# FUNCTORIAL DATA MIGRATION

DAVID I. SPIVAK

Abstract. In this paper we present a simple database definition language: that of categories and functors. A database schema is a category and a state is a set-valued functor. We show that morphisms of schemas induce three "data migration functors" that translate states from one schema to the other in canonical ways. Database states form a topos of which the classical "relational algebra" is a fragment. These ideas thus create a new denotational semantics for database theory.

Contents



## 1. Introduction

Database management is a huge industry, yet logical database design remains problematic. While the relational algebra is sufficient to understand single tables and operations that can be performed on them, it has no means to capture the structure of the database as a whole. The idea presented in this paper is that databases can be modeled by elementary category theory: a schema is a category $\mathcal{C}$ and database state on that schema is a functor $\mathcal{C} \to \mathbf{Set}$ (see Definition 1.1.1). This categorical understanding can be considered a new *denotational semantics* for databases.

While there have been many "categorificiations" of databases in the past (e.g. [RW],[JRW],[PS],[Ber],[DK],[Dis],[GB]), none of them has been so simple as this one. The main contribution we offer here is that one should consider data columns as foreign keys to 1-column tables; in this way a schema is a category (or a presentation thereof). For more on this, the reader can see slides and a video at http://www.galois.com/blog/2010/05/27/tech-talk-categories-are-databases/. Another categorical approach was given in [S1] and [S2]. That approach is more appropriate for assembling facts from a variety of sources (e.g. as in the semantic

This project was supported by ONR grant: N000141010841 and a generous contribution by Clark Barwick, Jacob Lurie, and the mathematics department at MIT.

web vision), whereas the present approach is more appropriate for modeling a local situation, (e.g. as in business).

The goal of this paper is to show the simplicity of the categorical model and the power of this simplicity by presenting a rigorous mathematical notion of data migration. Data migration is the process of transforming data modeled by schema $\mathcal{C}$ into data modeled by schema $\mathcal{D}$. As things currently stand, it is difficult to set up this process, let alone prove anything about its consistency. In order to transfer data, say into a data warehouse using the ETL process, one must know the structure of the database schemas at both ends and connect them in the appropriate ways. In the categorical model, this is simply finding a functor between the two schemas. This paper describes how, once such a functor is provided, data can be sent back and forth in canonical ways. The "data migration functors" presented here have good formal properties, which one can use to prove that a data transfer will have the desired results. These ideas may also be useful in studying issues such as data integration, data provenance, database federation, and the schema matching problem.

Perhaps surprisingly, many of the typical actions one performs on a database can be understood in terms of these data migration functors. For example, giving a user the ability to read or write certain types of data can be achieved in this way, as can joins and unions of tables. Finally, the fact that the data definition language (DDL) for databases is the same as for categories adds a lot of power to the system. For example a program could search for functors that relate schemas (and check the results of data-migration in well-known cases to ensure its correctness). Moreover, facts about the results of a given ETL process could be checked using a proof-checker such as Coq.

The main ideas of this paper may be understandable by a person who knows little to no category theory, but certainly the real substance is categorical. A working knowledge of adjoint functors would be useful. A good reference is [P1] or [M]. Some understanding of toposes might help, though it is certainly not assumed – the basic idea is given in Definition 5.1.1 but the curious reader should consult [MM]. The paper can also be read by a person who knows some category theory but nothing about databases.

1.1. **The basic idea: "category = schema".** The simplest way to understand databases in terms of mathematics is via the following definition.

**Definition 1.1.1.** A *database schema* or simply *schema* is a small category $\mathcal{C}$. An object $c$ of $\mathcal{C}$ is called a *table* in $\mathcal{C}$, and an arrow $f\colon c \to c'$ is called a *column of $c$ valued in $c'$*. The identity map $\mathrm{id}_c\colon c \to c$ is called the *primary id column of $c$*. A *leaf table* is an object $c \in \mathrm{Ob}(\mathcal{C})$ with no outgoing arrows (other than $\mathrm{id}_c$).

A *database state* or simply *state* on schema $\mathcal{C}$ is a functor $\gamma\colon \mathcal{C} \to \mathbf{Set}$. Given an object $c \in \mathrm{Ob}(\mathcal{C})$, an element $x \in \gamma(c)$ is called a *row of* $\gamma(c)$. We refer to a pair $(x, f)$, where $x$ is a row of $\gamma(c)$ and $f\colon c \to c'$ is a column of $c$, as *a cell in* $\gamma(c)$, and to each cell $(x, f)$ the element $\gamma(f)(x) \in \gamma(c')$ is called the *value of the $(x, f)$-cell*.

A *morphism of schemas from $\mathcal{C}$ to $\mathcal{C}'$* is a functor $F\colon \mathcal{C} \to \mathcal{C}'$. We write **Cat** to denote the category of schemas. A *morphism of database states from $\gamma$ to $\gamma'$* on $\mathcal{C}$ is a natural transformation $p\colon \gamma \to \gamma'$. We write $\mathcal{C}$–**Set** to denote the category of database states on $\mathcal{C}$ and sometimes refer to it as *the category of $\mathcal{C}$-sets*.

Definition 1.1.1 is precise enough for this paper, but there are some implicit abuses in terminology. These are detailed in Section 1.4.

*Remark* 1.1.2. Another way to look at Definition 1.1.1 is as a kind of "normal form" for a database. Suppose we say that a database is in *categorical normal form* if

- every table $t$ has a single primary key column $\mathrm{id}_t$, chosen at the outset. The cells in this column are called the row-ids of $t$;
- for every column $c$ of a table $t$ there is some table $t'$ such that the value in each cell of $c$ refers to some row-id of $t'$; and
- each "pure data" column of $t$ (with values in some set, say the set of strings) is considered a foreign key column to a 1-column table whose cells are all possible values of the given set (e.g. all strings). These 1-column tables do not have to be physically stored, but that issue is hidden.

*Example* 1.1.3. A database in categorical normal form consists of a bunch of tables. Each has a primary id column, and other columns. For example, consider these tables:

(1)

| **Employee** | | | | |
|---|---|---|---|---|
| **Id** | **First** | **Last** | **Mgr** | **Dpt** |
| 101 | David | Hilbert | 103 | q10 |
| 102 | Bertrand | Russell | 102 | x02 |
| 103 | Alan | Turing | 103 | q10 |

| **Department** | | |
|---|---|---|
| **Id** | **Name** | **Secr'y** |
| q10 | Sales | 101 |
| x02 | Production | 102 |

Every column in a table refers to the primary id column of another table – every column is a "key column." This will be made clear in Diagram (2), but lets look at the Employee table.

- the Id column refers to the Employee table,
- the First and last columns refer to the Strings table (see below),
- the Mgr column refers to the Employee table, and
- the Dpt column refers to the Department table.

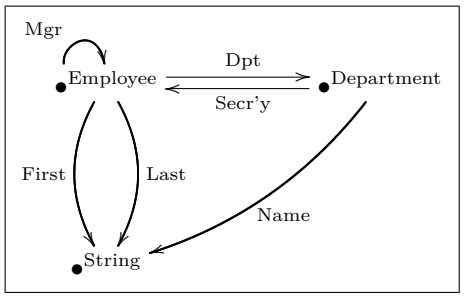
One should quickly check that all cells in the Dpt column refer to row-ids in the Department table.

As explained in Remark 1.1.2, even data columns (e.g. Name) refer to tables (e.g. the table of Strings). The table of strings can be considered "virtual" – one can never load its entirety into memory nor view it. Here is a sample of it:

| **Strings** |
|---|
| **Id** |
| a |
| b |
| ⋮ |
| z |
| aa |
| ab |
| ⋮ |

The way the above three tables interact (in terms of how their columns refer to one another) is called the schema for a database. The point of this paper is that

such a schema can be drawn as a category. In this case it is:

(2) $\mathcal{C} :=$ 


Each object (drawn as a dot) corresponds to a table. The arrows out of an object $t$ correspond to columns of $t$. Note that we never draw identity arrows (as they are implied) nor do we draw "free compo ions." For example the arrow

$$\text{First} \circ \text{Secr'y} \colon \bullet^{\text{Department}} \longrightarrow \bullet^{\text{String}}$$

is not drawn; this is akin to the fact that the Department table does not need a column for the secretary's first name. See Section 1.4

We can also impose "business rules" – composition laws. These are what differentiate categories from graphs. For example we could say that every employee must be in the same department as his or her manager. We could also say that the secretary of a department must be in that department. To impose these rules is to impose composition laws on $\mathcal{C}$:

$$\text{Dpt} \circ \text{Mgr} = \text{Dpt};$$
$$\text{Dpt} \circ \text{Secr'y} = \text{id}_{\text{Department}}.$$

Thus we see how this schema can be understood as a category, $\mathcal{C}$. One should also see that specifying a set of rows in the Employee table, the Department table, and the String table constitutes a functor $\gamma \colon \mathcal{C} \to \mathbf{Set}$. To each object in $\mathcal{C}$ we have written a set (the set of row-ids for that table), and to each arrow $f \colon c \to c'$ in $\mathcal{C}$ we have defined a function of sets (given by the $f$ column).

