Chasing Databases: The Theoretical Evolution of Data Migration

Ben Connors

August 29, 2022

Contents

1 Introduction

Relational databases are omnipresent in our modern world, used for storing any sort of persistent, shared data. Relational databases are databases storing data in a tabular form, which is natural for much of the data produced. A relational database consists of tables, each table modeling a certain type of object or entity. Each table consists of rows, each corresponding to a specific entity or object, and columns, each describing some attribute of that entity. The term "relational" comes from the recognition that these entities seldom exist in isolation: we often have many *relations* between objects of different types, which are precisely captured in by the concept of relations in relational databases.

We often wish to convert data in databases into different forms for different applications, like removing private information or combine multiple tables for easy access. Relational algebra provides the customary operations and semantics for accomplishing this. Though studied earlier in pure mathematics, relational algebra was first applied to databases in 1970 in [Cod70]. However, relational algebra is rarely formulated precisely, and even more seldom in practice: typical courses and texts on the subject rarely give concrete examples of the relational operations and instead appeal solely to examples. This extends to many real-world situations, where relational operations are rarely formulated formally: the desired operations are constructed using intuition and verified by testing them (not always on test databases). The fundamental operations of relational algebra described in [Cod70] are still in use today, with little change: they have proven to be sufficiently general to be able to do nearly any operation we require, and sufficiently simple to be implemented efficiently.

Category theory has found numerous applications throughout pure mathematics, especially in formalizing and generalizing abstract concepts. Applications to other areas, such as computer science, have been increasing in recent years. Many of the applications to that field have come through the angle of type theory and proof assistants: by modeling real-world situations and concepts using precise mathematical language (often category theory), we can use proof assistants and theorem provers to mathematically prove desired attributes and the nonexistence of many classes of errors.

This report will outline the general picture of the work done by Spivak et al. in the past decade to model relational databases and relational algebra using category theory. This application has been explored in the past by a number of other authors ([Spi12a] gives a number of such papers in its introduction), but there was a dearth of publications on the subject in the 2000s. We will explore each of the three major approaches taken by Spivak et al. in their historical order, which is also in order of increasing complexity. First we model databases very naturally using ordinary category theory (Section 2, based on [Spi12a]), which provides the foundation and the intuition for all further developments. Next, we explore the concept of "typed databases" (Section 3, based on [SW15]) to fix some of the problems encountered with the first approach. Finally, we sketch the basic ideas for modeling databases using the concept of algebraic profunctors (Section 4, based on [SSVW16]), which maintains the advantages of typed databases while avoiding certain technical constraints on the theory. The intention of this report is to collect the necessary background information, provide detailed examples and computations, and present the evolution of the theory in a unified way, justifying each new development.

We will assume some basic familiarity with category theory (categories, functors, natural transformations, (co)limts, adjunctions); this can be obtained from the first few chapters of any introductory text on category theory, such as [Spi14] or [Rie14] (for a more abstract perspective). Chapter 3 in [FS18] also gives a gentle introduction to category theory using databases (using the approach

outlined in Section 2 of this report) suitable for those wholly unfamiliar with the subject.

We do not assume much familiarity with relational databases, as the portions of the theory we will require are relatively intuitive and will be introduced with examples. The interested reader can consult any introductory text on the subject, e.g. [CB15], for a more detailed introduction.

This report was prepared at the University of Toronto under the supervision of Dr. Yun William Yu to partially fulfill the requirements of an MSc degree in mathematics.

2 Databases via Functor Categories

2.1 Schemas as Categories

In this section, we outline the constructions given in [Spi12a].

Consider the data tables in Table 1; these are an example of a database *instance*, a database that contains actual data. Each database instance is an instance of a *schema*: something that describes the format of our database. One presentation of a database schema is as an entity-relationship (ER) diagram; a simplified version of this for Table 1 is given in Figure 1.

ID	First	Last	Department	Salary
E01	Francis	Poulenc	D01	$15,000
E02	George	Dyson	D02	$14,500
E03	Charles	Gounod	D01	$16,000
E04	Georges	Bizet	D01	$24,000

ID	Location	Manager
D01	Paris	E04
D02	London	E02

Table 1: An example with two tables: one containing employees and another containing departments.

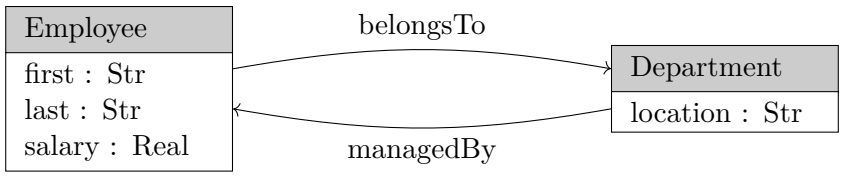


Figure 1: The schema for the database instance in Table 1.

The schema has two entities, "Employee" and "Department", and one relationship in each direction: each employee "belongsTo" a department, and each department is "managedBy" an employee. In our database instance these two relationships are represented by a column in each table, but for designing database schemas it's customary to represent them as arrows: this is because certain relationships, like many-to-many relationships (e.g. matching people to all of the books they've ever borrowed at a library: each person can borrow more than one book, and each book can be borrowe for Table 1d by more than one person), will require their own table to represent in most relational database softwares. Under each header is a list of attributes that the entity has, corresponding to the columns in our example tables. We also indicate the type of each attribute (e.g. an employee's first name is a **Str**ing of characters). We omit the unique ID field in each table that identifies employees and departments, leaving it implicit in the schema.

We may want to impose certain "integrity constraints" on our database: doing it at the database level can save writing and debugging code at the application level, especially if there are many different applications that are using the database. In this case, we want to ensure that the manager of a department actually works for that department: that is, if `dept` is a Department then (using the notation of the tables) `dept.Manager.Department` must be `dept`, the same Department we started with. We represent that in our diagram as a path constraint, written here at the bottom of the diagram; note that we follow the convention of graph theory here, in that the leftmost edge

in the list "managedBy; belongsTo" is the *first* edge in the path and the rightmost edge is the last (i.e. the opposite order of writing function composition in mathematics).

This format of a schema, as an ER diagram, gives us a natural way of interpreting a database schema in terms of category theory: an ER diagram is nothing but a graph with fancy labels and a list of path equivalences! We can easily turn such a graph into a category:

Definition 2.1. Let $G = (V, E, s, d)$ be a (directed, multi-) graph with a collection of vertices V, a collection of edges E, a function $s : E \to V$ which gives the source of an edge, and a function $d : E \to V$ which gives the destination of an edge. The *free category on* G, denoted FG, is the category with

- Objects the vertices of G, $\mathsf{Ob}\, FG := V$; and
- Basic morphisms $a \to b$ the collection of edges with source a and destination b; and
- Morphisms freely generated by the basic morphisms along with freely-adjoined identity morphisms, modulo the equivalence relation generated by associativity and composition by the identities. ⧫

This definition is somewhat annoying to write formally, but the intuition is simple: the objects are the vertices and the morphisms $a \to b$ in FG are all of the distinct paths from a to b. By the axioms of a category we must also add an identity morphism $a \to a$ for each vertex a. We also want to include the path equivalences in our construction:

Definition 2.2. Given a graph G with a set of path equivalences R, the *category generated by* G is the quotient category $FG/\sim$ where $\sim$ is the equivalence relation generated by the path equivalences R. ⧫

Again, the intuition is simple: we identify any two paths in our category that are equivalent, and also identify them in every other path (e.g. if we define the paths $A; B$ and $C; D; E$ to be equivalent, we must identify them and also all other paths that contain them like $F; G; A; B; H$ and $F; G; C; D; E; H$).

Now that we have a way of turning a graph into a category, we've reduced the problem to converting ER diagrams into ordinary graphs. The remaining question is how to interpret the fancy labels.

Our first approach will be to add a new vertex for each attribute; for our running example, this is in Figure 2.

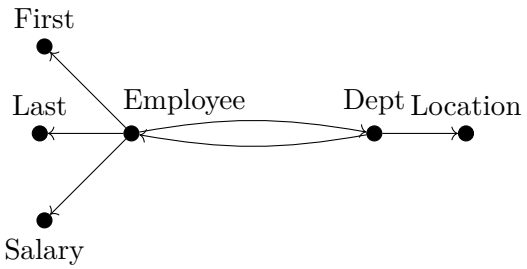


Figure 2: One graph corresponding to the schema in Figure 1.

This includes includes most of the information from our ER diagram; however, we're still missing two things. First, we've ignored the information on the *type* of each attribute—our graph says nothing about an employee's first name being a string or their salary being a real number. Second, we don't really have any way of distinguishing between entities and their attributes: both are vertices. Of course, we could define an attribute as a vertex with no edges leaving it, but eventually we will need to separate the two more explicitly. We will ignore these problems for now, deferring them both to our discussion of typed databases in Section 3.

Now that we have a way of converting a database schema from an ER diagram into a graph with some path equivalences, we can think of a database schema as a category (the category generated by its ER diagram interpreted as a graph). As is typical in category theory, it's more convenient to work in greater generality so we make the following definition:

Definition 2.3. We call any category $\mathcal{C}$ a *database schema* or just a *schema.* ⧫

In summary:

Procedure 2.4. To convert an ER diagram into a category:

1. Create a graph G with
   - A vertex for each entity and each attribute of that entry (excluding ID attributes and foreign keys);
   - An edge from A to B if:
     - B is an attribute of A; or
     - A has a relation to B
2. Form the free category on G, denoted FG
3. Take the quotient of FG by any path-equivalences given in the ER diagram. ⧫

2.2 Instances as Functors

The whole reason we care about databases is to store data: if we want to view databases through a categorical lense, we'd better figure out some way of representing database instances.

Consider our running example, begun in Table 1. We have a category (call it $\mathcal{C}$) that represents our schema, which is the category generated by the graph in Figure 2, modulo our path equivalences. In order to represent the data in Table 1, we need to give a list of employees and a list of departments. Equivalently, we need to give a list of the employee IDs and department IDs and a way to map these to their various attributes. This sounds a lot like a functor out of $\mathcal{C}$!

Definition 2.5. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. The collection of $\mathcal{D}$*-valued database instances on* $\mathcal{C}$ is the collection of functors $\mathcal{C} \to \mathcal{D}$. When the category $\mathcal{D}$ is implicit, this will be denoted by $\mathcal{C}$**-Inst**.⧫

For our purposes, $\mathcal{D} := \mathsf{Set}$: every database instance will be Set-valued. We may also want to work with $\mathcal{D} := \mathsf{FinSet}$ (finite sets), but for theoretical purposes we will want to assume that $\mathcal{D}$ is complete and cocomplete.

Let's figure out what this gets us: suppose we have a Set-valued instance on $\mathcal{C}$, i.e. a functor $I : \mathcal{C} \to \mathsf{Set}$. For each object $x \in \mathcal{C}$, we get a set $I(x)$. For each morphism $f : x \to y$ in $\mathcal{C}$, we get a function $If : Ix \to Iy$, and by functoriality this mapping must respect composition in $\mathcal{C}$.

In terms of our example, we get:

- A set of employee IDs $I(\text{Employee})$ and likewise for all other vertices;
- A function $I(\text{Employee}) \to I(\text{First})$ mapping an employee to their first name;
- So on for the other edges...

By functoriality, it also respects our path equivalences: that is, if $m : I(\text{Department}) \to I(\text{Employee})$ is the "managedBy" function mapping a department to its manager and $d : I(\text{Employee}) \to I(\text{Department})$ is the "belongsTo" function mapping an employee to their department, we must have $d \circ m = \text{id}$.

Thus, a functor $\mathcal{C} \to \mathcal{D}$ for some appropriate category $\mathcal{D}$ is, ignoring data types, exactly what we want: giving us a set for each entity and attribute and functions between them based on the edges connecting them is precisely giving the data that will be in our table, and functoriality is precisely ensuring that these assignments respect our path equivalences.

The collection of functors $\mathcal{C} \to \mathcal{D}$ has a natural notion of morphism: namely, natural transformation. What does a natural transformation between database instances mean? Given instances $I, I' : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : I \Rightarrow I'$, for each object $A \in \mathcal{C}$ we get a function $\alpha_A : I(A) \to I(A')$ with naturality saying that these functions respect our constraints. Thus, these morphisms give a way of relating two database instances. For example, we could use these morphisms to investigate how databases change over time. Thus, it seems reasonable to include this extra structure in our definition of $\mathcal{C}$-**Inst**:

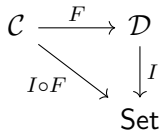
Definition 2.6. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. The *category of $\mathcal{D}$-valued database instances on $\mathcal{C}$* is the functor category $\mathcal{D}^{\mathcal{C}}$. When the category $\mathcal{D}$ is implicit, this will be denoted by $\mathcal{C}$-**Inst**. ♦

2.3 Data Migration

The real value in this perspective on databases comes with data migration, moving data between two schemas.

In relational algebra, we usually consider three types of operations. Projections "project" along certain attributes, removing unnecessary data from our schema, often for security or performance reasons. The other two, union and joins, combine multiple tables together in "dual" ways: a union is essentially the largest way of combining tables subject to imposed constraints, and a join is the smallest. We will show that all three operations appear very naturally in this categorical view.

Suppose we have two schemas (categories) $\mathcal{C}$ and $\mathcal{D}$. The natural way of relating the two is using functors; suppose we have a functor $F : \mathcal{C} \to \mathcal{D}$. We now want to use this to migrate actual instances from schema $\mathcal{C}$ to schema $\mathcal{D}$. However, without more tools we can only take instances from $\mathcal{D}$ to $\mathcal{C}$: if we have an instance $I : \mathcal{D} \to \mathsf{Set}$, we can compose



to get an instance $I \circ F : \mathcal{C} \to \mathsf{Set}$. This gives us a functor $F^* : \mathsf{Set}^D \to \mathsf{Set}^{\mathcal{C}}$ via precomposition. This is one of the three fundamental data migration functors:

Definition 2.7. Let $\mathcal{C}, \mathcal{D}$ be schemas and $F : \mathcal{C} \to \mathcal{D}$ a functor. The *delta operator corresponding to* F is the precomposition functor $\Delta_F := F^* : \mathcal{D}$**-Inst** $\to \mathcal{C}$**-Inst**. ⧫

Trivially:

Definition 2.8. For all schemas $\mathcal{C}, \mathcal{D}$, functors $F : \mathcal{C} \to \mathcal{D}$, and value category $\mathcal{E}$, the functor $\delta_F : \mathcal{D}$**-Inst** $\to \mathcal{C}$**-Inst** exists. ⧫

This delta operator essentially creates projections. Suppose we consider only the employee table in Figure 1. We want to give salary data to someone so they can do some statistics, but we don't want them to know the names or anything else about our employees. Our source schema $\mathcal{C}$ and destination schema $\mathcal{D}$ are in Figure 3.

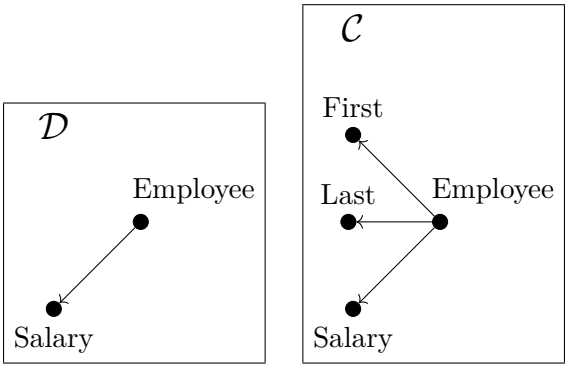


Figure 3: The destination schema (left) and source schema (right) our first migration.

We want to take a $\mathcal{C}$-instance (all data) and get a $\mathcal{D}$-instance (less data); since the delta operator flips the direction of a functor, we need to define a functor $F : \mathcal{D} \to \mathcal{C}$. We do so in the obvious way: we have a natural inclusion $\mathcal{D} \hookrightarrow \mathcal{C}$, so we define F to be this inclusion (i.e. we map Employee to Employee, Salary to Salary, and the arrow Employee $\to$ Salary to the arrow Employee $\to$ Salary). This gives us the functor $\Delta_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst**. Using our running example, the original instance and resulting instance are in Table 4.

ID	First	Last	Department	Salary
E01	Francis	Poulenc	D01	\$15,000
E02	George	Dyson	D02	\$14,500
E03	Charles	Gounod	D01	\$16,000
E04	Georges	Bizet	D01	\$24,000

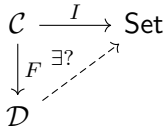
$\xmapsto{\Delta_F}$

ID	Salary
E01	\$15,000
E02	\$14,500
E03	\$16,000
E04	\$24,000

Figure 4: The result of our data migration using the delta functor $\Delta_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** for the schemas in Figure 3.

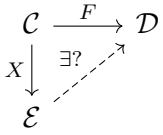
This gives a "proof by example" that taking the delta operator $\Delta_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** of a functor $F : \mathcal{D} \to \mathcal{C}$ allows us to represent any sort of projection operator: its extension to an actual proof is evident.

How do we go the other direction? Suppose we have $I : \mathcal{C} \to \mathsf{Set}$. We want:

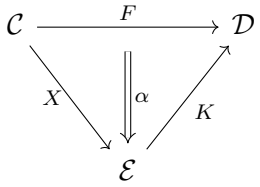


This amounts to trying to solve the given lifting problem. We can do this in many ways, but we have (up to) two universal ones called Kan extensions. These are explored in detail in most introductory category theory texts, e.g. Chapter 6 in [Rie14].

Definition 2.9. Let $F : \mathcal{C} \to \mathcal{D}$ and $X : \mathcal{C} \to \mathcal{E}$ be functors. Consider the lifting problem:



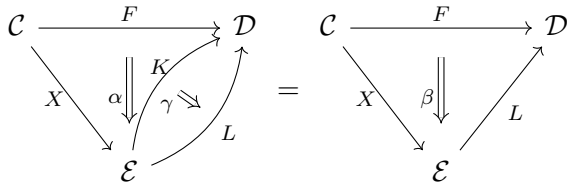
A left solution is a functor $K : \mathcal{E} \to \mathcal{D}$ and a natural transformation $\alpha : F \Rightarrow K \circ X$:



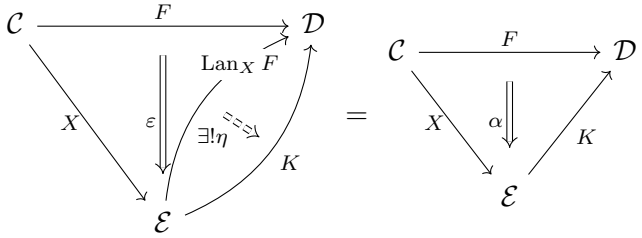
Note that the diagram need not commute.

