ"

The fact that F is a functor means relabeling behaves sensibly: relabeling by the identity does nothing, and relabeling twice is the same as relabeling by the composition.

Example 3.5 (Species of linear orders)**.** $L(A) = \{\text{linear orderings of } A\}$. For $A = \{1, 2, 3\}$, there are $3! = 6$ linear orders. A bijection $\sigma : A \to B$ turns an ordering of A into an ordering of B by relabeling.

Example 3.6 (Species of graphs)**.** $G(A) = \{\text{simple graphs with vertex set } A\}$. For $|A| = 3$, there are $2^{\binom{3}{2}} = 2^3 = 8$ graphs (each of the 3 potential edges is present or not).

Species turn combinatorics into category theory. Operations like "product of species" and "composition of species" correspond to combining structures, and these operations satisfy algebraic identities that generate combinatorial identities. It's a beautiful theory, though beyond what we can cover here.

3.6 Exercises

Exercise 3.1. What polynomial functor represents "a pair where the first component is from a 3-element set"? What is its generating function?

Exercise 3.2. Express the type `Either (a, b) c` as a polynomial in a, b, c. How many inhabitants does `Either (Bool, Bool) ()` have?

Exercise 3.3. Verify the functor laws for Maybe: $\text{Maybe}(\text{id}) = \text{id}$ and $\text{Maybe}(g \circ f) = \text{Maybe}(g) \circ \text{Maybe}(f)$.

Exercise 3.4. The type of binary trees with data at leaves is $\text{Tree}(A) = A + \text{Tree}(A)^2$. Expand the first few terms of this as a power series in A. What is the coefficient of A^3? (It counts binary tree shapes with 3 leaves.)

Exercise 3.5. Let $A = \{1, 2\}$ and $B = \{a, b, c\}$. List the elements of B^A and describe the evaluation map $\text{ev} : B^A \times A \to B$.

Exercise 3.6. Show that the currying correspondence $g : X \times A \to B \leftrightarrow \tilde{g} : X \to B^A$ is bijective. (Hint: give explicit formulas in both directions.)

4 Weeks 6–7: Graphs and Free Categories

Why Connect Graphs and Categories?

Graphs are everywhere in computer science: data structures, networks, state machines, dependency relations. Categories are about composition of morphisms. What's the connection?

It turns out that every graph generates a category in a natural way: the **free category** on the graph, whose morphisms are paths. And conversely, many categories can be understood as "graphs with extra structure" (the composition rules). This section explores that interplay.

4.1 Endomaps and Dynamical Systems

Let's start with the simplest kind of graph: one with a single node and edges looping back to it.

Definition 4.1 (Endomap)**.** An **endomap** on a set X is a function $a : X \to X$—a function from a set to itself.

Definition 4.2 (Category of endomaps)**.** The category **End** has:

- Objects: pairs (X, a) where X is a set and $a : X \to X$ is an endomap
- Morphisms $f : (X, a) \to (Y, b)$: functions $f : X \to Y$ such that $f \circ a = b \circ f$

That compatibility condition $f \circ a = b \circ f$ says: applying a then f equals applying f then b. In other words, f "respects the dynamics."

Think of (X, a) as a discrete-time dynamical system. The set X is the state space, and a is the transition function: $a(x)$ is the state you reach from x after one time step. Iterating gives $a^n(x)$, the state after n steps. A morphism f between dynamical systems translates states while preserving how they evolve.

Note. Here's an elegant perspective: a dynamical system (X, a) is the same as a functor from the monoid $(\mathbb{N}, +, 0)$ (viewed as a one-object category) to **Set**. The single object maps to X, and the number n maps to a^n.

4.2 Parts, Predicates, and the Power Set

The power set $\mathcal{P}(A)$ collects all "parts" of A—all the ways to select a subset. There's a nice categorical perspective here: subsets correspond to predicates.

Each subset $S \subseteq A$ corresponds to its characteristic function $\chi_S : A \to \{0, 1\}$, where $\chi_S(a) = 1$ iff $a \in S$. So $\mathcal{P}(A) \cong (A \to 2)$. Subsets *are* predicates.

Now here's something important about how subsets transform. Given $f : A \to B$:

- The direct image $f_* : \mathcal{P}(A) \to \mathcal{P}(B)$ pushes subsets forward: $f_*(S) = \{f(a) : a \in S\}$
- The preimage $f^{-1} : \mathcal{P}(B) \to \mathcal{P}(A)$ pulls subsets back: $f^{-1}(T) = \{a : f(a) \in T\}$

The preimage is contravariant (it goes "backwards") and behaves very nicely:

$$f^{-1}(S \cup T) = f^{-1}(S) \cup f^{-1}(T), \quad f^{-1}(S \cap T) = f^{-1}(S) \cap f^{-1}(T)$$

It preserves both unions and intersections! This is the categorical origin of "pulling back predicates" in logic and type theory.