*Remark* 1.1.4. In Example 1.1.3 we alluded to the fact that certain 1-column tables in a schema $\mathcal{C}$ should represent data types, like Strings or Integers. However a functor $\mathcal{C} \to \mathbf{Set}$ does not a priori have to enforce such data types. For example in 1.1.3 we chose $\gamma(\bullet^{\text{String}})$ to be the set of strings, but nothing in the definition enforced that.

The reader can rest assured that all of the ideas of this paper go through if one wants to assign data types to 1-column tables. However, for pedagogical reasons, the material in Sections 2, 3, and 4 are not presented that way. We finally make amends in Section 5.2 where it is shown how to enforce typing.

Before leaving this section, the reader should understand the following overview, at least in the context of Example 1.1.3:

**Database overview:** a database schema consists of tables and "column-headers"; a state on a schema coherently assigns to each table a set of rows and to each column their respective values.

**Category overview:** a category $\mathcal{C}$ consists of objects and arrows; a functor $\gamma \colon \mathcal{C} \to \mathbf{Set}$ coherently assigns to each object a set of "instances" and to each arrow their respective values.

1.2. **The ubiquity of schemas.** While most practitioners may think of a database schema $\mathcal{C}$ as a large complex object, it does not have to be. Any category is a schema, and this gives much for the curious mind to ponder: "what might my example category mean as a database schema?" Some of the most important categories in mathematics are the most simple. For example let $[n]$ denote the category with objects $\{0, 1, \ldots, n\}$ and a single morphism $i \to j$ if and only if $i \leq j$; we draw it as

$$[n] := \boxed{\bullet^0 \longrightarrow \bullet^1 \longrightarrow \cdots \longrightarrow \bullet^n}.$$

The category $[0]$ (which is the terminal object in **Cat**) represents a single set or perhaps a "type." The arrow category $[1]$ (with two objects and one morphism connecting them) represents a single function (useful for defining "controlled vocabularies"). In a database, the category $[n]$ may represent a hierarchical naming system, with a table $\bullet^0$ representing fine details which points downward toward broader and broader overviews culminating in a master table $\bullet^n$.

The categories $[n]$ are linear orders; for a completely different kind of category occurring in mathematics, consider the free monoid on one generator, denoted $\mathbb{N}$. As a database schema it models the dynamics of discrete dynamical systems (see Section 2.4).

Thus our definition of database schemas and database states reflects ideas found within in pure mathematics. While these particular uses may or may not be directly important to a database administrator, the idea should be. We will show in this paper that many views of databases (pun intended) are obtained by looking at small schemas mapping to a large one. Many procedures done by a practitioner are in fact carried out by such functors. For example, looking at a single table within a large schema is so obtained, as we now describe.

*Example* 1.2.1. Suppose that $\mathcal{D}$ is a (possibly large) database schema, $\delta\colon \mathcal{D} \to \mathbf{Set}$ is a database state on $\mathcal{D}$, and that the setup from Example 1.1.3 sits inside it as a subschema, $\mathcal{C}$. In the context of this paper we will interpret this as follows. There is a functor $i\colon \mathcal{C} \to \mathcal{D}$, and we can recover the three tables by composing the functors to get a state $\delta \circ i \in \mathcal{C}$–**Set**. In other words, viewing a table or set of tables within a larger database is understood simply as a composition of functors.

Other important morphisms between schemas generate more complex data migration possibilities; see Section 2.3 for a quick tour. It allows for such capabilities as privileged access, views, joins and unions, and various imports and exports between different data models. A database expert using the categorical definition of databases will begin to vastly generalize his or her notion of database schemas to include all categories, especially little ones, for they are most often overlooked and most useful to a human user.

1.3. **The basic data migration functor: data pull-back.** The main point of this paper is to explain the various senses in which a morphism $F\colon \mathcal{C} \to \mathcal{D}$ of database schemas allows one to migrate data between them. One such such sense was implicit in Example 1.2.1 above. Given a large database schema $\mathcal{D}$ and a table in it $i\colon \mathcal{C} \to \mathcal{D}$, one can take any database state on $\mathcal{D}$ and obtain a database state on $\mathcal{C}$. In fact one checks that this process is functorial; i.e. we have a functor $\mathcal{D}$–**Set** $\to \mathcal{C}$–**Set**.

**Definition 1.3.1.** Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of database schemas. There exists a functor

$$F^*\colon \mathcal{D}\text{–}\mathbf{Set} \longrightarrow \mathcal{C}\text{–}\mathbf{Set},$$

called *the data pull-back functor associated to* $F$ defined as follows. Given a $\mathcal{D}$–**Set**, say $\delta\colon \mathcal{D} \to \mathbf{Set}$, define $F^*(\delta)\colon \mathcal{C} \to \mathbf{Set}$ as

$$F^*(\delta) = \delta \circ F.$$

In other words, given a functor $F\colon \mathcal{C} \to \mathcal{D}$ and a functor $\delta\colon \mathcal{D} \to \mathbf{Set}$, we compose to get a functor $\mathcal{C} \to \mathbf{Set}$. Thus, the data pull-back functor takes data on $\mathcal{D}$ and brings it to $\mathcal{C}$ by "doing the obvious thing" – the reader should not go on before making this clear to him or herself.

*Remark* 1.3.2. Despite its name, there is no "pull-back" in the sense of category theory visible in Definition 1.3.1. In Section 2.6 we will show a sense in which there actually is a pull-back involved here, below the surface.

The data pull-back functor has both a left and a right adjoint, denoted $F_!$ and $F_*$ respectively. These are harder to understand but extremely useful in practice, for they create nearly all the basic functionality of RDMBSs, such as views, updates, privileges, joins, unions, ETL, etc. It is these that we shall discuss in this paper.

In Section 2 we will define these data migration functors and investigate some special cases. In Section 3 we shall discuss updates and how they appear locally (i.e. to the user doing the updating). In Section 4 we shall discuss how these updates affect the rest of the database. In Section 5 we will add typing information and discuss how database states on a schema form a topos (Proposition 5.1.2) and have a logic and a language ready-made. We also turn things around and discuss how databases may be used to classify methods and their inheritance or delegation rather than things and their attributes. Finally in Section 6 we discuss some possibilities for future work.

1.4. **Notation.** The category of sets and the (total) functions between them is denoted **Set**; the category of small categories and the functors between them is denoted **Cat**. The terminal object in **Cat** is denoted $[0]$, and the terminal object in **Set** is denoted $\{\star\}$.

Schemas, and categories that should be thought of as schemas, will usually be denoted with upper-case calligraphic letters (such as $\mathcal{C}$ and $\mathcal{D}$), and morphisms $\mathcal{C} \to \mathcal{D}$ between schemas (i.e. functors) will usually be denoted by other roman letters (such as $F, G, i$ and $j$). As morphisms between schemas are functors, we use the adjectives for functors to describe morphisms of schemas; for example we may speak of a fully faithful morphism of schemas. Database states (i.e. functors from a schema to **Set**) will usually be denoted with greek letters (such as $\gamma\colon \mathcal{C} \to \mathbf{Set}$).

If between objects $c$ and $d$ in $\mathcal{C}$ there is only one morphism $f\colon c \to d$ then we often denote the corresponding column of table $c$ as $d$, rather than as $f$. We also sometimes denote identity morphisms (such as $\mathrm{id}_c\colon c \to c$) simply by the name of their object (here, $c$). This should not cause the reader much trouble.

Definition 1.1.1 is slightly misleading from a purely categorical point of view; here we explicate the abuses. Given a database state $\gamma\colon \mathcal{C} \to \mathbf{Set}$ and an object $c \in \mathcal{C}$, we write $\gamma(c)$ to denote the table whose columns are arrows out of $c$ and whose rows are indexed by the elements of the set $\gamma(c)$. Perhaps $\gamma(c/)$ would be better notation, but it is not as nice typographically. Furthermore, notice that not

all arrows out of $c$ need to be included as columns of the table $\gamma(c)$ – this issue is called "indirection" in database parlance. Given a commutative triangle

$$c \xrightarrow{f} d \xrightarrow{g} e, \qquad c \xrightarrow[h]{} e,$$

we only need include $f$ as a column in table $c$ and $g$ as a column in table $d$ – the morphism $h$ is obtained by "indirection." This suggests that a schema is really more like a *presentation of a category* or a *category sketch* (in the sense of [Mak]) rather than a category, but we do not make the distinction here. In other words, by abuse we do not distinguish between a presentation of a category $\mathcal{C}$ and the category $\mathcal{C}$ itself. [In math jargon, we can specify such a presentation as an object in the category $\mathbf{Pre}(\mathbf{\Delta}_{\leq 2})$ of 2-skeletal simplicial sets.]

