Chapter 3

Databases: Categories, functors, and universal constructions

3.1 What is a database?

Integrating data from disparate sources is a major problem in industry today. A study in 2008 [BH08] showed that data integration accounts for 40% of IT (information technology) budgets, and that the market for data integration software was $2.5 billion in 2007 and increasing at a rate of more than 8% per year. In other words, it is a major problem; but what is it?

A database is a system of interlocking tables. Data becomes information when it is stored *in* a given *formation*. That is, the numbers and letters don't mean anything until they are organized, often into a system of interlocking tables. An organized system of interlocking tables is called a database. Here is a favorite example:

Employee	**FName**	**WorksIn**	**Mngr**
1	Alan	101	2
2	Ruth	101	2
3	Kris	102	3

Department	**DName**	**Secr**
101	Sales	1
102	IT	3

(3.1)

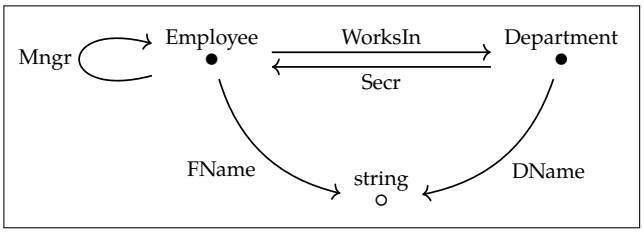
These two tables interlock by use of a special left-hand column, demarcated by a vertical line; it is called the ID column. The ID column of the first table is called 'Employee,' and the ID column of the second table is called 'Department.' The entries in the ID column—e.g. 1, 2, 3 or 101, 102—are like row labels; they indicate a whole row of the table they're in. Thus each row label must be unique (no two rows in a table can have the same label), so that it can unambiguously specify its row.

Each table's ID column, and the set of unique identifiers found therein, is what allows for the interlocking mentioned above. Indeed, other entries in various tables can reference rows in a given table by use of its ID column. For example, each entry in the WorksIn column references a department for each employee; each entry in the Mngr (manager) column references an employee for each employee, and each entry in the Secr (secretary) column references an employee for each department. Managing all this cross-referencing is the purpose of databases.

Looking back at Eq. (3.1), one might notice that every non-ID column, found in either table, is a reference to a label of some sort. Some of these, namely WorksIn, Mngr, and Secr, are *internal references*, often called *foreign keys*; they refer to rows (keys) in the ID column of some (foreign) table. Others, namely FName and DName, are *external references*; they refer to strings or integers, which can also be thought of as labels, whose meaning is known more broadly. Internal reference labels can be changed as long as the change is consistent—1 could be replaced by 1001 everywhere without changing the meaning—whereas external reference labels certainly cannot! Changing Ruth to Bruce everywhere would change how people understood the data.

The reference structure for a given database—i.e. how tables interlock via foreign keys—tells us something about what information was intended to be stored in it. One may visualize the reference structure for Eq. (3.1) graphically as follows:

easySchema :=



(3.2)

This is a kind of "Hasse diagram for a database," much like the Hasse diagrams for preorders in Remark 1.39. How should you read it?

The two tables from Eq. (3.1) are represented in the graph (3.2) by the two black nodes, which are given the same name as the ID columns: Employee and Department. There is another node—drawn white rather than black—which represents the external reference type of strings, like "Alan," "Alpha," and "Sales". The arrows in the diagram represent non-ID columns of the tables; they point in the direction of reference: WorksIn refers an employee to a department.

Exercise 3.3. Count the number of non-ID columns in Eq. (3.1). Count the number of arrows (foreign keys) in Eq. (3.2). They should be the same number in this case; is this a coincidence? ◊

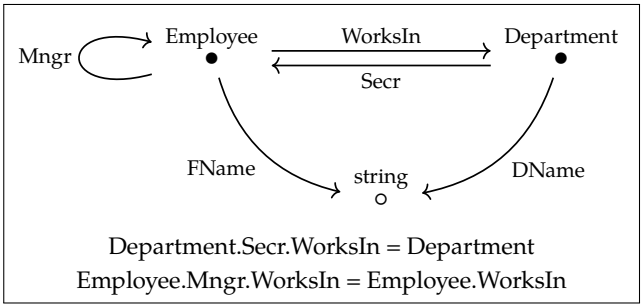
A Hasse-style diagram like the one in Eq. (3.2) can be called a *database schema*; it represents how the information is being organized, the formation in which the data is kept. One may add rules, sometimes called 'business rules' to the schema, in order to ensure the integrity of the data. If these rules are violated, one knows that data being

entered does not conform to the way the database designers intended. For example, the designers may enforce rules saying

- every department's secretary must work in that department;
- every employee's manager must work in the same department as the employee.

Doing so changes the schema, say from 'easySchema' (3.2) to 'mySchema' below.

mySchema :=



(3.4)

In other words, the difference is that easySchema plus constraints equals mySchema.

We will soon see that database schemas are categories $\mathcal{C}$, that the data itself is given by a 'set-valued' functor $\mathcal{C} \to \mathbf{Set}$, and that databases can be mapped to each other via functors $\mathcal{C} \to \mathcal{D}$. In other words, there is a relatively large overlap between database theory and category theory. This has been worked out in a number of papers; see Section 3.6. It has also been implemented in working software, called FQL, which stands for *functorial query language*. Here is example FQL code for the schema shown above:

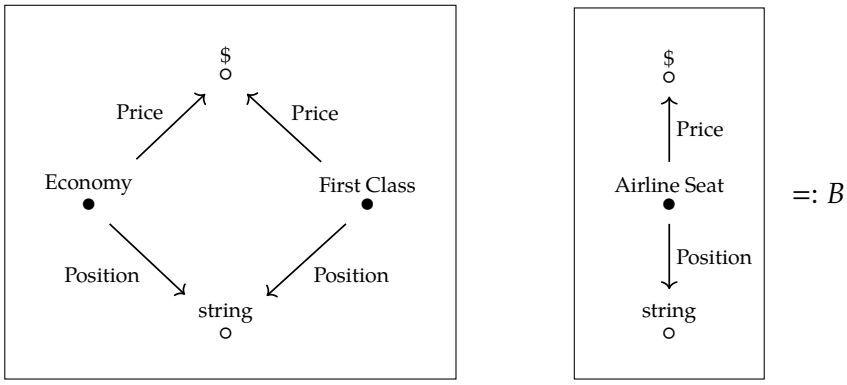
```
schema mySchema = {
   nodes
      Employee, Department;
   attributes
      DName : Department -> string,
      FName : Employee   -> string;
   arrows
      Mngr    : Employee   -> Employee,
      WorksIn : Employee   -> Department,
      Secr    : Department -> Employee;
   equations
      Department.Secr.WorksIn = Department,
      Employee.Mngr.WorksIn   = Employee.WorksIn;
}
```

Communication between databases. We have said that databases are designed to store information about something. But different people or organizations might view the same sort of thing in different ways. For example, one bank stores its financial records according to European standards and another does so according to Japanese standards. If these two banks merge into one, they will need to be able to share their data despite differences in the shape of their database schemas.

Such problems are huge and intricate in general, because databases often comprise hundreds or thousands of interlocking tables. Moreover, these problems occur more frequently than just when companies want to merge. It is quite common that a given company moves data between databases on a daily basis. The reason is that different ways of organizing information are convenient for different purposes. Just like we pack our clothes in a suitcase when traveling but use a closet at home, there is generally not one best way to organize anything.

Category theory provides a mathematical approach for translating between these different organizational forms. That is, it formalizes a sort of automated reorganizing process called *data migration*, which takes data that fits snugly in one schema and moves it into another.

Here is a simple case. Imagine an airline company has two different databases, perhaps created at different times, that hold roughly the same data.

$A :=$ 

$=: B$ (3.5)

Schema A has more detail than schema B—an airline seat may be in first class or economy—but they are roughly the same. We will see that they can be connected by a functor, and that data conforming to A can be migrated through this functor to schema B and vice versa.

The statistics at the beginning of this section show that this sort of problem—when occurring at enterprise scale—continues to prove difficult and expensive. If one attempts to move data from a source schema to a target schema, the migrated data could fail to fit into the target schema or fail to satisfy some of its constraints. This happens surprisingly often in the world of business: a night may be spent moving data, and the next morning it is found to have arrived broken and unsuitable for further use. In fact, it is believed that over half of database migration projects fail.

In this chapter, we will discuss a category-theoretic method for migrating data. Using categories and functors, one can prove up front that a given data migration will not fail, i.e. that the result is guaranteed to fit into the target schema and satisfy all its constraints.

The material in this chapter gets to the heart of category theory: in particular, we discuss categories, functors, natural transformations, adjunctions, limits, and colimits. In fact, many of these ideas have been present in the discussion above:

- The schema pictures, e.g. Eq. (3.4) depict categories $\mathcal{C}$.

- The instances, e.g. Eq. (3.1) are functors from $\mathcal{C}$ to a certain category called **Set**.
- The implicit mapping in Eq. (3.5), which takes economy and first class seats in A to airline seats in B, constitutes a functor $A \to B$.
- The notion of data migration for moving data between schemas is formalized by adjoint functors.

We begin in Section 3.2 with the definition of categories and a bunch of different sorts of examples. In Section 3.3 we bring back databases, in particular their instances and the maps between them, by discussing functors and natural transformations. In Section 3.4 we discuss data migration by way of adjunctions, which generalize the Galois connections we introduced in Section 1.4. Finally in Section 3.5 we give a bonus section on limits and colimits.[1]

3.2 Categories

A category $\mathcal{C}$ consists of four pieces of data—objects, morphisms, identities, and a composition rule—satisfying two properties.

Definition 3.6. To specify a *category* $\mathcal{C}$:

(i) one specifies a collection[2] $\mathrm{Ob}(\mathcal{C})$, elements of which are called *objects*.

(ii) for every two objects c, d, one specifies a set $\mathcal{C}(c,d)$,[3] elements of which are called *morphisms* from c to d.

(iii) for every object $c \in \mathrm{Ob}(\mathcal{C})$, one specifies a morphism $\mathrm{id}_c \in \mathcal{C}(c,c)$, called the *identity morphism* on c.

(iv) for every three objects $c, d, e \in \mathrm{Ob}(\mathcal{C})$ and morphisms $f \in \mathcal{C}(c,d)$ and $g \in \mathcal{C}(d,e)$, one specifies a morphism $f \fatsemi g \in \mathcal{C}(c,e)$, called *the composite of f and g*.

We will sometimes write an object $c \in \mathcal{C}$, instead of $c \in \mathrm{Ob}(\mathcal{C})$. It will also be convenient to denote elements $f \in \mathcal{C}(c,d)$ as $f : c \to d$. Here, c is called the *domain* of f, and d is called the *codomain* of f.

These constituents are required to satisfy two conditions:

(a) *unitality*: for any morphism $f : c \to d$, composing with the identities at c or d does nothing: $\mathrm{id}_c \fatsemi f = f$ and $f \fatsemi \mathrm{id}_d = f$.

(b) *associativity*: for any three morphisms $f : c_0 \to c_1$, $g : c_1 \to c_2$, and $h : c_2 \to c_3$, the following are equal: $(f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h)$. We write this composite simply as $f \fatsemi g \fatsemi h$.

[1] By "bonus," we mean that although not strictly essential to the understanding of this particular chapter, limits and colimits will show up throughout the book and throughout one's interaction with category theory, and we think the reader will especially benefit from this material in the long run.

[2] Here, a *collection* can be thought of as a bunch of things, just like a set, but that may be too large to formally be a set. An example is the collection of all sets, which would run afoul of Russell's paradox if it were itself a set.

[3] This set $\mathcal{C}(c,d)$ is often denoted $\mathrm{Hom}_{\mathcal{C}}(c,d)$, and called the "hom-set from c to d." The word "hom" stands for homomorphism, of which the word "morphism" is a shortened version.

Our next goal is to give lots of examples of categories. Our first source of examples is that of free and finitely-presented categories, which generalize the notion of Hasse diagram from Remark 1.39.

3.2.1 Free categories

Recall from Definition 1.36 that a graph consists of two types of thing: vertices and arrows. From there one can define paths, which are just head-to-tail sequences of arrows. Every path p has a start vertex and an end vertex; if p goes from v to w, we write $p\colon v \to w$. To every vertex v, there is a trivial path, containing no arrows, starting and ending at v; we often denote it by id_v or simply by v. We may also concatenate paths: given $p\colon v \to w$ and $q\colon w \to x$, their concatenation is denoted $p \fatsemi q$, and it goes $v \to x$.