We can form the collection of solutions into a category as follows:

- Objects are pairs $(K : \mathcal{E} \to \mathcal{D}, \alpha : F \Rightarrow K \circ X)$;
- Morphisms $(K, \alpha) \to (L, \beta)$ are natural transformations $\gamma : K \to L$ such that $(\gamma \cdot X) \circ \alpha = \beta$, i.e.:



The left Kan extension of F along X, denoted $\mathrm{Lan}_X F : \mathcal{E} \to \mathcal{D}$, is the initial object in this category (with a natural transformation $\varepsilon : F \Rightarrow \mathrm{Lan}_X F \circ X$), i.e. for any solution (K, α) we have a unique factorization $\alpha = \eta \circ \varepsilon$



Dually, the right Kan extension, $\mathrm{Ran}_X F$, is the terminal object in the category of right solutions (reverse the direction of the natural transformation), which is the left Kan extension when the categories $\mathcal{C}, \mathcal{D}, \mathcal{E}$ are replaced with their opposites, $\mathcal{C}^{\mathsf{op}}, \mathcal{D}^{\mathsf{op}}, \mathcal{E}^{\mathsf{op}}$. ⧫

Definition 2.10. Recall that if $\mathcal{C}$ is a category, the dual category denoted $\mathcal{C}^{\mathsf{op}}$ has the same objects as $\mathcal{C}$ but all morphisms and compositions are flipped, i.e. $\mathcal{C}^{\mathsf{op}}(x, y) := \mathcal{C}(y, x)$ and $f \circ^{\mathsf{op}} g := g \circ f$.⧫

This first definition of a Kan extension is sometimes called a "local Kan extension" and is the most general, since we only require a specific Kan extension $\mathrm{Lan}_X F$ to exist for a fixed functor F. This next definition is theoretically nicer and more convenient, but less general.

Definition 2.11. Let $X : \mathcal{C} \to \mathcal{E}$ be a functor and $\mathcal{D}$ a category. We have the induced functor given by precomposition:

$$\mathcal{D}^{\mathcal{C}} \xleftarrow[X^*]{} \mathcal{D}^{\mathcal{E}}$$

If this functor has a left adjoint, we call it the left Kan extension operation along X and denoted it Lan_X. Dually, if this functor has a right adjoint we call it the right Kan extension operation along X and denote it Ran_X:

$$\mathcal{D}^{\mathcal{C}} \underset{X^*}{\overset{\mathrm{Lan}_X}{\overset{\longleftarrow}{\underset{\longrightarrow}{\overset{\mathrm{Ran}_X}{\longrightarrow}}}}} \mathcal{D}^{\mathcal{E}} \qquad (\bot,\ X^* \bot) \tag{2.1}$$

⧫

The intuition is that the left Kan extension is the "smallest" left solution and the right Kan extension is the "largest" right solution.

This intuition is entirely correct in the case of posets, as will be explored in detail when we discuss profunctors in Appendix A: given a partial order $\leq$ on a set X, we can form a category PX with

- Objects the elements of X;
- The morphisms $PX(x, y)$ either empty if $x \not\leq y$ or containing a unique morphism if $x \leq y$.

If we have two poset categories PX and PY, a functor $F : PX \to PY$ is precisely a monotone increasing map between the underlying posets, $f : X \to Y$. A natural transformation between two functors $F, G : PX \to PY$ asserts that $F(x) \leq G(x)$ for all objects $x \in PX$ (naturality is degenerate since we have at most one morphism between two objects). If we consider the lifting problem:

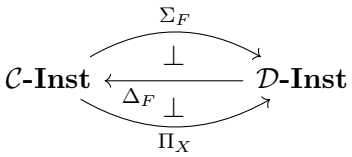
$$\begin{array}{ccc} PX & \xrightarrow{F} & PY \\ {\scriptstyle X}\downarrow & \overset{\exists?}{\nearrow} & \\ PZ & & \end{array}$$

a left solution $G : PZ \to PY$ comes with a natural transformation $\alpha : F \Rightarrow G \circ X$, i.e. $F(x) \leq (G \circ X)(x)$ for all $x \in PX$. The universal property of the left Kan extension means we have both a natural transformation $\beta : F \Rightarrow \mathrm{Lan}_X F \circ X$ and a natural transformation $\gamma : \mathrm{Lan}_X F \Rightarrow G$, so we have $F(x) \leq (\mathrm{Lan}_X F \circ X)(x)$ for all x and also $(\mathrm{Lan}_X F)(z) \leq G(z)$ for all $z \in PZ$, so $\mathrm{Lan}_X F$ is indeed the smallest left solution in a precise sense. Dually, $\mathrm{Ran}_X F$ is the largest right solution.

Returning to databases, given a functor $F : \mathcal{C} \to \mathcal{D}$ we can come up with two other functors $\mathrm{Lan}_F, \mathrm{Ran}_F : \mathcal{C}$-**Inst** $\to \mathcal{D}$-**Inst** that don't flip the direction of our functor. We give these special names:

Definition 2.12. Let $\mathcal{C}, \mathcal{D}$ be schemas and $F : \mathcal{C} \to \mathcal{D}$ a functor. The *sigma (union) operator corresponding to* F is the functor $\Sigma_F := \mathrm{Lan}_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** (when it exists). The *pi (join) operator corresponding to* F is the functor $\Pi_F := \mathrm{Ran}_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** (when it exists). ♦

Using these names, we can rewrite Equation 2.1 as:



In short, $\Sigma_F \dashv \Delta_F \dashv \Pi_F$. As suggested by the names in the definition, we will show that Σ_F corresponds to unions of tables while Π_F corresponds to joins.

Consider an example instance in Table 2. The schema is similar to our running example, except

ID	First	Last	Salary
E01	Francis	Poulenc	$15,000
E02	George	Dyson	$14,500
E03	Charles	Gounod	$16,000
E04	Georges	Bizet	$24,000

ID	First	Last	Position
F01	Georges	Bizet	Junior Intern
F02	Charles	Gounod	Associate Fax Guru
F03	Francis	Poulenc	Executive Vice-CEO

Table 2: An example with two tables: one containing salaries and another containing positions.

now we have one table with the employee's salary and another with their position. We want to combine these into one table with all of the information. Unfortunately there's no explicit connection between the IDs of the two tables, so we have to match data based on the employees' first and last names. In relational terminology, we want to *join* these two tables along the "First" and "Last" columns Our source and desination schema are in Figure 5.

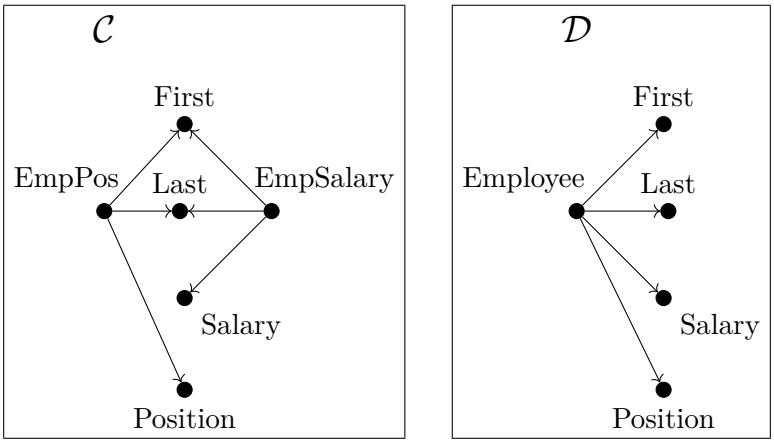


Figure 5: The source schema (left) and desination schema (right) for our join operation.

Our functor $G : \mathcal{C} \to \mathcal{D}$ will be the obvious one: we send EmpPos and EmpSalary to Employee and everything else to "itself": we map EmpPos $\to$ First and EmpSalary $\to$ First both to Employee $\to$ First and likewise for Second, and so on for the other two. For our instance in Table 2, we will compute its right Kan extension along G, i.e. its image under Π_G. We will make use of the following theorem which tells us how to compute Kan extensions as limits and colimits:

Theorem 2.13 (Theorem 6.2.1 in [Rie14]). Given $F : \mathcal{C} \to \mathcal{D}$ and $\gamma : \mathcal{C} \to \mathcal{E}$, if for every $D \in \mathcal{D}$ the colimit:

$$\mathrm{Lan}_F \gamma(D) := \mathrm{colim}(F \downarrow D \xrightarrow{\Pi^d} \mathcal{C} \xrightarrow{\gamma} \mathcal{E}) \tag{2.2}$$

exists, then they define the left Kan extension $\mathrm{Lan}_F \gamma : \mathcal{D} \to \mathcal{E}$. Likewise for the other:

$$\mathrm{Ran}_F \gamma(D) := \lim(D \downarrow F \xrightarrow{\Pi_d} \mathcal{C} \xrightarrow{\gamma} \mathcal{E})$$

defines $\mathrm{Ran}_F \gamma : \mathcal{D} \to \mathcal{E}$. ⧫

This gives us an existence result about our functors:

Corollary 2.14. Let $\mathcal{E}$ be a category, $\mathcal{C}, \mathcal{D}$ be schemas, and $F : \mathcal{C} \to \mathcal{D}$ a functor. If $\mathcal{C}$**-Inst** and $\mathcal{D}$**-Inst** are the categories of $\mathcal{E}$-valued $\mathcal{C}$- and $\mathcal{D}$-instances, respectively, then:

1. If $\mathcal{E}$ is cocomplete, the union functor $\Sigma_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** exists.
2. If $\mathcal{E}$ is complete, the join functor $\Pi_F : \mathcal{C}$**-Inst** $\to \mathcal{D}$**-Inst** exists. ⧫

We recall some definitions needed for the theorem:

Definition 2.15. Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{E} \to \mathcal{D}$ be functors. The *comma category of F and G*, denoted $F \downarrow G$, has:

- Objects the triples (C, E, h) for $C \in \mathsf{Ob}\,\mathcal{C}$, $E \in \mathsf{Ob}\,\mathcal{E}$, and $h : FC \to GE$ a morphism in $\mathcal{D}$;
- Morphisms $(C, E, h) \to (C', E', h')$ the pairs (f, g) for $f : C \to C'$ a morphism in $\mathcal{C}$ and $g : E \to E'$ a morphism in $\mathcal{E}$ such that the following commutes:

$$\begin{array}{ccc} FC & \xrightarrow{Ff} & FC' \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle h'} \\ GE & \xrightarrow{Gg} & GE' \end{array}$$

⧫

We have projection functors $\Pi_d : F \downarrow G \to \mathcal{C}$ given by $\Pi_d(C, E, h) := C$ and $\Pi_d(f, g) := f$ and likewise $\Pi^d : F \downarrow G \to \mathcal{E}$ for the second coordinate.

Thus, the functor on the right hand side of Equation 2.2 is explicitly:

$$\begin{gathered} D \downarrow F \xrightarrow{\Pi_d} \mathcal{C} \xrightarrow{\gamma} \mathsf{Set} \\ (X, h : D \to FX) \mapsto \gamma(X) \in \mathsf{Set} \\ (f : X \to X') \mapsto (\gamma(f) : \gamma(X) \to \gamma(X')) \end{gathered}$$

For our case, we will use the following formula giving a limit as an equalizer of products, which is easy to compute in Set:

Theorem 2.16 (Theorem 3.2.13 in [Rie14]). The limit of $\gamma : \mathcal{C} \to \mathsf{Set}$ may be computed as the equalizer:

$$\lim \gamma \longrightarrow \prod_{X \in \mathsf{Ob}\,\mathcal{C}} \gamma X \underset{d}{\overset{c}{\rightrightarrows}} \prod_{f \in \mathsf{Mor}\,\mathcal{C}} \gamma(\mathsf{cod}\, f)$$

where

$$\begin{aligned} c &:= \langle \pi_{\mathsf{cod}\, f} \rangle_{f \in \mathsf{Mor}\,\mathcal{C}} \\ d &:= \langle \gamma f \circ \pi_{\mathsf{dom}\, f} \rangle_{f \in \mathsf{Mor}\,\mathcal{C}} \end{aligned}$$

⧫

For the computation, we will use the schemas in Figure 6; the only change is the names of the objects.

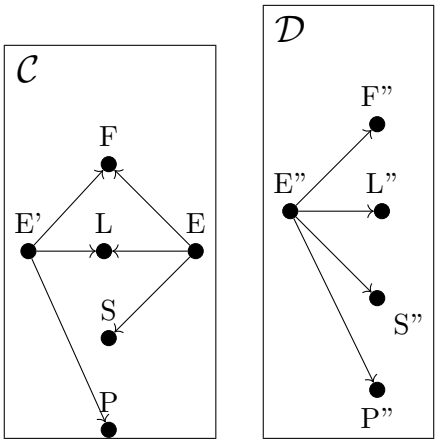


Figure 6: The revised source schema (left) and destination schema (right) for our join operation (c.f. Figure 5). All objects have been renamed for brevity but the schemas are otherwise identical.

Call our instance $I : \mathcal{C} \to \mathsf{Set}$. For each object of our *destination* schema $D \in \mathcal{D}$, we need to compute the limit of the functor:

$$D \downarrow G \xrightarrow{\Pi_d} \mathcal{C} \xrightarrow{I} \mathsf{Set}$$

which, by Theorem 2.16, will require us to compute the following two products:

$$\prod_{X \in \mathsf{Ob}(D\downarrow G)} I(X), \qquad \prod_{f \in \mathsf{Mor}(D\downarrow G)} I(\mathsf{cod}\, f)$$

We will only compute this for E'', the table of IDs in our destination schema; using the specific definition of the limit in Theorem 2.16, this will tell us all of the data. First, we must compute the slice category $E'' \downarrow G$, whose objects may be identified with the functions $E'' \to GX$ for $X \in \mathsf{Ob}\,\mathcal{C}$, our source category; these are:

$$\begin{aligned} \mathrm{id}_{E''} &: E'' \to GE = E'' \\ \mathrm{id}_{E''} &: E'' \to GE' = E'' \\ & E'' \to GF \\ & E'' \to GL \\ & E'' \to GP \\ & E'' \to GS \end{aligned}$$

so our first product is:

$$\prod_{X \in \mathsf{Ob}(D\downarrow G)} I(X) = I(E) \times I(E') \times I(F) \times I(L) \times I(P) \times I(S)$$

The morphisms in our category $X \to Y$ are the precisely the morphisms $X \to Y$ in $\mathcal{C}$; the commutativity condition is degenerate in our case since there is at most one morphism between any two

objects. These are:

$$\begin{array}{ll} E \to E & E' \to E' \\ E \to F & E' \to F \\ E \to L & E' \to L \\ E \to P & E' \to S \end{array}$$

so our second product has eight terms:

$$\prod_{f \in \mathsf{Mor}(D \downarrow G)} I(\mathsf{cod}\, f) = I(E) \times I(F) \times I(L) \times I(P) \times I(E') \times I(F) \times I(L) \times I(S)$$

The function c in Theorem 2.16 between the two is:

$$c(e, e', f, l, p, s') \mapsto (e, f, l, p, e', f, l, s)$$

and the function d is:

$$d(e, e', f, l, p, s) \mapsto (e, F(e), L(e), P(e), e', F(e'), L(e'), S(e'))$$

where $F(e)$ is the first name of an employee (i.e. $(I(E \to F))(e)$), $F(e')$ is the other first name $(I(E' \to F))(e')$, and so forth. The equalizer of the two functions c, d is the subset:

$$\{(e, e', f, l, p, s) : f = F(e) = F(e'), l = L(e) = L(e'), p = P(e), s = S(e')\} \tag{2.3}$$

which we can write as the table in Table 3. In that representation we've counted the first two columns as the "ID" column, but using our computation the actual set of IDs would be as in Equation 2.3, i.e. an element of a product. Notice that we lost all of the rows that didn't have an entry with a matching First and Last name in both tables. This offers a proof-by-example that the

ID1	ID2	First	Last	Salary	Position
E01	F03	Francis	Poulenc	\$15,000	Executive Vice-CEO
E03	F02	Charles	Gounod	\$16,000	Associate Fax Guru
E04	F01	Georges	Bizet	\$24,000	Junior Intern

Table 3: The result of our join of the two tables in Table 2 along First and Last.

Π operator handles joins of tables.

Our last operator, Σ, is dual: it handles unions of tables. We can run the same computation to see what Σ_G does by dualizing Theorem 2.16, since the colimit of $G : \mathcal{C} \to \mathcal{D}$ is the limit of $G^{\mathsf{op}} : \mathcal{C}^{\mathsf{op}} \to \mathcal{D}^{\mathsf{op}}$:

Theorem 2.17. The colimit of $\gamma : \mathcal{C} \to \mathsf{Set}$ may be computed as the coequalizer:

$$\bigsqcup_{f \in \mathsf{Mor}\,\mathcal{C}} \gamma(\mathsf{dom}\, f) \underset{d}{\overset{c}{\rightrightarrows}} \bigsqcup_{X \in \mathsf{Ob}\,\mathcal{C}} \gamma(X) \longrightarrow \operatorname{colim} \gamma$$

where

$$\begin{aligned} c &:= \langle \iota_{\mathsf{dom}\, f} \rangle_{f \in \mathsf{Mor}\,\mathcal{C}} \\ d &:= \langle \iota_{\mathsf{cod}\, f} \circ \gamma f \rangle_{f \in \mathsf{Mor}\,\mathcal{C}} \end{aligned}$$

♦

We are working with the other comma category $G \downarrow E''$, which has two objects:

$$GE \to E'' \qquad GE' \to E''$$

and only the identity morphisms, since there are no morphisms $E \to E'$ or $E' \to E$ in $\mathcal{C}$. Our left coproduct is:

$$I(E) \sqcup I(E')$$

The right:

$$I(E) \sqcup I(E')$$

The function c is the identity, and so is d. In this case, our resulting table is the disjoint union of the two, given in Table 4. In the join case, we used the most information possible from the "along First and Last" part of the join: we merged the pairs of rows, one from each table, that had the same first and last name. In the union case, we used the least information possible: the resulting table has as columns the unions of the columns in the two tables, taking from "along First and Last" to mean only that we must identify the two First and Last columns.