These abuses should not scare anyone who is unfamiliar with category theory because they are minor and easily accounted for. Summary:

- A schema should not be a category, it should be a 2-graph, i.e. an object in $\mathbf{Pre}(\mathbf{\Delta}_{\leq 2})$, a presentation of a category. Similarly morphisms of schemas should be morphisms of 2-graphs. **We are implicitly identifying a category with its presentation.**
- An object $c$ in a schema $\mathcal{C}$ should be called a *type*, not a table. The corresponding table-schema should really be the slice 1-graph $\mathrm{Ob}(\mathcal{C}_{c/})$. **We are implicitly identifying a table with its primary id column.**
- Along the same lines, given an object $c \in \mathrm{Ob}(\mathcal{C})$ and a state $\gamma\colon \mathcal{C} \to \mathbf{Set}$, an element $x \in \gamma(c)$ should be called a *row-id*, not a row. **We are implicitly identifying a row with its primary id cell.**

1.5. **Acknowledgements.** I would like to thank Peter Gates, Dave Balaban, John Launchbury, and Greg Morrisett for many useful conversations.

## 2. The data migration functors

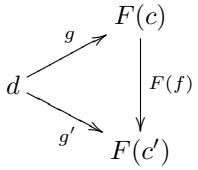
The basic data migration functor (data pull-back) was defined in Definition 1.3.1. Given a morphism of schemas $F\colon \mathcal{C} \to \mathcal{D}$, we can "pull back" data on $\mathcal{D}$ to get data on $\mathcal{C}$, and we write $F^*\colon \mathcal{D}$–**Set** $\to \mathcal{C}$–**Set**. As we mentioned at the time, the workings of this functor are in some sense "obvious"; however it has has both a left adjoint and a right adjoint, whose workings are less obvious. We define them in this section. Readers who are either unfamiliar with category theory or totally familiar with the adjoints of $F^*$ can skip to Section 2.2 or perhaps Section 2.5.

2.1. **Definitions of the data push-forward functors.** Suppose $F\colon \mathcal{C} \to \mathcal{D}$ is a functor. To calculate the effects of its left (respectively right) push-forward functor we need to recall or define the left (respectively right) Kan extension in this setting. To be clear, we are taking the left or right Kan extension *of* a state $\mathcal{C} \to \mathbf{Set}$ *along* the morphism $\mathcal{C} \xrightarrow{F} \mathcal{D}$ (see [M, Section X.3] or [MM, VII.2.2]). Most of this section can be skipped or skimmed on a first reading – we only ask that the reader make good note of the content and symmetry in Definitions 2.1.2 and 2.1.3.

We begin by establishing some notation. Given an object $d \in \mathrm{Ob}(\mathcal{D})$, let $(d \downarrow F)$ denote the category with

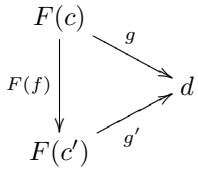
$$\mathrm{Ob}(d \downarrow F) = \{(c, g) \mid c \in \mathrm{Ob}(\mathcal{C}), g\colon d \to F(c) \in \mathcal{D}\}$$
$$\mathrm{Hom}_{(d\downarrow F)}((c,g),(c',g')) = \{f\colon c \to c' \in \mathcal{C} \mid F(f) \circ g = g'\}$$

In other words, an object in $(d \downarrow F)$ is a morphism from $d$ to the image of an object from $\mathcal{C}$, and the morphisms are commutative triangles




Similarly, let $(F \downarrow d)$ denote the category with

$$\mathrm{Ob}(F \downarrow d) = \{(c, g) \mid c \in \mathrm{Ob}(\mathcal{C}), g\colon F(c) \to d \in \mathcal{D}\}$$
$$\mathrm{Hom}_{(F\downarrow d)}((c,g),(c',g')) = \{f\colon c \to c' \in \mathcal{C} \mid g' \circ F(f) = g\}$$




For each object $d \in \mathrm{Ob}(\mathcal{D})$ there are "forgetful" functors

$$\pi^F(d)\colon (d \downarrow F) \to \mathcal{C} \text{ and}$$
$$\pi_F(d)\colon (F \downarrow d) \to \mathcal{C},$$

both of which are given by "$(c, g) \mapsto c$." We consider the category $(d \downarrow F)$ together with the functor $\pi^F(d)\colon (d \downarrow F) \to \mathcal{C}$ as an object of $\mathbf{Cat}_{/\mathcal{C}}$. Similarly, we consider the category $(F \downarrow d)$ together with the functor $\pi_F(d)\colon (F \downarrow d) \to \mathcal{C}$ as an object of $\mathbf{Cat}_{/\mathcal{C}}$.

In fact, all of this is functorial in $d$. That is, we have functors

$$\pi^F\colon \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}_{/\mathcal{C}} \text{ and}$$
$$\pi_F\colon \mathcal{D} \to \mathbf{Cat}_{/\mathcal{C}}.$$

**Definition 2.1.1.** Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of schemas. The functor $\pi^F\colon \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}_{/\mathcal{C}}$ is called *the $F$-right schema functor on $\mathcal{D}$* and the functor $\pi_F\colon \mathcal{D} \to \mathbf{Cat}_{/\mathcal{C}}$ is called *the $F$-left schema functor on $\mathcal{D}$*. To any object $d \in \mathrm{Ob}(\mathcal{D})$ the category $(d \downarrow F)$ is called *the schema $F$-right of $d$* and the category $(F \downarrow d)$ is called *the schema $F$-left of $d$*.

This is much simpler than it looks, but one has to work with it to get it (choose a simple $F\colon \mathcal{C} \to \mathcal{D}$ and work out $\pi_F$). However, let us flesh out the idea a bit more. Given a map $p\colon d \to d'$ in $\mathcal{D}$, composition with $p$ turns an object $F$-right of $d'$ into an object $F$-right of $d$; similarly, composition with $p$ turns an object $F$-left

of $d$ into an object $F$-left of $d'$:

$$(d' \downarrow F) \xrightarrow{\pi^F(p)} (d \downarrow F) \qquad (F \downarrow d) \xrightarrow{\pi_F(p)} (F \downarrow d')$$

is given via

$$\begin{array}{ccc} d' & \xleftarrow{\quad p \quad} & d \\ \downarrow & & \\ F(c') & & \end{array}$$

is given via

$$\begin{array}{ccc} F(c) & & \\ \downarrow & & \\ d & \xrightarrow[\quad p \quad]{} & d' \end{array}$$

To see $\pi^F$ and $\pi_F$ used in an example, see Section 2.2.2. We are ready to define the data push-forward functors.

**Definition 2.1.2.** Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of schemas and $F^*\colon \mathcal{D}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\mathbf{Set}$ be the associated data pull-back functor (see Definition 1.3.1). There exists a right adjoint to $F^*$ called the *right push-forward functor associated to* $F$, denoted

$$F_*\colon \mathcal{C}\text{–}\mathbf{Set} \to \mathcal{D}\text{–}\mathbf{Set},$$

and defined as follows.

Given an object $\gamma\colon \mathcal{C} \to \mathbf{Set}$ in $\mathcal{C}\text{–}\mathbf{Set}$ define $F_*\gamma$ on an object $d \in \mathrm{Ob}(\mathcal{D})$ as

$$F_*\gamma(d) := \lim_{(d \downarrow F)} \Big(\gamma \circ \pi^F(d)\Big). \tag{3}$$

This is simply the limit of the functor $(d \downarrow F) \xrightarrow{\pi^F(d)} \mathcal{C} \xrightarrow{\gamma} \mathbf{Set}$. Given a map $g\colon d \to d'$ in $\mathcal{D}$ one obtains a map $F_*\gamma(g)\colon F_*\gamma(d) \to F_*\gamma(d')$ by the universal property of limits.

The idea is this. We have some $\mathcal{C}$-set $\gamma$ and we want a $\mathcal{D}$-set $F_*\gamma$. To each object in $d$ we look at the objects in $\mathcal{C}$ which are sent to the right of $d$ (i.e. those equipped with a chosen morphism from $d$). Each has been assigned by $\gamma$ some set of rows; we take the limit of all these sets and assign that to $F_*\gamma(d)$.

The description for $F_!$ is appropriately "dual."

**Definition 2.1.3.** Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of schemas and $F^*\colon \mathcal{D}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\mathbf{Set}$ be the associated data pull-back functor (see Definition 1.3.1). There exists a left adjoint to $F^*$ called the *left push-forward functor associated to* $F$, denoted

$$F_!\colon \mathcal{C}\text{–}\mathbf{Set} \to \mathcal{D}\text{–}\mathbf{Set},$$

and defined as follows.