In Chapter 1, we used graphs to depict preorders $(V, \leq)$: the vertices form the elements of the preorder, and we say that $v \leq w$ if there is a path from v to w in G. We will now use graphs in a very similar way to depict certain categories, known as *free categories*. Then we will explain a strong relationship between preorders and categories in Section 3.2.3.

Definition 3.7. For any graph $G = (V, A, s, t)$, we can define a category $\mathbf{Free}(G)$, called the *free category on* G, whose objects are the vertices V and whose morphisms from c to d are the paths from c to d. The identity morphism on an object c is simply the trivial path at c. Composition is given by concatenation of paths.

For example, we define $\mathbf{2}$ to be the free category generated by the graph shown below:

$$\mathbf{2} := \mathbf{Free}\left(\boxed{ v_1 \bullet \xrightarrow{f_1} \bullet v_2 } \right) \tag{3.8}$$

It has two objects v_1 and v_2, and three morphisms: $\mathrm{id}_{v_1}\colon v_1 \to v_1$, $f_1\colon v_1 \to v_2$, and $\mathrm{id}_{v_2}\colon v_2 \to v_2$. Here id_{v_1} is the path of length 0 starting and ending at v_1, f_1 is the path of length 1 consisting of just the arrow f_1, and id_{v_2} is the length 0 path at v_2. As our notation suggests, id_{v_1} is the identity morphism for the object v_1, and similarly id_{v_2} for v_2. As composition is given by concatenation, we have, for example $\mathrm{id}_{v_1} \fatsemi f_1 = f_1$, $\mathrm{id}_{v_2} \fatsemi \mathrm{id}_{v_2} = \mathrm{id}_{v_2}$, and so on.

From now on, we may elide the difference between a graph and the corresponding free category $\mathbf{Free}(G)$, at least when the one we mean is clear enough from context.

Exercise 3.9. For $\mathbf{Free}(G)$ to really be a category, we must check that this data we specified obeys the unitality and associativity properties. Check that these are obeyed for any graph G. ◊

Exercise 3.10. The free category on the graph shown here:[4]

$$\mathbf{3} := \mathrm{Free}\left(\boxed{v_1 \bullet \xrightarrow{f_1} \bullet v_2 \xrightarrow{f_2} \bullet v_3} \right) \tag{3.11}$$

has three objects and six morphisms: the three vertices and six paths in the graph.

Create six names, one for each of the six morphisms in **3**. Write down a six-by-six table, label the rows and columns by the six names you chose.

1. Fill out the table by writing the name of the composite in each cell, when there is a composite.
2. Where are the identities? ◊

Exercise 3.12. Let's make some definitions, based on the pattern above:

1. What is the category **1**? That is, what are its objects and morphisms?
2. What is the category **0**?
3. What is the formula for the number of morphisms in **n** for arbitrary $n \in \mathbb{N}$? ◊

Example 3.13 (Natural numbers as a free category). Consider the following graph:

(3.14)

It has only one vertex and one arrow, but it has infinitely many paths. Indeed, it has a unique path of length n for every natural number $n \in \mathbb{N}$. That is, Path $= \{z, s, (s \fatsemi s), (s \fatsemi s \fatsemi s), \ldots\}$, where we write z for the length 0 path on z; it represents the morphism id_z. There is a one-to-one correspondence between Path and the natural numbers, $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

This is an example of a category with one object. A category with one object is called a *monoid*, a notion we first discussed in Example 2.6. There we said that a monoid is a tuple $(M, *, e)$ where $*\colon M \times M \to M$ is a function and $e \in M$ is an element, and $m * 1 = m = 1 * m$ and $(m * n) * p = m * (n * p)$.

The two notions may superficially look different, but it is easy to describe the connection. Given a category $\mathcal{C}$ with one object, say $\bullet$, let $M := \mathcal{C}(\bullet, \bullet)$, let $e = \mathrm{id}_\bullet$, and let $*\colon \mathcal{C}(\bullet, \bullet) \times \mathcal{C}(\bullet, \bullet) \to \mathcal{C}(\bullet, \bullet)$ be the composition operation $* = \fatsemi$. The associativity and unitality requirements for the monoid will be satisfied because $\mathcal{C}$ is a category.

Exercise 3.15. In Example 3.13 we identified the paths of the loop graph (3.14) with numbers $n \in \mathbb{N}$. Paths can be concatenated. Given numbers $m, n \in \mathbb{N}$, what number corresponds to the concatenation of their associated paths? ◊

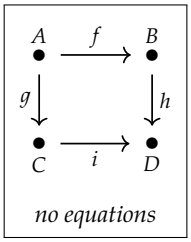
[4] As mentioned above, we elide the difference between the graph and the corresponding free category.

3.2.2 Presenting categories via path equations

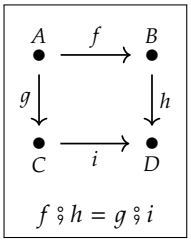
So for any graph G, there is a free category on G. But we don't have to stop there: we can add equations between paths in the graph, and still get a category. We are only allowed to equate two paths p and q when they are *parallel*, meaning they have the same source vertex and the same target vertex.

A finite graph with path equations is called a *finite presentation* for a category, and the category that results is known as a *finitely-presented category*. Here are two examples:

Free_square :=



Comm_square :=



Both of these are presentations of categories: in the left-hand one, there are no equations so it presents a free category, as discussed in Section 3.2.1. The free square category has ten morphisms, because every path is a unique morphism.

Exercise 3.16.

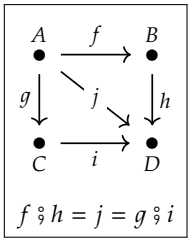
1. Write down the ten paths in the free square category above.
2. Name two different paths that are parallel.
3. Name two different paths that are not parallel. ◊

On the other hand, the category presented on the right has only nine morphisms, because $f \fatsemi h$ and $g \fatsemi i$ are made equal. This category is called the "commutative square." Its morphisms are

$$\{A, B, C, D, f, g, h, i, f \fatsemi h\}$$

One might say "the missing one is $g \fatsemi i$," but that is not quite right: $g \fatsemi i$ is there too, because it is equal to $f \fatsemi h$. As usual, A denotes id_A, etc.

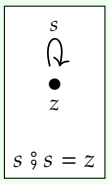
Exercise 3.17. Write down all the morphisms in the category presented by the following diagram:

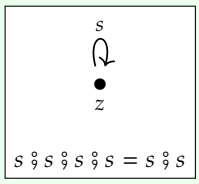


◊

Example 3.18. We should also be aware that enforcing an equation between two morphisms often implies additional equations. Here are two more examples of presenta-

tions, in which this phenomenon occurs:

$\mathcal{C} :=$ 

$\mathcal{D} :=$ 

In $\mathcal{C}$ we have the equation $s \fatsemi s = z$. But this implies $s \fatsemi s \fatsemi s = z \fatsemi s = s$! And similarly we have $s \fatsemi s \fatsemi s \fatsemi s = z \fatsemi z = z$. The set of morphisms in $\mathcal{C}$ is in fact merely $\{z, s\}$, with composition described by $s \fatsemi s = z \fatsemi z = z$, and $z \fatsemi s = s \fatsemi z = s$. In group theory, one would speak of a group called $\mathbb{Z}/2\mathbb{Z}$.

Exercise 3.19. Write down all the morphisms in the category $\mathcal{D}$ from Example 3.18. ◊

Remark 3.20. We can now see that the schemas in Section 3.1, e.g. Eqs. (3.2) and (3.4) are finite presentations of categories. We will come back to this idea in Section 3.3.

3.2.3 Preorders and free categories: two ends of a spectrum

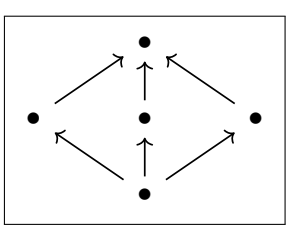
Now that we have used graphs to depict preorders in Chapter 1 and categories above, one may want to know the relationship between these two uses. The main idea we want to explain now is that

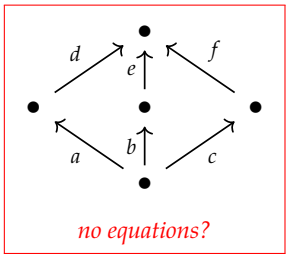
> "A preorder is a category where every two parallel arrows are the same."

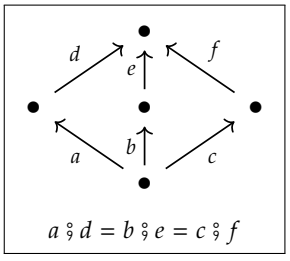
Thus any preorder can be regarded as a category, and any category can be somehow "crushed down" into a preorder. Let's discuss these ideas.

Preorders as categories. Suppose $(P, \leq)$ is a preorder. It specifies a category $\mathcal{P}$ as follows. The objects of $\mathcal{P}$ are precisely the elements of P; that is, $\mathrm{Ob}(\mathcal{P}) = P$. As for morphisms, $\mathcal{P}$ has exactly one morphism $p \to q$ if $p \leq q$ and no morphisms $p \to q$ if $p \not\leq q$. The fact that $\leq$ is reflexive ensures that every object has an identity, and the fact that $\leq$ is transitive ensures that morphisms can be composed. We call $\mathcal{P}$ the *category corresponding to the preorder* $(P, \leq)$.

In fact, a Hasse diagram for a preorder can be thought of a presentation of a category where, for all vertices p and q, every two paths from $p \to q$ are declared equal. For example, in Eq. (1.5) we saw a Hasse diagram that was like the graph on the left:

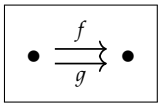
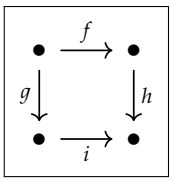
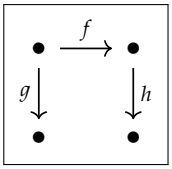






The Hasse diagram (left) might look the most like the free category presentation (middle) which has no equations, but that is not correct. The free category has three morphisms (paths) from bottom object to top object, whereas preorders are categories with *at most one* morphism between two given objects. Instead, the diagram on the right, with these paths from bottom to top made equal, is the correct presentation for the preorder on the left.

Exercise 3.21. What equations would you need to add to the graphs below in order to present the associated preorders?

$G_1 =$  $G_2 =$  $G_3 =$  $G_4 =$  ◊

The preorder reflection of a category. Given any category $\mathcal{C}$, one can obtain a preorder $(C, \leq)$ from it by destroying the distinction between any two parallel morphisms. That is, let $C := \mathrm{Ob}(\mathcal{C})$, and put $c_1 \leq c_2$ iff $\mathcal{C}(c_1, c_2) \neq \emptyset$. If there is one, or two, or fifty, or infinitely many morphisms $c_1 \to c_2$ in $\mathcal{C}$, the preorder reflection does not see the difference. But it does see the difference between some morphisms and no morphisms.

Exercise 3.22. What is the preorder reflection of the category $\mathbb{N}$ from Example 3.13? ◊

We have only discussed adjoint functors between preorders, but soon we will discuss adjoints in general. Here is a statement you might not understand exactly, but it's true; you can ask a category theory expert about it and they should be able to explain it to you:

> Considering a preorder as a category is right adjoint to turning a category into a preorder by preorder reflection.

Remark 3.23 (Ends of a spectrum). The main point of this subsection is that both preorders and free categories are specified by a graph without path equations, but they denote opposite ends of a spectrum. In both cases, the vertices of the graph become the objects of a category and the paths become morphisms. But in the case of free categories, there are no equations so each path becomes a different morphism. In the case of preorders, all parallel paths become the same morphism. Every category presentation, i.e. graph with some equations, lies somewhere in between the free category (no equations) and its preorder reflection (all possible equations).

3.2.4 Important categories in mathematics

We have been talking about category presentations, but there are categories that are best understood directly, not by way of presentations. Recall the definition of category

from Definition 3.6. The most important category in mathematics is the category of sets.

Definition 3.24. The *category of sets*, denoted **Set**, is defined as follows.

(i) $\mathrm{Ob}(\mathbf{Set})$ is the collection of all sets.

(ii) If S and T are sets, then $\mathbf{Set}(S,T) = \{f : S \to T \mid f \text{ is a function}\}$.

(iii) For each set S, the identity morphism is the function $\mathrm{id}_S : S \to S$ given by $\mathrm{id}_S(s) := s$ for each $s \in S$.

(iv) Given $f : S \to T$ and $g : T \to U$, their composite is the function $f \fatsemi g : S \to U$ given by $(f \fatsemi g)(s) := g(f(s))$.

These definitions satisfy the unitality and associativity conditions, so **Set** is indeed a category.