ID1	ID2	First	Last	Salary	Position
E01	–	Francis	Poulenc	$15,000	–
E02	–	George	Dyson	$14,500	–
E03	–	Charles	Gounod	$16,000	–
E04	–	Georges	Bizet	$24,000	–
–	F01	Georges	Bizet	–	Junior Intern
–	F02	Charles	Gounod	–	Associate Fax Guru
–	F03	Francis	Poulenc	–	Executive Vice-CEO

Table 4: The result of our union of the two tables in Table 2 along First and Last.

Thus, by these examples we observe that this theory of data migration at least subsumes the notion of data migration from relational algebra. Given a functor $F : \mathcal{C} \to \mathcal{D}$ between schemas, we have three very natural ways of migrating instances between schemas: precomposition $\Delta_F : \mathcal{D}\textbf{-Inst} \to \mathcal{C}\textbf{-Inst}$, right Kan extension $\Pi_F : \mathcal{C}\textbf{-Inst} \to \mathcal{D}\textbf{-Inst}$, and left Kan extension $\Sigma_F : \mathcal{C}\textbf{-Inst} \to \mathcal{D}\textbf{-Inst}$. These correspond to the three fundamental notions of relational algebra: projection, join, and union, respectively.

2.4 The Problem

There is a problem with this approach; we danced around it in our computations of examples of data migration. We define Π_F and Σ_F for some functor $F : \mathcal{C} \to \mathcal{D}$ using limits and colimits, respectively. In our computation, we used the specific equation for a limit given in Theorem 2.16. However, *a limit is only defined up to isomorphism*: in Set, this means that when we compute a limit, any other set of the same cardinality will suffice.

It turns out that for databases Theorem 2.16 gives us a very nice limit object, but theoretically speaking the resulting limit could have nothing to do with our input data. This is not a problem for ID columns like Employee in our example: we don't really care what the IDs are, so long as they're unique. However, it does ruin our attribute columns, like First: the resulting set of first names in our migrated instance doesn't need to be a subset or superset of our original set, or even

ID1	ID2	First	Last	Salary	Position
A05	G03	Sheridan	3:00 PM EST	★★★★	𝕐 𝕆 𝕊 ℍ 𝕀
E19	E02	Limited	Network	Connectivity	Forty-seven
E22	G01	Apples	⊔	Paris	According to all known laws of aviation, there is no way

Table 5: Another possible result of our join of the two tables in Table 2 along First and Last.

have any elements in common. For example, the result of our join operation could be given by Table 5 instead of Table 3.

This leads us to consider a different approach, moving the attribute columns from our schema category into a separate category in Section 3. We will have the same fundamental data migration operations that are defined in much the same way, but we will need to place some more conditions on the schemas to ensure that they always exist.

2.5 A Diversion to Monads

Monads are ubiquitous in category theory and computer science, though in applications they are not often called monads, nor is their full structure often used. Using our theory of databases as categories and instances as functors, we can use monads to represent certain special types of databases. This section is explored in greater detail in [Spi12b].

The main example for this section will be fuzzy databases. It is very difficult to pin down a single definition for a fuzzy database, and all implementations found by the author (or proposed implementations) make use of conventional databases with multiple columns representing each attribute (e.g. [GUP05]). The one we will be using is the following, which is not too helpful for implementation but generalizes all of the definitions the author has seen:

Definition 2.18. A fuzzy database is a database in which every relation and attribute gives a probability distribution on the possible values of that relation/attribute, instead of simply a value.♦

The idea is that this type of database can store "fuzzy" information: we may know that a person's height is between 6' and 6'6", or that they live within 5km of the Bahen building, or that they're most likely married to Susan but might be married to Karen. We can represent all of these cases using the above definition: for height, the set of values would be the positive real numbers (perhaps up to 10') and we may represent "between 6' and 6'6"" by assigning to it a uniform distribution on $[6, 6.5]$. By changing our definition of database instances to include a certain construction using monads, we can implement these fuzzy databases as well as other exotic instances.

First, we give the basic definitions.

Definition 2.19. Let $\mathcal{C}$ be a category. A monad on $\mathcal{C}$ is a triple (T, μ, η) of

- A (covariant) endofunctor $T : \mathcal{C} \to \mathcal{C}$;
- A natural transformation $\mu : T^2 \Rightarrow T$ called multiplication;
- A natural transformation $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ called the unit transformation

such that μ is associative; for each $A \in \mathcal{C}$:

$$\begin{array}{ccc} T^3A & \xrightarrow{T\mu_A} & T^2A \\ \downarrow{\scriptstyle \mu_{TA}} & & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow{\mu_A} & TA \end{array}$$

and η and μ are compatible; for each $A \in \mathcal{C}$:

$$\begin{array}{ccc} TA & \xrightarrow{\eta_{TA}} & T^2A \\ \downarrow{\scriptstyle T\eta_A} & \searrow & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow{\mu_A} & TA \end{array}$$

♦

Examples abound: every adjunction $F : \mathcal{C} \leftrightarrows \mathcal{D} : U$ with $F \dashv U$ (F left adjoint to G) induces a monad on $\mathcal{C}$ via $U \circ F : \mathcal{C} \to \mathcal{C}$, with the unit of the adjunction being the unit of the monad, and if $\varepsilon : F \circ U \Rightarrow \mathrm{id}_{\mathcal{D}}$ is the counit then $U\varepsilon_F : UF \circ UF \Rightarrow UF$ is the multiplication. In particular, any "free" object in algebra gives a monad on Set:

- For a fixed ring R, the free R-module on a set, specializing to the free vector space and free abelian group.
- The free group on a set.
- The free monoid on a set.
- etc.

These are a bit abstract; we will discuss in some detail two examples used in computer science, the list monad and the maybe monad.

The list monad is another name for the free monoid monad: it is a functor $L : \mathsf{Set} \to \mathsf{Set}$ that maps each set to the set consisting of finite lists of elements from that set, i.e.:

$$L(X) := \{(x_1, \ldots, x_n) : n \in \mathbb{Z}_{\geq 0}, x_i \in X\}$$

We will write $[X] := L(X)$, which is the notation used in e.g. Haskell for lists. This functor acts on functions $f : X \to Y$ in the obvious way: to get a function $[f] : [X] \to [Y]$ we apply f to each entry of the input list. This is precisely the `map` function present in many programming languages.

The "multiplication" operation $\mu : [[X]] \to [X]$ is list concatenation: $[[X]]$ is the set of lists of [lists of elements of X], and to get a list of elements of X we concatenate all lists in the input list of lists. The "unit" transformation $\mu : X \to [X]$ maps an element $x \in X$ to the singleton list $[x]$ containing just x. The compatibility criterion is easily checked, or is immediate by observing that this comes from the adjunction $\mathsf{Set} \to \mathsf{Mon}$ mapping a set to its free monoid.

The maybe monad can be used for exception handling; it can also be used to implement the null coalescing operator "`.?`" present in some languages, e.g. C#. This is the functor $M : \mathsf{Set} \to \mathsf{Set}$ that maps each set to itself with a disjoint basepoint added:

$$M(X) := X \sqcup \{*\}$$

The action on functions $f : X \to Y$ is defined by setting $Mf(x) := f(x)$ for all $x \in X$ and $Mf(*) := *$, preserving the basepoints. The multiplication operator identifies the two basepoints:

$$\mu((X \sqcup \{*\}) \sqcup \{*\}) := X \sqcup \{*\}$$

i.e. we map both basepoints in our input to the same basepoint in the output. The unit operator $X \to X \sqcup \{*\}$ is the evident inclusion. This is helpful for programming since we can chain together functions that *might* return a value, or might fail; if they succeed, we keep going down the chain, passing the output along, but if they fail we stop immediately. We can extend this to add any finite number of basepoints to the set to represent an arbitrary number of distinct exceptions or failure states.

For our case, the most important monad will be the Giry monad (or one of its variants). We will need a definition:

Definition 2.20. Let X be a set and $p : X \to [0, 1]$ a probability distribution on X, i.e. a function such that $\sum_{x \in X} p(x) = 1$. We call p *finitary* if $p(x) \neq 0$ for only finitely many $x \in X$. The *support* of p, denoted $\operatorname{supp} p$, is the subset of X on which p is nonzero:

$$\operatorname{supp} p := \{x \in X : p(x) \neq 0\}$$ ♦

We define the (finitary, Set-valued) Giry monad $G : \mathsf{Set} \to \mathsf{Set}$ as follows: we map a set X to the set of finite probability distributions on X and on a function $f : X \to Y$ to be the function sending:

$$G(f) : GX \to GY$$
$$G(f)(p : X \to [0, 1]) : Y \to [0, 1]$$
$$G(f)(p)(y) := \sum_{x \in f^{-1}(y)} p(x)$$

The multiplication is given by the weighted sum:

$$\mu_X : G(GX) \to GX$$
$$(\mu_X(f : GX \to [0, 1]))(x) := \sum_{p \in \operatorname{supp} f} f(p) \cdot p(x)$$

The unit is given by the Kronecker delta, i.e.:

$$\eta_X : X \to GX$$
$$\eta_X(x) : X \to [0, 1]$$
$$(\eta_X(x))(y) := \begin{cases} 1 & \text{if } y = x \\ 0 & \text{otherwise} \end{cases}$$

The usual Giry monad is the monad $G : \mathsf{Meas} \to \mathsf{Meas}$ defined on the category of measurable spaces and measurable functions which sends a measurable space $(X, \mathcal{M})$ (we will typically omit the σ-algebra $\mathcal{M} \subseteq \mathcal{P}(X)$ from the notation) to the set of all probability measures on that space with the σ-algebra generated by the evaluation maps $\mathrm{ev}_U : G(X) \to [0, 1]$ given by evaluating a probability measure on the measurable subset $U \subseteq X$ for all $U \in \mathcal{M}$. The action on functions is:

$$G(f) : G(X, \mathcal{M}) \to G(Y, \mathcal{M}')$$
$$G(f)(p) : \mathcal{M}' \to [0, 1]$$
$$(G(f)(p))(U) := \int_{f^{-1}(U)} dp$$

The multiplication is:

$$\mu_X : G(GX) \to GX$$
$$\mu_X(p) : \mathcal{M} \to [0,1]$$
$$\mu_X(p)(U) := \int_{q \in GX} \mathrm{ev}_U(q)\, dp$$

and the unit is given by the Dirac measure (point mass):

$$\eta_X : X \to GX$$
$$\eta_X(x) : \mathcal{M} \to [0,1]$$
$$(\eta_X(x))(U) := \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise} \end{cases}$$

How do we connect this to databases? Our motivating example is that of fuzzy databases, so we want to have our attributes to be probability distributions over the possible values. The solution is the Kleisli category:

Definition 2.21. Let (T, μ, η) be a monad on a category $\mathcal{C}$. The Kleisli category of T, denoted $\mathrm{KI}(T)$, has

- as objects the objects of $\mathcal{C}$;
- as morphisms $A \to B$ the morphisms $A \to TB$ in $\mathcal{C}$

with composition of two morphisms $f : A \to B$ (really $f : A \to TB$) and $g : B \to C$ (really $g : B \to TC$) given by the composite

$$A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu} TC$$

The identity morphisms are the components of the unit map $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$. ⧫

Thus, instead of considering Set-instances $I : \mathcal{C} \to \mathsf{Set}$, we should instead consider *Kleisli* instances $I : \mathcal{C} \to \mathrm{KI}(T)$ for some fixed monad T.

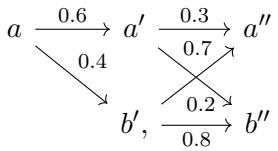
Definition 2.22. Let $T : \mathcal{D} \to \mathcal{D}$ be a monad and $\mathcal{C}$ a database schema. A *Kleisli T-instance of* $\mathcal{C}$ is a functor $\mathcal{C} \to \mathrm{KI}(T)$. ⧫

In the case of fuzzy databases, what is an instance $I : \mathcal{C} \to \mathrm{KI}(G)$, where $G : \mathsf{Set} \to \mathsf{Set}$ is the finitary Giry monad? The objects of $\mathrm{KI}(G)$ are the same as the objects of the (co)domain of G, so for each attribute in our schema we still get a set. However, the morphisms $X \to Y$ in $\mathrm{KI}(G)$ are instead morphisms $X \to GY$ in Set; that is, instead of getting a function $If : IX \to IY$, we instead get a function $If : IX \to G(IY)$, i.e. for each attribute we get a finitary probability distribution on the possible values, as we wanted.

This doesn't quite implement our example: for example, we couldn't represent a *continuous* range like $[6, 6.5]$ as a uniform probability distribution since we need to give finitary probability distributions. The two solutions are either to switch to discrete values (necessary for implementation anyways) or to use the full Giry monad $G : \mathsf{Meas} \to \mathsf{Meas}$ (perhaps helpful for theoretical purposes).

What does functoriality of $I : \mathcal{C} \to \mathrm{KI}(G)$ mean? This is answered by considering composition in the Kleisli category. This will be different for every monad, but for the Giry monad this means that our compositions must respect conditional probabilities. The composition is unenlightening to write down generally, but consider the following specific example; each column is a different set and we define functions $f : A \to B$ and $g : B \to C$ using the arrows labeled with probabilities:

$$A \xrightarrow{f} B \xrightarrow{g} C$$



The probability of a'' given a should be

$$\mathrm{P}(a''|a) = 0.6 \cdot 0.3 + 0.4 \cdot 0.2 = 0.26$$

and this is precisely what composition is in the Kleisli category: we have $(g \circ f)(a)(a'') = 0.26$. Thus, a valid database instance $I : \mathcal{C} \to \mathrm{KI}(G)$ (i.e. one that is actually a functor) would be required to respect conditional probabilities in this fashion.

The other monads offer similar functionality: the maybe monad allows us to have "labelled null" values, i.e. we can set certain attributes to be a globally-identifiable "null" value (the basepoint) if we have no data for that entry. The list monad allows us to take as values lists of values, instead of single values.

The data migration functors can certainly be extended to this notion of instances: they were defined for instances valued in arbitrary categories. However, Kleisli categories in general fail to have many limits and colimits; thus, we cannot appeal to e.g. Corollary 2.14 to prove the existence of the union and join migration functors (projection always exists).

The other limitation is that we are stuck with the same monad for the entire database: if we use the Giry monad every value must be a probability distribution, with the list monad a list, and so on. However, this is still an interesting example of how other natural concepts in category theory interact in interesting ways with our notion of databases.

3 Databases via Slice Categories

We observed in Section 2 that viewing databases as categories can be done in a natural way; our computations in Section 2.3 showed that this perspective naturally gives rise to the fundamental data migration operations from relational algebra. However, we ran into some problems in Section 2.4: the notion of isomorphism in Set (up to cardinality) is too weak to properly deal with actual data. This problem is not fatal: what if we restrict our schemas to only contain the *ID objects*, and leave out the attributes?

In this case, the weak notion of isomorphism is no longer an issue: as long as the IDs are unique and we can keep track of how they change through migrations, we don't actually care what they are. We only need to find a way to reintroduce attributes that will behave properly with respect to migration.

This section will present a (small) subset of the material in [SW15].

3.1 Typed Instances

Consider again the example database with data in Table 1 and schema in Figure 1. We wish to convert this into a category using our new perspective: we do the same as in Procedure 2.4, but we forget about any attributes for now. This gives us the graph in Figure 7 (c.f. Figure 2).

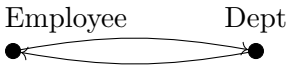
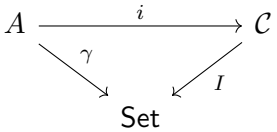


Figure 7: The new graph for the schema in Figure 1.

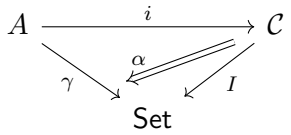
How do we add attributes to this? Suppose we have a schema $\mathcal{C}$. A $\mathcal{C}$-instance is a functor $I : \mathcal{C} \to \mathsf{Set}$, which now only gives a set of unique IDs for each object in our schema and functions between them corresponding to the relations; for our running example, we get a set of employee IDs and a set of department IDs. We want each employee to have a first name, last name, and a salary and each department to have a location. Our original schema in Figure 1 also gives specific *types* for each of these, i.e. sets that they must be contained in. Suppose we define a new category A with one object for each attribute we want:

$$\mathsf{Ob}\, A := \{\text{First}, \text{Last}, \text{Salary}, \text{Location}\}$$

The type of each attribute assigns a set to each object of this category; if we let A be a discrete category (a category with no non-identity morphisms), then this typing gives a functor $\gamma : A \to \mathsf{Set}$ mapping First to the set of strings, Salary to the set of positive (nonnegative?) real numbers, and so forth. We also have a natural functor $i : A \to \mathcal{C}$ assigning each attribute to its entity: First, Last, and Salary are mapped to Employee and Location is mapped to Dept (since A is discrete, it suffices to define these functors only on the objects of A, since functoriality tells us where to send identities). We now have the following situation; the diagram need not commute:



We've captured all of the information given in the schema in Figure 1 using the pair of functors $\gamma : A \to \mathsf{Set}$ (types of attributes) and $i : A \to \mathcal{C}$ (assigning attributes to entities). All that remains is to add attributes to our instance $I : \mathcal{C} \to \mathsf{Set}$. This is done by specifying a natural transformation $\alpha : I \circ i \Rightarrow \gamma$:

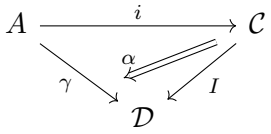


What does this give us? For each object $x \in A$, we get a function from $(I \circ i)(a) \to \gamma(a)$, so for each attribute $x \in A$ we get a function from the set of IDs for its associated object $I(i(x))$ to its set of possible values $\gamma(x)$. If $E := I(\text{Employee})$ is our set of employee IDs, then $(I \circ i)(\text{First}) = E$ so we get a function from E to the set of strings which maps an employee to their first name; likewise the other attributes. Naturality of α ensures that these assignments reflect composition in our schema $\mathcal{C}$. We now revise our definition of schemas and instances to reflect this:

Definition 3.1. A *schema* is a category $\mathcal{C}$.

Let $\mathcal{D}$ be a category. A *$\mathcal{D}$-valued typing for a schema $\mathcal{C}$* is a triple (A, i, γ) of a category A, called the *attribute category*, and a pair of functors $\gamma : A \to \mathcal{D}$ called the *attribute typing* and $i : A \to \mathcal{C}$ called the *attribute assignment*. We call the category $\mathcal{C}$ the *entity category*.