Given an object $\gamma\colon \mathcal{C} \to \mathbf{Set}$ in $\mathcal{C}\text{–}\mathbf{Set}$ define $F_!\gamma$ on an object $d \in \mathrm{Ob}(\mathcal{D})$ as

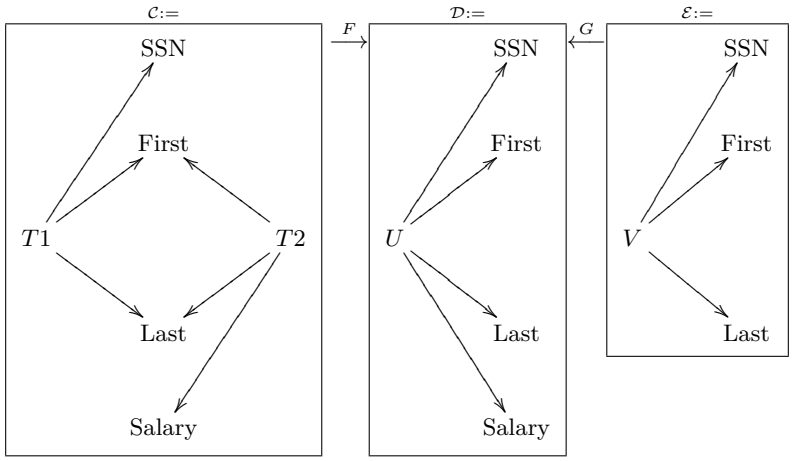
$$F_!\gamma(d) := \operatorname*{colim}_{(F \downarrow d)} \Big(\gamma \circ \pi_F(d)\Big). \tag{4}$$

This is simply the colimit of the functor $(F \downarrow d) \xrightarrow{\pi_F(d)} \mathcal{C} \xrightarrow{\gamma} \mathbf{Set}$. Given a map $g\colon d \to d'$ in $\mathcal{D}$ one obtains a map $F_!\gamma(g)\colon F_!\gamma(d) \to F_!\gamma(d')$ by the universal property of colimits.

Here, we have some $\mathcal{C}$-set $\gamma$ and we want an $\mathcal{D}$-set $F_!\gamma$. To each object in $d$ we look at the objects in $\mathcal{C}$ which are sent to the left of $d$ (i.e. those equipped with a chosen morphism to $d$). Each has been assigned by $\gamma$ some set of rows; we take the colimit of all these sets and assign that to $F_!\gamma(d)$.

2.2. **A basic example.** Let $\mathcal{C} \xrightarrow{F} \mathcal{D} \xleftarrow{G} \mathcal{E}$ be the categories depicted in the boxes below:

(5)




Here the morphism (functor) $F\colon \mathcal{C} \to \mathcal{D}$ is given by sending both $T1$ and $T2$ to $U$ and by identity on everything else; the morphism $G\colon \mathcal{E} \to \mathcal{D}$ is given by inclusion (sending $V \mapsto U$). We will describe $F^*, F_*$, and $F_!$ (respectively for $G$) in this case.

Before beginning, the reader who wishes to really understand the constructions given in Section 2.1 should do a basic task: Determine what objects in $\mathcal{C}$ are $F$-left of $U \in \mathrm{Ob}(\mathcal{D})$ and which objects in $\mathcal{C}$ are $F$-right of $U$.

2.2.1. *The pull-back functors $F^*$ and $G^*$.* Suppose first that $\delta\colon \mathcal{D} \to \mathbf{Set}$ is a database state on $\mathcal{D}$. We can represent it as five tables. Four of these are arbitrary 1-column tables (or just sets): a set of SSN's, a set of First's, a set of Last's, and a set of Salary's, which we will not fix here for space and readability reasons. The fifth is a "fact" table such as

$\delta(U) :=$

| **U** | | | | |
|---|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** | **Salary** |
| x11 | 101-22-0411 | David | Hilbert | 150 |
| x12 | 220-39-7479 | Bertrand | Russell | 200 |
| x13 | 775-33-2819 | Alan | Turing | 200 |

The requirement is that each cell in a given column represents a row in the corresponding 1-column table. For example, the 1-column table $\delta$(Salary) can be the set of integers less than 32,000 or it can be a set with only two elements ($\{150{,}200\}$); it simply must contain the cells in the Salary column, or $\delta$ would not be a functor.

A state $\mathcal{C} \to \mathbf{Set}$ will be similar. It will have six tables, four of which are 1-column tables as above. There will be two 4-column tables, one of which has facts relating SSN, First, and Last, and the other of which has facts relating First, Last, and Salary.

As we mentioned above, $F^*(\delta)$ is obtained by "doing the obvious thing": given an object in $\mathcal{C}$, map it to $\mathcal{D}$ and see what $\delta$ does to it. Thus $F^*(\delta)$ will not change

the four 1-column tables of $\delta$. The two 4-column tables will be the projections

$F^*\delta(T1) =$

| T1 | | | |
|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** |
| x11 | 101-22-0411 | David | Hilbert |
| x12 | 220-39-7479 | Bertrand | Russell |
| x13 | 775-33-2819 | Alan | Turing |

$F^*\delta(T2) =$

| T2 | | | |
|---|---|---|---|
| **Id** | **First** | **Last** | **Salary** |
| x11 | David | Hilbert | 150 |
| x12 | Bertrand | Russell | 200 |
| x13 | Alan | Turing | 200 |

The reader should work out the case of $G^*\delta$.

2.2.2. *The right push-forward functors $F_*$ and $G_*$.* Now suppose that $\gamma\colon \mathcal{C} \to \mathbf{Set}$ is a database state on $\mathcal{C}$. Again, it will include four 1-column tables, which we will not write out, and two 4-column tables, which we arbitrarily choose to be:

$\gamma(T1) :=$

| T1 | | | |
|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** |
| x11 | 101-22-0411 | David | Hilbert |
| x12 | 220-39-7479 | Bertrand | Russell |
| x13 | 775-33-2819 | Bertrand | Russell |

$\gamma(T2) :=$

| T2 | | | |
|---|---|---|---|
| **Id** | **First** | **Last** | **Salary** |
| y1 | David | Hilbert | 150 |
| y2 | Bertrand | Russell | 200 |
| y3 | Bertrand | Russell | 225 |
| y4 | Alan | Turing | 200 |

Before calculating $F_*\gamma$, let us quickly say what $\pi^F\colon \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}_{/\mathcal{C}}$ is. On the each of the four objects SSN, First, Last, and Salary, $\pi^F$ yields the subcategory of $\mathcal{C}$ consisting of that object alone: for example $\pi^F(\mathrm{SSN}) = (\{SSN\} \to \mathcal{C})$. On the object $U$, it yields the entirety of $\mathcal{C}$. It follows that $F_*\gamma$ applied to SSN, First, Last, or Salary is just $\gamma$ applied to that object of $\mathcal{C}$. It also follows that $F_*\gamma$ applied to $U$ is the limit of the diagram $\gamma$, namely an element (row) in $F_*\gamma(U)$ is a row in $T1$ and a row in $T2$ such that their images in First and Last agree.

While the above description is long, it is straightforward. But what does it really mean? One computes that $F_*\gamma$ in fact yields the join of $T1$ and $T2$ along First and Last. In other words it yields

$F_*\gamma(U) =$

| U | | | | |
|---|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** | **Salary** |
| (x11,y1) | 101-22-0411 | David | Hilbert | 150 |
| (x12,y2) | 220-39-7479 | Bertrand | Russell | 200 |
| (x11,y3) | 220-39-7479 | Bertrand | Russell | 225 |
| (x13,y2) | 775-33-2819 | Bertrand | Russell | 200 |
| (x13,y3) | 775-33-2819 | Bertrand | Russell | 225 |

There are several nice things about the fact that $F_*$, whose description does not outwardly suggest anything about joins can indeed compute joins. First, it shows that data migration functors are more than meets the eye – if they can compute joins, what else can they do? Second, one should recognize that the picture in (5) tells the story. In this picture, tables $T1$ and $T2$ are being merged together into table $U$; everything else is the same. With learned intuition, a DBA would not need to compute what $F_*$ will do – he or she will consider it obvious that $F_*$ will make $U$ as the join of $T1$ and $T2$ along their common columns. At the same time, the DMBS can actually make the computation in a rigorous way, while theorem-provers could reason about it.

We now repeat the above ideas for $G_*$. Let $\epsilon\colon \mathcal{E} \to \mathbf{Set}$ be a database state with some choice (which we do not write down here) of the three 1-column tables and with $\epsilon(V)$ given by

$\epsilon(V) :=$

| **V** | | | |
|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** |
| x11 | 101-22-0411 | David | Hilbert |
| x12 | 220-39-7479 | Bertrand | Russell |
| x13 | 775-33-2819 | Alan | Turing |

Its right push-forward will not know what to do with the new 1-column table Salary, nor what to do with the Salary-column of $U$. And yet something canonical must occur! To determine what it is, one must compute $\pi^G(\text{Salary})$, which turns out to be the empty subcategory of $\mathcal{E}$. Its limit is a one-point set.