Closely related is the category **FinSet**. This is the category whose objects are finite sets and whose morphisms are functions between them.

Exercise 3.25. Let $\underline{2} = \{1,2\}$ and $\underline{3} = \{1,2,3\}$. These are objects in the category **Set** discussed in Definition 3.24. Write down all the elements of the set $\mathbf{Set}(\underline{2},\underline{3})$; there should be nine. ◊

Remark 3.26. You may have wondered what categories have to do with $\mathcal{V}$-categories (Definition 2.46); perhaps you think the definitions hardly look alike. Despite the term 'enriched category', $\mathcal{V}$-categories are not categories with extra structure. While some sorts of $\mathcal{V}$-categories, such as **Bool**-categories, i.e. preorders, can naturally be seen as categories, other sorts, such as **Cost**-categories, cannot.

The reason for the importance of **Set** is that, if we generalize the definition of enriched category (Definition 2.46), we find that categories in the sense of Definition 3.6 are exactly **Set**-categories—so categories are $\mathcal{V}$-categories for a very special choice of $\mathcal{V}$. We'll come back to this in Section 4.4.4. For now, we simply remark that just like a deep understanding of the category **Cost**—for example, knowing that it is a quantale—yields insight into Lawvere metric spaces, so the study of **Set** yields insights into categories.

There are many other categories that mathematicians care about:

- **Top**: the category of topological spaces (neighborhood)
- **Grph**: the category of graphs (connection)
- **Meas**: the category of measure spaces (amount)
- **Mon**: the category of monoids (action)
- **Grp**: the category of groups (reversible action, symmetry)
- **Cat**: the category of categories (action in context, structure)

But in fact, this does not at all do justice to the diversity of categories mathematicians think about. They work with whatever category they find fits their purpose at the time, like 'the category of connected Riemannian manifolds of dimension at most 4'.

Here is one more source of examples: take any category you already have and reverse all its morphisms; the result is again a category.

Example 3.27. Let $\mathcal{C}$ be a category. Its *opposite*, denoted $\mathcal{C}^{\mathrm{op}}$, is the category with the same objects, $\mathrm{Ob}(\mathcal{C}^{\mathrm{op}}) := \mathrm{Ob}(\mathcal{C})$, and for any two objects $c, d \in \mathrm{Ob}(\mathcal{C})$, one has $\mathcal{C}^{\mathrm{op}}(c,d) := \mathcal{C}(d,c)$. Identities and composition are as in $\mathcal{C}$.

3.2.5 Isomorphisms in a category

The previous sections have all been about examples of categories: free categories, presented categories, and important categories in math. In this section, we briefly switch gears and talk about an important concept in category theory, namely the concept of isomorphism.

In a category, there is often the idea that two objects are interchangeable. For example, in the category **Set**, one can exchange the set $\{\blacksquare, \square\}$ for the set $\{0,1\}$ and everything will be the same, other than the names for the elements. Similarly, if one has a preorder with elements a, b, such that $a \leq b$ and $b \leq a$, i.e. $a \cong b$, then a and b are essentially the same. How so? Well they act the same, in that for any other object c, we know that $c \leq a$ iff $c \leq b$, and $c \geq a$ iff $c \geq b$. The notion of isomorphism formalizes this notion of interchangeability.

Definition 3.28. An *isomorphism* is a morphism $f : A \to B$ such that there exists a morphism $g : B \to A$ satisfying $f \fatsemi g = \mathrm{id}_A$ and $g \fatsemi f = \mathrm{id}_B$. In this case we call f and g *inverses*, and we often write $g = f^{-1}$, or equivalently $f = g^{-1}$. We also say that A and B are *isomorphic* objects.

Example 3.29. The set $A := \{a, b, c\}$ and the set $\underline{3} = \{1, 2, 3\}$ are isomorphic; that is, there exists an isomorphism $f : A \to \underline{3}$ given by $f(a) = 2$, $f(b) = 1$, $f(c) = 3$. The isomorphisms in the category **Set** are the bijections.

Recall that the cardinality of a finite set is the number of elements in it. This can be understood in terms of isomorphisms in **FinSet**. Namely, for any finite set $A \in \mathbf{FinSet}$, its cardinality is the number $n \in \mathbb{N}$ such that there exists an isomorphism $A \cong \underline{n}$. Georg Cantor defined the cardinality of any set X to be its isomorphism class, meaning the equivalence class consisting of all sets that are isomorphic to X.

Exercise 3.30.

1. What is the inverse $f^{-1} : \underline{3} \to A$ of the function f given in Example 3.29?
2. How many distinct isomorphisms are there $A \to \underline{3}$? ◊

Exercise 3.31. Show that in any given category $\mathcal{C}$, for any given object $c \in \mathcal{C}$, the identity id_c is an isomorphism. ◊

Exercise 3.32. Recall Examples 3.13 and 3.18. A monoid in which every morphism is an isomorphism is known as a *group*.

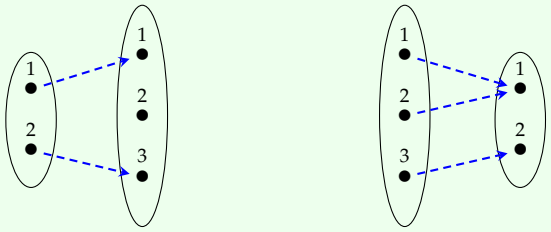
1. Is the monoid in Example 3.13 a group?

2. What about the monoid $\mathcal{C}$ in Example 3.18? ◊

Exercise 3.33. Let G be a graph, and let **Free**(G) be the corresponding free category. Somebody tells you that the only isomorphisms in **Free**(G) are the identity morphisms. Is that person correct? Why or why not? ◊

Example 3.34. In this example, we will see that it is possible for g and f to be almost—but not quite—inverses, in a certain sense.

Consider the functions $f : \underline{2} \to \underline{3}$ and $g : \underline{3} \to \underline{2}$ drawn below:



Then the reader should be able to instantly check that $f \fatsemi g = \mathrm{id}_{\underline{2}}$ but $g \fatsemi f \neq \mathrm{id}_{\underline{3}}$. Thus f and g are not inverses and hence not isomorphisms. We won't need this terminology, but category theorists would say that f and g form a *retraction*.

3.3 Functors, natural transformations, and databases

In Section 3.1 we showed some database schemas: graphs with path equations. Then in Section 3.2.2 we said that graphs with path equations correspond to finitely-presented categories. Now we want to explain what the data in a database is, as a way to introduce functors. To do so, we begin by noticing that sets and functions—the objects and morphisms in the category **Set**—can be captured by particularly simple databases.

3.3.1 Sets and functions as databases

The first observation is that any set can be understood as a table with only one column: the ID column.

Planet of Sol
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune

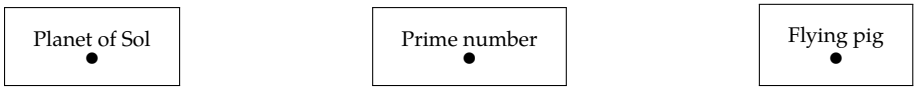
Prime number
2
3
5
7
11
13
17
⋮

Flying pig

Rather than put the elements of the set between braces, e.g. $\{2, 3, 5, 7, 11, \ldots\}$, we write them down as rows in a table.

In databases, single-column tables are often called controlled vocabularies, or master data. Now to be honest, we can only write out every single entry in a table when its set of rows is finite. A database practitioner might find the idea of our prime number table a bit unrealistic. But we're mathematicians, so since the idea makes perfect sense abstractly, we will continue to think of sets as one-column tables.

The above databases have schemas consisting of just one vertex:



Obviously, there's really not much difference between these schemas, other than the label of the unique vertex. So we could say "sets are databases whose schema consists of a single vertex." Let's move on to functions.

A function $f : A \to B$ can almost be depicted as a two-column table

Beatle	**Played**
George	Lead guitar
John	Rhythm guitar
Paul	Bass guitar
Ringo	Drums

except it is unclear whether the elements of the right-hand column exhaust all of B. What if there are rock-and-roll instruments out there that none of the Beatles played? So a function $f : A \to B$ requires two tables, one for A and its f column, and one for B:

Beatle	**Played**
George	Lead guitar
John	Rhythm guitar
Paul	Bass guitar
Ringo	Drums

Rock-and-roll instrument
Bass guitar
Drums
Keyboard
Lead guitar
Rhythm guitar

Thus the database schema for any function is just a labeled version of **2**:

The lesson is that an instance of a database takes a presentation of a category, and turns every vertex into a set, and every arrow into a function. As such, it describes a map from the presented category to the category **Set**. In Section 2.4.2 we saw that maps of $\mathcal{V}$-categories are known as $\mathcal{V}$-functors. Similarly, we call maps of plain old categories, functors.

3.3.2 Functors

A functor is a mapping between categories. It sends objects to objects and morphisms to morphisms, all while preserving identities and composition. Here is the formal definition.

Definition 3.35. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. To specify a *functor from* $\mathcal{C}$ *to* $\mathcal{D}$, denoted $F\colon \mathcal{C} \to \mathcal{D}$,

(i) for every object $c \in \mathrm{Ob}(\mathcal{C})$, one specifies an object $F(c) \in \mathrm{Ob}(\mathcal{D})$;

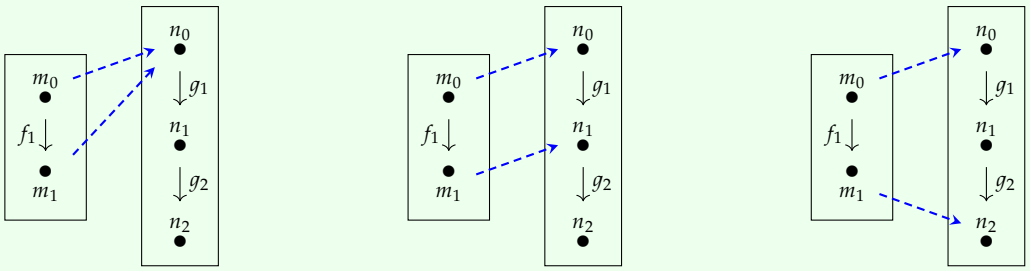
(ii) for every morphism $f\colon c_1 \to c_2$ in $\mathcal{C}$, one specifies a morphism $F(f)\colon F(c_1) \to F(c_2)$ in $\mathcal{D}$.

The above constituents must satisfy two properties:

(a) for every object $c \in \mathrm{Ob}(\mathcal{C})$, we have $F(\mathrm{id}_c) = \mathrm{id}_{F(c)}$.

(b) for every three objects $c_1, c_2, c_3 \in \mathrm{Ob}(\mathcal{C})$ and two morphisms $f \in \mathcal{C}(c_1, c_2)$, $g \in \mathcal{C}(c_2, c_3)$, the equation $F(f \fatsemi g) = F(f) \fatsemi F(g)$ holds in $\mathcal{D}$.

Example 3.36. For example, here we draw three functors $F\colon \mathbf{2} \to \mathbf{3}$:



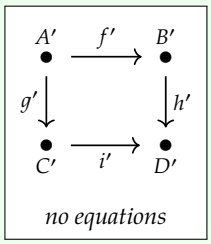
In each case, the dotted arrows show what the functor F does to the vertices in $\mathbf{2}$; once that information is specified, it turns out—in this special case—that what F does to the three paths in $\mathbf{2}$ is completely determined. In the left-hand diagram, F sends every path to the trivial path, i.e. the identity on n_0. In the middle diagram $F(m_0) = n_0$, $F(f_1) = g_1$, and $F(m_1) = n_1$. In the right-hand diagram, $F(m_0) = n_0$, $F(m_1) = n_2$, and $F(f_1) = g_1 \fatsemi g_2$.

Exercise 3.37. Above we wrote down three functors $\mathbf{2} \to \mathbf{3}$. Find and write down all the remaining functors $\mathbf{2} \to \mathbf{3}$. ◊

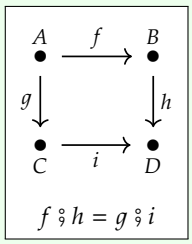
Example 3.38. Recall the categories presented by Free_square and Comm_square in Section 3.2.2. Here they are again, with ′ added to the labels in Free_square to help

distinguish them:

Free_square :=



Comm_square :=



There are lots of functors from the free square category (let's call it $\mathcal{F}$) to the commutative square category (let's call it $\mathcal{C}$).

However, there is exactly one functor $F\colon \mathcal{F} \to \mathcal{C}$ that sends A' to A, B' to B, C' to C, and D' to D. That is, once we have made this decision about how F acts on objects, each of the ten paths in $\mathcal{F}$ is forced to go to a certain path in $\mathcal{C}$: the one with the right source and target.