Let $\mathcal{C}$ be a schema and (A, i, γ) a $\mathcal{D}$-valued typing. A *typed $\mathcal{C}$-instance* is a functor $I : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : I \circ i \Rightarrow \gamma$:



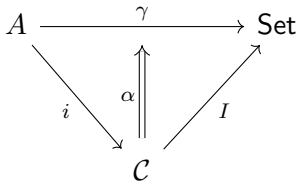
(3.1)

Note that the diagram need not commute. ♦

We will work with $\mathcal{D} := \mathsf{Set}$ throughout.

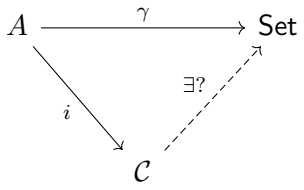
In the definition, we could require A to be discrete as in the preceding discussion; however, allowing A to have non-identity morphisms mean that we can have attributes that are wholly determined by the value of other attributes (e.g. net earnings are the difference of gross earnings and expenses). These can be added later, but putting them in the typing A ensures that they will be preserved by any data migration; we will explore this further in Section 4.

We wish to form a category $\bar{\mathcal{C}}$**-Inst** of typed instances as we did with untyped instances; what is the correct notion of "morphism" of typed instances? If we rotate the diagram in Equation 3.1 a bit:



This looks like something related to a Kan extension, Definition 2.9: it says that (I, α) is a *right*

solution to the lifting problem:

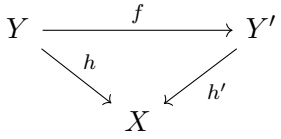


(3.2)

So we can equivalently define an instance to be a right solution to this lifting problem. The advantage of this is that we have a natural notion of a category of instances: the slice category of $\mathsf{Set}^{\mathcal{C}}$ (or our original $\mathcal{C}$-**Inst**) over the right Kan extension $\mathrm{Ran}_i\,\gamma : \mathcal{C} \to \mathsf{Set}$. We give the definition of the slice category; recall the comma category, Definition 2.15.

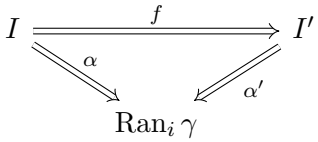
Definition 3.2. Let $\mathcal{C}$ be a category and $X \in \mathsf{Ob}\,\mathcal{C}$. View X as the constant functor $X : \mathbf{1} \to \mathcal{C}$ where $\mathbf{1}$ is the category with one object and one morphism, which are sent to X. We call the comma category $\mathrm{id}_{\mathcal{C}} \downarrow X$ the *slice category of* $\mathcal{C}$ *over* X and denote it by $\mathcal{C}/X$. Removing redundant terms, it has:

- Objects the pairs (Y, h) for $Y \in \mathsf{Ob}\,\mathcal{C}$ and $h : Y \to X$ a morphism in $\mathcal{C}$;
- Morphisms $(Y, h) \to (Y', h')$ the morphisms $f : Y \to Y'$ in $\mathcal{C}$ such that the following commutes:

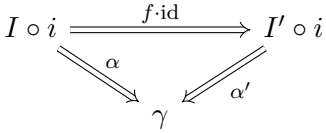


Likewise, we have a *slice category of* $\mathcal{C}$ *under* X given by $X/\mathcal{C} := X \downarrow \mathrm{id}_{\mathcal{C}}$. ♦

Let $\eta : \mathrm{Ran}_i\,\gamma \circ i \Rightarrow \gamma$ be the universal natural transformation of the right Kan extension. The universal property of the right Kan extension says that giving a right solution (I, α) to our lifting problem in Equation 3.2 is the same as giving a functor $I\mathcal{C} \to \mathsf{Set}$ and a natural transformation $\alpha' : I \Rightarrow \mathrm{Ran}_i\,\gamma$, since every natural transformation $\alpha : I \circ i \Rightarrow \gamma$ factors uniquely through $\eta : \mathrm{Ran}_i\,\gamma \circ i \Rightarrow \gamma$. We now have a candidate category for the category of typed instances: $\mathcal{C}$-**Inst**$/\,\mathrm{Ran}_i\,\gamma$. What are the morphisms in this category? Explicitly, a morphism $I \to I'$ (for (I, α) and $(I', \alpha') \in \mathcal{C}$-**Inst**$/\,\mathrm{Ran}_i\,\gamma$) is a natural transformation $f : I \Rightarrow I'$ such that



By the universal property of the right Kan extension, this is equivalent to, if we replace α with $\eta \cdot \alpha$ and α' with $\eta \cdot \alpha'$:



which means a natural transformation $f : I \Rightarrow I'$ such that for each $x \in A$ and $y \in I(a)$ we have

$$\alpha_a(y) = (\alpha'_a \circ f_{i(a)})(y)$$

so f preserves the attributes, i.e. cannot change any attributes of any rows. We define:

Definition 3.3. Let $\mathcal{C}$ be a schema and (A, i, γ) a typing. The *category of typed $\mathcal{C}$-instances*, denoted $\bar{\mathcal{C}}$**-Inst**, is the category with

- Objects the typed $\mathcal{C}$-instances, i.e. pairs $I : \mathcal{C} \to \mathcal{D}$ and $\alpha : I \circ i \Rightarrow \gamma$;
- Morphisms the typed homomorphisms, the natural transformations $f : I \Rightarrow I'$ that preserve attributes. ⧫

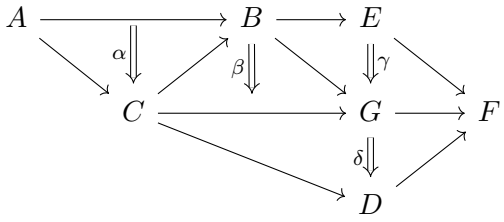
Lemma 3.4. $\bar{\mathcal{C}}$**-Inst** is isomorphic to the slice category $\mathcal{D}^{\mathcal{C}} / \operatorname{Ran}_i \gamma = \mathcal{C}$**-Inst**$/\Pi_i \gamma$. ⧫

Proof. Right solutions $I : \mathcal{C} \to \mathcal{D}$ and $\alpha : I \circ i \Rightarrow \gamma$ are in bijection with pairs $I : \mathcal{C} \to \mathcal{D}$ and $\alpha : I \Rightarrow \operatorname{Ran}_i \gamma$ by the universal property of $\operatorname{Ran}_i \gamma$. □

3.2 A Primer on Pasting Diagrams

We will need to deal extensively with natural transformations when extending data migration to typed instances; it will be helpful to draw pasting diagrams.

A pasting diagram is a commutative diagram of natural transformations; that is, a diagram with nodes representing categories, arrows between nodes representing functors, and arrows between arrows representing natural transformations. These diagrams need not commute when the natural transformations are removed, but any compositions of natural transformations will commute. For example, the diagram:

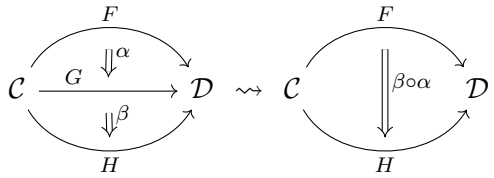


(3.3)

says that all possible compositions of the natural transformations will be equal, but says nothing about the equality of compositions of the functors.

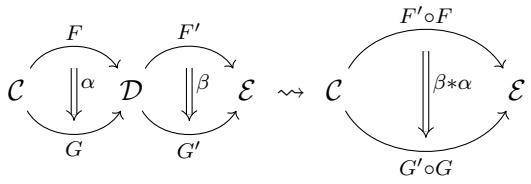
We can compose natural transformations in two ways:

Definition 3.5. Let $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ be natural transformations for $F, G, H : \mathcal{C} \to \mathcal{D}$ functors. The *vertical composition* of α and β, denoted $\beta \circ \alpha$, is the natural transformation $F \Rightarrow H$:



with components $(\beta \circ \alpha)_X := \beta_X \circ \alpha_X$.

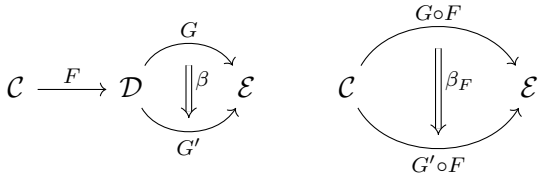
Let $\alpha : F \Rightarrow G$ for $F, G : \mathcal{C} \to \mathcal{D}$ and $\beta : F' \Rightarrow G'$ for $F', G' : \mathcal{D} \to \mathcal{E}$. The *horizontal composition* of α and β, denoted $\beta * \alpha$, is the natural transformation $F' \circ F \Rightarrow G' \circ G$:



whose components are given by

$$(\beta * \alpha)_X := (G'\alpha_X) \circ \beta_{FX} = \beta_{GX} \circ (F'\alpha_X)$$ ♦

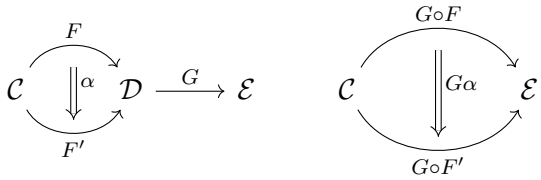
An important special case of horizontal composition is when one of the two parallel pairs of functors are the same, e.g.:



We get the "whiskered" natural transformation $\beta * \mathrm{id}_F =: \beta_F : G \circ F \Rightarrow G' \circ F$ whose components are, as suggested by the notation:

$$(\beta_F)_X := \beta_{FX}$$

and the other case:

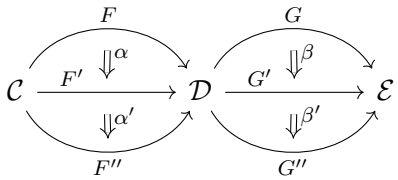


we get the "whiskered" natural transformation $\mathrm{id}_G * \alpha =: G\alpha : G \circ F \Rightarrow G \circ F'$ whose components are:

$$(G\alpha)_X := G(\alpha_X)$$

These compositions interact nicely:

Proposition 3.6 (Middle Exchange Law). In the situation:



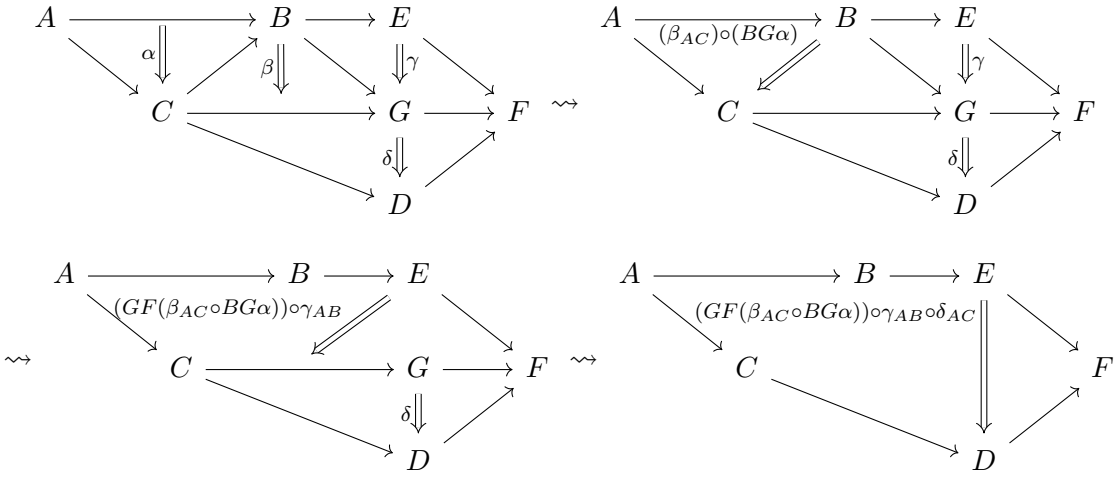
we have that

$$(\beta' \circ \beta) * (\alpha' \circ \alpha) = (\beta' * \alpha') \circ (\beta * \alpha)$$ ♦

This says that horizontal composition followed by vertical composition is the same thing as vertical composition followed by horizontal composition. The result is that, given any pasting diagram,

we can compose natural transformations in either direction (horizontally or vertically) and end up with the same result.

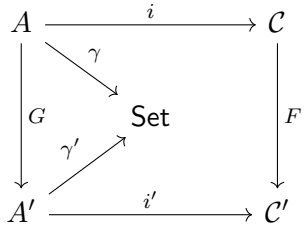
Going back to the example in Equation 3.3, one such sequence of compositions is, using e.g. AB to refer to the unique arrow $A \to B$:



and any other sequence of compositions would give the same result by the middle exchange law.

3.3 Typed Data Migration

We take the same approach as with schemas to extend data migration to typed schemas: use the same procedure as before and add anything necessary to deal with typing. Here we will add the requirement that the value categories of the schemas are the same; we will assume they are Set. Since a typed schema is now a pair of categories with some extra functors, we should need, for instances $(\mathcal{C}, A, i, \gamma)$ and $(\mathcal{C}', A', i', \gamma')$ a pair of functors $F : \mathcal{C} \to \mathcal{D}$ and $G : A \to A'$ that fit into our schema diagrams; this means (the following diagram commutes):



This definition also makes sense categorically: observe that if we define the functor:

$$\mathrm{id} \times \mathsf{Set} : \mathsf{CAT} \to \mathsf{CAT}, \mathcal{C} \mapsto \mathcal{C} \times \mathsf{Set}$$

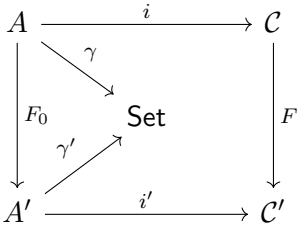
then we can view typed Set-valued schemas as elements of the comma category $\mathrm{id} \downarrow (\mathrm{id} \times \mathsf{Set})$, which are triples $(A, \mathcal{C}, \langle i, \gamma \rangle : A \to \mathcal{C} \times \mathsf{Set})$ (we need to use CAT, the category of large categories, since $\mathsf{Set} \notin \mathsf{Cat}$). The morphisms in this comma category are precisely the pairs $(G : A \to A', F : \mathcal{C} \to \mathcal{C}')$ such that

$$\begin{array}{ccc} A & \xrightarrow{G} & A' \\ \downarrow{\scriptstyle \langle i,\gamma \rangle} & & \downarrow{\scriptstyle \langle i',\gamma' \rangle} \\ \mathcal{C} \times \mathsf{Set} & \xrightarrow{F \times \mathrm{id}} & \mathcal{C}' \times \mathsf{Set} \end{array}$$

which, unraveling the products, corresponds precisely to the diagram in Equation 3.3. We will name this:

Definition 3.7. The category of Set-valued schemas, denoted SetSchemas, is the comma category $\mathrm{id} \downarrow (\mathrm{id} \times \mathsf{Set})$. ⧫

Definition 3.8. Let $(\mathcal{C}, A, i, \gamma)$ and $(\mathcal{C}', A', i', \gamma')$. A typed schema morphism is a pair (F, F_0) of functors $F : \mathcal{C} \to \mathcal{C}'$ and $F_0 : A \to A'$ such that the following commutes:



⧫

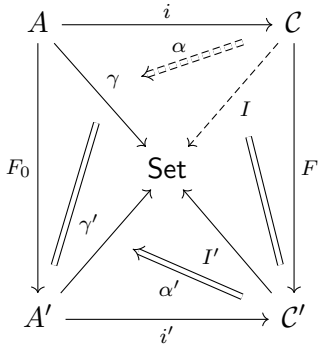
The existence of the various data migration functors is harder now: given $F : \mathcal{C} \to \mathcal{D}$ and $G : A \to A'$, we can no longer start simply with precomposition $\Delta_F := F^* : \mathcal{D}$**-Inst** $\to \mathcal{C}$**-Inst**; we must instead define some new functor $\Delta_{F,G} : \overline{\mathcal{C}'}$**-Inst** $\to \overline{\mathcal{C}}$**-Inst**. We refer directly to [SW15] for sufficient conditions; throughout, (F, F_0) is a typed schema morphism $(\mathcal{C}, A, i, \gamma) \to (\mathcal{C}', A', i', \gamma')$.

Proposition 3.9 (Proposition 9.1.7 in [SW15]). The pullback functor $\Delta_F : \mathcal{C}'$**-Inst** $\to \mathcal{C}$**-Inst** of untyped instances extends to a functor of typed instances

$$\Delta_{\overline{F}} : \overline{\mathcal{C}'}\textbf{-Inst} \to \overline{\mathcal{C}}\textbf{-Inst}$$

⧫

Proof. Suppose we have a $\mathcal{C}'$ instance $I' : \mathcal{C}' \to \mathsf{Set}, \alpha' : I' \circ i' \Rightarrow \gamma'$. We are in the following situation:



(3.4)

We need to define $I : \mathcal{C} \to \mathcal{C}'$ and $\alpha : I \circ i \Rightarrow \gamma$. We define $I := \Delta_F I' = I' \circ F$ to be the usual precomposition. The natural transformation $\alpha : \Delta_i I \Rightarrow \gamma$ is the composite:

$$\Delta_i I = \Delta_i \Delta_F I' = \Delta_{F_0} \Delta_{i'} I' \xRightarrow{\alpha'_{F_0}} \Delta_{F_0} \gamma' = \gamma$$

That is, the composite of the bottom three natural transformations (two of which are identies) in Equation 3.4. □

Corollary 3.10. The above defines a contravariant functor $\Delta : \mathsf{SetSchemas} \to \mathsf{Cat}$ with

$$\overline{\mathcal{C}} \mapsto \overline{\mathcal{C}}\textbf{-Inst}$$
$$\overline{F} : \overline{\mathcal{C}} \to \overline{\mathcal{C}'} \mapsto \Delta_{\overline{F}} : \overline{\mathcal{C}'}\textbf{-Inst} \to \overline{\mathcal{C}}\textbf{-Inst}$$

⧫

We need more for the join operator:

Definition 3.11. Let $\overline{F} = (F, F_0)$ be a typed schema morphism $(\mathcal{C}, A, i, \gamma) \to (\mathcal{C}', A', i', \gamma')$. We call $\overline{F}$ *join-ready* if $A = A'$, $\gamma = \gamma'$, and F_0 is the identity $\mathrm{id}_A : A \to A$. ♦

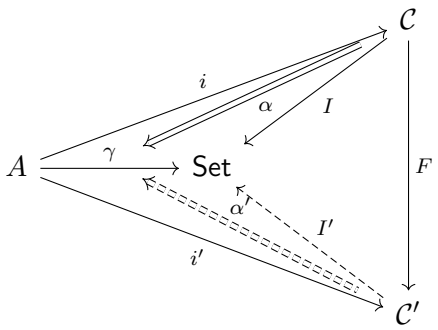
This means that these join-ready morphisms cannot remove, add, re-type, or otherwise mess with the attributes in our attribute category. This seems like a restriction, but we will recover the generality using *compositions* of the various data migration functors.