Thus, the 1-column table $G_*\epsilon(\text{Salary})$ consists of a single value, say $\star$. This is not an integer; as mentioned in Remark 1.1.4 we are not enforcing data types at this time (see Section 5 for how to enforce data types). In fact, there could be no good choice of $G_*\epsilon$ if we forced $\star$ to be an integer but did not include any such information in $\epsilon$. Here, $\star$ simply represents "unknown." But now we can see that there is no hardship in computing $G_*\epsilon(U)$ because there is no choice necessary for the Salary column:

$G_*\epsilon(U) =$

| **U** | | | | |
|---|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** | **Salary** |
| x11 | 101-22-0411 | David | Hilbert | $\star$ |
| x12 | 220-39-7479 | Bertrand | Russell | $\star$ |
| x13 | 775-33-2819 | Alan | Turing | $\star$ |

2.2.3. *The left push-forward functors $F_!$ and $G_!$.* We introduced a database state $\gamma\colon \mathcal{C} \to \mathbf{Set}$ on $\mathcal{C}$ and a database state $\epsilon\colon \mathcal{E} \to \mathbf{Set}$ on $\mathcal{E}$ above in Section 2.2.2. For convenience we repeat $\gamma$ here.

(6) $\gamma(T1) :=$

| T1 | | | |
|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** |
| x11 | 101-22-0411 | David | Hilbert |
| x12 | 220-39-7479 | Bertrand | Russell |
| x13 | 775-33-2819 | Bertrand | Russell |

(7) $\gamma(T2) :=$

| T2 | | | |
|---|---|---|---|
| **Id** | **First** | **Last** | **Salary** |
| y1 | David | Hilbert | 150 |
| y2 | Bertrand | Russell | 200 |
| y3 | Bertrand | Russell | 225 |
| y4 | Alan | Turing | 200 |

In this section we will explore the left push-forward $F_!\gamma$ of $\gamma$ associated to $F$. It will clearly be a $\mathcal{D}$-set and as such consist of four 1-column tables and one 5-column table $U$. We will leave $G_!\epsilon$ as an exercise for the reader.

What is $\pi_F \colon \mathcal{D} \to \mathbf{Cat}_{/\mathcal{C}}$ in this case? One checks that on the objects of $\mathcal{D}$ we have

(8)
$$\begin{aligned}
\pi_F(U) &= \boxed{\bullet^{T1} \qquad \bullet^{T2}} \\
\pi_F(\text{SSN}) &= \boxed{\bullet^{T1} \longrightarrow \bullet^{\text{SSN}} \qquad \bullet^{T2}} \\
\pi_F(\text{First}) &= \boxed{\bullet^{T1} \longrightarrow \bullet^{\text{First}} \longleftarrow \bullet^{T2}} \\
\pi_F(\text{Last}) &= \boxed{\bullet^{T1} \longrightarrow \bullet^{\text{Last}} \longleftarrow \bullet^{T2}} \\
\pi_F(\text{Salary}) &= \boxed{\bullet^{T1} \qquad \bullet^{\text{Salary}} \longleftarrow \bullet^{T2}}.
\end{aligned}$$

and the reader should be able to guess and check what $\pi_F$ does to morphisms in $\mathcal{D}$. Each of the above categories comes with a canonical inclusion into $\mathcal{C}$ which we compose with $\gamma$ to get a functor to **Set**. To compute $F_!\gamma$, we are interested in the colimit of each of these functors. They will be (in order)

$$\begin{aligned}
\gamma(T1) \amalg \gamma(T2);& \\
\gamma(\text{SSN}) \amalg \gamma(T2);& \\
\gamma(\text{First});& \\
\gamma(\text{Last});& \\
\gamma(T1) \amalg \gamma(\text{Salary}).&
\end{aligned}$$

We are ready to calculate $F_!\gamma$ and we need only consider $F_!\gamma(U)$. The formula in (8) says to take the coproduct of the rows in $\gamma(T1)$ and $\gamma(T2)$. One checks that:

$F_!\gamma(U) =$

| U | | | | |
|---|---|---|---|---|
| **Id** | **SSN** | **First** | **Last** | **Salary** |
| x11 | 101-22-0411 | David | Hilbert | x11Salary |
| x12 | 220-39-7479 | Bertrand | Russell | x12Salary |
| x13 | 775-33-2819 | Bertrand | Russell | x13Salary |
| y1 | y1SSN | David | Hilbert | 150 |
| y2 | y2SSN | Bertrand | Russell | 200 |
| y3 | y3SSN | Bertrand | Russell | 225 |
| y4 | y4SSN | Alan | Turing | 200 |

Here, every "unexpected cell" is filled in with a uniquely-assigned representative or *Skolem variable*. We have chosen some arbitrary name (e.g. "x11Salary") but any other choice will work, as long as it is uniquely chosen. Of course, we are not making this uniqueness rule – it is forced upon us by the category theory. These almost take on the role of nulls, but with more clearly defined meaning and better formal properties.

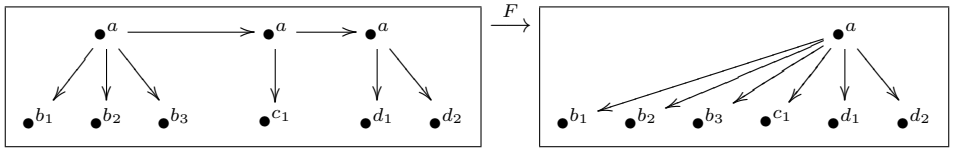
Whereas the right push-forward $F_*$ produces limits (joins), the left push-forward $F_!$ produces colimits (unions). Note that although there was repetition in $F_!\gamma(U)$ (e.g. too many Bertrand Russells for comfort!), it could have been avoided if $\mathcal{C}$ had contained a table that identified certain rows of $T1$ with certain rows of $T2$ (e.g. x12 with y2).

On the 1-column tables, $F_!\gamma$ simply repeats what is found in $\gamma$, and then adds a new row for every "uniquely-assigned representative" (e.g. in $F_!\gamma$(Salary) we find x11Salary).

$G_!$ is similar and is left as an exercise to the reader.

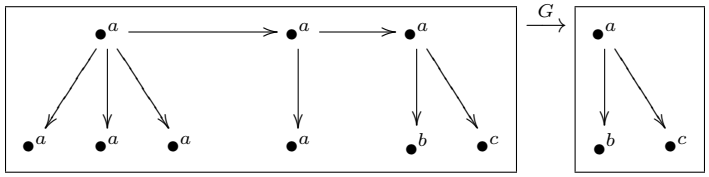
2.3. **Visualizing joins.** In this section we invite the reader to visualize joins using the categorical approach. We offer three examples of schema morphisms under which the right data migration functor creates useful joins.

*Example* 2.3.1. Consider the morphism of schemas drawn as




where every object is mapped to the one of the same label. The idea is that we begin with three tables (with 3,1, and 2 data columns respectively) arranged in a detail to master hierarchy, and we want to join them into one table (with 6 data columns). This is done using the right push-forward functor $F_*$. To check ones intuition, the number of rows in the final "join" table will equal the number of rows in the original 3-column table.

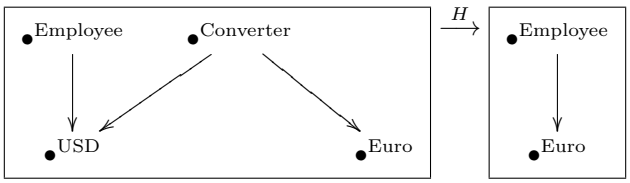
*Example* 2.3.2. Consider the morphism of schemas drawn as




where every object is mapped to the one of the same label. The idea here is that we begin with three tables (with 3,1, and 2 data columns respectively) arranged in a hierarchy and want to use "indirection" to view the $b, c$ attributes of the rows in the first table. This is done using the right push-forward functor $G_*$. To check ones intuition, the number of rows in the final "join" table will equal the number of rows in the original 3-column table.

*Example* 2.3.3. Suppose that one has a table of employees and their salaries in US Dollars, and we wish to convert the salaries into Euros. We have a table that

converts Dollars to Euros. In this situation we have the following morphism of schemas




where everything but Euro in the first schema is sent under $H$ to Employee. Then the right push-forward functor $H_*$ will output the table of employees and their salary in Euros. If every USD has exactly one conversion to Euros, then the number of rows in the Employee table will remain constant under $H_*$.

2.4. **Time steps in discrete dynamical systems.** Here is another example, which is different in nature from that presented in Section 2.2. While discrete dynamical systems may not be very relevant to the normal operations of databases as they are currently conceived, it is the author's hope that the scope of databases will increase to include situations such as these. Readers who are not interested in how data migration can be applied to discrete dynamical systems should skip to Section 2.5.

A *discrete dynamical system* is a pair $(S, f)$ where $S$ is a set and $S \xrightarrow{f} S$ is a function. Each element $s \in S$ is transformed into $f(s)$ in one "time step" and to $f^n(s)$ in $n$ time steps. Given a natural number $n$, we can perform an *$n$-step time-lapse view* on $(S, f)$ by considering instead $(S, f^n)$ as a new dynamical system. We have the same set of states (elements of $S$), but we only take account of every $n$th change.

Now we present the categorical view of this situation. Consider the free monoid on one generator,

$$N := \boxed{\overset{t}{\circlearrowright}\bullet}.$$

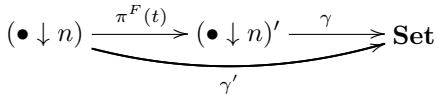
It is a category with one object, hence a database schema with one table. This table has one non-id column with values in the id-column. The $n$th step in the dynamical system is obtained by $n$-fold indirection.