4.3 Graphs Generate Categories

Here's the main event of this section: turning graphs into categories.

Definition 4.3 (Quiver/Directed graph)**.** A **quiver** (category theorists' name for a directed graph) Q consists of:

- A set Q_0 of **vertices**
- A set Q_1 of **edges**
- Source and target functions $s, t : Q_1 \to Q_0$

Definition 4.4 (Free category on a graph)**.** The **free category** $\mathrm{Path}(Q)$ on a quiver Q has:

- Objects: vertices of Q
- Morphisms from u to v: directed paths from u to v (including the empty path when $u = v$)
- Composition: path concatenation
- Identity: the empty path at each vertex

The word "free" means: this is the category you get by adding *only* what the category axioms require (identities and composites) and nothing more. No extra equations, no collapsing of paths.

Example 4.1. For the path graph $1 \xrightarrow{a} 2 \xrightarrow{b} 3$:

The morphisms of $\mathrm{Path}(Q)$ are: $\mathrm{id}_1, \mathrm{id}_2, \mathrm{id}_3$ (the empty paths), $a : 1 \to 2$, $b : 2 \to 3$, and $b \circ a : 1 \to 3$. That's six morphisms total, and the category is finite.

Example 4.2. For a cycle $1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{c} 1$:

Now $\mathrm{Path}(Q)$ is infinite! The morphisms from 1 to 1 are: $\mathrm{id}_1, cba, (cba)^2, (cba)^3, \ldots$—you can go around the cycle any number of times. Cycles in the graph create infinitely many paths.

4.4 Diagrams as Functors

Here's a beautiful unification: a *diagram* in a category is just a functor from a "shape" category.

Definition 4.5. A **diagram of shape J in category $\mathcal{C}$** is a functor $D : J \to \mathcal{C}$.

The shape J specifies what kind of diagram we're drawing:

- $J = \bullet \to \bullet$ (two objects, one morphism): a diagram of this shape picks out a single morphism in $\mathcal{C}$
- $J = \bullet \rightrightarrows \bullet$ (two objects, two parallel morphisms): picks out a parallel pair
- $J =$ a commutative square: picks out a commuting square in $\mathcal{C}$

The functor D assigns objects and morphisms of $\mathcal{C}$ to the "slots" in the shape, preserving composition. Commutativity of the diagram is automatic: if two paths in J are equal (compose to the same morphism), their images under D are equal too.

Application (Database schemas). Here's a practical application: relational databases are functors!

A database schema is a quiver where vertices are tables and edges are foreign key relationships. A database *instance* (actual data) is a functor from Path(Schema) to **Set**:

- Each table T maps to a set $F(T)$ of rows
- Each foreign key $e : T \to T'$ maps to a function $F(e) : F(T) \to F(T')$ (looking up the referenced row)

Composition of foreign keys must be respected—if you can get from table A to table C via B, either path gives the same function. This is referential integrity!

4.5 Adjacency Matrices Count Paths

Remember from linear algebra: the (i, j) entry of A^n counts paths of length n from vertex i to vertex j. Now we can see why.

Theorem 4.1. *For the free category Path(Q), if A is the adjacency matrix of Q, then:*

$$(A^n)_{ij} = |\mathrm{Hom}_{Path(Q)}(i, j) \cap \{\textit{paths of length } n\}|$$

Matrix multiplication is computing composition in the free category, with the ring $(\mathbb{N}, +, \times)$ counting how many ways compositions can happen. The "free category" perspective explains why this works: paths are morphisms, and matrix multiplication is tracking compositions.

4.6 Connected Components Functor

Let's see another functor arising from graphs.

Define $\pi_0 : \mathbf{Graph} \to \mathbf{Set}$ by sending a graph to its set of connected components. Given a graph homomorphism $f : G \to H$, we get a function $\pi_0(f) : \pi_0(G) \to \pi_0(H)$ by mapping each component of G to the component of H containing its image.

This is a functor: it respects identities and composition. (Check: if f and g are composable graph homomorphisms, the component you land in by doing f then g is the same as doing $g \circ f$ directly.)