As suggested by the name:

Proposition 3.12 (Proposition 9.1.10 in [SW15]). Let $\overline{F}$ be a join-ready typed schema morphism $(\mathcal{C}, A, i, \gamma) \to (\mathcal{C}, A, i', \gamma)$. Then the join functor $\Pi_F : \mathcal{C}\textbf{-Inst} \to \mathcal{C}'\textbf{-Inst}$ of untyped instances extends to a functor of typed instances

$$\Pi_{\overline{F}} : \overline{\mathcal{C}}\textbf{-Inst} \to \overline{\mathcal{C}'}\textbf{-Inst}$$ ♦

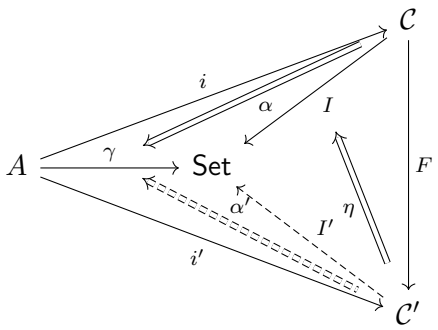
Proof. Suppose we have a $\mathcal{C}$ instance $I : \mathcal{C} \to \mathsf{Set}$, $\alpha : I \circ i \Rightarrow \gamma$. Since the attribute categories are the same, we are in the following situation:



We want an extension, so we set $I' := \Pi_F I$. This gives us a natural transformation $\eta : \Delta_F \Pi_F I \Rightarrow I$, and we set $\alpha' : \Delta_{i'} \Pi_F I$ to be the composite:

$$\Delta_{i'} \Pi_F I = \Delta_i \Delta_F \Pi_F I \xRightarrow{\Delta_i \eta} \Delta_i I \xRightarrow{\alpha} \gamma$$

That is, the composite of the two upper natural transformations in:



□

Observe also that $\eta : \Delta_F \Pi_F I \Rightarrow I$ is the I-th component of the counit $\Delta_F \Pi_F \Rightarrow \mathrm{id}$ of the adjunction $\Delta_F \dashv \Pi_F$. This is nearly a functor:

Corollary 3.13. Let ΠSetSchemas denote the subcategory of SetSchemas with morphisms the join-ready typed signature morphisms. The above defines a covariant pseudofunctor $\Pi : \Pi\mathsf{SetSchemas} \to \mathsf{Cat}$ with

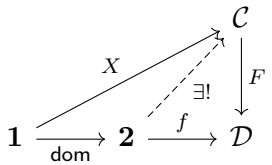
$$\overline{\mathcal{C}} \mapsto \overline{\mathcal{C}}\textbf{-Inst}$$
$$\overline{F} : \overline{\mathcal{C}} \to \overline{\mathcal{C}'} \mapsto \Pi_{\overline{F}} : \overline{\mathcal{C}}\textbf{-Inst} \to \overline{\mathcal{C}'}\textbf{-Inst}$$ ♦

Proof. Composition of left Kan extensions is associative up to natural isomorphism. □

Unions require more complicated conditions:

Definition 3.14. Let $\mathcal{C}, \mathcal{D}$ be categories. A functor $F : \mathcal{C} \to \mathcal{D}$ is a *discrete opfibration* if for each $X \in \mathcal{C}$ and $g : Y \to Y'$ for $Y, Y' \in \mathcal{D}$ with $F(X) = Y$ there is a unique morphism $\overline{g} : X \to X'$ in $\mathcal{C}$ for some $X' \in \mathcal{C}$ such that $F(\overline{g}) = g$. ♦

This is almost like a path-lifting condition from algebraic topology; if **2** is the category with two objects and one arrow and **1** is the category with one object and no non-identity arrows, this says for each pair of functors $g : \mathbf{2} \to \mathcal{D}$ (i.e. each morphism $g \in \mathsf{Mor}\,\mathcal{D}$) and $X : \mathbf{1} \to \mathcal{D}$ (i.e. each object $X \in \mathsf{Ob}\,\mathcal{C}$) such that $g \circ \mathsf{dom} = F \circ X$ (the domain of g is $F(X)$) we have a unique lift:



That is, for each lift of the starting point (domain) of the path (morphism) $g : Y \to Y'$ in $\mathcal{D}$ we have a unique lift to a path (morphism) in $\mathcal{C}$ starting at the given point (object).

Definition 3.15. Let $\overline{F} = (F, F_0)$ be a typed schema morphism $(\mathcal{C}, A, i, \gamma) \to (\mathcal{C}', A', i', \gamma')$. We call $\overline{F}$ *union-ready* if

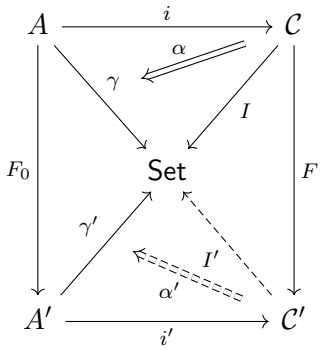
1. F_0 is a discrete opfibration; and
2. The following is a pullback diagram:

$$\begin{array}{ccc} A & \xrightarrow{i} & \mathcal{C} \\ {\scriptstyle F_0}\downarrow & \lrcorner & \downarrow{\scriptstyle F} \\ A' & \xrightarrow{i'} & \mathcal{C}' \end{array} \tag{3.5}$$
♦

Proposition 3.16 (Proposition 9.1.13 in [SW15]). Let $\overline{F}$ be a union-ready typed schema morphism $(\mathcal{C}, A, i, \gamma) \to (\mathcal{C}, A', i', \gamma')$. Then the union functor $\Sigma_F : \mathcal{C}\textbf{-Inst} \to \mathcal{C}'\textbf{-Inst}$ of untyped instances extends to a functor of typed instances

$$\Sigma_{\overline{F}} : \overline{\mathcal{C}}\textbf{-Inst} \to \overline{\mathcal{C}'}\textbf{-Inst}$$ ♦

Proof. Suppose we have a $\mathcal{C}$ instance $I : \mathcal{C} \to \mathsf{Set}$, $\alpha : I \circ i \Rightarrow \gamma$. We are in the following situation:



We define $I' := \Sigma_F I$. To get the rest, we rely on technical lemmas from [SW15]; we have a natural transformation $\zeta : \Sigma_{F_0}\Delta_i \Rightarrow \Delta_{i'}\Sigma_F$ given by

$$\Sigma_{F_0}\Delta_i \xRightarrow{\eta_F} \Sigma_{F_0}\Delta_i\Delta_F\Sigma_F = \Sigma_{F_0}\Delta_{i'}\Delta F_0\Sigma_F \xRightarrow{\varepsilon_{F_0}} \Delta_{i'}\Sigma_F$$

where $\eta : \mathrm{id} \Rightarrow \Delta_*\Sigma_*$ is the unit of the adjunction $\Delta_* \dashv \Sigma_*$ and $\varepsilon : \Sigma_*\Delta_* \Rightarrow \mathrm{id}$ is the counit. Since $F_0 : A \to A'$ is a discrete opfibration and the diagram in Equation 3.5 is a pullback diagram, by Proposition 8.4.16 of [SW15] this ζ is a natural isomorphism. We compose:

$$\Delta_{i'}\Sigma_F I \xRightarrow{\zeta^{-1}} \Sigma_{F_0}\Delta_i I \xRightarrow{\Sigma_{F_0}\alpha} \Sigma_{F_0}\gamma = \Sigma_{F_0}\Delta_{F_0}\gamma' \xRightarrow{\varepsilon_{F_0}} \gamma' \qquad \square$$

3.4 Parsing the Data Migration Functors

We run into a problem if we try to use the data migration functors in the obvious way. The projection functor Δ works exactly as expected: we needed no constraints to ensure its existence. However, the constraints on the other functors cause more trouble. Consider our join example in Table 2. The immediate way of translating these schemas into typed schemas might be like Figure 8. Note that we are forced to have the exact same attributes in both schemas, since the attribute categories must be the same. The typed morphism is the obvious one: we map EmpPos and EmpSalary both to Employee, and the attribute functor is the identity.

We run into a problem immediately. If I is a $\overline{\mathcal{C}}$-instance, then the functor part of the joined instance $\Pi_{\overline{F}} I : \mathcal{C}' \to \mathsf{Set}$ is defined as $\Pi_F I$: it incorporates none of our typing data! The resulting object will simply be the product of the two ID columns, ignoring any attributes that are the same.

However, if we have a database like our first example in Table 1, we could certainly remove one of the two foreign keys between them and join them based on the remaining one; by the computations for untyped tables, this would give us the expected result.

We need another way of translating schemas into typed schemas to have a meaningful way of joining tables; we will first consider the method proposed in [SW15]. Instead of assigning attributes directly to tables, we instead add a "dummy" object D, and give that object a single attribute; no other attributes are present in our database. For each attribute we want, we add a morphism from its object to D. For example, schema $\overline{\mathcal{C}}$ in Figure 8 would be something like the somewhat floral category in Figure 9.

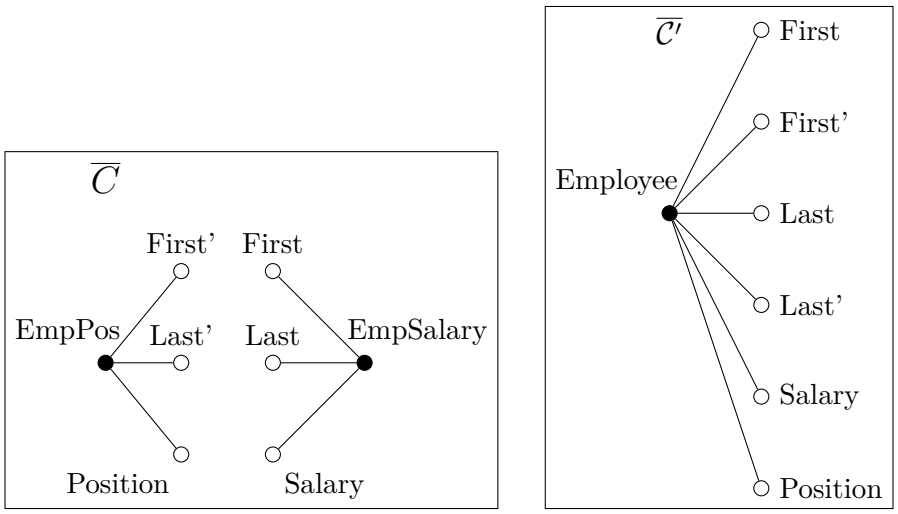


Figure 8: Our first attempt at a typed source schema (left) and desination schema (right) for our join operation. Filled nodes are elements of the foreign key category, hollow nodes are attributes, and lines from filled to hollow give attribute assignments.

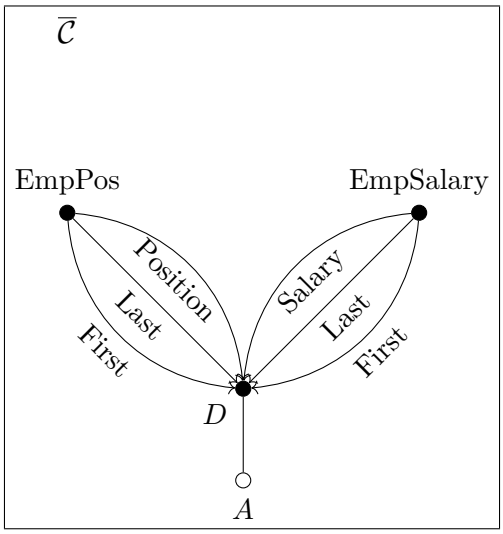


Figure 9: One approach to fixing unions and joins along attributes; c.f. Figure 8.

We still have an attribute for each distinct morphism into D: for each instance $I : \mathcal{C} \to \mathsf{Set}$, $\alpha : i \circ I \Rightarrow \gamma$, the morphism of sets will assign to each column of each row of the tables an ID in D, and the natural transformation will give the values to each attribute. We join along attributes by identifying morphisms in our functors; since all of our schemas only have one attribute (A), the condition for the existence of the join functor is always satsified.

Nonetheless, this approach itself is nonetheless unsatisfying: we have essentially removed the attribute category from the equation, which means we no longer have a categorical way of asserting that different attributes have different types (all attributes must take values in the same set, the type of the only attribute). Another approach is to "merge" the attribute category with the schema:

Definition 3.17. Let $\overline{\mathcal{C}} = (\mathcal{C}, A, i, \gamma)$ be a typed schema with A discrete. The merged typed

schema, denoted $[\overline{\mathcal{C}}]$, is the typed schema $([\mathcal{C}], A, [i], \gamma)$ with:

- Schema category $[\mathcal{C}]$ the category with:
  - Objects the disjoint union $\mathsf{Ob}\,\mathcal{C} \sqcup \mathsf{Ob}\,A$;
  - Morphisms generated by the morphisms of $\mathcal{C}$, freely adjoined with a unique morphism $i(a) \to a$ for each $a \in A$.
- Attribute category the same category A;
- Attribute mapping $[i] : A \to [\mathcal{C}]$ given by $[i](a) := a$, viewing the right a as $a \in \mathsf{Ob}[\mathcal{C}]$
- Attribute typing the same functor γ. ⧫

This is easily understood by example; see Figure 10. The idea behind this is to return to our original approach of converting schemas into categories, while using the attribute category A to "pick out" all of the objects which are concrete: those objects whose only importance is the value under the natural transformation $\alpha : I \circ i \Rightarrow \gamma$, not the value under the functor $I : \mathcal{C} \to \mathsf{Set}$.

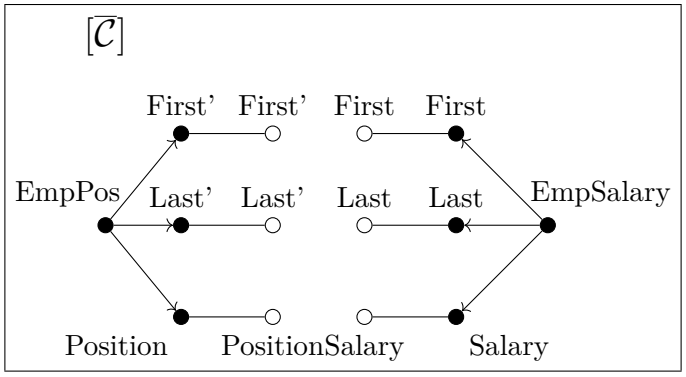
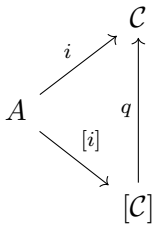


Figure 10: Another approach to joins along attributes using "merged" schemas; c.f. Figure 8.

We can use our data migration operations to transfer between these schemas: we can define a functor $q : [\mathcal{C}] \to \mathcal{C}$ that maps all of the objects from A back to the object in $\mathcal{C}$ they are assigned to; explicitly:

$$q(X) := \begin{cases} X & \text{if } X \in \mathcal{C} \\ i(X) & \text{if } X \in A \end{cases}$$

This gives a typed schema morphism:



so we can use the functor $\Delta_{\overline{q}}$ to transform a $\overline{\mathcal{C}}$-instance into a $[\overline{\mathcal{C}}]$-instance. Explicitly, given a $\overline{\mathcal{C}}$-instance $I : \mathcal{C} \to \mathsf{Set}$, for $a \in A$ we have $I(a) = I(i(a))$, so each new object is a copy of the object it was assigned to by i.

This functor has a left inverse Q:[1] intuitively, we compose along the natural inclusion $\iota : \mathcal{C} \hookrightarrow [\mathcal{C}]$; however, this inclusion doesn't give a typed schema morphism (attributes are moved). Explicitly, given an instance $I : [\mathcal{C}] \to \mathsf{Set}$, $\alpha : I \circ [i] \Rightarrow \gamma$ we get an instance $\underline{I} : \mathcal{C} \to \mathsf{Set}$, $\underline{\alpha} : \underline{I} \circ i \Rightarrow \gamma$ by setting $\underline{I} := I \circ \iota$ and $\underline{\alpha}_a := \alpha_a \circ I(i(a) \to a)$.

Suppose we want to join the schema $\overline{\mathcal{C}}$ in Figure 8 along First and Last. We proceed in a few steps.

First, form the merged schema $[\overline{\mathcal{C}}]$ in Figure 10. We move $\overline{\mathcal{C}}$ instances to this schema using $\Delta_{\overline{q}}$.

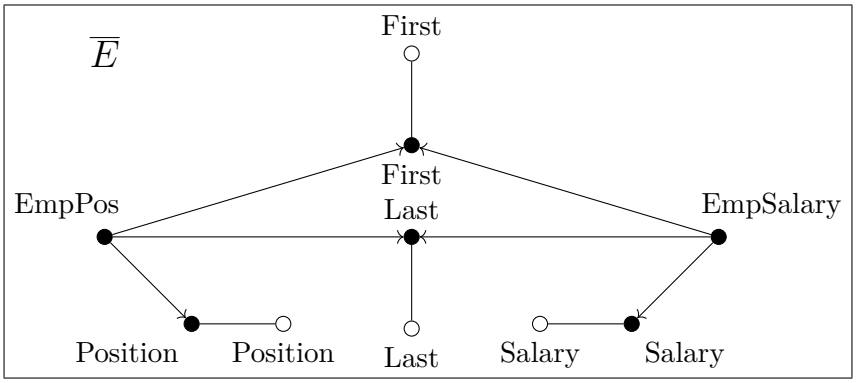


Figure 11: The category derived from Figure 10 by merging the First and Last objects.

Second, merge the attributes to be joined on and form a new schema: in our case, we merge [First and First'] and [Last and Last'] to get the schema $\mathcal{E}$ in Figure 11. We then move an instance from $[\overline{\mathcal{C}}]$ to $\mathcal{E}$ by taking the union $\Sigma_{\overline{F}}$ along the natural projection $F : [\overline{\mathcal{C}}] \to \mathcal{E}$ (since $\mathcal{E}$ is really a quotient of $[\overline{\mathcal{C}}]$). One may verify that this projection and the associated projection between the two attribute categories satisfy the conditions of Proposition 3.16. The effect of this step is to merge the [First and First'] columns (and likewise the other) into one object.