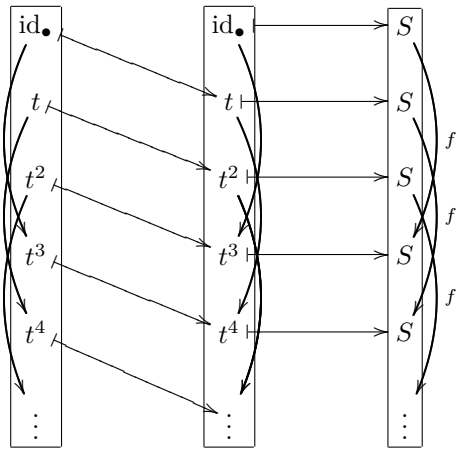
Consider the morphism of schemas $n\colon N \to N$ given by sending $t$ to $t^n$. We call this morphism the *$n$-step morphism*. A discrete dynamical system is then a state $\delta\colon N \to \mathbf{Set}$ and one may check that $n^*\delta$ has the same set of states, but everything moves "$n$ times as far" in each step.

What are $n_*\delta$ and $n_!\delta$? This is quite interesting. Given a dynamical system, how can one canonically create two new dynamical systems which move $n$-times slower? We will only work through the $n_*$ example, leaving the $n_!$ example, which is in some sense dual, to the reader.

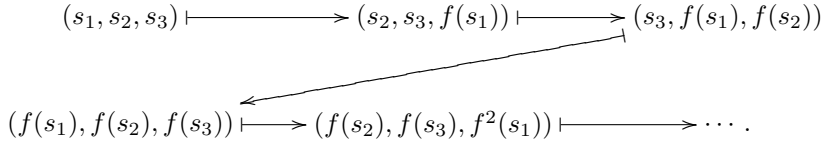
Let us explore the case $n(t) = t^3$, the 3-step time lapse morphism. Suppose that $\gamma\colon N \to \mathbf{Set}$ is a dynamical system whose underlying set is denoted $S$ and whose map is denoted $f\colon S \to S$. Without going into too many details, the category $(\bullet \downarrow n)$, the endo-functor $\pi^F(t)$, and the composite functor $(\bullet \downarrow n) \to N \xrightarrow{\gamma} \mathbf{Set}$

(which we simply write as $\gamma'$) can be displayed as follows:




We are ready to say what the dynamical system $n_*\gamma$ is. Its underlying set is the limit of $\gamma'$, which is simply $S \times S \times S$, and the induced action on it is $(s_1, s_2, s_3) \mapsto (s_2, s_3, f(s_1))$. Going a bit further we see the pattern:




This indeed moves three times slower than $(S, f)$ does.

So for example, if each object in $S$ represents a musical note and $f$ tells us how each note should be changed into another note, then $n_*(S, f)$ is the dynamical system whose states are something like chords of 3-notes and whose transition is given by changing the first note in the chord, then the second note, then the third, repeatedly. One can check that if we look at $n_!$ instead of $n_*$ then the result is more like three different instruments playing in succession: one instrument chooses a note, plays it, and then passes it to the next instrument; on the passage from the third instrument to the first, the note is changed as specified by $f$.

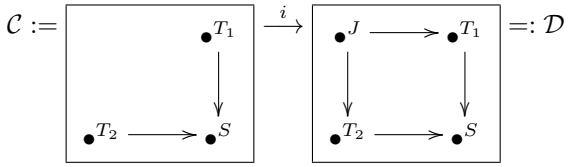
Here is perhaps a more interesting application of these ideas. A *continuous dynamical system* is an $M$-set, where $M = (\mathbb{R}_{\geq 0}, +, 0)$ is the monoid of nonnegative real numbers under addition. The obvious inclusion functor $N \to M$ has associated data migration functors. One could presumably use these to understand how well discrete dynamical systems can model continuous ones.

### 2.5. **Using push-forwards to construct joins, fixed points, unions, orbits, and images.**

One can use right push-forwards to construct joins, as well as things like the set of fixed points of a monoid action. One can use left push-forwards to construct

unions (and insertions), as well as things like the set of orbits of a monoid action. One can use the two push-forwards in tandem to construct images of functions. We present the simplest scenario of each of the above constructions and then give a general formula for it in the examples below.

*Example* 2.5.1 (Joins). Consider tables $T_1$ and $T_2$ each with one (non-identity) attribute valued in a leaf table $S$ (e.g. a person table and a car table, each with an address column). The schema $\mathcal{C}$ for this setup is the left-hand category below:




The morphism $i\colon \mathcal{C} \to \mathcal{D}$ is an inclusion; for any database state $\gamma\colon \mathcal{C} \to \mathbf{Set}$ on $\mathcal{C}$, the right push-forward $i_*\gamma$ will leave all tables from $\mathcal{C}$ as they were, and put the join (along $S$) of $T_1$ and $T_2$ as the value of $i_*\gamma$ at $J$,

$$i_*\gamma(J) = \gamma(T_1) \times_{\gamma(S)} \gamma(T_2).$$

In particular if we want the cartesian product of $T_1$ and $T_2$, we put $\gamma(S) = \{\star\}$ or just leave $S$ out of the picture altogether.

In general, for a category $\mathcal{C}$, let $\mathcal{C}^\triangleleft$ denote the result of freely adding an initial object to $\mathcal{C}$; this category will be called the *left cone on* $\mathcal{C}$ and its unique initial object will be called the *cone point of* $\mathcal{C}^\triangleleft$. There is a canonical inclusion $i\colon \mathcal{C} \to \mathcal{C}^\triangleleft$ and $i_*$ applied to any $\mathcal{C}$-set $\gamma$ will assign to the cone point the join (limit) of the tables in $\gamma$.

*Example* 2.5.2 (Fixed points). Given a monoid $M$, considered as a category with one object, an $M$-set is a functor $\gamma\colon M \to \mathbf{Set}$; the image of the unique object of $M$ is a set which we denote $|\gamma|$. For each monoid element $m \in M$ and $x \in |\gamma|$ we write $m \cdot x$ to denote $\gamma(m)(x) \in |\gamma|$. A fixed point of $\gamma$ is an element $x \in |\gamma|$ such that $m \cdot x = x$ for all $m \in M$.

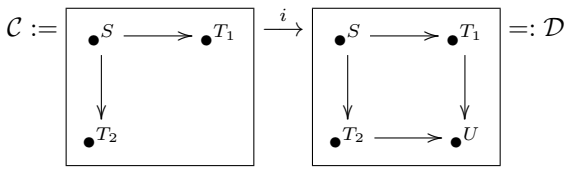
To obtain the set of fixed points of $\gamma$ using right push-forwards, one simply considers the unique functor $t\colon M \to [0]$, where $[0]$ is the terminal category. Then $t_*\gamma \in \mathbf{Set}$ will be the set of fixed points of $\gamma$. There is a canonical inclusion $i^*i_*\gamma \to \gamma$ which allows us to view the fixed points of $\gamma$ inside of $\gamma$.

As a database schema, a monoid is seen as a table $t$ all of whose columns point back to the primary id column of $t$; i.e. the value in every cell is a row-id in $t$. A fixed point is a row-id such that every cell in that row has the same value.

In general, given any schema (category) $\mathcal{C}$ and $\mathcal{C}$-set $\gamma$, one can take the right push-forward $t_*\gamma$ of $\gamma$ along the terminal morphism $t\colon \mathcal{C} \to [0]$. This set is called *the set of global elements of* $\gamma$. For example, if every table in a database state has a default value (and these are coherently defined), then that default value will be a global element of the database state.

*Example* 2.5.3 (Unions and insertions). Consider two tables $T_1$ and $T_2$ (e.g. classical-music lovers and jazz lovers) of which we want to take the union. Suppose $S$ is a table of "common elements" (e.g. known lovers of both classical and jazz). The

schema $\mathcal{C}$ for this setup is the left-hand category below:




The morphism $i\colon \mathcal{C} \to \mathcal{D}$ is an inclusion; for any database state $\gamma\colon \mathcal{C} \to \mathbf{Set}$ on $\mathcal{C}$, the left push-forward $i_!\gamma$ will leave all tables from $\mathcal{C}$ as they were, and put the union (along $S$) of $T_1$ and $T_2$ as the value of $i_!\gamma$ at $U$,

$$i_!\gamma(U) = \gamma(T_1) \amalg_{\gamma(S)} \gamma(T_2).$$

In particular if we want the disjoint union of $T_1$ and $T_2$, we put $\gamma(S) = \emptyset$ or leave $S$ out of the picture all together.

In general, for a category $\mathcal{C}$, let $\mathcal{C}^{\triangleright}$ denote the result of freely adding an final object to $\mathcal{C}$; this category will be called the *right cone on* $\mathcal{C}$ and its unique final object will be called the *cone point of* $\mathcal{C}^{\triangleright}$. There is a canonical inclusion $i\colon \mathcal{C} \to \mathcal{C}^{\triangleright}$ and $i_!$ applied to any $\mathcal{C}$-set $\gamma$ will assign to the cone point the union (colimit) of the tables in $\gamma$.