Exercise 3.39. Say where each of the ten morphisms in $\mathcal{F}$ is sent under the functor F from Example 3.38. ◊

All of our example functors so far have been completely determined by what they do on objects, but this is usually not the case.

Exercise 3.40. Consider the free categories $\mathcal{C} = \boxed{\bullet \to \bullet}$ and $\mathcal{D} = \boxed{\bullet \rightrightarrows \bullet}$. Give two functors $F, G\colon \mathcal{C} \to \mathcal{D}$ that act the same on objects but differently on morphisms. ◊

Example 3.41. There are also lots of functors from the commutative square category $\mathcal{C}$ to the free square category $\mathcal{F}$, but *none* that sends A to A', B to B', C to C', and D to D'. The reason is that if F were such a functor, then since $f \fatsemi h = g \fatsemi i$ in $\mathcal{C}$, we would have $F(f \fatsemi h) = F(g \fatsemi i)$, but then the rules of functors would let us reason as follows:

$$f' \fatsemi h' = F(f) \fatsemi F(h) = F(f \fatsemi h) = F(g \fatsemi i) = F(g) \fatsemi F(i) = g' \fatsemi i'$$

The resulting equation, $f' \fatsemi h' = g' \fatsemi i'$ does not hold in $\mathcal{F}$ because it is a free category (there are "no equations"): every two paths are considered different morphisms. Thus our proposed F is not a functor.

Example 3.42 (Functors between preorders are monotone maps). Recall from Section 3.2.3 that preorders are categories with at most one morphism between any two objects. A functor between preorders is exactly a monotone map.

For example, consider the preorder $(\mathbb{N}, \leq)$ considered as a category $\mathcal{N}$ with objects $\mathrm{Ob}(\mathcal{N}) = \mathbb{N}$ and a unique morphism $m \to n$ iff $m \leq n$. A functor $F\colon \mathcal{N} \to \mathcal{N}$ sends each object $n \in \mathbb{N}$ to an object $F(n) \in \mathbb{N}$. It must send morphisms in $\mathcal{N}$ to morphisms in $\mathbb{N}$. This means if there is a morphism $m \to n$ then there had better be a morphism $F(m) \to F(n)$. In other words, if $m \leq n$, then we had better have $F(m) \leq F(n)$. But as

long as $m \leq n$ implies $F(m) \leq F(n)$, we have a functor.

Thus a functor $F\colon \mathcal{N} \to \mathcal{N}$ and a monotone map $\mathbb{N} \to \mathbb{N}$ are the same thing.

Exercise 3.43 (The category of categories). Back in the primordial ooze, there is a category **Cat** in which *the objects are themselves categories*. Your task here is to construct this category.

1. Given any category $\mathcal{C}$, show that there exists a functor $\mathrm{id}_{\mathcal{C}}\colon \mathcal{C} \to \mathcal{C}$, known as the *identity functor on* $\mathcal{C}$, that maps each object to itself and each morphism to itself.

Note that a functor $\mathcal{C} \to \mathcal{D}$ consists of a function from $\mathrm{Ob}(\mathcal{C})$ to $\mathrm{Ob}(\mathcal{D})$ and for each pair of objects $c_1, c_2 \in \mathcal{C}$ a function from $\mathcal{C}(c_1, c_2)$ to $\mathcal{D}(F(c_1), F(c_2))$.

2. Show that given $F\colon \mathcal{C} \to \mathcal{D}$ and $G\colon \mathcal{D} \to \mathcal{E}$, we can define a new functor $(F \fatsemi G)\colon \mathcal{C} \to \mathcal{E}$ just by composing functions.
3. Show that there is a category, call it **Cat**, where the objects are categories, morphisms are functors, and identities and composition are given as above. ◊

3.3.3 Database instances as Set-valued functors

Let $\mathcal{C}$ be a category, and recall the category **Set** from Definition 3.24. A functor $F\colon \mathcal{C} \to$ **Set** is known as a *set-valued functor* on $\mathcal{C}$. Much of database theory (not how to make them fast, but what they are and what you do with them) can be cast in this light.

Indeed, we already saw in Remark 3.20 that any database schema can be regarded as (presenting) a category $\mathcal{C}$. The next thing to notice is that the data itself—any instance of the database—is given by a set-valued functor $I\colon \mathcal{C} \to$ **Set**. The only additional detail is that for any white node, such as $c = \overset{\text{string}}{\circ}$, we want to force I to map to the set of strings. We suppress this detail in the following definition.

Definition 3.44. Let $\mathcal{C}$ be a schema, i.e. a finitely-presented category. A $\mathcal{C}$*-instance* is a functor $I\colon \mathcal{C} \to$ **Set**.[5]

Exercise 3.45. Let **1** denote the category with one object, called 1, one identity morphism id_1, and no other morphisms. For any functor $F\colon \mathbf{1} \to$ **Set** one can extract a set $F(1)$. Show that for any set S, there is a functor $F_S\colon \mathbf{1} \to$ **Set** such that $F_S(1) = S$. ◊

The above exercise reaffirms that the set of planets, the set of prime numbers, and the set of flying pigs are all set-valued functors—instances—on the schema **1**. Similarly, set-valued functors on the category **2** are functions. All our examples so far are for the situation where the schema is a free category (no equations). Let's try an example of a category that is not free.

[5]Warning: a $\mathcal{C}$-instance is a state of the database "at an instant in time." The term "instance" should not be confused with its usage in object oriented programming, which would correspond more to what we call a row $r \in I(c)$.

Example 3.46. Consider the following category:

$$\mathcal{C} := \boxed{\begin{array}{c} s \\ \circlearrowright \\ \bullet \\ z \\ \\ s \fatsemi s = s \end{array}} \tag{3.47}$$

What is a set-valued functor $F\colon \mathcal{C} \to \mathbf{Set}$? It will consist of a set $Z := F(z)$ and a function $S := F(s)\colon Z \to Z$, subject to the requirement that $S \fatsemi S = S$. Here are some examples

- Z is the set of US citizens, and S sends each citizen to her or his president. The president's president is her- or him-self.
- $Z = \mathbb{N}$ is the set of natural numbers and S sends each number to 0. In particular, 0 goes to itself.
- Z is the set of all well-formed arithmetic expressions, such as $13+(2*4)$ or -5, that one can write using integers and the symbols $+, -, *, (,)$. The function S evaluates the expression to return an integer, which is itself a well-formed expression. The evaluation of an integer is itself.
- $Z = \mathbb{N}_{\geq 2}$, and S sends n to its smallest prime factor. The smallest prime factor of a prime is itself.

$\mathbb{N}_{\geq 2}$	**smallest prime factor**
2	2
3	3
4	2
⋮	⋮
49	7
50	2
51	3
⋮	⋮

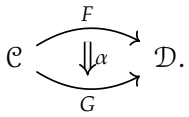
Exercise 3.48. Above, we thought of the sort of data that would make sense for the schema (3.47). Give an example of the sort of data that would make sense for the following schemas: 1. $\boxed{\begin{array}{c} s \\ \circlearrowright \\ \bullet \\ z \\ \\ s \fatsemi s = z \end{array}}$ 2. $\boxed{\begin{array}{c} a \quad f \quad b \quad g \quad c \\ \bullet \longrightarrow \bullet \rightrightarrows \bullet \\ \qquad\qquad\quad h \\ \\ f \fatsemi g = f \fatsemi h \end{array}}$ ◊

The main idea is this: a database schema is a category, and an instance on that schema—the data itself—is a set-valued functor. All the constraints, or business rules, are ensured by the rules of functors, namely that functors preserve composition.[6]

[6] One can put more complex constraints, called *embedded dependencies*, on a database; these correspond category theoretically to what are called "lifting problems" in category theory. See [Spi14b] for more on this.

3.3.4 Natural transformations

If $\mathcal{C}$ is a schema—i.e. a finitely-presented category—then there are many database instances on it, which we can organize into a category. But this is part of a larger story, namely that of natural transformations. An abstract picture to have in mind is this:



Definition 3.49. Let $\mathcal{C}$ and $\mathcal{D}$ be categories, and let $F, G\colon \mathcal{C} \to \mathcal{D}$ be functors. To specify a *natural transformation* $\alpha\colon F \Rightarrow G$,

(i) for each object $c \in \mathcal{C}$, one specifies a morphism $\alpha_c\colon F(c) \to G(c)$ in $\mathcal{D}$, called the *c-component of* α.

These components must satisfy the following, called the *naturality condition*:

(a) for every morphism $f\colon c \to d$ in $\mathcal{C}$, the following equation must hold:

$$F(f) \fatsemi \alpha_d = \alpha_c \fatsemi G(f).$$

A natural transformation $\alpha\colon F \to G$ is called a *natural isomorphism* if each component α_c is an isomorphism in $\mathcal{D}$.

The naturality condition can also be written as a so-called *commutative diagram*. A diagram in a category is drawn as a graph whose vertices and arrows are labeled by objects and morphisms in the category. For example, here is a diagram that's relevant to the naturality condition in Definition 3.49:

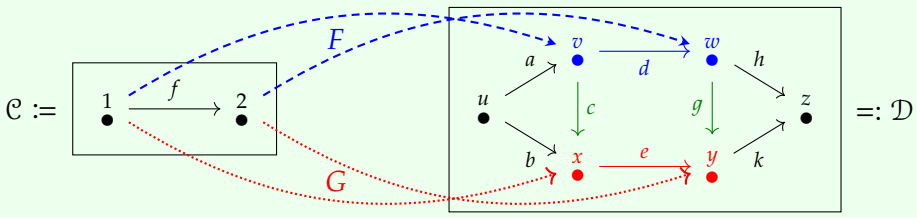
$$\begin{array}{ccc} F(c) & \xrightarrow{\alpha_c} & G(c) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(d) & \xrightarrow[\alpha_d]{} & G(d) \end{array} \tag{3.50}$$

Definition 3.51. A *diagram* D in $\mathcal{C}$ is a functor $D\colon \mathcal{J} \to \mathcal{C}$ from any category $\mathcal{J}$, called the *indexing category* of the diagram D. We say that D *commutes* if $D(f) = D(f')$ holds for every parallel pair of morphisms $f, f'\colon a \to b$ in $\mathcal{J}$.[7]

In terms of Eq. (3.50), the only case of two parallel morphisms is that of $F(c) \rightrightarrows G(d)$, so to say that the diagram commutes is to say that $F(f) \fatsemi \alpha_d = \alpha_c \fatsemi G(f)$. This is exactly the naturality condition from Definition 3.49.

[7] We could package this formally by saying that D commutes iff it factors through the preorder reflection of $\mathcal{J}$.

Example 3.52. A representative picture is as follows:



We have depicted, in blue and red respectively, two functors $F, G\colon \mathcal{C} \to \mathcal{D}$. A natural transformation $\alpha\colon F \Rightarrow G$ is given by choosing components $\alpha_1\colon v \to x$ and $\alpha_2\colon w \to y$. We have highlighted the only choice for each in green; namely, $\alpha_1 = c$ and $\alpha_2 = g$.

The key point is that the functors F and G are ways of viewing the category $\mathcal{C}$ as lying inside the category $\mathcal{D}$. The natural transformation α, then, is a way of relating these two views using the morphisms in $\mathcal{D}$. Does this help you to see and appreciate the notation $\mathcal{C} \overset{F}{\underset{G}{\Downarrow \alpha}} \mathcal{D}$?

Example 3.53. We said in Exercise 3.45 that a functor $\mathbf{1} \to \mathbf{Set}$ can be identified with a set. So suppose A and B are sets considered as functors $A, B\colon \mathbf{1} \to \mathbf{Set}$. A natural transformation between these functors is just a function between the sets.

Definition 3.54. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. We denote by $\mathcal{D}^{\mathcal{C}}$ the category whose objects are functors $F\colon \mathcal{C} \to \mathcal{D}$ and whose morphisms $\mathcal{D}^{\mathcal{C}}(F, G)$ are the natural transformations $\alpha\colon F \to G$. This category $\mathcal{D}^{\mathcal{C}}$ is called the *functor category*, or the *category of functors from* $\mathcal{C}$ *to* $\mathcal{D}$.

Exercise 3.55. Let's look more deeply at how $\mathcal{D}^{\mathcal{C}}$ is a category.

1. Figure out how to compose natural transformations. (Hint: an expert tells you "for each object $c \in \mathcal{C}$, compose the c-components.")
2. Propose an identity natural transformation on any object $F \in \mathcal{D}^{\mathcal{C}}$, and check that it is unital (i.e. that it obeys condition (a) of Definition 3.6). ◊

Example 3.56. In our new language, Example 3.53 says that $\mathbf{Set}^{\mathbf{1}}$ is equivalent to $\mathbf{Set}$.