Third, "dedupe" the attributes to be joined on. In taking this "merged schema" approach, we are converting attributes into foreign keys and joining along these new foreign keys. We know that these foreign keys should be entirely determined by their value under the natural transformation: that is, we should identify those elements of $(\Sigma_{\overline{F}} \circ \Delta_{\overline{q}} \circ I)(\text{First})$ that correspond to the same actual value, and likewise for Last. We do this by defining a new map[2] D taking an instance I, α to the "deduped" instance along First and Last; more generally, if $\mathfrak{a} \subseteq A$ is a subset of attributes, we define $D_{\mathfrak{a}}$ as:

$$D_{\mathfrak{a}} I(X) := \begin{cases} I(X) & \text{if } X \notin i(\mathfrak{a}) \\ \operatorname{Im} \alpha_a & \text{if } X = i(a),\ a \in \mathfrak{a} \end{cases}$$

and on the interesting morphisms:

$$D_{\mathfrak{a}} I(Y \to i(a)) := \alpha_a \circ I(Y \to i(a))$$

The effect is to identify any elements of the given concrete columns (i.e. meaningless IDs) that have the same actual value.

Fourth, conduct the actual join: define the destination schema $\overline{\mathcal{C}'}$, form the merged schema $[\overline{\mathcal{C}'}]$ (in Figure 12), and conduct the join along the natural functor $G : \mathcal{E} \to \mathcal{C}'$, given by $\Pi_{\overline{G}}$.

Finally, use the functor Q to return to the unmerged schema $\mathcal{C}'$ (in Figure 13).

[1] The author suspects that this functor is $Q = \Pi_q = \Sigma_q$.

[2] This map can likely be extended to a functor in the evident way.

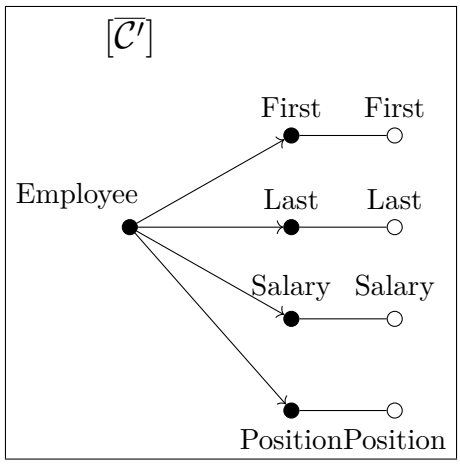


Figure 12: The final merged category in our join; the original category is in Figure 13.

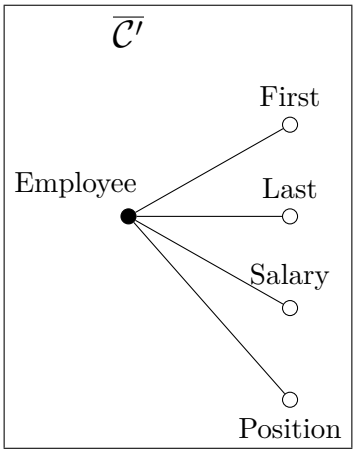


Figure 13: The final category in our join.

Applying this rather lengthy procedure to our example instance in Table 2 gives us exactly what we originally wanted: the instance given in Table 3. This time, due to the added structure of the natural transformations and the conditions placed on functors, that is the only possible result (excepting changes in ID values).

We can use a similar procedure to use the approach in [SW15] (i.e. the revised schema in Figure 9): convert ordinary schemas to the "floral" form, do the operations along attributes, and convert back. If the desired operations "respect" the typing of the attributes, the loss of typing in the intermediate stages can be undone at the end; however, the method outlined here of merged categories maintains the typing information the entire way through.

3.5 Smaller Problems

This theory of databases is significantly developed in [SW15]; it shows that many data migrations can be represented as a composition of a join, a union, and a projection, and (using the construction in Figure 9) proves that this theory implements any operations possible in conventional relational algebra. It also discusses in detail the semantics and implementation of various queries that can be

conducted using this construction. There is an implementation of this perspective as a computer program called FQL: it has been superseded by CQL (at [cql]), but can be found at [SSW].

This perspective of databases fixes the problems outlined in Section 2.4: joins and unions are forced to behave properly, and we can also enforce data types for each attribute. Nonetheless, this method has obvious pitfalls; namely, the difficulty in constructing the data migration functors. We needed to apply certain conditions to our functors between schemas in order to ensure their existence and their proper behaviour. We also worked primarily with *discrete* attribute categories: this means that we can add path equivalences for foreign keys, but not for attributes; it would be advantageous to be able to constrain attributes as well. The final development explored in this paper, profunctors, addresses these issues.

4 Databases via Algebraic Profunctors

We can use a concept called "profunctors" to fix the deficiencies of typed databases; in this section, we sketch out the basic constructions in [SSVW16].

The idea of a profunctor is to generalize functors, just as relations between sets generalize functions between sets; some literature calls profunctors "distributors" in analogy with this relation. This generalization is discussed in detail in Appendix A, in which composition of profunctors is also discussed and justified.

4.1 Profunctors and Algebraic Theories

The definition of a profunctor is rather simple:

Definition 4.1. Let A and B be categories. A *profunctor from A to B*, denoted $F : A \nrightarrow B$, is a functor $F : A^{\mathsf{op}} \times B \to \mathsf{Set}$. ⧫

The algebraic literature typically reverses the order, i.e. a profunctor from A to B is a functor $B^{\mathsf{op}} \times A \to \mathsf{Set}$; this difference is explained in Appendix A and is relevant only to the intuition.

The second piece of this view, the "algebraic" part, comes from the theory of algebraic theories.

Definition 4.2. An *algebraic theory* is a category $\mathcal{T}$ with finite products and a collection $S_{\mathcal{T}}$ of *base sorts* such that every object in $\mathcal{T}$ is uniquely a finite product of elements of $S_{\mathcal{T}}$.

Given two algebraic theories $\mathcal{T}, \mathcal{S}$, a *morphism of algebraic theories* $\mathcal{T} \to \mathcal{S}$ is a finite-product-preserving functor $F : \mathcal{T} \to \mathcal{S}$ such that $F(t) \in S_{\mathcal{S}}$ for each base sort $t \in S_{\mathcal{T}}$, i.e. F preserves base sorts. ⧫

These algebraic theories essentially "model" a specific type of algebraic object: for example, to represent the theory of groups using an algebraic theory $\mathcal{T}$, we would have a single base sort $S_{\mathcal{T}} := \{x\}$; every object of $\mathcal{T}$ would be a finite product of x, i.e. x^n for $n \geq 0$. We would also have a binary morphism multiplication $\cdot : x \times x \to x$, a unary morphism inversion $\cdot^{-1} : x \to x$, and a nullary morphism representing the identity $e : () = x^0 \to x$. The morphisms in $\mathcal{T}$ would be freely generated by these morphisms, quotiented by the relations expressing associativity of multiplication, that $e : () \to x$ is the identity, and that the product of an element with its inverse is the identity. We can use this to define groups:

Definition 4.3. Let $\mathcal{T}$ be an algebraic theory. An *algebra* or *model* of $\mathcal{T}$ is a finite-product-preserving functor $\mathcal{T} \to \mathsf{Set}$.

Let $F, G : \mathcal{T} \to \mathsf{Set}$ be algebras. A *$\mathcal{T}$-algebra morphism* $F \to G$ is a natural transformation $\alpha : F \Rightarrow G$.

The *category of $\mathcal{T}$-algebras*, denoted **$\mathcal{T}$-Alg**, is the category whose objects are $\mathcal{T}$-algebras and morphisms are $\mathcal{T}$-algebra morphisms. ⧫

That is, **$\mathcal{T}$-Alg** is the full subcategory (not all objects, but all morphisms between included objects) of $\mathsf{Set}^{\mathcal{T}}$ whose objects are the $\mathcal{T}$-algebras.

Definition 4.4. Let $\mathcal{T}$ be an algebraic theory and $\mathcal{C}$ a category. An *algebraic profunctor* $\mathcal{C} \nrightarrow \mathcal{T}$ is a profunctor $F : \mathcal{C} \nrightarrow \mathcal{T}$ such that for each $X \in \mathcal{C}$, the *functor* $F(X, -) : \mathcal{T} \to \mathsf{Set}$ is a $\mathcal{T}$-algebra. ⧫

4.2 Schemas as Algebraic Profunctors

We now define a schema as:

Definition 4.5. Let $\mathcal{T}$ be an algebraic theory. A *schema* is a pair (S_e, S_0) of a category S_e, called the *entity category*, and an algebraic profunctor $S_0 : S_e \nrightarrow \mathcal{T}$, called the *observables profunctor*. ♦

Comparing with a typed database schema $\overline{\mathcal{C}} = (\mathcal{C}, A, i, \gamma)$, S_e plays the exact same role as the entity category $\mathcal{C}$ while $S_0 : S_e \nrightarrow \mathcal{T}$ replaces all of the typing data A, i, γ.

How does this describe a database schema? It will be helpful to define a specific algebraic theory, **Type**. It will have three base sorts: $S_{\mathbf{Type}} := \{\mathrm{Str}, \mathrm{Int}, \mathrm{Bool}\}$. The Int sort comes with two nullary morphisms $0, 1 : () \to \mathrm{Int}$ as well as addition (which lets us define 2 as $2 := +(1, 1)$ and likewise n) and multiplications, modulo all the relations that we wish to hold; Bool is likewise, modeling boolean algebra, as Str models concatenation of strings. We can add whatever operations we want to this category: we may wish to check equality of strings or compare integers, corresponding to morphisms $= : (\mathrm{Str}, \mathrm{Str}) \to \mathrm{Bool}$ and $\leq : (\mathrm{Int}, \mathrm{Int}) \to \mathrm{Bool}$, respectively, quotiented by appropriate relations.

The algebraic profunctor $S_0 : S_e \nrightarrow \mathcal{T}$ gives us our typing data: for each entity $e \in S_e$, $S_0(e, -)$ must be a **Type**-algebra, meaning for each sort $\{\mathrm{Str}, \mathrm{Int}, \mathrm{Bool}\}$ we are given a collection of *observables* about the entity e as well as functions between these observables corresponding to the morphisms in $\mathcal{S}_e$ and **Type**. For our running example started in Table 1, the Str observables corresponding to the entity Employee would be generated by First and Last and the Int observables would be generated by Salary.

These would not be all of the observables: we necessarily have functions between these sets of observables $\mathcal{S}_e(\mathrm{Employee}, \mathrm{Str})$, $\mathcal{S}_e(\mathrm{Employee}, \mathrm{Int})$, and $\mathcal{S}_e(\mathrm{Employee}, \mathrm{Bool})$ corresponding to the various morphisms in the **Type** category. For example, using string concatenation, we would have a Str observable for Employee corresponding to the concatenation of the first and last names, and a Bool observable corresponding to whether or not their name is Joe.

4.3 Collages and Instances

How do we represent instances using this formulation? We need the concept of the collage of a profunctor. This is defined more generally, but we will use only the instance our case of ordinary profunctors:

Definition 4.6. Let $F : \mathcal{C} \nrightarrow \mathcal{D}$ be a profunctor. The *collage of* F, denoted $\widetilde{F}$, is the category with

- Objects the disjoint union $\mathsf{Ob}\,\widetilde{F} := \mathsf{Ob}\,\mathcal{C} \sqcup \mathsf{Ob}\,\mathcal{D}$;
- Morphisms $X \to Y$ given by:

$$\widetilde{F}(X, Y) := \begin{cases} \mathcal{C}(X, Y) & \text{if } X, Y \in \mathcal{C} \\ F(X, Y) & \text{if } X \in \mathcal{C}, Y \in \mathcal{D} \\ \varnothing & \text{if } X \in \mathcal{D}, Y \in \mathcal{C} \\ \mathcal{D}(X, Y) & \text{if } X, Y \in \mathcal{D} \end{cases}$$

Composition is given by composition in $\mathcal{C}$, $\mathcal{D}$, and functoriality of F.

We have a natural functor $q : \widetilde{F} \to \mathbf{2}$ given via:

$$q(X) := \begin{cases} 0 & \text{if } X \in \mathcal{C} \\ 1 & \text{if } X \in \mathcal{D} \end{cases}$$

$$q(f : X \to Y) := \begin{cases} \mathrm{id}_0 & \text{if } X, Y \in \mathcal{C} \\ \mathrm{id}_1 & \text{if } X, Y \in \mathcal{D} \\ 0 \to 1 & \text{if } X \in \mathcal{C}, Y \in \mathcal{D} \end{cases}$$

and natural inclusions $i_{\mathcal{C}} : \mathcal{C} \hookrightarrow \widetilde{F}$, $i_{\mathcal{D}} : \mathcal{D} \hookrightarrow \widetilde{F}$ satisfying, with $\mathbf{2} := \{0 \to 1\}$ being the category with two objects $0, 1$ and one non-identity morphism $0 \to 1$:

$$\begin{array}{ccccc} \mathcal{C} & \xrightarrow{i_{\mathcal{C}}} & \widetilde{F} & \xleftarrow[i_{\mathcal{D}}]{} & \mathcal{D} \\ \downarrow & \lrcorner & \downarrow{\scriptstyle q} & \llcorner & \downarrow \\ \{*\} & \xrightarrow{0} & \mathbf{2} & \xleftarrow[1]{} & \{*\} \end{array}$$

♦

This is a more general form of the "merged typed schema" construction of Definition 3.17: we merge the two categories $\mathcal{C}$ and $\mathcal{D}$ and use the profunctor $F : \mathcal{C} \nrightarrow \mathcal{D}$ to define morphisms between objects of $\mathcal{C}$ and objects of $\mathcal{D}$; functoriality of F ensures that we can compose these "straddling" morphisms with morphisms in $\mathcal{C}$ and $\mathcal{D}$ on either end. Here the direction of the profunctor is important: we have no morphisms from objects of $\mathcal{D}$ to objects of $\mathcal{C}$.

We use this to define instances:

Definition 4.7. Let $\mathcal{S} := (S_e, S_o : S_e \nrightarrow \mathcal{T})$ be a schema. An *$\mathcal{S}$-instance* is a functor $I : \widetilde{S}_o \to \mathsf{Set}$ such that the restriction $I_t := I \circ i_{\mathcal{T}}$ along the type inclusion preserves finite products, i.e. is a $\mathcal{T}$-algebra. ♦

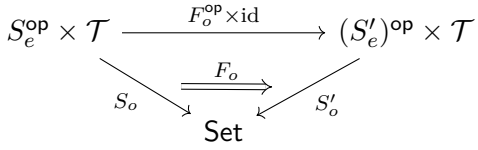
What does this get us? For each entity $e \in S_e$, we get a set of IDs, as usual. We also get a $\mathcal{T}$-algebra $I_t = I \circ i_{\mathcal{T}}$: in the case of $\mathcal{T} := \mathbf{Type}$, this means we get a set $I(\mathrm{Str})$ of things that behave like strings, $I(\mathrm{Int})$ behaving like integers, and $I(\mathrm{Bool})$ behaving like booleans. We also get functions between these sets corresponding to the morphisms in our schema: with our running example, we get functions $I(\mathrm{First}), I(\mathrm{Last}) : I(\mathrm{Employee}) \to I(\mathrm{Str})$, but also things like $I(\mathrm{FullName}) : I(\mathrm{Employee}) \to I(\mathrm{Str})$ which must be equal to the concatenation of their first and last name.

Thus, we elegantly recover the notion of typing using this construction, and more: we have access to all of the operations we would like to use on our data types, like integer addition and string concatenation, without having to define these explicitly in an attribute category as with typed databases.

4.4 Data Migration

The data migration functors also arise more naturally in this context, and they exist without restriction; we first need to define morphisms of schemas:

Definition 4.8. Let $\mathcal{S}, \mathcal{S}'$ be schemas with type $\mathcal{T}$. A *morphism of schemas* $\mathcal{S} \to \mathcal{S}'$ is a pair (F_e, F_o) of a functor functor $F_e : S_e \to S'_e$ and a natural transformation $F_o : S_o \Rightarrow S'_o \circ F_e^{\mathsf{op}} \times \mathrm{id}$:



♦

We can define the projection functor:

Definition 4.9. Let $\mathcal{S}, \mathcal{S}'$ be schemas and $F : \mathcal{S} \to \mathcal{S}'$ a schema morphism; let $\widetilde{F} : \widetilde{S}_o \to \widetilde{S}'_o$ be the induced functor between the collages. Given an instance $I' : \widetilde{S}'_o \to \mathsf{Set}$, we define:

$$\Delta_F(I) := I \circ \widetilde{F}$$

♦

In this situation, this functor actually has the necessary adjoints:

Proposition 4.10 (Propositions 7.3 and 7.4 in [SSVW16]). For any schema morphism $F : \mathcal{S} \to \mathcal{S}'$, the functor Δ_F has left and right adjoints, denoted Σ_F and Π_F, respectively. ♦

4.5 Further Development

For further reading in this perspective, the reader is directed to [SSVW16]: that paper gives a significantly more detailed treatment of the topic and excellent examples. This summary should suffice to provide the background necessary to understand the terminology and the important points of the proofs in Sections 6 and 7 of [SSVW16], where this theory of databases is developed.

5 Conclusion

In this report, we have developed three different theories for modeling relational databases using category theory, beginning with simple functor categories and ending with algebraic profunctors. Each interpretation arose naturally from objects already present in category theory, with little modification necessary to accomodate practical concerns.

Category theory has already found myriad applications within pure mathematics and theoretical computer science, but has yet to even be heard of in most practical fields. We have explored here an example of how category theory can be applied to model a concrete concept; this example is not isolated. Many authors have explored other beautiful applications; Fong and Spivak provide an excellent overview of seven in [FS18] applying many concepts of category theory.

The author hopes this report not only outlines such an application, but also provides convincing evidence that many more such applications can and will be found in the future.

References

[Bé00] Jean Bénabou. Distributors at work, 2000. Available at http://www.mathematik.tu-darmstadt.de/~streicher/FIBR/DiWo.pdf.

[CB15] Thomas Connolly and Carolyn Begg. Database Systems: A Practical Approach to Design, Implementation, and Management. Addison Wesley, 6 edition, 2015.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, jun 1970.

[cql] Categorial query language. Available at https://github.com/CategoricalData/CQL.

[FS18] Brendan Fong and David I. Spivak. An Invitation to Applied Category Theorem. Cambridge University Press, 2018.

[GUP05] Jose Galindo, Angelica Urrutia, and Mario Piattini. Fuzzy Databases: Modeling, Design, and Implementation. Idea Group, Hershey, PA, 2005.

[Rie14] Emily Riehl. Category Theory in Context. Dover, Mineola, NY, 2014. Available at https://emilyriehl.github.io/files/context.pdf.