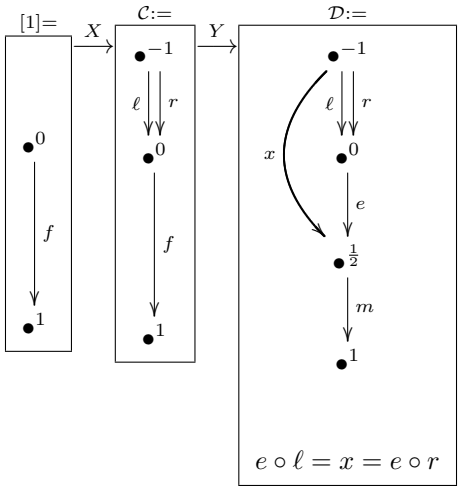
*Example* 2.5.4 (Orbits). Given a monoid $M$, considered as a category with one object or schema with one table, an $M$-set is a state $\gamma\colon M \to \mathbf{Set}$; the image of the unique object of $M$ is a set which we denote $|\gamma|$. For each monoid element $m \in M$ and $x \in |\gamma|$ we write $m \cdot x$ to denote $\gamma(m)(x) \in |\gamma|$. An orbit of $\gamma$ is an equivalence class of element $x \in |\gamma|$, where we consider elements $x$ and $x'$ to be equivalent if there exists $m \in M$ such that $m \cdot x = x'$.

To obtain the set of orbits of $\gamma$ using left push-forwards, one simply considers the unique morphism $t\colon M \to [0]$. Then $t_!\gamma \in \mathbf{Set}$ will be the set of orbits of $\gamma$. There is a canonical projection $\gamma \to t^*t_!\gamma$ sending an element to its orbit.

In general, given any schema (category) $\mathcal{C}$ and $\mathcal{C}$-set $\gamma$, one can take the left push-forward $t_!\gamma$ of $\gamma$ along the terminal morphism $t\colon \mathcal{C} \to [0]$. This set is called *the set of orbits of* $\gamma$. Such a set of orbits may have some relevance to "data-mining."

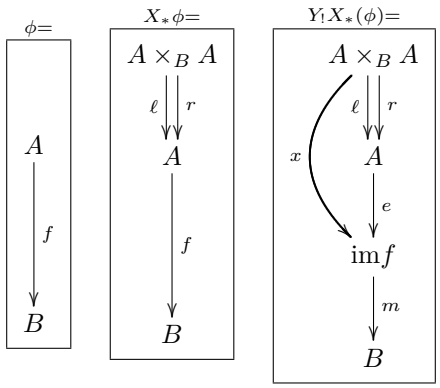
*Example* 2.5.5 (Images). Suppose that $f\colon A \to B$ is a function, considered as a functor $\phi\colon [1] \to \mathbf{Set}$. Suppose we would like to find the image of this function. We show here how one can construct that image using a right push-forwards followed

by a left push-forwards. Consider the two morphisms pictured below




where $X$ and $Y$ preserve the labels on objects, so $f \mapsto m \circ e$.

One checks that $\phi$, $X_*\phi$ and $Y_!X_*\phi$ are respectively




In the last picture $Y_!X_*\phi$ we have the "epi-mono factorization" of $f$; i.e. $m \circ e = f$, $m$ is injective, and $e$ is surjective. The table $A \times_B A$ at position $\bullet^{-1}$ represents the kernel of $f$: pairs of elements in $A$ that have the same $f$-attribute. Thus we have constructed the image of $f$ as a "basic view" as defined below in Definition 3.3.

2.6. **The Grothendieck construction.** The Grothendieck construction is a way to transform functors into categories. Given a category $\mathcal{D}$ and a functor $\delta\colon \mathcal{D} \to \mathbf{Set}$ (respectively $\delta\colon \mathcal{D} \to \mathbf{Cat}$), one obtains a new category $\int \delta$ and an op-fibration $\pi\colon \int \delta \to \mathcal{D}$. For each object $d \in \mathrm{Ob}(\mathcal{D})$ the fiber $\pi^{-1}(d)$ is a set (resp. a category) that is isomorphic to $\delta(d)$. We give the formal definition for the case of set-valued functors now.

**Definition 2.6.1.** Let $\mathcal{D}$ be a category and $\delta\colon \mathcal{D} \to \mathbf{Set}$ be a $\mathcal{D}$-set. The *category of elements of* $\delta$, denoted $\int \delta$, is a category whose set of objects is $\{(d, x) | d \in \mathrm{Ob}(\mathcal{D}), x \in \delta(d)\}$ and whose hom-sets are given by

$$\mathrm{Hom}_{\int \delta}((d, x), (d', x')) := \{g\colon d \to d' | g(x) = x'\}.$$

There is a natural functor $\pi\colon \int \delta \to \mathcal{D}$ given by taking $(d, x)$ to $d$ and $g\colon (d, x) \to (d', x')$ to $g\colon d \to d'$. It is a discrete op-fibration, and we call $\pi$ the *category of elements of $\delta$ over $\mathcal{D}$*.

The idea here is that given any database state, we can break it down into its fundamental units or atoms of information. These can be conceived of as 2-column tables (one non-id attribute) or as ⟨subject, attribute, object⟩ triples; see Remark 2.6.3. This construction is most natural in category theory (see [MM, p. 44]).

*Remark* 2.6.2. We learned the following factorization system from [Joy]. In the category **Cat**, every morphism $F\colon \mathcal{C} \to \mathcal{D}$ can be uniquely factored as

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{i} & \mathcal{E} \\ & \searrow^{F} & \downarrow{\scriptstyle p} \\ & & \mathcal{D}, \end{array}$$

where $i$ is an initial functor and $p$ is a discrete op-fibration. A functor $i\colon \mathcal{C} \to \mathcal{E}$ is initial if, for every object $e \in \mathrm{Ob}(\mathcal{E})$ the category $(i \downarrow e)$ is (non-empty and) connected. A functor $p\colon \mathcal{E} \to \mathcal{D}$ is a discrete op-fibration if, for every object $e \in \mathrm{Ob}(\mathcal{E})$ and morphism $g$ in $\mathcal{D}$ with source $p(e)$, there is a unique arrow $f$ in $\mathcal{E}$ with $p(f) = g$.

Two of the three data migration functors can be easily explained using the Grothendieck construction. Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism between schemas. Given a state $\delta\colon \mathcal{D} \to \mathbf{Set}$, consider the category of elements $\pi\colon \int \delta \to \mathcal{D}$ over $\mathcal{D}$. The data pull-back functor $F^*\delta$ is just the fiber product

$$\begin{array}{ccc} F^*\delta & \longrightarrow & \int \delta \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ \mathcal{C} & \xrightarrow[F]{} & \mathcal{D}. \end{array}$$

This may explain the association between the terminology "pull-back" and the functor $F^*$; see Remark 1.3.2.

Again let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of schemas and suppose that $\gamma\colon \mathcal{C} \to \mathbf{Set}$ is a $\mathcal{C}$-set; let $\pi\colon \int \gamma \to \mathcal{C}$ be the category of elements over $\mathcal{C}$. The composition

$$\begin{array}{ccc} \int \gamma & & \\ {\scriptstyle \pi}\downarrow & & \\ \mathcal{C} & \xrightarrow[F]{} & \mathcal{D} \end{array}$$

is a category over $\mathcal{D}$ but it is not in general a discrete op-fibration. There exists an initial object in the category of discrete op-fibrations over it (using the factorization system of Remark 2.6.2), and this is $F_!\delta$. In other words, one "completes" the map $\int \gamma \to \mathcal{D}$ in the minimal way. In fact, this way to think about or calculate $F_!$ is often easier than the one presented in Definition 2.1.3.

The right push-forward does not appear to have an illuminating description in terms of Grothendieck constructions.

*Remark* 2.6.3. The category of elements for a $\mathcal{C}$-set $\gamma\colon \mathcal{C} \to \mathbf{Set}$ could be called *the RDF category* of $\gamma$. Its objects could be called *URIs* and its morphisms could be called *triples*. This terminology comes from that used in the Semantic Web – see http://www.w3.org/TR/rdf-concepts/.

A more database-oriented naming system would be that $\int \gamma$ is called *the category of rows-ids and cells* of $\gamma$. The objects are called *row-ids* and the morphisms are called *cells*. The functor $\pi$ is called *the location functor* because for every row-id (resp. cell) $\pi$ returns the table (resp. column) in which it is located.

## 3. Updates and their local effects

In this section we discuss certain kinds of "views" on a database, and focus our attention on what we call "data-entry views." These are small portions of the database through which a user can add or delete data and then update the system in a sensible way. We also discuss more general views in Section 3.3.