Example 3.57. Let $\mathcal{N}$ denote the category associated to the preorder $(\mathbb{N}, \leq)$, and recall from Example 3.42 that we can identify a functor $F\colon \mathcal{N} \to \mathcal{N}$ with a non-decreasing sequence $(F_0, F_1, F_2, \ldots)$ of natural numbers, i.e. $F_0 \leq F_1 \leq F_2 \leq \cdots$. If G is another functor, considered as a non-decreasing sequence, then what is a natural transformation

$\alpha : F \to G$?

Since there is at most one morphism between two objects in a preorder, each component $\alpha_n : F_n \to G_n$ has no data, it just tells us a fact: that $F_n \leq G_n$. And the naturality condition is vacuous: every square in a preorder commutes. So a natural transformation between F and G exists iff $F_n \leq G_n$ for each n, and any two natural transformations $F \Rightarrow G$ are the same. In other words, the category $\mathcal{N}^{\mathcal{N}}$ is itself a preorder; namely the preorder of monotone maps $\mathbb{N} \to \mathbb{N}$.

Exercise 3.58. Let $\mathcal{C}$ be an arbitrary category and let $\mathcal{P}$ be a preorder, thought of as a category. Consider the following statements:

1. For any two functors $F, G : \mathcal{C} \to \mathcal{P}$, there is at most one natural transformation $F \to G$.
2. For any two functors $F, G : \mathcal{P} \to \mathcal{C}$, there is at most one natural transformation $F \to G$.

For each, if it is true, say why; if it is false, give a counterexample. ◊

Remark 3.59. Recall that in Remark 2.71 we said the category of preorders is equivalent to the category of **Bool**-categories. We can now state the precise meaning of this sentence. First, there exists a category **PrO** in which the objects are preorders and the morphisms are monotone maps. Second, there exists a category **Bool-Cat** in which the objects are **Bool**-categories and the morphisms are **Bool**-functors. We call these two categories equivalent because there exist functors $F : \mathbf{PrO} \to \mathbf{Bool\text{-}Cat}$ and $G : \mathbf{Bool\text{-}Cat} \to \mathbf{PrO}$ such that there exist natural isomorphisms $F \fatsemi G \cong \mathrm{id}_{\mathbf{PrO}}$ and $G \fatsemi F \cong \mathrm{id}_{\mathbf{Bool\text{-}Cat}}$ in the sense of Definition 3.49.

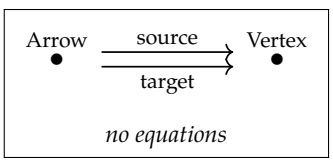
3.3.5 The category of instances on a schema

Definition 3.60. Suppose that $\mathcal{C}$ is a database schema and $I, J : \mathcal{C} \to \mathbf{Set}$ are database instances. An *instance homomorphism* between them is a natural transformation $\alpha : I \to J$. Write $\mathcal{C}$-**Inst** $:= \mathbf{Set}^{\mathcal{C}}$ to denote the functor category as defined in Definition 3.54.

We saw in Example 3.53 that **1-Inst** is equivalent to the category **Set**. In this subsection, we will show that there is a schema whose instances are graphs and whose instance homomorphisms are graph homomorphisms.

Extended example: the category of graphs as a functor category. You may find yourself back in the primordial ooze (first discussed in Section 2.3.2), because while previously we have been using graphs to present categories, now we obtain graphs themselves as database instances on a specific schema (which is itself a graph):

Gr :=



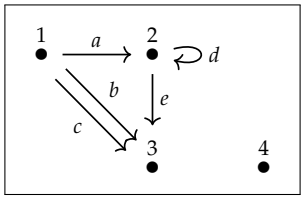
Here's an example **Gr**-instance, i.e. set-valued functor $I\colon \mathbf{Gr} \to \mathbf{Set}$, in table form:

Arrow	source	target
a	1	2
b	1	3
c	1	3
d	2	2
e	2	3

Vertex
1
2
3
4

(3.61)

Here $I(\text{Arrow}) = \{a, b, c, d, e\}$, and $I(\text{Vertex}) = \{1, 2, 3, 4\}$. One can draw the instance I as a graph:

$I =$ 

Every row in the Vertex table is drawn as a vertex, and every row in the Arrow table is drawn as an arrow, connecting its specified source and target. Every possible graph can be written as a database instance on the schema **Gr**, and every possible **Gr**-instance can be represented as a graph.

Exercise 3.62. In Eq. (3.2), a graph is shown (forget the distinction between white and black nodes). Write down the corresponding **Gr**-instance, as in Eq. (3.61). (Do not be concerned that you are in the primordial ooze.) ◊

Thus the objects in the category **Gr-Inst** are graphs. The morphisms in **Gr-Inst** are called *graph homomorphisms*. Let's unwind this. Suppose that $G, H\colon \mathbf{Gr} \to \mathbf{Set}$ are functors (i.e. **Gr**-instances); that is, they are objects $G, H \in \mathbf{Gr\text{-}Inst}$. A morphism $G \to H$ is a natural transformation $\alpha\colon G \to H$ between them; what does that entail?

By Definition 3.49, since **Gr** has two objects, α consists of two components,

$$\alpha_{\text{Vertex}}\colon G(\text{Vertex}) \to H(\text{Vertex}) \qquad \text{and} \qquad \alpha_{\text{Arrow}}\colon G(\text{Arrow}) \to H(\text{Arrow}),$$

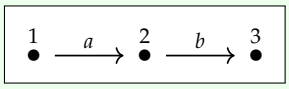
both of which are morphisms in **Set**. In other words, α consists of a function from vertices of G to vertices of H and a function from arrows of G to arrows of H. For these functions to constitute a graph homomorphism, they must "respect source and target" in the precise sense that the naturality condition, Eq. (3.50) holds. That is, for every morphism in **Gr**, namely source and target, the following diagrams must commute:

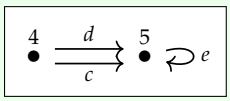
$$\begin{array}{ccc} G(\text{Arrow}) & \xrightarrow{\alpha_{\text{Arrow}}} & H(\text{Arrow}) \\ {\scriptstyle G(\text{source})}\downarrow & & \downarrow{\scriptstyle H(\text{source})} \\ G(\text{Vertex}) & \xrightarrow[\alpha_{\text{Vertex}}]{} & H(\text{Vertex}) \end{array} \qquad \begin{array}{ccc} G(\text{Arrow}) & \xrightarrow{\alpha_{\text{Arrow}}} & H(\text{Arrow}) \\ {\scriptstyle G(\text{target})}\downarrow & & \downarrow{\scriptstyle H(\text{target})} \\ G(\text{Vertex}) & \xrightarrow[\alpha_{\text{Vertex}}]{} & H(\text{Vertex}) \end{array}$$

These may look complicated, but they say exactly what we want. We want the functions α_{Vertex} and α_{Arrow} to respect source and targets in G and H. The left diagram says "start

with an arrow in G. You can either apply α to the arrow and then take its source in H, or you can take its source in G and then apply α to that vertex; either way you get the same answer." The right-hand diagram says the same thing about targets.

Example 3.63. Consider the graphs G and H shown below

$G :=$ 

$H :=$ 

Here they are, written as database instances—i.e. set-valued functors—on **Gr**:

$G :=$

Arrow	source	target
a	1	2
b	2	3

Vertex
1
2
3

$H :=$

Arrow	source	target
c	4	5
d	4	5
e	5	5

Vertex
4
5

The top row is G and the bottom row is H. They are offset so you can more easily complete the following exercise.

Exercise 3.64. We claim that—with G, H as in Example 3.63—there is exactly one graph homomorphism $\alpha\colon G \to H$ such that $\alpha_{\text{Arrow}}(a) = d$.

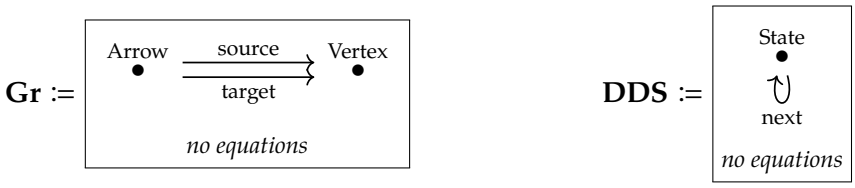
1. What is the other value of α_{Arrow}, and what are the three values of α_{Vertex}?
2. In your own copy of the tables of Example 3.63, draw α_{Arrow} as two lines connecting the cells in the ID column of $G(\text{Arrow})$ to those in the ID column of $H(\text{Arrow})$. Similarly, draw α_{Vertex} as connecting lines.
3. Check the source column and target column and make sure that the matches are natural, i.e. that "alpha-then-source equals source-then-alpha" and similarly for "target." ◊

3.4 Adjunctions and data migration

We have talked about how set-valued functors on a schema can be understood as filling that schema with data. But there are also functors between schemas. When the two sorts of functors are composed, data is migrated. This is the simplest form of data migration; more complex ways to migrate data come from using adjoints. All of the above is the subject of this section.

3.4.1 Pulling back data along a functor

To begin, we will migrate data between the graph-indexing schema **Gr** and the loop schema, which we call **DDS**, shown below



We begin by writing down a sample instance $I\colon \mathbf{DDS} \to \mathbf{Set}$ on this schema:

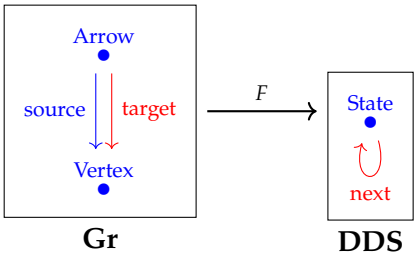
State	next
1	4
2	4
3	5
4	5
5	5
6	7
7	6

(3.65)

We call the schema **DDS** to stand for discrete dynamical system. Indeed, we may think of the data in the **DDS**-instance of Eq. (3.65) as listing the states and movements of a deterministic machine: at every point in time the machine is in one of the listed states, and given the machine in one of the states, in the next instant it moves to a uniquely determined next state.

Our goal is to migrate the data in Eq. (3.65) to data on **Gr**; this will give us the data of a graph and so allow us to visualise our machine.

We will use a functor connecting these schemas in order to move data between them. The reader can create any functor she likes, but we will use a specific functor $F\colon \mathbf{Gr} \to \mathbf{DDS}$ to migrate data in a way that makes sense to us, the authors. Here we draw F, using colors to hopefully aid understanding:



The functor F sends both objects of **Gr** to the 'State' object of **DDS** (as it must). On morphisms, it sends the 'source' morphism to the identity morphism on 'State', and the 'target' morphism to the morphism 'next'.

A sample database instance on **DDS** was given in Eq. (3.65); recall this is a functor $I\colon \mathbf{DDS} \to \mathbf{Set}$. So now we have two functors as follows:

$$\mathbf{Gr} \xrightarrow{F} \mathbf{DDS} \xrightarrow{I} \mathbf{Set}. \tag{3.66}$$

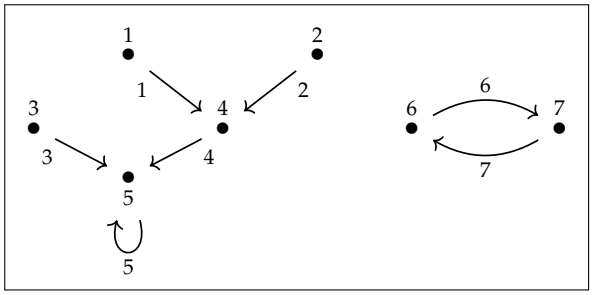
Objects in **Gr** are sent by F to objects in **DDS**, which are sent by I to objects in **Set**, which are sets. Morphisms in **Gr** are sent by F to morphisms in **DDS**, which are sent by I to morphisms in **Set**, which are functions. This defines a composite functor $F \fatsemi I\colon \mathbf{Gr} \to \mathbf{Set}$. Both F and I respect identities and composition, so $F \fatsemi I$ does too. Thus we have obtained an instance on **Gr**, i.e. we have converted our discrete dynamical system from Eq. (3.65) into a graph! What graph is it?

For an instance on **Gr**, we need to fill an Arrow table and a Vertex table. Both of these are sent by F to State, so let's fill both with the rows of State in Eq. (3.65). Similarly, since F sends 'source' to the identity and sends 'target' to 'next', we obtain the following tables:

Arrow	**source**	**target**
1	1	4
2	2	4
3	3	5
4	4	5
5	5	5
6	6	7
7	7	6

Vertex
1
2
3
4
5
6
7

Now that we have a graph, we can draw it.