[Spi12a] David I. Spivak. Functorial data migration. Information and Computation, 217:31–51, 2012. Available at https://arxiv.org/abs/1009.1166.

[Spi12b] David I. Spivak. Kleisli database instances. CoRR, abs/1209.1011, 2012. Available at http://arxiv.org/abs/1209.1011.

[Spi14] David I. Spivak. Category Theory for the Sciences. MIT Press, 2014.

[SSVW16] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic databases. Theory and Applications of Categories, 32:547–619, 2016. Available at http://www.tac.mta.ca/tac/volumes/32/16/32-16.pdf.

[SSW] Patrick Schultz, David I. Spivak, and Ryan Wisnesky. Functional query language. Available at https://github.com/CategoricalData/FQL.

[SW15] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. Proceedings of the International Symposium on Database Programming Languages (DBPL), 2015. Available at http://arxiv.org/abs/1212.5303.

A Motivating Profunctors

The concept of a profunctor is a simultaneous generalization of two seeming unrelated concepts: relations between sets and bimodules. This generalization is outlined in a number of papers, e.g. [Bé00], but requires more detail to be convincing; this is the goal of this appendix.

A.1 Profunctors as Generalized Relations

We roughly follow the abstract portions of Chapter 2 in [FS18].

First, we need to view a relation between two sets A and B in the correct way. The usual way is:

Definition A.1. A relation from A to B is a subset of $A \times B$. ⧫

The more helpful way is:

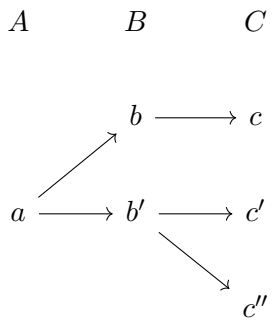
Definition A.2. A relation from A to B is a function $A \to \mathcal{P}(B)$. ⧫

It is not too hard to see that these are the same thing: essentially, we assign to each element of A the subset of B consisting of elements to which it is related. Abstractly, we use two correspondences. The first is subsets of a set B and arbitrary functions $B \to \mathbf{2}$, where $\mathbf{2} := \{\text{false}, \text{true}\}$ is a two-element set. The second is the fact that Set is cartesian-closed, so we have the "currying" correspondence:

$$A \times B \to \mathbf{2} \leftrightarrow A \to \mathbf{2}^B$$

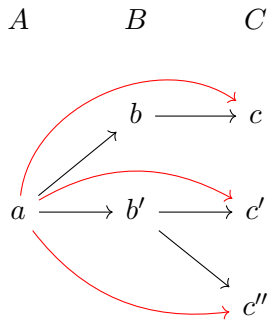
and we have a bijection (isomorphism) $\mathbf{2}^B \cong \mathcal{P}(B)$ by sending a subset $X \subseteq B$ to the function $B \to \mathbf{2}$ taking the value "true" precisely on the elements of X.

Suppose we have a relation from A to B, call it R_1, and a relation from B to C, call it R_2. Suppose for $a \in A$, $b, b' \in B$, and $c, c', c'' \in C$ we have the following relations:



i.e. $a\ R_1\ b$, $a\ R_1\ b'$, $b\ R_2\ c$, $b'\ R_2\ c'$, and $b'\ R_2\ c''$. One natural way of "composing" these two relations would be by "composing" the relations exactly as the diagram suggests, i.e. we define the

new relation R_3 from A to C by the red arrows:



Formally, we would define our new relation as:

$$R_3 := \{(x, z) \in A \times C : (x, b) \in R_1 \text{ and } (b, z) \in R_2 \text{ for some } b \in B\}$$

We can generalize this further to relations on partial orders.

Definition A.3. A partial order on a set A is a subset $\leq$ of $A \times A$ such that the following are satisfied for all $a, b, c \in A$:

1. Reflexivity: $a \leq a$.
2. Antisymmetry: if $a \leq b$ and $b \leq a$ then $a = b$.
3. Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$.

For partial orders $(A, \leq)$ and $(B, \preceq)$, a monotone map $f : A \to B$ is a function $f : A \to B$ such that for all $a, b \in A$, if $a \leq b$ then $f(a) \leq f(b)$.

The partial orders and monotone maps form a category Poset. ⧫

Consider partial orders $(A, \leq)$ and $(B, \preceq)$. If A and B were just sets, a relation between them would be an arbitrary function $A \to \mathbf{2}^B$; for partial orders, it makes sense to consider something like monotone maps $A \to \mathbf{2}^B$. The object $\mathbf{2}^B$ is the collection of all monotone maps $B \to \mathbf{2}$, with $f \leq g$ if $f(x) \leq g(x)$ for all $x \in B$. We can describe this in a different way:

Definition A.4. Let $(A, \leq) \in \mathsf{Poset}$. A subset $X \subseteq A$ is called upward-closed if for all $y \in A$, if $x \leq y$ for some $x \in X$ then $y \in X$.

The collection of upward-closed subsets of A, denoted $\uparrow A$, is:

$$\uparrow A := \{X \in \mathcal{P}(A) : X \text{ is upward-closed}\}$$

This has a natural partial order given by superset-containment, i.e. $X \leq Y$ if $Y \subseteq X$ for $X, Y \in \uparrow A$.

A primitive upward-closed subset of A is a set of the form

$$\uparrow a := \{x \in A : a \leq x\}$$

for some $a \in A$. ⧫

Lemma A.5. For $(A, \leq) \in \mathsf{Poset}$ we have $\mathbf{2}^A \cong \uparrow A$. ⧫

Proof. We first define a bijection between them. Let $f : A \to \mathbf{2}$ be monotone; this corresponds uniquely to a subset $F \subseteq A$. We claim that F is upward-closed. Let $y \in A$ with $x \leq y$ for some $x \in F$. Then $f(x) = \text{true}$ and f is monotone so $f(x) \leq f(y)$ and thus $f(y) = \text{true}$, so $y \in F$; we do the same to show that an upward-closed subset defines a monotone map $A \to \mathbf{2}$. That these maps are monotone is immediate from the correspondence of subsets and functions $A \to \mathbf{2}$. □

Generalizing slightly:

Lemma A.6. Let A, B, C be posets. We have an isomorphism between monotone maps $A \times B \to C$ and monotone maps $B \to C^A$ (where C^A consists of monotone maps $A \to C$ where $f \leq g$ if $f(x) \leq g(x)$ for all $x \in A$):

$$\mathsf{Hom}(A \times B, C) \cong \mathsf{Hom}(B, C^A)$$

natural in both B and C, called the "currying" isomorphism. ♦

This will be seen as a special case of certain enriched categorical constructions seen in a later section.

Now we wish to try and define relations on partial orders. Our first naïve attempt might be to consider monotone maps $A \times B \to \mathbf{2}$; however, we need to be a bit more careful.

Suppose we have two posets A and B, each with two elements $a \leq a'$ and $b \leq b'$, respectively, and we try to define a relation $\varphi : A \times B \to \mathbf{2}$ by declaring that a is related to b, i.e. we set $\varphi(a, b) := \text{true}$. We are in the following situation:

$$\begin{array}{ccc} A & & B \\ \\ a & \longrightarrow & b \\ \downarrow & & \downarrow \\ a' & & b' \end{array}$$

We have the added structure that now we not only have a notion of horizontal composition (composing a relation from A to B with one from B to C), but also a combined sort of composition of vertical morphisms and horizontal relations: since $b \leq b'$, in $A \times B$ we have $(a, b) \leq (a, b')$; thus, by monotonicity of φ we must also set $\varphi(a, b') := \text{true}$. The trouble comes with composing in A: suppose we instead start with defining $\varphi(a', b') := \text{true}$:

$$\begin{array}{ccc} A & & B \\ \\ a & & b \\ \downarrow & & \downarrow \\ a' & \longrightarrow & b' \end{array}$$

Then it makes sense intuitively for us to want $\varphi(a, b') = \text{true}$, since we can get from a to a' in A and from a' to b' using the relation. However, this is not guaranteed by monotonicity: monotonicity only says that $\varphi(a, b') \leq \varphi(a', b')$. Thus, what we really want to do is to generalize a relation as a monotone map $A^{\mathsf{op}} \times B \to \mathbf{2}$, to take this into account.

Definition A.7. Let A and B be posets. A relation from A to B is a monotone map $A^{\mathsf{op}} \times B \to \mathbf{2}$.♦

Via the curring isomorphism, we could equivalently talk about monotone maps $B \to \mathbf{2}^{A^{\mathsf{op}}}$.

Definition A.8. Let $(A, \leq) \in \mathsf{Poset}$. A subset $X \subseteq A$ is called downward-closed if for all $y \in A$, if $y \leq x$ for some $x \in X$ then $y \in X$.

The set of downward-closed subsets of A, denotes $\downarrow A$, is the set:

$$\downarrow A := \{X \in \mathcal{P}(A) : X \text{ is downward-closed}\}$$

This has a natural partial order given by subset-containment, i.e. $X \leq Y$ if $Y \subseteq X$ for $X, Y \in \uparrow A$.

A primitive downward-closed subset of A is a set of the form

$$\downarrow a := \{x \in A : x \leq a\}$$

for some $a \in A$. ⧫

As with upward-closed sets:

Lemma A.9. $\downarrow A \cong \mathbf{2}^{A^{\mathsf{op}}}$. ⧫

Using this, we can show that our concept of a relation between posets does indeed generalize the concept of relations on sets: for any set A, we can turn A into a discrete poset by declaring that $a \leq b$ iff $a = b$. We also have a natural poset structure on $\mathcal{P}(A)$ given by subset inclusion; when a set A becomes a poset in this fashion, we have $\downarrow A \cong \mathcal{P}(A)$. Thus, if B is another set then a relation from A to B is precisely a monotone map $B \to \mathcal{P}(A)$, which is precisely a function $B \to \mathcal{P}(A)$ since B is a discrete poset.

We can generalize composition in two ways: we can do it directly, or we can exploit the monadic structure that we've been ignoring so far.

We viewed relations between sets A and B as functions $B \to \mathcal{P}(A)$, and relations between posets A and B as monotone maps $B \to \downarrow A$; both the covariant power set functor $\mathcal{P} : \mathsf{Set} \to \mathsf{Set}$ and the downward-closed-subsets functor $\downarrow\colon \mathsf{Poset} \to \mathsf{Poset}$ form monads on their respective categories.

Consider the covariant power set functor $\mathcal{P} : \mathsf{Set} \to \mathsf{Set}$. The action on a map $f : A \to B$ is given by the direct image:

$$(\mathcal{P}f)(X \subseteq A) := \{f(x) : x \in X\} \subseteq B$$

the multiplication natural transformation $\mu : \mathcal{P}^2 \Rightarrow \mathcal{P}$ is given by union:

$$\mu_A(\{A_i : i \in I, A_i \subseteq A\}) := \bigcup_{i \in I} A_i$$

and the unit natural transformation $\eta : \mathrm{id}_{\mathsf{Set}} \Rightarrow \mathcal{P}$ is given by singleton sets:

$$\eta_A(a) := \{a\}$$

For $\downarrow\colon \mathsf{Poset} \to \mathsf{Poset}$, the action of the functor on a monotone map $f : A \to B$ is given by smallest downward-closed subset containing the direct image:

$$(\downarrow f)(X \subseteq A \text{ downward-closed}) := \{b \in B : b \leq f(x) \text{ for some } x \in X\}$$

the multiplication natural transformation $\mu : \downarrow\downarrow \Rightarrow \downarrow$ is given by union:

$$\mu_A(\{A_i : i \in I, A_i \subseteq A \text{ downward-closed}\}) := \bigcup_{i \in I} A_i$$

observing that the result is downward-closed, and the unit natural transformation $\eta : \mathrm{id}_{\mathsf{Poset}} \Rightarrow \downarrow$ is given by

$$\eta_A(a) := \{x \in A : x \leq a\}$$

The final observation is that composition in both cases is given by composition in the Kleisli categories of the monads (recall Definition 2.21). The morphisms in the Kleisli category are almost our relations: a relation from A to B (sets) is a map $B \to \mathcal{P}(A)$ (using our convention), and likewise for posets; to use our convention, which is natural for composing relations, we really need to work in the *opposite* of the Kleisli category.

We will now compute explicitly the composition in the opposite Kleisli category of $\mathcal{P}$, and show it agrees with our original notion of composition. Suppose we have two relations: one from A to B, $f : B \to \mathcal{P}(A)$, and one from B to C, $g : C \to \mathcal{P}(B)$. Then we need to compute $f \circ g$ in the regular Kleisli category:

$$\begin{aligned}(f \circ g)(c) :&= \mu((\mathcal{P}f)(g(c))) \\ &= \mu(\{f(b) : b \in g(c)\}) \\ &= \bigcup_{b \in g(c)} f(b)\end{aligned}$$

which is exactly what we wanted: $c \in C$ is related to every $a \in A$ such that a is related to some $b \in B$ (i.e. $a \in f(b)$) that is related to c (i.e. $b \in g(c)$).

Generalizing by analogy, what do we get when we compute composition in the Kleisli category for $\downarrow$? Let A, B, C be posets, $f : B \to\downarrow A$ a relation from A to B, and $g : C \to\downarrow B$ a relation from B to C. We need to compute two things: where the composition $f \circ g$ sends each $c \in C$:

$$\begin{aligned}(f \circ g)(c) :&= \mu((\downarrow f)(g(c))) \\ &= \mu(\{\alpha \in\downarrow A : \alpha \subseteq f(b) \text{ for some } b \in g(c)\}) \\ &= \{a \in A : a \in f(b) \text{ for some } b \in g(c)\} \\ &= \bigcup_{b \in g(c)} f(b)\end{aligned}$$

with the same interpretation as in the powerset case. Converting back to viewing a relation as a monotone map $A^{\mathsf{op}} \times B \to \mathbf{2}$, we can compute this explicitly:

$$(f \circ g)(a, c) = \bigvee_{b \in B} (f(a, b) \wedge g(b, c))$$

That is, a and c are related in the composite iff $a\ f\ b$ and $b\ g\ c$ for some $b \in B$, as we expected; by the same logic, this formula also holds for the powerset.

A.2 Profunctors as Generalized Bimodules

A similar number of hoops are required to generalize bimodules in this fashion.

Definition A.10. Let S, R be rings. An (S, R)-bimodule is an abelian group M such that M is a left S-module and a right R-module and the two actions are compatible:

1. For all $s \in S$, $r \in R$, $x \in M$ we have $s \cdot (m \cdot r) = (s \cdot m) \cdot r$. ♦

We have a notion of "composing" two modules: given an (S, R)-bimodule M and an (R, T)-bimodule N, we can form the tensor product over R:

$$M \otimes_R N$$

which naturally becomes an (S, T)-bimodule; viewing the (S, R)-bimodule M as some sort of arrow $M : S \nrightarrow R$ and N as $N : R \nrightarrow T$, the result is $M \otimes_R N : S \nrightarrow T$. This "composition" is not literally associative, but is associative up to a unique isomorphism since the tensor product is a universal object.

In this setting, "composition" is much more natural; viewing this construction in terms of functors $?\times? \to ?$ is more complicated. This begins by considering the following statement:

> " 'A monad is a monoid in the category of endofunctors, what's the problem?'
> –Philip Wadler"
> –James Iry

in an even less helpful instance:

> "A ring is a monoid in the category of abelian groups, what's the problem?"
> –Nobody sane

Unhelpful for an intuitive understanding of a ring, but helpful for understanding profunctors. We need to make a brief detour into enriched category theory to understand this statement.

Definition A.11. A monoidal category is a triple $(\mathcal{C}, \otimes, I)$ where

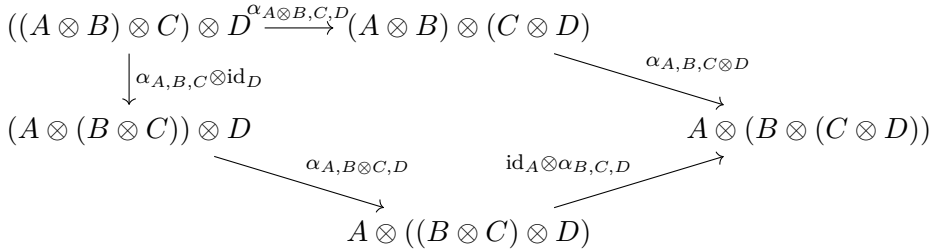
- $\mathcal{C}$ is a category;
- $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a bifunctor, with $\otimes(A, B)$ typically written as $A \otimes B$;
- $I \in \mathcal{C}$ is any object, called the unit object

such that $\otimes$ is associative up to natural isomorphism and I is a left and right identity for $\otimes$ up to natural isomorphism, subject to coherence axioms.

Specifically, we have natural isomorphisms

- $\alpha : (- \otimes -) \otimes - \cong - \otimes (- \otimes -)$ called the associator;
- $\lambda : I \otimes - \cong -$ called the left unitor;
- $\rho : - \otimes I \cong -$ called the right unitor

that satisfy the "MacLane pentagon"



and the triangle identity

$$\begin{array}{ccc} (A \otimes I) \otimes B & \xrightarrow{\alpha_{A,I,B}} & A \otimes (I \otimes B) \\ & {\scriptstyle \rho_A \otimes \mathrm{id}_B} \searrow \quad \swarrow {\scriptstyle \mathrm{id}_A \otimes \lambda_B} & \\ & A \otimes B & \end{array}$$

♦

This is best understood through the examples:

- Cat or Set with the usual $\times$; the unit object is the singleton set/category, the associator witnesses the idea that $(a, (b, c))$ is morally the same as $((a, b), c)$, and the unitors witness that $(a, *)$ and $(*, a)$ are morally the same as a. Even more generally, this construction works on any category with (binary) products and a terminal object.
- The category of abelian groups Ab with the direct sum $\oplus$, since $\oplus$ is a product and the zero group 0 is both terminal and initial.
- Ab with the tensor product of abelian groups $\otimes$; in this case, the unit is the integers $\mathbb{Z}$.

The first two are the simplest to work with, but the last is what will help us parse the statement.

We have two equivalent descriptions of the tensor product: a universal property and a construction. For completeness, we define the tensor product of modules; abelian groups are precisely $\mathbb{Z}$-modules.

Definition A.12. Let A be a right R-module, B a left R-module, and C an abelian group. A map $\varphi : A \times B \to C$ is called R-balanced if for all $a, a' \in A$, $b, b' \in B$, and $r \in R$

1. $\varphi(a + a', b) = \varphi(a, b) + \varphi(a', b)$;
2. $\varphi(a, b + b') = \varphi(a, b) + \varphi(a, b')$;
3. $\varphi(a \cdot r, b) = \varphi(a, r \cdot b)$.