Remark. The functor π_0 has a nice property: it preserves coproducts (disjoint unions). The components of $G \sqcup H$ are the components of G together with the components of H. In categorical language, π_0 is *left adjoint* to the discrete-graph functor (which turns a set into a graph with no edges).

4.7 Exercises

Exercise 4.1. For the graph $a \xrightarrow{f} b \xrightarrow{g} c$, $c \xrightarrow{h} b$:

(a) List all morphisms from a to c in Path(Q).

(b) List all morphisms from b to b.

(c) Is Path(Q) finite or infinite?

Exercise 4.2. Draw a database schema for: Users, Posts, Comments, where Posts have authors (Users) and Comments reference both a Post and a User. Write down what a functor from this schema to **Set** looks like.

Exercise 4.3. Let (X, a) and (Y, b) be endomaps, and let $f : X \to Y$ satisfy $f \circ a = b \circ f$. Prove by induction on n that $f \circ a^n = b^n \circ f$ for all $n \geq 0$.

Exercise 4.4. Show that f^{-1} preserves unions and intersections: for any $f : A \to B$ and $S, T \subseteq B$, prove $f^{-1}(S \cup T) = f^{-1}(S) \cup f^{-1}(T)$ and $f^{-1}(S \cap T) = f^{-1}(S) \cap f^{-1}(T)$.

Exercise 4.5. Compute $\pi_0(G)$ for a graph G with vertices $\{1, 2, 3, 4\}$ and edges $\{1, 2\}, \{2, 3\}$. Then describe π_0 on a graph homomorphism that collapses $\{1, 2, 3\}$ to a single vertex in a 2-vertex graph.

5 Week 8: Initial Algebras and Catamorphisms

Why Initial Algebras?

In the main course this week, you're seeing trees as algebraic datatypes: a tree is either a leaf or a node with subtrees. You're learning that this recursive structure gives you structural recursion (for defining functions) and structural induction (for proving properties).

This section explains *why* that works. The answer involves one of the most beautiful ideas in category theory: initial algebras. An algebraic datatype is an initial algebra for a functor, and the "fold" pattern that defines all recursive functions is the unique morphism out of an initial object.

Once you see this, you'll understand why structural recursion always terminates, why there's exactly one function satisfying any given recursive equations, and how the same pattern applies to natural numbers, lists, trees, and any other algebraic datatype.

5.1 F-Algebras

Let's start with a general framework for "things with constructors."

Definition 5.1 (F-algebra)**.** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An F**-algebra** is a pair (A, α) where:

- A is an object (the **carrier**—the set of "values")
- $\alpha : F(A) \to A$ is a morphism (the **structure map**—the "constructors")

The functor F describes the "shape" of a single layer of construction. The structure map α says how to build an A from an F-shaped collection of As.

Definition 5.2 (Algebra homomorphism)**.** A **homomorphism** $h : (A, \alpha) \to (B, \beta)$ of F-algebras is a morphism $h : A \to B$ such that this diagram commutes:

$$\begin{tikzcd}
F(A) \arrow[r, "\alpha"] \arrow[d, "F(h)"'] & A \arrow[d, "h"] \\
F(B) \arrow[r, "\beta"'] & B
\end{tikzcd}$$

In equations: $h \circ \alpha = \beta \circ F(h)$.

This says: translating via h then constructing in B equals constructing in A then translating. The homomorphism "respects the constructors."

Example 5.1 (Algebras for $F(X) = 1 + X$)**.** This functor describes "either nothing, or one thing." An F-algebra (A, α) consists of:

- A set A

- A function $\alpha : 1 + A \to A$

But a function from $1 + A$ to A is the same as specifying two things:

- Where to send the element of 1: call this $z \in A$ (the "zero")
- Where to send each element of A: call this $s : A \to A$ (the "successor")

So an F-algebra is a triple (A, z, s): a set with a distinguished element and an endofunction. Sound familiar? This is exactly the structure of natural numbers!

5.2 Initial Algebras

Now the key definition. Among all the F-algebras, is there a "most fundamental" one?

Definition 5.3 (Initial algebra)**.** An **initial F-algebra** $(\mu F, \mathrm{in})$ is an F-algebra such that for every F-algebra (A, α), there exists a *unique* homomorphism $[\![\alpha]\!] : \mu F \to A$.

This unique morphism is called a **catamorphism** (from Greek "cata" = down, "morph" = form). In programming, it's called **fold**.

Think about what initiality means: the initial algebra can map to any other algebra, and there's only one way to do it. The initial algebra is "the freest"—it has no extra equations beyond what the functor requires.