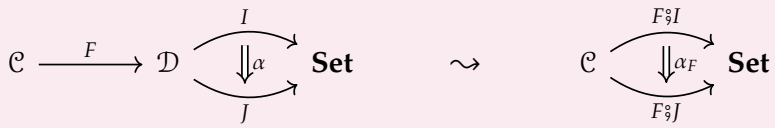
Each arrow is labeled by its source vertex, as if to say, "What I do next is determined by what I am now."

Exercise 3.67. Consider the functor $G\colon \mathbf{Gr} \to \mathbf{DDS}$ given by sending 'source' to 'next' and sending 'target' to the identity on 'State'. Migrate the same data, called I in Eq. (3.65), using the functor G. Write down the tables and draw the corresponding graph. ◊

We refer to the above procedure—basically just composing functors as in Eq. (3.66)—as "pulling back data along a functor." We just now pulled back data I along functor F.

Definition 3.68. Let $\mathcal{C}$ and $\mathcal{D}$ be categories and let $F\colon \mathcal{C} \to \mathcal{D}$ be a functor. For any set-valued functor $I\colon \mathcal{D} \to \mathbf{Set}$, we refer to the composite functor $F \fatsemi I\colon \mathcal{C} \to \mathbf{Set}$ as the *pullback of I along F*.

Given a natural transformation $\alpha\colon I \Rightarrow J$, there is a natural transformation $\alpha_F\colon F\fatsemi I \Rightarrow F \fatsemi J$, whose component $(F \fatsemi I)(c) \to (F \fatsemi J)(c)$ for any $c \in \mathrm{Ob}(\mathcal{C})$ is given by $(\alpha_F)_c \coloneqq \alpha_{Fc}$.



This uses the data of F to define a functor $\Delta_F\colon \mathcal{D}\textbf{-Inst} \to \mathcal{C}\textbf{-Inst}$.

Note that the term pullback is also used for a certain sort of limit, for more details see Remark 3.100.

3.4.2 Adjunctions

In Section 1.4 we discussed Galois connections, which are adjunctions between preorders. Now that we've defined categories and functors, we can discuss adjunctions in general. The relevance to databases is that the data migration functor Δ from Definition 3.68 always has two adjoints of its own: a left adjoint which we denote Σ and a right adjoint which we denote Π.

Recall that an adjunction between preorders P and Q is a pair of monotone maps $f\colon P \to Q$ and $g\colon Q \to P$ that are *almost* inverses: we have

$$f(p) \leq q \text{ if and only if } p \leq g(q). \tag{3.69}$$

Recall from Section 3.2.3 that in a preorder P, a hom-set $P(a, b)$ has one element when $a \leq b$, and no elements otherwise. We can thus rephrase Eq. (3.69) as an isomorphism of sets $Q(f(p), q) \cong P(p, g(q))$: either both are one-element sets or both are 0-element sets. This suggests how to define adjunctions in the general case.

Definition 3.70. Let $\mathcal{C}$ and $\mathcal{D}$ be categories, and $L\colon \mathcal{C} \to \mathcal{D}$ and $R\colon \mathcal{D} \to \mathcal{C}$ be functors. We say that L *is left adjoint to* R (and that R *is right adjoint to* L) if, for any $c \in \mathcal{C}$ and $d \in \mathcal{D}$, there is an isomorphism of hom-sets

$$\alpha_{c,d}\colon \mathcal{C}(c, R(d)) \xrightarrow{\cong} \mathcal{D}(L(c), d)$$

that is natural in c and d.[8]

Given a morphism $f\colon c \to R(d)$ in $\mathcal{C}$, its image $g \coloneqq \alpha_{c,d}(f)$ is called its *mate*. Similarly, the mate of $g\colon L(c) \to d$ is f.

To denote an adjunction we write $L \dashv R$, or in diagrams,

$$\mathcal{C} \underset{R}{\overset{L}{\rightleftarrows}} \mathcal{D}$$

with the $\Rightarrow$ in the direction of the left adjoint.

Example 3.71. Recall that every preorder $\mathcal{P}$ can be regarded as a category. Galois connections between preorders and adjunctions between the corresponding categories are exactly the same thing.

Example 3.72. Let $B \in \mathrm{Ob}(\mathbf{Set})$ be any set. There is an adjunction called 'currying B,' after the logician Haskell Curry:

$$\mathbf{Set} \underset{(-)^B}{\overset{-\times B}{\rightleftarrows}} \mathbf{Set} \qquad\qquad \mathbf{Set}(A \times B, C) \cong \mathbf{Set}(A, C^B)$$

Abstractly we write it as on the left, but what this means is that for any sets A, C, there is a natural isomorphism as on the right.

To explain this, we need to talk about exponential objects in **Set**. Suppose that B and C are sets. Then the set of functions $B \to C$ is also a set; let's denote it C^B. It's written this way because if C has 10 elements and B has 3 elements then C^B has 10^3 elements, and more generally for any two finite sets $|C^B| = |C|^{|B|}$.

The idea of currying is that given sets A, B, and C, there is a one-to-one correspondence between functions $(A \times B) \to C$ and functions $A \to C^B$. Intuitively, if I have a function f of two variables a, b, I can "put off" entering the second variable: if you give me just a, I'll return a function $B \to C$ that's waiting for the B input. This is the curried version of f. As one might guess, there is a formal connection between exponential objects and what we called hom-elements $b \multimap c$ in Definition 2.79.

Exercise 3.73. In Example 3.72, we discussed an adjunction between functors $- \times B$ and $(-)^B$. But we only said how these functors worked on objects: for an arbitrary set X, they return sets $X \times B$ and X^B respectively.

[8] This naturality is between functors $\mathcal{C}^{\mathrm{op}} \times \mathcal{D} \to \mathbf{Set}$. It says that for any morphisms $f : c' \to c$ in $\mathcal{C}$ and $g : d \to d'$ in $\mathcal{D}$, the following diagram commutes:

$$\begin{array}{ccc} \mathcal{C}(c, Rd) & \xrightarrow{\alpha_{c,d}} & \mathcal{D}(Lc, d) \\ {\scriptstyle \mathcal{C}(f, Rg)}\downarrow & & \downarrow{\scriptstyle \mathcal{D}(Lf, g)} \\ \mathcal{C}(c', Rd') & \xrightarrow[\alpha_{c',d'}]{} & \mathcal{D}(Lc', d') \end{array}$$

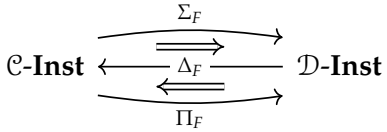
1. Given a morphism $f\colon X \to Y$, what morphism should $- \times B\colon X \times B \to Y \times B$ return?
2. Given a morphism $f\colon X \to Y$, what morphism should $(-)^B\colon X^B \to Y^B$ return?
3. Consider the function $+\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, which sends $(a, b) \mapsto a + b$. Currying $+$, we get a certain function $p\colon \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$. What is $p(3)$? ◊

Example 3.74. If you know some abstract algebra or topology, here are some other examples of adjunctions.

1. Free constructions: given any set you get a free group, free monoid, free ring, free vector space, etc.; each of these is a left adjoint. The corresponding right adjoint takes a group, a monoid, a ring, a vector space etc. and forgets the algebraic structure to return the underlying set.
2. Similarly, given a graph you get a free preorder or a free category, as we discussed in Section 3.2.3; each is a left adjoint. The corresponding right adjoint is the underlying graph of a preorder or of a category.
3. Discrete things: given any set you get a discrete preorder, discrete graph, discrete metric space, discrete category, discrete topological space; each of these is a left adjoint. The corresponding right adjoint is again underlying set.
4. Codiscrete things: given any set you get a codiscrete preorder, complete graph, codiscrete category, codiscrete topological space; each of these is a right adjoint. The corresponding left adjoint is the underlying set.
5. Given a group, you can quotient by its commutator subgroup to get an abelian group; this is a left adjoint. The right adjoint is the inclusion of abelian groups into groups.

3.4.3 Left and right pushforward functors, Σ and Π

Given $F\colon \mathcal{C} \to \mathcal{D}$, the data migration functor Δ_F turns $\mathcal{D}$-instances into $\mathcal{C}$-instances. This functor has both a left and a right adjoint:

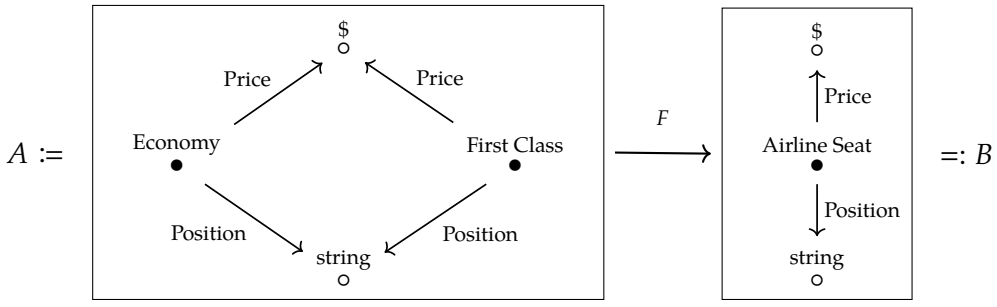


Using the names Σ and Π in this context is fairly standard in category theory. In the case of databases, they have the following helpful mnemonic:

Migration Functor	Pronounced	Reminiscent of	Database idea
Δ	Delta	Duplicate or destroy	Duplicate or destroy tables or columns
Σ	Sigma	Sum	Union (sum up) data
Π	Pi	Product	Pair[9] and query data

Just like we used Δ_F to pull back any discrete dynamical system along $F\colon \mathbf{Gr} \to \mathbf{DDS}$ and get a graph, the migration functors Σ_F and Π_F can be used to turn any graph into a discrete dynamical system. That is, given an instance $J\colon \mathbf{Gr} \to \mathbf{Set}$, we can get instances $\Sigma_F(J)$ and $\Pi_F(J)$ on **DDS**. This, however, is quite technical, and we leave it to the adventurous reader to compute an example, with help perhaps from [Spi14a], which explores the definitions of Σ and Π in detail. A less technical shortcut is simply to code up the computation in the open-source FQL software.

To get the basic idea across without getting mired in technical details, here we shall instead discuss a very simple example. Recall the schemas from Eq. (3.5). We can set up a functor between them, the one sending black dots to black dots and white dots to white dots:



With this functor F in hand, we can transform any B-instance into an A-instance using Δ_F. Whereas Δ was interesting in the case of turning discrete dynamical systems into graphs in Section 3.4.1, it is not very interesting in this case. Indeed, it will just copy—Δ for duplicate—the rows in Airline seat into both Economy and First Class.

Δ_F has two adjoints, Σ_F and Π_F, both of which transform any A-instance I into a B-instance. The functor Σ_F does what one would most expect from reading the names on each object: it will put into Airline Seat the union of Economy and First Class:

$$\Sigma_F(I)(\text{Airline Seat}) = I(\text{Economy}) \sqcup I(\text{First Class}).$$

The functor Π_F puts into Airline Seat the set of those pairs (e, f) where e is an Economy seat, f is a First Class seat, and e and f have the same price and position.

[9] This is more commonly called "join" by database programmers.

In this particular example, one imagines that there should be no such seats in a valid instance I, in which case $\Pi_F(I)$(Airline Seat) would be empty. But in other uses of these same schemas, Π_F can be a useful operation. For example, in the schema A replace the label 'Economy' by 'Rewards Program', and in B replace 'Airline Seat' by 'First Class Seats'. Then the operation Π_F finds those first class seats that are also rewards program seats. This operation is a kind of database query; querying is the operation that databases are built for.

The moral is that complex data migrations can be specified by constructing functors F between schemas and using the "induced" functors Δ_F, Σ_F, and Π_F. Indeed, in practice essentially all useful migrations can be built up from these. Hence the language of categories provides a framework for specifying and reasoning about data migrations.