♦

Lemma A.13. Let A be a right R-module and B a left R-module. The tensor product $A \otimes_R B$, unique up to unique isomorphism, is the abelian group such that for any abelian group C and R-balanced map $f : A \times B \to C$ we have a unique map:

$$\begin{array}{ccc} & & A \otimes B \\ & \nearrow & \downarrow {\scriptstyle \exists!} \\ A \times B & \xrightarrow{f} & C \end{array}$$

We may construct $A \otimes B$ as the quotient of $R(A \times B)/\ \sim$, where $R(A \times B)$ is the free R-module on $A \times B$ and $\sim$ is the smallest equivalence relation generated by the following:

- $(a + a', b) \sim (a, b) + (a', b)$;
- $(a, b + b') \sim (a, b) + (a, b')$;
- $(a \cdot r, b) \sim (a, r \cdot b)$.

If A is an (S,R)-bimodule and B is an (R,T)-bimodule, the tensor product $A \otimes_R B$ is naturally an (S,T)-bimodule by considering the R-balanced maps $(a,b) \mapsto (s \cdot a, b)$ and $(a,b) \mapsto (a, b \cdot t)$ for $s \in S$ and $t \in T$. ⧫

Corollary A.14. For a fixed (R,T)-bimodule B we have an adjunction

$$- \otimes_R B : \mathsf{Mod} - R \leftrightarrows \mathsf{Mod} - S : \mathsf{Hom}_R(B, -)$$

and likewise for left modules. ⧫

Definition A.15. Let $(\mathcal{C}, \otimes, I)$ be a monoidal category. A monoid object in $\mathcal{C}$ is an object $M \in \mathsf{Ob}\,\mathcal{C}$ with a multiplication morphism $\mu : M \otimes M \to M$ and an identity morphism $\eta : I \to M$ satisfying

- Associativity:

$$\begin{array}{ccccc} (M \otimes M) \otimes M & \xrightarrow{\alpha} & M \otimes (M \otimes M) & \xrightarrow{1 \otimes \mu} & M \otimes M \\ & \searrow^{\mu \otimes \mathrm{id}_M} & & & \downarrow \mu \\ & & M \otimes M & \xrightarrow{\mu} & M \end{array}$$

- Unit laws:

$$\begin{array}{ccccc} I \otimes M & \xrightarrow{\eta \otimes \mathrm{id}_M} & M \otimes M & \xleftarrow{\mathrm{id} \otimes \eta} & M \otimes I \\ & \searrow^{\lambda} & \downarrow \mu & \swarrow_{\rho} & \\ & & M & & \end{array}$$

A morphism of monoids $(M, \mu, \eta) \to (M', \mu', \eta')$ is a morphism $f : M \to M'$ such that $f \circ \mu = \mu' \circ (f \otimes f)$ and $f \circ \eta = \eta'$. ⧫

The only way to understand these definitions is to compute a few examples with favourite monoids. We have three important examples:

- Monads on a category $(\mathcal{C}, \times)$ are precisely monoids in the category of endofunctors $\mathcal{C}^{\mathcal{C}}$.
- Monoid objects in $(\mathsf{Set}, \times)$ are precisely the usual monoids.
- Monoid objects in $(\mathsf{Ab}, \otimes)$ are precisely rings. Distributivity and associativity of ring multiplication follows immediately from the axioms and the requirement that each induced map $x \mapsto r \times x$ and $x \mapsto x \times r$ is a group homomorphism.

In all cases, the morphisms of monoids are exactly what you would suspect.

We now need the concept of an enriched category:

Definition A.16. Let $(\mathcal{C}, \otimes, I)$ be a monoidal category. A $\mathcal{C}$-category $\mathcal{D}$ (or category enriched over $\mathcal{C}$) consists of

- A set of objects of $\mathcal{D}$, $\mathsf{Ob}\,\mathcal{D}$;
- For each pair $A, B \in \mathsf{Ob}\,\mathcal{D}$ a Hom-object $\mathcal{D}(A,B) \in \mathsf{Ob}\,\mathcal{C}$;

- For each triple $A, B, C \in \mathsf{Ob}\,\mathcal{D}$ a composition map $\circ : \mathcal{D}(B,C) \otimes \mathcal{D}(A,B) \to \mathcal{D}(A,C) \in \mathsf{Mor}\,\mathcal{C}$;
- For each object $A \in \mathsf{Ob}\,\mathcal{D}$ a morphism $\mathrm{id}_A : I \to \mathcal{D}(A,A)$ called the identity map

subject to coherence diagrams (associativity of $\circ$ and unit laws) that won't be bothered with here. For two $\mathcal{C}$-enriched categories $\mathcal{D}$ and $\mathcal{E}$, a $\mathcal{C}$-functor $F : \mathcal{D} \to \mathcal{E}$ consists of

- For each object $A \in \mathsf{Ob}\,\mathcal{D}$ an object $FA \in \mathsf{Ob}\,\mathcal{E}$;
- For each pair $A, B \in \mathsf{Ob}\,\mathcal{D}$ a map $\mathcal{D}(A,B) \to \mathcal{E}(FA, FB) \in \mathsf{Mor}\,\mathcal{C}$

which respects composition and units (also diagrams that will not be bothered with). ⧫

Examples abound:

- Ordinary categories are precisely $(\mathsf{Set}, \times)$-enriched categories.
- Ab is Ab-enriched, since $\mathsf{Ab}(A,B)$ is naturally an abelian group.
- $R - \mathsf{Mod}$ is both Ab-enriched and $R - \mathsf{Mod}$ enriched.
- 2-categories are precisely $(\mathsf{Cat}, \times)$-enriched categories.

One example worth going into in detail is $\mathbf{2}$-enriched categories: $\mathbf{2} = \{\mathrm{false}, \mathrm{true}\}$ becomes a monoidal category using logical AND, $\wedge$, with true as the unit. The axioms tell us that for every object a we have an identity morphism $\mathrm{true} \leq \mathbf{2}(a,a)$; this is reflexivity, since it means $a \leq a$. For every triple of objects a, b, c we have a composition map $\mathbf{2}(b,c) \wedge \mathbf{2}(a,b) \leq \mathbf{2}(a,c)$, which is transitivity. Thus, $\mathbf{2}$-enriched categories are precisely the preoders, which are reflexive and transitive relations (posets without antisymmetry). For $\mathbf{2}$-functors, functoriality is essentially degenerate: it follows from transitivity of the categories and the fact that we have only zero or one morphism in each hom-object. The fact that a functor gives us a map between the hom-objects is precisely monotonicity, so functors correspond to monotone maps.

Specializing a bit, every poset can be viewed as a $\mathbf{2}$-category (a skeletal one, since antisymmetry ensures each isomorphism class has only one object).

Many concepts from ordinary category theory can be defined in enriched category theory: commutative diagrams, natural transformations, limits, colimits, etc. all have natural generalizations.

Definition A.17. Let $(\mathcal{C}, \otimes, I)$ be a monoidal category. We say that $\mathcal{C}$ is a closed monoidal category if for all $B \in \mathcal{C}$ the functor $- \otimes B : \mathcal{C} \to \mathcal{C}$ admits a right adjoint $\mathsf{Hom}(-, B) : \mathcal{C} \to \mathcal{C}$. ⧫

All of the examples of monoidal categories given are in fact closed monoidal categories.

We are now in a position to generalize bimodules; we begin by viewing rings as categories.

Definition A.18. Let (M, μ, η) be a monoid object in a closed monoidal category $(\mathcal{C}, \otimes, I)$. We turn M into a $\mathcal{C}$-category BM with

- The only object being M;

- The morphisms $M \to M$ being the image of μ under the currying isomorphism:

$$\mathcal{C}(M \otimes M, M) \cong \mathcal{C}(M, \mathcal{C}(M, M))$$ ♦

We can also go backwards via that same isomorphism: the monoids in $\mathcal{C}$ correspond to the single-object $\mathcal{C}$-enriched categories. One more step:

Lemma A.19. Let R be a ring. A left R-module is an Ab-enriched functor $BR \to \mathsf{Ab}$. ♦

This is proven by unpeeling the many layers of the onion: Ab-enrichedness gives distributivity, functoriality gives associativity of $\cdot$, etc. Now we can finally get to the candidate for generalization:

Lemma A.20. For rings R and S, an (S, R)-bimodule is precisely an Ab-enriched functor $R^{\mathsf{op}} \times S \to \mathsf{Ab}$. ♦

This is the source of the confusion between $S^{\mathsf{op}} \times R$ and $R^{\mathsf{op}} \times S$ for profunctors $S \nrightarrow R$: it depends on whether one considers covariant or contravariant functors (and hence left or right modules) to be the primitive notion, and is thus wholly irrelevant to the theory: a right R-module is precisely a left R^{op} module, where R^{op} is the ring with multiplication in the opposite order and addition unchanged.

The tensor product of bimodules gives us a way of "composing" two bimodules, associative and unital up to unique natural isomorphism. How can we do this? Given an (S, R)-bimodule and an (R, T)-bimodule, viewed as $F : S^{\mathsf{op}} \times R \to \mathsf{Ab}$ and $R^{\mathsf{op}} \times T \to \mathsf{Ab}$, how can we get an (S, T)-bimodule?

Thus, the generalization is as follows: instead of considering enriched functors $A^{\mathsf{op}} \times B \to \mathsf{Ab}$ of single-object Ab-categories A and B, why not consider enriched functors $A^{\mathsf{op}} \times B \to \mathcal{C}$ for arbitrary categories enriched over some monoidal category $(\mathcal{C}, \otimes, I)$? We then specialize this to the case $\mathcal{C} = \mathsf{Set}$ to consider functors $A^{\mathsf{op}} \times B \to \mathsf{Set}$ for categories A and B.

A.3 Composing Profunctors

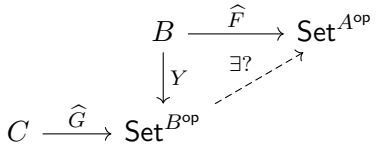
We now return to considering general profunctors $A^{\mathsf{op}} \times B \to \mathsf{Set}$. How can we compose them? Composition came up naturally in applications, but we need some abstract nonsense to compose them in general. Suppose we have two profunctors $F : A \nrightarrow B$, i.e. $F : A^{\mathsf{op}} \times B \to \mathsf{Set}$, and $G : B \nrightarrow C$, i.e. $G : B^{\mathsf{op}} \times C \to \mathsf{Set}$. We want some profunctor $F; G : A \nrightarrow C$. We have the following things at our disposal:

- The curried functor $\widehat{F} : B \to \mathsf{Set}^{A^{\mathsf{op}}}$;
- The curried functor $\widehat{G} : C \to \mathsf{Set}^{B^{\mathsf{op}}}$;
- The Yoneda embedding $Y : B \to \mathsf{Set}^{B^{\mathsf{op}}}$.

The only way to fit these together is:

$$\begin{array}{ccc} B & \xrightarrow{\widehat{F}} & \mathsf{Set}^{A^{\mathsf{op}}} \\ \downarrow{\scriptstyle Y} & & \\ C \xrightarrow{\widehat{G}} \mathsf{Set}^{B^{\mathsf{op}}} & & \end{array}$$

We want a functor $A^{\mathsf{op}} \times C \to \mathsf{Set}$, or equivalently $C \to \mathsf{Set}^{A^{\mathsf{op}}}$; we could do this as the bottom composite if we can find a dashed functor:



We can't always do this exactly, but we can approximate this best using either the left or right Kan extension (Definition 2.9) of $\widehat{F}$ along Y. Which Kan extension makes more sense in our case?

We need to move sideways to figure this out: consider instead **2**-enriched functors $A^{\mathsf{op}} \times B \to \mathbf{2}$ for **2**-enriched categories A and B; that is, return to the situation of posets. We have an "enriched Yoneda lemma" posets that says we can embed $A \hookrightarrow \mathbf{2}^{A^{\mathsf{op}}} = \downarrow A$. The contravariant Yoneda embedding by definition maps

$$Y(a) := \mathbf{2}(-, a)$$

With our interpretation of $\mathbf{2}^{A^{\mathsf{op}}}$ as $\downarrow A$, this is precisely

$$Y(a) := \{a' \in A : a \leq A\}$$

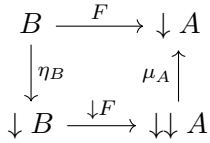
which is the unit η_A from our monad! To go backwards, a is the unique maximal element in $\eta(a)$.

To talk about Kan extensions, we also need a notion of natural transformation between functors (monotone maps) $F, G : A \to B$. In analogy with Cat, a natural transformation $\alpha : F \Rightarrow G$ amounts to specifying for each $a \in A$ a morphism $Fa \to Ga$, which in our case means asserting that $Fa \leq Ga$ for all a. Since we have either one or zero morphisms between any two objects, the naturality square that should commute is redundant for our case.

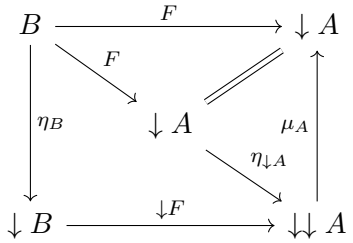
To compose the relations $F : A \nrightarrow B$ and $G : B \nrightarrow C$, we did:

$$C \xrightarrow{G} \downarrow B \xrightarrow{\downarrow F} \downarrow\downarrow A \xrightarrow{\mu_A} \downarrow A$$

so the dashed functor $E : \downarrow B \to \downarrow A$ must be the composite of the last two arrows, $\mu_A \circ \downarrow F$. Observe that the diagram



doesn't just commute up to some natural transformation, it actually commutes; by naturality of η the bottom triangle commutes:



Clearly the upper triangle commutes, and by the unit laws for the monad $\downarrow$ the right triangle commutes. So for this specific case, the identity provides a natural transformation $\mathrm{id}_A : F \Rightarrow E \circ Y$

and $\mathrm{id}_A : E \circ Y \Rightarrow F$. Suppose we have another such left extension $(K :\downarrow B \to\downarrow A, \alpha : F \Rightarrow K \circ Y)$. We claim that we have $E \leq K$, i.e. a natural transformation $\alpha : E \Rightarrow K$. We need to show this for all objects, so fix $X \in\downarrow B$. By assumption, $F \leq K$ so we have the inequality

$$K(X) \geq \bigcup_{b \in X} F(b)$$

by monotonicity of K, since we have $\alpha : F \Rightarrow K \circ Y$ so K must be at least F on all primitive downward-closed subsets of B. Since $K(X)$ is downward closed, it must also contain the smallest downward-closed subset containing the right side; this is $(\mu\circ \downarrow F)(X)$! Thus, we have a natural transformation $E \Rightarrow K$, so E must be the left Kan extension.

This agrees with the bimodule case; this will not be proven explicitly, but the intuition is as follows. We would define composition using the tensor product, so for bimodules the functor $- \otimes A$ for some fixed bimodule A would have a right adjoint; if we define composition using the left Kan extension, the precomposition functor $- \circ f$ will have a right adjoint (under some appropriate cocompletness assumptions on the involved categories). The same is true for precomposition in Cat (Kan extensions), so this also agrees with ordinary functor composition in that sense.

Alternatively, we can observe that we are dealing with a "pseudomonad" on Cat (and ignore size issues for the moment). For each small category $\mathcal{C}$, the free (small) cocompletion of $\mathcal{C}$ (that is, freely adjoining all colimits to $\mathcal{C}$) is $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$: the category of presheaves $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ is cocomplete and every presheaf $\mathcal{C}^{\mathsf{op}} \to \mathsf{Set}$ is canonically a limit of representable functors (for locally-small categories, $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ is no longer necessarily the free cocompletion, since we are only considering colimits of functors with small domains).

Consider the 2-category Cocomp consisting of cocomplete categories, cocontinuous functors (functors preserving colimits), and natural transformations. We can define a 2-functor (a functor that is functorial up to natural isomorphism) $P : \mathsf{Cat} \to \mathsf{Cocomp}$ via the Yoneda embedding: we set $P(\mathcal{C}) := \mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$, and we send a functor $F : \mathcal{C} \to \mathcal{D}$ to $\mathrm{Lan}_Y Y \circ F$:

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F} & \mathcal{D} \\ \downarrow{\scriptstyle Y} & & \downarrow{\scriptstyle Y} \\ \mathsf{Set}^{\mathcal{C}^{\mathsf{op}}} & \xrightarrow[\mathrm{Lan}_Y Y \circ F]{} & \mathsf{Set}^{\mathcal{D}^{\mathsf{op}}} \end{array}$$

or alternatively we send it to $\mathrm{Lan}_F^{\mathsf{op}} : \mathsf{Set}^{\mathcal{C}^{\mathsf{op}}} \to \mathsf{Set}^{\mathcal{D}^{\mathsf{op}}}$; since both share the same universal property, they are isomorphic. In particular, $PF \dashv F^*$ so PF preserves colimits and $PF : P\mathcal{C} \to P\mathcal{D}$ is well-defined. We need to deal with 2-functors because *a priori* we don't have $P(g \circ f) = Pg \circ Pf$, but will be true up to canonical natural isomorphism by uniqueness of colimits.

It turns out that this functor P defines a (pseudo) left adjoint to the forgetful functor $U : \mathsf{Cocomp} \to \mathsf{Cat}$ (ignoring size issues): we have a natural equivalence

$$\mathsf{Cocomp}(P\mathcal{C}, \mathcal{D}) \simeq \mathsf{Cat}(\mathcal{C}, U\mathcal{D})$$

that follows from the Yoneda lemma. Thus, we get a "pseudomonad" structure on Cat via the composite $U \circ P$. From the computations in the other monad and the definition of our functor P, it is intuitively clear that composition of $F : \mathcal{C} \to \mathsf{Set}^{\mathcal{D}^{\mathsf{op}}}$ and $G : \mathcal{D} \to \mathsf{Set}^{\mathcal{E}^{\mathsf{op}}}$ in some relaxed version of the Kliesli category for this pseudomonad will be given by (a factorization of)

$$\mathcal{C} \xrightarrow{F} \mathsf{Set}^{\mathcal{D}^{\mathsf{op}}} \xrightarrow{\mathrm{Lan}_Y G} \mathsf{Set}^{\mathcal{E}^{\mathsf{op}}}$$

exactly as it was in the poset case. Thus, our definition of composition of arbitrary profunctors is in some sense a direct generalization of the intuition given by the monads on Poset and Set.