Key Result

[Lambek's Lemma] If $(\mu F, \mathrm{in})$ is an initial F-algebra, then the structure map $\mathrm{in} : F(\mu F) \to \mu F$ is an **isomorphism**.
In other words: $\mu F \cong F(\mu F)$.

This is remarkable! It says the initial algebra is a "fixed point" of the functor. For $F(X) = 1+X$, we get $\mathbb{N} \cong 1+\mathbb{N}$—the natural numbers are "either zero or a successor of a natural number." That's exactly what the recursive definition says!

Example 5.2 (Natural numbers)**.** For $F(X) = 1 + X$, the initial algebra is $(\mathbb{N}, [\mathrm{zero}, \mathrm{succ}])$ where:

- $\mathrm{zero} : 1 \to \mathbb{N}$ picks out 0
- $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$ is the successor function

Given any algebra (A, z, s)—that is, any set with an element z and a function s—there's a unique homomorphism $[\![z, s]\!] : \mathbb{N} \to A$. It's defined by:

$$[\![z, s]\!](0) = z \qquad [\![z, s]\!](n + 1) = s([\![z, s]\!](n))$$

This is primitive recursion! The initiality of $\mathbb{N}$ *is* the principle of recursive definition.

Example 5.3 (Lists)**.** For $F_A(X) = 1 + A \times X$, an algebra specifies what to do with "nil" (the 1 case) and "cons" (the $A \times X$ case).

The initial algebra is $(\mathrm{List}(A), [\mathrm{nil}, \mathrm{cons}])$—lists with the usual constructors.

The unique catamorphism to any algebra (B, z, f) is exactly `foldr`:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Every function defined by "recursion on the structure of a list" is a catamorphism.

Example 5.4 (Binary trees)**.** For $F_A(X) = A + X \times X$ ("either a leaf with data, or two subtrees"), the initial algebra is binary trees with data at leaves.

Catamorphisms are the functions from the main course material:

- Sum leaves: $\text{leaf}(a) = a$, $\text{node}(x, y) = x + y$
- Height: $\text{leaf}(a) = 0$, $\text{node}(x, y) = 1 + \max(x, y)$
- Flatten: $\text{leaf}(a) = [a]$, $\text{node}(x, y) = x +\!\!+\, y$

You specify the "algebra"—what to do for leaves and how to combine subtree results—and you get a unique function for free.

5.3 Why Structural Recursion Terminates

Here's the deep reason structural recursion works. The initiality property guarantees three things:

1. **Existence:** There *is* a function satisfying the recursive equations.
2. **Uniqueness:** There's *only one* such function—the definition is unambiguous.
3. **Correctness:** The function actually satisfies the equations (it's a homomorphism).

Now think about termination. If a recursive definition didn't terminate on some input, then no function would exist for that input—violating existence. But the initial algebra exists, so the catamorphism must be total.

This is why "structural recursion always terminates": you're not just following a syntactic pattern, you're constructing the unique homomorphism out of an initial object. The mathematics guarantees it works.

And here's the connection to structural induction: proving a property by structural induction is *defining* a function into the type of proofs. If you can give the base case (a proof for leaves) and the inductive case (a way to combine proofs for subtrees), you've defined a catamorphism—and initiality guarantees it gives a proof for every input.

5.4 Catamorphism Fusion

Initial algebras give us a powerful optimization principle.

Theorem 5.1 (Fusion)**.** *If $h : (A, \alpha) \to (B, \beta)$ is an algebra homomorphism, then:*

$$h \circ [\![\alpha]\!] = [\![\beta]\!]$$

In words: if h respects the algebra structures, then folding to A and then applying h equals folding directly to B. You can "fuse" the post-processing into the fold.

Programming Connection

This is the theoretical basis for **fusion** optimizations in functional compilers:

```
map f . map g = map (f . g)   -- fuse two traversals into one
sum . map f   = foldr (\x a -> f x + a) 0
```

Instead of building an intermediate list (map) and then consuming it (sum), you fold once. The fusion theorem tells you when this transformation is valid.

5.5 Exercises

Exercise 5.1. For $F(X) = 1 + X$:

(a) Define an F-algebra on $\{\text{even}, \text{odd}\}$ with $z = \text{even}$ and $s = \text{flip}$.

(b) Compute the unique catamorphism from $\mathbb{N}$ to this algebra.

(c) What function does this catamorphism compute?

Exercise 5.2. For $F_{\mathbb{N}}(X) = 1 + \mathbb{N} \times X$:

(a) Define an algebra on $\mathbb{N}$ that computes the sum of a list.