3.4.4 Single set summaries of databases

To give a stronger idea of the flavor of Σ and Π, we consider another special case, namely where the target category $\mathcal{D}$ is equal to **1**; see Exercise 3.12. In this case, there is exactly one functor $\mathcal{C} \to \mathbf{1}$ for any $\mathcal{C}$; let's denote it

$$!\colon \mathcal{C} \to \mathbf{1}. \tag{3.75}$$

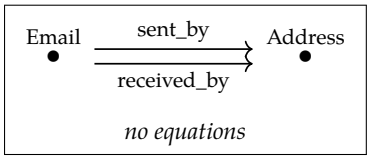
Exercise 3.76. Describe the functor $!\colon \mathcal{C} \to \mathbf{1}$ from Eq. (3.75). Where does it send each object? What about each morphism? ◊

We want to consider the data migration functors $\Sigma_!\colon \mathcal{C}$-**Inst** $\to$ **1-Inst** and $\Pi_!\colon \mathcal{C}$-**Inst** $\to$ **1-Inst**. In Example 3.53, we saw that an instance on **1** is the same thing as a set. So let's identify **1-Inst** with **Set**, and hence discuss

$$\Sigma_!\colon \mathcal{C}\text{-}\mathbf{Inst} \to \mathbf{Set} \qquad \text{and} \qquad \Pi_!\colon \mathcal{C}\text{-}\mathbf{Inst} \to \mathbf{Set}.$$

Given any schema $\mathcal{C}$ and instance $I\colon \mathcal{C} \to \mathbf{Set}$, we will get sets $\Sigma_!(I)$ and $\Pi_!(I)$. Thinking of these sets as database instances, each corresponds to a single one-column table—a controlled vocabulary—summarizing an entire database instance on the schema $\mathcal{C}$.

Consider the following schema

$\mathcal{G} :=$ 

(3.77)

Here's a sample instance $I\colon \mathcal{G} \to \mathbf{Set}$:

Email	**sent_by**	**received_by**
Em_1	Bob	Grace
Em_2	Grace	Pat
Em_3	Bob	Emmy
Em_4	Sue	Doug
Em_5	Doug	Sue
Em_6	Bob	Bob

Address
Bob
Doug
Emmy
Grace
Pat
Sue

Exercise 3.78. Note that $\mathcal{G}$ from Eq. (3.77) is isomorphic to the schema **Gr**. In Section 3.3.5 we saw that instances on **Gr** are graphs. Draw the above instance I as a graph. ◊

Now we have a unique functor $!\colon \mathcal{G} \to \mathbf{1}$, and we want to say what $\Sigma_!(I)$ and $\Pi_!(I)$ give us as single-set summaries. First, $\Sigma_!(I)$ tells us all the emailing groups—the "connected components"—in I:

1
Bob-Grace-Pat-Emmy
Sue-Doug

This form of summary, involving identifying entries into common groups, or quotients, is typical of Σ-operations.

The functor $\Pi_!(I)$ lists the emails from I which were sent from a person to her- or him-self.

1
Em_6

This is again a sort of query, selecting the entries that fit the criterion of self-to-self emails. Again, this is typical of Π-operations.

Where do these facts—that $\Pi_!$ and $\Sigma_!$ act the way we said—come from? Everything follows from the definition of adjoint functors (3.70): indeed we hope this, together with the examples given in Example 3.74, give the reader some idea of how general and useful adjunctions are, both in mathematics and in database theory.

One more point: while we will not spell out the details, we note that these operations are also examples of constructions known as colimits and limits in **Set**. We end this chapter with bonus material, exploring these key category theoretic constructions. The reader should keep in mind that, in general and not just for functors to **1**, Σ-operations are built from colimits in **Set**, and Π-operations are built from limits in **Set**.

3.5 Bonus: An introduction to limits and colimits

What do products of sets, the results of $\Pi_!$-operations on database instances, and meets in a preorder all have in common? The answer, as we shall see, is that they are all examples of limits. Similarly, disjoint unions of sets, the results of $\Sigma_!$-operations on database instances, and joins in a preorder are all colimits. Let's begin with limits.

Recall that $\Pi_!$ takes a database instance $I\colon \mathcal{C} \to \mathbf{Set}$ and turns it into a set $\Pi_!(I)$. More generally, a limit turns a functor $F\colon \mathcal{C} \to \mathcal{D}$ into an object of $\mathcal{D}$.

3.5.1 Terminal objects and products

Terminal objects and products are each a sort of limit. Let's discuss them in turn.

Terminal objects. The most basic limit is a terminal object.

Definition 3.79. Let $\mathcal{C}$ be a category. Then an object Z in $\mathcal{C}$ is a *terminal object* if, for each object C of $\mathcal{C}$, there exists a unique morphism $!\colon C \to Z$.

Since this *unique* morphism exists *for all* objects in $\mathcal{C}$, we say that terminal objects have a *universal property*.

Example 3.80. In **Set**, any set with exactly one element is a terminal object. Why? Consider some such set $\{\bullet\}$. Then for any other set C we need to check that there is exactly one function $!\colon C \to \{\bullet\}$. This unique function is the one that does the only thing that can be done: it maps each element $c \in C$ to the element $\bullet \in \{\bullet\}$.

Exercise 3.81. Let $(P, \leq)$ be a preorder, let $z \in P$ be an element, and let $\mathcal{P}$ be the corresponding category (see Section 3.2.3). Show that z is a terminal object in $\mathcal{P}$ if and only if it is a *top element* in P: that is, if and only if for all $c \in P$ we have $c \leq z$. ◊

Exercise 3.82. Name a terminal object in the category **Cat**. (Hint: recall Exercise 3.76.) ◊

Exercise 3.83. Not every category has a terminal object. Find one that doesn't. ◊

Proposition 3.84. All terminal objects in a category $\mathcal{C}$ are isomorphic.

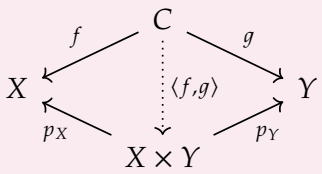
Proof. This is a simple, but powerful standard argument. Suppose Z and Z' are both terminal objects in some category $\mathcal{C}$. Then there exist (unique) maps $a\colon Z \to Z'$ and $b\colon Z' \to Z$. Composing these, we get a map $a \fatsemi b\colon Z \to Z$. Now since Z is terminal, this map $Z \to Z$ must be unique. But id_Z is also such a map. So we must have $a \fatsemi b = \mathrm{id}_Z$. Similarly, we find that $b \fatsemi a = \mathrm{id}_{Z'}$. Thus a is an isomorphism, with inverse b. □

Remark 3.85 ("The limit" vs. "a limit"). Not only are all terminal objects isomorphic, there is a unique isomorphism between any two. We hence say "terminal objects are unique up to unique isomorphism." To a category theorist, this is very nearly the same thing as saying "all terminal objects are equal." Thus we often abuse terminology and talk of 'the' terminal object, rather than "a" terminal object. We will do the same for any sort of limit or colimit, e.g. speak of "the product" of two sets, rather than "a product." We saw a similar phenomenon in Definition 1.81.

Products. Products are slightly more complicated to formalize than terminal objects, but they are familiar in practice.

Definition 3.86. Let $\mathcal{C}$ be a category, and let X, Y be objects in $\mathcal{C}$. A *product* of X and Y is an object, denoted $X \times Y$, together with morphisms $p_X\colon X \times Y \to X$ and $p_Y\colon X \times Y \to Y$ such that for all objects C together with morphisms $f\colon C \to X$ and $g\colon C \to Y$, there exists a unique morphism $C \to X \times Y$, denoted $\langle f, g \rangle$, for which the following diagram

commutes:



We will try to bring this down to earth in Example 3.87. Before we do, note that $X \times Y$ is an object equipped with morphisms to X and Y. Roughly speaking, it is like "the best object-equipped-with-morphisms-to-X-and-Y" in all of $\mathcal{C}$, in the sense that any other object-equipped-with-morphisms-to-X-and-Y maps to it uniquely. This is called a *universal property*. It's customary to use a dotted line to indicate the unique morphism that exists because of some universal property.

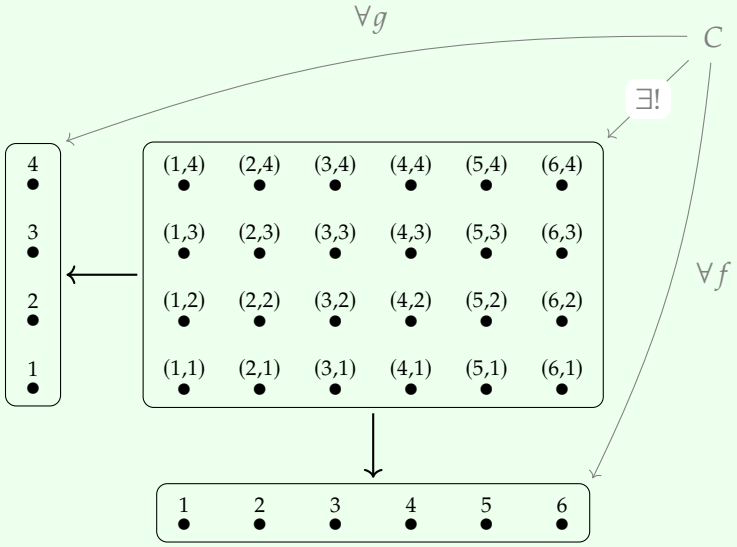
Example 3.87. In **Set**, a product of two sets X and Y is their usual cartesian product

$$X \times Y \coloneqq \{(x, y) \mid x \in X, y \in Y\},$$

which comes with two projections $p_X \colon X \times Y \to X$ and $p_Y \colon X \times Y \to Y$, given by $p_X(x, y) \coloneqq x$ and $p_Y(x, y) \coloneqq y$.

Given any set C with functions $f \colon C \to X$ and $g \colon C \to Y$, the unique map from C to $X \times Y$ such that the required diagram commutes is given by $\langle f, g \rangle(c) \coloneqq (f(c), g(c))$.

Here is a picture of the product $\underline{6} \times \underline{4}$ of sets $\underline{6}$ and $\underline{4}$.



Exercise 3.88. Let $(P, \leq)$ be a preorder, let $x, y \in P$ be elements, and let $\mathcal{P}$ be the corresponding category. Show that the product $x \times y$ in $\mathcal{P}$ agrees with their meet $x \wedge y$ in P. ◊

Example 3.89. Given two categories $\mathcal{C}$ and $\mathcal{D}$, their product $\mathcal{C} \times \mathcal{D}$ may be given as follows. The objects of this category are pairs (c, d), where c is an object of $\mathcal{C}$ and d is an object of $\mathcal{D}$. Similarly, morphisms $(c, d) \to (c', d')$ are pairs (f, g) where $f : c \to c'$ is a morphism in $\mathcal{C}$ and $g : d \to d'$ is a morphism in $\mathcal{D}$. Composition of morphisms is simply given by composing each entry in the pair separately, so $(f, g) \fatsemi (f', g') = (f \fatsemi f', g \fatsemi g')$.

Exercise 3.90.

1. What are the identity morphisms in a product category $\mathcal{C} \times \mathcal{D}$?
2. Why is composition in a product category associative?
3. What is the product category $\mathbf{1} \times \mathbf{2}$?
4. What is the product category $\mathcal{P} \times \mathcal{Q}$ when P and Q are preorders and $\mathcal{P}$ and $\mathcal{Q}$ are the corresponding categories? ◊

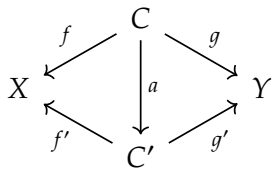
These two constructions, terminal objects and products, are subsumed by the notion of limit.

3.5.2 Limits

We'll get a little abstract. Consider the definition of product. This says that given any pair of maps $X \xleftarrow{f} C \xrightarrow{g} Y$, there exists a unique map $C \to X \times Y$ such that certain diagrams commute. This has the flavor of being terminal—there is a unique map to $X \times Y$—but it seems a bit more complicated. How are the two ideas related?

It turns out that products *are* terminal objects, but of a different category, which we'll call $\mathbf{Cone}(X, Y)$, *the category of cones over X and Y in $\mathcal{C}$*. We will see in Exercise 3.91 that $X \xleftarrow{p_X} X \times Y \xrightarrow{p_Y} Y$ is a terminal object in $\mathbf{Cone}(X, Y)$.

An object of $\mathbf{Cone}(X, Y)$ is simply a pair of maps $X \xleftarrow{f} C \xrightarrow{g} Y$. A morphism from $X \xleftarrow{f} C \xrightarrow{g} Y$ to $X \xleftarrow{f'} C' \xrightarrow{g'} Y$ in $\mathbf{Cone}(X, Y)$ is a morphism $a : C \to C'$ in $\mathcal{C}$ such that the following diagram commutes:



Exercise 3.91. Check that a product $X \xleftarrow{p_X} X \times Y \xrightarrow{p_Y} Y$ is exactly the same as a terminal object in $\mathbf{Cone}(X, Y)$. ◊

We're now ready for the abstract definition. Don't worry if the details are unclear; the main point is that it is possible to unify terminal objects, maximal elements, and meets, products of sets, preorders, and categories, and many other familiar friends under the scope of a single definition. In fact, they're all just terminal objects in different categories.