(b) Define an algebra on $\mathbb{N}$ that computes the length of a list.

(c) Define an algebra on $\mathbb{N}$ that computes the product of a list.

Exercise 5.3. Verify Lambek's Lemma for natural numbers: show explicitly that $[\text{zero}, \text{succ}] : 1 + \mathbb{N} \to \mathbb{N}$ is an isomorphism by constructing its inverse.

Exercise 5.4. Express each tree traversal (preorder, inorder, postorder) as a catamorphism by specifying the algebra.

6 Week 9: Coalgebras and Automata

Why Coalgebras?

Algebras are about *construction*: how do you build data? Coalgebras are about *observation*: how do you examine data? How does it behave over time?

This duality is profound. Lists are an initial algebra: you build them from nil and cons, and fold consumes them. Streams (infinite lists) are a final coalgebra: you observe them by taking head and tail, and unfold produces them.

In this section, we'll see how automata—the topic of the main course—are coalgebras. This perspective illuminates what it means for two automata to "accept the same language" and gives a beautiful characterization of DFA minimization.

6.1 F-Coalgebras

Coalgebras are the **dual** of algebras. Where algebras have structure maps *into* the carrier, coalgebras have structure maps *out of* the carrier.

Definition 6.1 (F-coalgebra)**.** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An F**-coalgebra** is a pair (A, α) where:

- A is an object (the **state space**)
- $\alpha : A \to F(A)$ is a morphism (the **observation/transition map**)

The arrow goes the other way! Instead of "how to construct an A from F-many As," we have "how to observe an A and get F-much information."

Definition 6.2 (Coalgebra homomorphism)**.** A homomorphism $h : (A, \alpha) \to (B, \beta)$ is a morphism $h : A \to B$ such that:

$$\begin{tikzcd} A \arrow[r, "\alpha"] \arrow[d, "h"'] & F(A) \arrow[d, "F(h)"] \\ B \arrow[r, "\beta"'] & F(B) \end{tikzcd}$$

commutes: $\beta \circ h = F(h) \circ \alpha$.

This says: observing in A then translating equals translating then observing in B. The homomorphism preserves the observable behavior.

6.2 Deterministic Finite Automata as Coalgebras

Here's the payoff: DFAs are coalgebras!

Theorem 6.1. *A DFA over alphabet Σ is exactly a coalgebra for the functor:*

$$F(X) = 2 \times X^{\Sigma}$$

where $2 = \{accept, reject\}$ and X^{Σ} denotes functions from Σ to X.

Unpack this: from any state, you can observe two things:

1. Is this state accepting? (That's the 2 part.)
2. For each symbol $a \in \Sigma$, what state do you transition to? (That's the X^{Σ} part.)

A coalgebra $(Q, \langle o, \delta \rangle)$ consists of:

- Q: the set of states
- $o : Q \to 2$: the output function (accepting or not?)
- $\delta : Q \to Q^{\Sigma}$: the transition function (equivalently, $\delta : Q \times \Sigma \to Q$)

That's exactly a DFA! The coalgebra perspective emphasizes what you can *observe* about a state, rather than how states were constructed.

6.3 Final Coalgebras

Just as algebras have initial objects, coalgebras have final objects—and they're equally important.

Definition 6.3 (Final coalgebra)**.** A **final F-coalgebra** $(\nu F, \mathrm{out})$ is an F-coalgebra such that for every F-coalgebra (A, α), there exists a unique homomorphism $(\!|\alpha|\!) : A \to \nu F$.

This unique morphism is called an **anamorphism** ("ana" = up) or **unfold**.

The final coalgebra is the "most complex" coalgebra: every other coalgebra maps uniquely into it. Where the initial algebra is "built from nothing," the final coalgebra "observes everything."

Theorem 6.2. *For the DFA functor $F(X) = 2 \times X^{\Sigma}$, the final coalgebra is:*

$$(\mathcal{P}(\Sigma^*), \langle \varepsilon?, derivatives \rangle)$$

where:

- $\mathcal{P}(\Sigma^*)$ *is the set of* all *languages over* Σ
- $\varepsilon?(L) =$ *accept iff* $\varepsilon \in L$ *(does* L *contain the empty string?)*
- $\partial_a(L) = \{w : aw \in L\}$ *is the Brzozowski derivative (what's left after reading* a*?)*

The elements of the final coalgebra are *languages*. Each language "behaves like" a DFA state: it either accepts or rejects the empty string, and reading a symbol gives you a new language (the derivative).