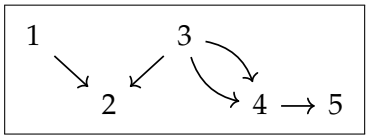
Recall from Definition 3.51 that formally speaking, a diagram in $\mathcal{C}$ is just a functor $D : \mathcal{J} \to \mathcal{C}$. Here $\mathcal{J}$ is called the *indexing category* of the diagram D.

Definition 3.92. Let $D : \mathcal{J} \to \mathcal{C}$ be a diagram. A *cone* (C, c_*) over D consists of
(i) an object $C \in \mathcal{C}$;
(ii) for each object $j \in \mathcal{J}$, a morphism $c_j : C \to D(j)$.
To be a cone, these must satisfy the following property:
(a) for each $f : j \to k$ in $\mathcal{J}$, we have $c_k = c_j \fatsemi D(f)$.

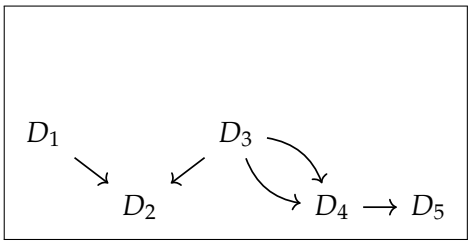
A *morphism of cones* $(C, c_*) \to (C', c'_*)$ is a morphism $a : C \to C'$ in $\mathcal{C}$ such that for all $j \in \mathcal{J}$ we have $c_j = a \fatsemi c'_j$. Cones over D, and their morphisms, form a category $\mathbf{Cone}(D)$.

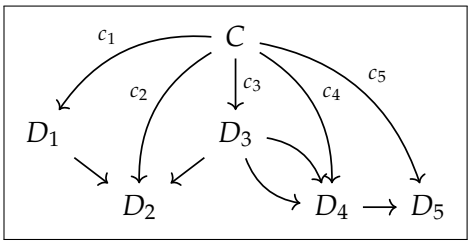
The *limit* of D, denoted $\lim(D)$, is the terminal object in the category $\mathbf{Cone}(D)$. Say it is the cone $\lim(D) = (C, c_*)$; we refer to C as the *limit object* and the map c_j for any $j \in \mathcal{J}$ as the j^{th} *projection map*.

For visualization purposes, if $\mathcal{J}$ is the free category on the graph



with five objects and five non-identity morphisms, then we may draw a diagram $D : \mathcal{J} \to \mathcal{C}$ inside $\mathcal{C}$ as on the left below, and a cone on it as on the right:





Here, any two parallel paths that start at C are considered the same. Note that both these diagrams depict a collection of objects and morphisms inside the category $\mathcal{C}$.

Example 3.93. Terminal objects are limits where the indexing category is empty, $\mathcal{J} = \varnothing$.

Example 3.94. Products are limits where the indexing category consists of two objects v, w and no arrows, $\mathcal{J} =$ 

.

3.5.3 Finite limits in Set

Recall that this discussion was inspired by wanting to understand Π-operations, and in particular $\Pi_!$. We can now see that a database instance $I : \mathcal{C} \to \mathbf{Set}$ is a diagram in

Set. The functor $\Pi_!$ takes the limit of this diagram. In this subsection we give a formula describing the result. This captures *all finite limits in* **Set**.

In database theory, we work with categories $\mathcal{C}$ that are presented by a finite graph plus equations. We won't explain the details, but it's in fact enough just to work with the graph part: as far as limits are concerned, the equations in $\mathcal{C}$ don't matter. For consistency with the rest of this section, let's denote the database schema by $\mathcal{J}$ instead of $\mathcal{C}$.

Theorem 3.95. Let $\mathcal{J}$ be a category presented by the finite graph (V, A, s, t) together with some equations, and let $D\colon \mathcal{J} \to \mathbf{Set}$ be a set-valued functor. Write $V = \{v_1, \ldots, v_n\}$. The set

$$\lim_{\mathcal{J}} D := \big\{(d_1, \ldots, d_n) \mid d_i \in D(v_i) \text{ for all } 1 \le i \le n \text{ and } \\ \text{for all } a\colon v_i \to v_j \in A, \text{ we have } D(a)(d_i) = d_j\big\}.$$

together with the projection maps $p_i\colon (\lim_{\mathcal{J}} D) \to D(v_i)$ given by $p_i(d_1, \ldots, d_n) := d_i$, is a limit of D.

Example 3.96. If J is the empty graph $\boxed{}$, then $n = 0$: there are no vertices. There is exactly one empty tuple $(\,)$, which vacuously satisfies the properties, so we've constructed the limit as the singleton set $\{(\,)\}$ consisting of just the empty tuple. Thus the limit of the empty diagram, i.e. the terminal object in **Set** is the singleton set. See Remark 3.85.

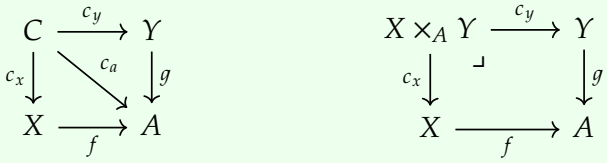
Exercise 3.97. Show that the limit formula in Theorem 3.95 works for products. See Example 3.94. ◊

Exercise 3.98. If $D\colon \mathbf{1} \to \mathbf{Set}$ is a functor, what is the limit of D? Compute it using Theorem 3.95, and check your answer against Definition 3.92. ◊

Pullbacks. In particular, the condition that the limit of $D\colon \mathcal{J} \to \mathbf{Set}$ selects tuples $(d_1, \ldots, d_n)$ such that $D(a)(d_i) = d_j$ for each morphism $a\colon i \to j$ in $\mathcal{J}$ allows us to use limits to select data that satisfies certain equations or constraints. This is what allows us to express queries in terms of limits. Here is an example.

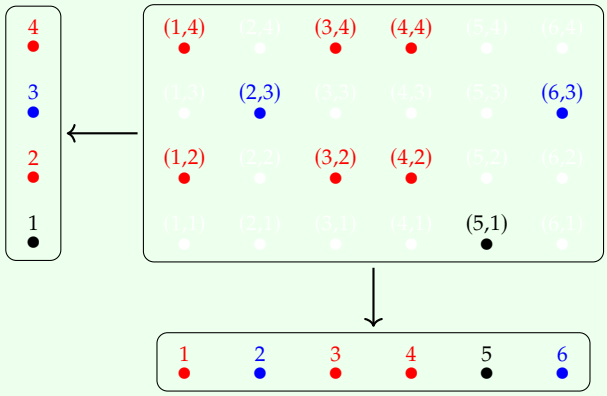
Example 3.99. If J is presented by the *cospan* graph $\boxed{\bullet^{x} \xrightarrow{f} \bullet^{a} \xleftarrow{g} \bullet^{y}}$, then its limit is known as a *pullback*. Given the diagram $X \xrightarrow{f} A \xleftarrow{g} Y$, the pullback is the cone shown

on the left below:



The fact that the diagram commutes means that the diagonal arrow c_a is in some sense superfluous, so one generally denotes pullbacks by dropping the diagonal arrow, naming the cone point $X \times_A Y$, and adding the ⌟ symbol, as shown to the right above.

Here is a picture to help us unpack the definition in **Set**. We take $X = \underline{6}$, $Y = \underline{4}$, and A to be the set of colors {red, blue, black}.



The functions $f : \underline{6} \to A$ and $g : \underline{4} \to A$ are expressed in the coloring of the dots: for example, $g(2) = g(4) =$ red, while $f(5) =$ black. The pullback selects pairs $(i, j) \in \underline{6} \times \underline{4}$ such that $f(i)$ and $g(j)$ have the same color.

Remark 3.100. As mentioned following Definition 3.68, this definition of pullback is not to be confused with the pullback of a set-valued functor along a functor; they are for now best thought of as different concepts which accidentally have the same name. Due to the power of the primordial ooze, however, the pullback along a functor is a special case of pullback as the limit of a cospan: it can be understood as the pullback of a certain cospan in **Cat**. To unpack this, however, requires the notions of category of elements and discrete opfibration; ask your friendly neighborhood category theorist.

3.5.4 A brief note on colimits

Just like upper bounds have a dual concept—namely that of lower bounds—so limits have a dual concept: colimits. To expose the reader to this concept, we provide a succinct definition of these using opposite categories and opposite functors. The point, however, is just exposure; we will return to explore colimits in detail in Chapter 6.

Exercise 3.101. Recall from Example 3.27 that every category $\mathcal{C}$ has an opposite $\mathcal{C}^{\mathrm{op}}$. Let $F : \mathcal{C} \to \mathcal{D}$ be a functor. How should we define its opposite, $F^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$? That

is, how should F^{op} act on objects, and how should it act on morphisms? ◊

Definition 3.102. Given a category $\mathcal{C}$ we say that a *cocone* in $\mathcal{C}$ is a cone in $\mathcal{C}^{\mathrm{op}}$.

Given a diagram $D : \mathcal{J} \to \mathcal{C}$, we may take the limit of the functor $D^{\mathrm{op}} : \mathcal{J}^{\mathrm{op}} \to \mathcal{C}^{\mathrm{op}}$. This is a cone in $\mathcal{C}^{\mathrm{op}}$, and so by definition a cocone in $\mathcal{C}$. The *colimit* of D is this cocone.

Definition 3.102 is like a compressed file: useful for transmitting quickly, but completely useless for working with, unless you can successfully unpack it. We will unpack it later in Chapter 6 when we discuss electric circuits.

3.6 Summary and further reading

Congratulations on making it through one of the longest chapters in the book! We apologize for the length, but this chapter had a lot of work to do. Namely it introduced the "big three" of category theory—categories, functors, and natural transformations—as well as discussed adjunctions, limits, and very briefly colimits.

That's really quite a bit of material. For more on all these subjects, one can consult any standard book on category theory, of which there are many. The bible (old, important, seminal, and requires a priest to explain it) is [Mac98]; another thorough introduction is [Bor94]; a logical perspective is given in [Awo10]; a computer science perspective is given in [BW90] and [Pie91] and [Wal92]; math students should probably read [Lei14] or [Rie17] or [Gra18]; a general audience might start with [Spi14a].

We presented categories from a database perspective, because data is pretty ubiquitous in our world. A database schema—i.e. a system of interlocking tables—can be captured by a category $\mathcal{C}$, and filling it with data corresponds to a functor $\mathcal{C} \to \mathbf{Set}$. Here **Set** is the category of sets, perhaps the most important category to mathematicians.

The perspective of using category theory to model databases has been rediscovered several times. It seems to have first been discussed by various authors around the mid-90's [IP94; CD95; PS95; TG96]. Bob Rosebrugh and collaborators took it much further in a series of papers including [FGR03; JR02; RW92]. Most of these authors tend to focus on sketches, which are more expressive categories. Spivak rediscovered the idea again quite a bit later, but focused on categories rather than sketches, so as to have all three data migration functors Δ, Σ, Π; see [Spi12; SW15b]. The version of this story presented in the chapter, including the white and black nodes in schemas, is part of a larger theory of algebraic databases, where a programming language such as Java or Haskell is attached to a database. The technical details are worked out in [Sch+17], and its use in database integration projects can be found in [SW15a; Wis+15].

Before we leave this chapter, we want to emphasize two things: coherence conditions and universal constructions.

Coherence conditions. In the definitions of category, functor, and natural transformations, we have data (indexed by (i)) that is required to satisfy certain properties (indexed

by (a)). Indeed, for categories it was about associativity and unitality of composition, for functors it was about respecting composition and identities, and for natural transformations it was the naturality condition. These conditions are often called *coherence conditions*: we want the various structures to cohere, to work well together, rather than to flop around unattached.

Understanding why these particular structures and coherence conditions are "the right ones" is more science than mathematics: we empirically observe that certain combinations result in ideas that are both widely applicable and also strongly compositional. That is, we become satisfied with coherence conditions when they result in beautiful mathematics down the road.

Universal constructions. Universal constructions are one of the most important themes of category theory. Roughly speaking, one gives some specified shape in a category and says "find me the best solution!" And category theory comes back and says "do you want me to approximate from the left or the right (colimit or limit)?" You respond, and either there is a best solution or there is not. If there is, it's called the (co)limit; if there's not we say "the (co)limit does not exist."

Even data migration fits this form. We say "find me the closest thing in $\mathcal{D}$ that matches my $\mathcal{C}$-instance using my functor $F\colon \mathcal{C} \to \mathcal{D}$." In fact this approach—known as Kan extensions—subsumes the others. One of the two founders of category theory, Saunders Mac Lane, has a section in his book [Mac98] called "All concepts are Kan extensions," a big statement, no?

18.S097 Applied Category Theory

January IAP 2019