And here's the beautiful part: the unique coalgebra morphism from any DFA to this final coalgebra sends each state q to **the language accepted from state** q! The finality of languages explains what DFA equivalence means: two states are equivalent iff they map to the same language.

6.4 Bisimulation

The coalgebraic perspective gives us a clean definition of equivalence.

Definition 6.4 (Bisimulation)**.** Two states (possibly in different automata) are **bisimilar** if they map to the same element of the final coalgebra.

For DFAs, bisimilarity is language equivalence: two states are bisimilar iff they accept the same language. The minimal DFA is obtained by quotienting by bisimilarity—identifying states that can't be distinguished by any sequence of observations.

This is much more general than DFAs. Bisimulation is the right notion of equivalence for any coalgebraic system: two states are "the same" if no sequence of observations can tell them apart.

6.5 Streams as a Final Coalgebra

Let's see another final coalgebra, one that produces infinite data rather than classifying finite inputs.

Theorem 6.3. *For* $F(X) = A \times X$*, the final coalgebra is:*

$$(A^\omega, \langle \mathit{head}, \mathit{tail} \rangle)$$

where A^ω *is the set of infinite streams over* A*.*

The structure map sends a stream to its head (an element of A) and its tail (another stream). The isomorphism $A^\omega \cong A \times A^\omega$ says: "an infinite stream is a head followed by another infinite stream."

Compare to lists: $\mathrm{List}(A) \cong 1 + A \times \mathrm{List}(A)$ (a list is either empty *or* a head and tail). Streams have no "empty" case—they go on forever.

Example 6.1 (Generating streams)**.** The Fibonacci sequence as an anamorphism (unfold):

```
fibs = unfold (\(a,b) -> (a, (b, a+b))) (0, 1)
     = [0, 1, 1, 2, 3, 5, 8, 13, ...]
```

The seed is $(0, 1)$. At each step, output the first component (current Fibonacci number) and update the seed to $(b, a + b)$. The unfold produces an infinite stream—no base case needed.

6.6 Algebra vs Coalgebra: Summary

Here's the duality at a glance:

Concept	**Algebra**	**Coalgebra**
Structure map	$F(A) \to A$ (build)	$A \to F(A)$ (observe)
Universal object	Initial (smallest)	Final (largest)
Universal morphism	Catamorphism (fold)	Anamorphism (unfold)
Canonical examples	$\mathbb{N}$, lists, trees	Streams, automata
Data	Finite (inductive)	Potentially infinite (coinductive)
Perspective	Constructors	Observers/destructors
Proof principle	Induction	Coinduction

Algebras and coalgebras are two sides of the same coin. Understanding both gives you a complete picture of recursive and corecursive data.

6.7 Exercises

Exercise 6.1. Write the DFA accepting "strings ending in 01" as an F-coalgebra for $F(X) = 2 \times X^{\{0,1\}}$.

Exercise 6.2. Compute the Brzozowski derivatives $\partial_0(L)$ and $\partial_1(L)$
for $L = \{w : w \text{ has even number of 1s}\}$.

Exercise 6.3. Define the stream of powers of 2 as an anamorphism.

Exercise 6.4. Two DFAs have state sets $Q_1 = \{a, b\}$ and $Q_2 = \{x, y, z\}$. State a accepts $\{0^n 1^n : n \geq 0\}$ (note: not regular, but pretend). State x accepts all strings. Are a and x bisimilar? Why or why not?

7 Week 10: Monoids and Cost

Why Monoids?

A monoid is one of the simplest algebraic structures: a set with an associative operation and an identity. Addition on numbers. Concatenation on strings. Composition of functions.

From the categorical viewpoint, monoids are even simpler: a monoid *is* a category with just one object. All the action is in the morphisms (the monoid elements), and composition is the monoid operation.

This perspective connects monoids to everything we've done. It explains why monoids appear in algorithm analysis (tracking costs), in automata theory (syntactic monoids recognize languages), and in parallel computing (combining results associatively).

7.1 Monoids as One-Object Categories

Definition 7.1 (Monoid)**.** A **monoid** $(M, \cdot, e)$ is:

- A set M
- An associative binary operation $\cdot : M \times M \to M$
- An identity element $e \in M$ with $e \cdot m = m \cdot e = m$

Theorem 7.1. *A monoid is exactly a category with one object.*

- *The single object: call it* $*$
- *Morphisms* $* \to *$*: elements of* M
- *Composition: the monoid operation* $\cdot$
- *Identity morphism: the identity element* e

The category axioms (associativity of composition, identity laws) are exactly the monoid axioms. A group is a monoid where every element has an inverse—equivalently, a one-object category where every morphism is an isomorphism.

Example 7.1 (Common monoids)**.**

Monoid	**Set**	**Operation**	**Identity**
$(\mathbb{N}, +, 0)$	natural numbers	addition	0
$(\mathbb{N}, \times, 1)$	natural numbers	multiplication	1
$(\Sigma^*, \cdot, \varepsilon)$	strings	concatenation	empty string
$(\mathbb{N} \cup \{\infty\}, \max, 0)$	extended naturals	maximum	0
$(\mathbb{N} \cup \{\infty\}, \min, \infty)$	extended naturals	minimum	∞

Each of these is a one-object category. The morphisms are numbers (or strings), and composition is the operation.

7.2 Free Monoids

The "free" construction for monoids is one you already know: strings.

Definition 7.2 (Free monoid)**.** The **free monoid** on a set Σ (an "alphabet") is $(\Sigma^*, \cdot, \varepsilon)$—strings over Σ with concatenation.

Why "free"? Because strings impose no equations beyond associativity and identity. The string ab is different from ba; the string aa is different from a. You get exactly the combinations the monoid axioms allow, and nothing more.

Theorem 7.2 (Universal property)**.** *For any monoid* M *and function* $f : \Sigma \to M$*, there exists a unique monoid homomorphism* $f^* : \Sigma^* \to M$ *extending* f*.*

This says: to define a homomorphism out of Σ^*, you just need to say where each letter goes. The rest is forced by the monoid laws: $f^*(w_1 w_2) = f^*(w_1) \cdot f^*(w_2)$.

7.3 Free/Forgetful Adjunction

This universal property is an adjunction in disguise.

Let $U : \mathbf{Mon} \to \mathbf{Set}$ be the forgetful functor (forget the operation, just keep the underlying set) and $F : \mathbf{Set} \to \mathbf{Mon}$ be the free monoid functor ($F(\Sigma) = \Sigma^*$).

The universal property says:

$$\mathrm{Hom}_{\mathbf{Mon}}(\Sigma^*, M) \cong \mathrm{Hom}_{\mathbf{Set}}(\Sigma, U(M))$$

Monoid homomorphisms from the free monoid correspond bijectively to functions from the generators. This is the free/forgetful adjunction $F \dashv U$.

Adjunctions capture the pattern of "free constructions" across mathematics: free groups, free vector spaces, free categories on graphs. The left adjoint builds the free structure; the right adjoint forgets the structure.

7.4 Regular Languages and Syntactic Monoids

Here's a beautiful connection between monoids and automata theory.

Theorem 7.3. *A language $L \subseteq \Sigma^*$ is regular if and only if there exists:*

1. *A* finite *monoid M*
2. *A monoid homomorphism $h : \Sigma^* \to M$*
3. *A subset $F \subseteq M$ (the "accepting" elements)*

such that $L = h^{-1}(F)$ (a string is in L iff its image is in F).

The monoid M is playing the role of the DFA, but algebraically. Reading a string computes a monoid element; accepting means landing in F.

The smallest such M that works for a given L is called the **syntactic monoid** of L. It captures exactly the distinctions L makes between strings.

Example 7.2. For $L = \{w : |w|_1 \equiv 0 \pmod 3\}$ (strings where the number of 1s is divisible by 3):

The syntactic monoid is $\mathbb{Z}_3 = \{0, 1, 2\}$ with addition mod 3. The homomorphism $h : \{0, 1\}^* \to \mathbb{Z}_3$ sends a string to its count of 1s mod 3. The accepting set is $F = \{0\}$.

7.5 Cost Monoids

Here's a practical application: tracking computational costs.

Definition 7.3 (Cost monoid)**.** A **cost monoid** is a monoid used to measure computational resources:

- $(\mathbb{N}, +, 0)$: counting operations (sequential time—costs add up)
- $(\mathbb{N}, \max, 0)$: parallel time (the slowest branch determines total time)
- $(\mathbb{N} \times \mathbb{N}, +, (0, 0))$: tracking time AND space simultaneously

The choice of monoid determines how costs combine. Sequential composition adds; parallel composition takes the max.

Example 7.3 (Merge sort cost)**.** The recurrence $T(n) = 2T(n/2) + \Theta(n)$ describes merge sort.

Using $(\mathbb{N}, +, 0)$ for sequential time: we add the costs of the two recursive calls and the merge step. This gives $T(n) = \Theta(n \log n)$.

Using $(\mathbb{N}, \max, 0)$ for parallel time: the two recursive calls happen simultaneously, so we take the max (not the sum). This gives $T(n) = T(n/2) + \Theta(\log n) = \Theta(\log^2 n)$—much faster when you can parallelize!

7.6 Enriched Categories (Brief)

Here's a glimpse of a more advanced topic that ties together graphs, costs, and categories.

A **category enriched over a monoidal category** V replaces hom-sets with objects of V. Instead of "the set of morphisms from A to B," you have "a V-object measuring the 'distance' from A to B."

Example 7.4 (Weighted graphs as enriched categories)**.** A weighted graph with non-negative weights is a category enriched over $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$:

- Objects: vertices
- $\mathrm{Hom}(u, v)$: the edge weight (∞ if no edge)
- Composition: $d(u, w) = \min_v(d(u, v) + d(v, w))$
- Identity: $d(v, v) = 0$

The composition law says: the distance from u to w is the minimum over all intermediate vertices v of going through v. This is exactly what shortest-path algorithms compute!

Dijkstra's algorithm, Bellman-Ford, Floyd-Warshall—they're all computing composition in an enriched category.

7.7 Exercises

Exercise 7.1. Verify that $(\mathbb{N}, \max, 0)$ is a monoid.

Exercise 7.2. How many monoid structures are there on the set $\{0, 1\}$? List them all.

Exercise 7.3. Show that the length function $|\cdot| : \Sigma^* \to \mathbb{N}$ is a monoid homomorphism.

Exercise 7.4. Find the syntactic monoid of $L = \{w \in \{a, b\}^* : w \text{ contains } ab\}$.

Exercise 7.5. Let $U : \mathbf{Mon} \to \mathbf{Set}$ be the forgetful functor and $F : \mathbf{Set} \to \mathbf{Mon}$ the free monoid functor. Prove that giving a monoid homomorphism $h : \Sigma^* \to M$ is equivalent to giving a function $\Sigma \to U(M)$.

Exercise 7.6. Model the cost of linear search and binary search using the cost monoid $(\mathbb{N}, +, 0)$.

8 Conclusion: The Categorical Perspective

8.1 What Have We Learned?

If you've made it through this companion, you've seen category theory not as abstract nonsense, but as a *language for patterns*. The same structures appear across mathematics and computer science; category theory gives us vocabulary to recognize and exploit them.

8.2 Recurring Themes

Let's recap the big ideas:

1. Universal Properties. Many constructions are characterized not by *what they are*, but by *what they do*—by their relationship to everything else. Products, coproducts, free constructions, initial and final objects are all defined by universal properties. Once you prove something satisfies a universal property, you get uniqueness (up to isomorphism) for free.

2. Functors Preserve Structure. The maps between structures (functors, homomorphisms) are often more revealing than the structures themselves. When you find a functor, you've found a way to translate problems from one domain to another while preserving the essential structure.

3. Duality. Concepts come in pairs: products/coproducts, monos/epis, algebras/coalgebras, initial/final, fold/unfold. The duality isn't just formal symmetry—it reflects real phenomena. Finite vs. infinite data. Construction vs. observation. Understanding one side illuminates the other.

4. Diagrams as Specifications. Commutative diagrams express invariants. When we say "this diagram commutes," we're stating a law that any implementation must satisfy. Diagrams are the "type signatures" of mathematics.

5. Free Constructions. Given generators, build the simplest structure containing them. Free monoids (strings), free categories (paths), initial algebras (recursive data). The universal property of free constructions explains why recursive definitions work.

8.3 Where to Go From Here

This companion has only scratched the surface. If you want to go deeper:

- **Seven Sketches in Compositionality** by Fong & Spivak: applied category theory for working scientists, free online. Emphasizes databases, circuits, and systems.
- **Category Theory for Programmers** by Milewski: a programmer's introduction, Haskell-focused, free online. Great for connecting category theory to functional programming.
- **Algebra of Programming** by Bird & de Moor: folds, unfolds, and program calculation. The definitive text on using category theory for program derivation.
- **Categories for the Working Mathematician** by Mac Lane: the classic graduate text. Dense but comprehensive.

The categorical perspective isn't just theoretical elegance—it's practical. It tells you why your recursive functions terminate, how to optimize your data transformations, what "equivalence" really means for automata, and how to design composable abstractions.

Category theory is a tool. Now you have a sense of how to use it.