### 3.1. Data-entry views and updates.

**Lemma 3.1.1.** *Let $i\colon \mathcal{C} \to \mathcal{D}$ be a functor. The following are equivalent:*

1. *$i$ is fully faithful;*
2. *$i_!\colon \mathcal{C}$–**Set** $\to \mathcal{D}$–**Set** is fully faithful;*
3. *$i_*\colon \mathcal{C}$–**Set** $\to \mathcal{D}$–**Set** is fully faithful;*
4. *the counit map $i^*i_* \to \mathrm{id}_{\mathcal{C}\text{–}\mathbf{Set}}$ is an isomorphism; and*
5. *the unit map $\mathrm{id}_{\mathcal{C}\text{–}\mathbf{Set}} \to i^*i_!$ is an isomorphism.*

*Proof.* The equivalence of (2), (3), (4), and (5) is proven in [MM, Lemma VII.4.1]. The equivalence of (3) and (1) is proven in [MM, Section VII.4].

□

**Definition 3.1.2.** Let $\mathcal{D}$ be a schema and let $i\colon \mathcal{C} \hookrightarrow \mathcal{D}$ be a fully faithful morphism. Given a $\mathcal{D}$-set $\delta$, we call $i^*\delta$ the *data-entry view of $\delta$ associated to $i$*. A *progressive change of $i^*\delta$* is a natural transformation $a\colon i^*\delta \to \gamma$. A *regressive change of $i^*\delta$* is a natural transformation $b\colon \beta \to i^*\delta$.

If $a\colon i^*\delta \to \gamma$ is a progressive change, then the *progressive $a$-update of $\delta$ made by $i$*, denoted $P(i,a)\colon \delta \to \delta(i,a)$, is the right-hand map in the push-out square

(9)
$$\begin{array}{ccc} i_!i^*\delta & \xrightarrow{\epsilon} & \delta \\ {\scriptstyle i_!a}\downarrow & \ulcorner & \downarrow{\scriptstyle P(i,a)} \\ i_!\gamma & \longrightarrow & \delta(i,a) \end{array}$$

in $\mathcal{D}$–**Set**, where $\epsilon$ is the counit.

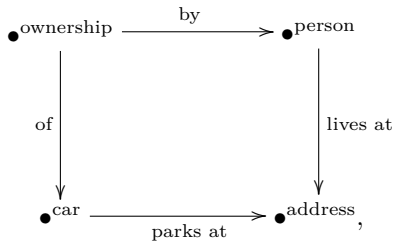
If $b\colon \beta \to i^*\delta$ is a regressive change, then the *regressive $b$-update of $\delta$ made by $i$*, denoted $R(i,b)\colon \delta(i,b) \to \delta$, is the left map in the pull-back

(10)
$$\begin{array}{ccc} \delta(i,b) & \longrightarrow & i_*\beta \\ {\scriptstyle R(i,b)}\downarrow & \lrcorner & \downarrow{\scriptstyle i_*b} \\ \delta & \xrightarrow[\eta]{} & i_*i^*\delta \end{array}$$

in $\mathcal{D}$–**Set**, where $\eta$ is the unit.

A progressive update of $\delta$ involves inserting some new records or identifying records in $\delta$. A regressive update involves deleting some records, or differentiating between records that were once assumed the same.

*Example* 3.1.3. Suppose that a database tracks people, the cars they own, and the addresses of both the car and the person. In other words, let $\mathcal{D}$ be the commutative square




suppose that $\mathcal{C} = \boxed{\bullet^{\text{person}} \longrightarrow \bullet^{\text{address}}}$, and let $U\colon \mathcal{C} \to \mathcal{D}$ be the fully faithful inclusion. We are interested in the effects of progressive and regressive changes that "user" $U$ can make. In this example, we'll focus on a progressive change; in Example 3.1.4 we'll focus on a regressive change. Suppose the 1-column table of addresses is fixed.

Let $\delta\colon \mathcal{D} \to \mathbf{Set}$ be a database state on $\mathcal{D}$ which user $U$ sees as

(11)
$$U^*\delta = \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{\textbf{Person}} \\ \hline \textbf{Id} & \textbf{Address} \\ \hline \text{Hilb1} & \text{16 Wilson Dr.} \\ \hline \text{Hilb2} & \text{16 Wilson Dr.} \\ \hline \text{Russ1} & \text{37 Maple St.} \\ \hline \end{array}.$$

There are two atomic kinds of progressive changes $U$ can make: add a row or declare two rows equivalent. To add a row, let $a\colon U^*\delta \to \gamma$, where

(12)
$$\gamma = \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{\textbf{Person}} \\ \hline \textbf{Id} & \textbf{Address} \\ \hline \text{Hilb1} & \text{16 Wilson Dr.} \\ \hline \text{Hilb2} & \text{16 Wilson Dr.} \\ \hline \text{Russ1} & \text{37 Maple St.} \\ \hline \text{Turi1} & \text{21 Main St.} \\ \hline \end{array}.$$

If we are working in a typed setting (see Section 5.2), then the only two rows that $U$ could possibly declare equivalent are Hilb1 and Hilb2, by using $a'\colon U^*\delta \to \gamma'$ where

$$\gamma' = \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{\textbf{Person}} \\ \hline \textbf{Id} & \textbf{Address} \\ \hline \text{Hilb} & \text{16 Wilson Dr.} \\ \hline \text{Turi1} & \text{21 Main St.} \\ \hline \end{array}.$$

In an untyped setting, we could actually set any two rows of $\gamma$ to be the same, and this change would propagate into the 1-column table of addresses to declare possibly different addresses to be the same.

Regardless, let's continue with the example using the insertion $a\colon U^*\delta \to \gamma$ of (12). According to the procedure set out in (9), we need to compute the maps in

the diagram

$$\begin{array}{ccc} U_!U^*\delta & \longrightarrow & \delta \\ \downarrow & & \\ U_!\gamma & & \end{array}$$

and push out. It is easy to show that for any $x \in \mathcal{C}$–**Set**, (e.g. $x = U^*\delta$ or $x = \gamma$) one has

$$\begin{aligned} U_!x(\bullet^{\text{ownership}}) &= \emptyset, \\ U_!x(\bullet^{\text{car}}) &= \emptyset, \\ U_!x(\bullet^{\text{person}}) &= x(\bullet^{\text{person}}), \text{ and} \\ U_!x(\bullet^{\text{address}}) &= x(\bullet^{\text{address}}). \end{aligned}$$

It follows that the update $\delta \to \delta(U, a)$ is quite benign: it makes no change to the ownership nor the car tables, it only adds the new "Turi1" row. This is the expected result. See also Proposition 4.1.3.

*Example* 3.1.4. We continue with the setup of Example 3.1.3, with $U\colon \mathcal{C} \to \mathcal{D}$ and $\delta$ as above so that $U^*\delta$ is as in (11). A regressive change $b\colon \beta \to i^*\delta$ is a deletion or new distinction. For example

(13)
$$\beta = \begin{array}{|c|c|} \hline \multicolumn{2}{|c|}{\textbf{Person}} \\ \hline \textbf{Id} & \textbf{Address} \\ \hline \text{Hilb1} & \text{16 Wilson Dr.} \\ \hline \text{Hilb2} & \text{16 Wilson Dr.} \\ \hline \end{array}.$$

One can see $U^*\delta$ as an insertion to $\beta$ or see $\beta$ as a deletion to $U^*\delta$; this explains the symmetry in Definition 3.1.2. A new distinction could be created by realizing that the one row Russ1 in $U^*\delta$ was actually representing two different people (say, Bertrand Russell and his wife). Given such a realization, one might think that the best thing to do would be to (progressively) add a new row for Russell's wife, but perhaps making a distinction (with two elements of $\beta$ mapping to Russ1 in $U^*\delta$) would leave the data more consistent.

Regardless, let's continue with the example using the deletion $b\colon \beta \to U^*\delta$. According to the procedure set out in (10), we need to compute the maps in the diagram

$$\begin{array}{ccc} & & U_*\beta \\ & & \downarrow \\ \delta & \longrightarrow & U_*U^*\delta \end{array}$$

and pull back. It is easy to show that for any $x \in \mathcal{C}$–**Set** (e.g. $x = \beta$ or $x = U^*\delta$), one has

$$\begin{aligned} U_*x(\bullet^{\text{ownership}}) &= x(\bullet^{\text{person}}), \\ U_*x(\bullet^{\text{car}}) &= x(\bullet^{\text{address}}), \\ U_*x(\bullet^{\text{person}}) &= x(\bullet^{\text{person}}), \text{ and} \\ U_*x(\bullet^{\text{address}}) &= x(\bullet^{\text{address}}). \end{aligned}$$

The result of the pull back is the regressive change $\delta(U, b)$ which will be a subset of $\delta$. For every table it will consist of the subset of rows in $\delta$ whose image in person or address does not include the deleted row

$$\boxed{\text{Russ1}} \,\|\, \boxed{\text{37 Maple St.}}. \tag{14}$$

This is again the expected result.

3.2. **Local effects of an update.** The following proposition shows that the $i$-view of an update made by $i$ is just the change that generates it.

**Proposition 3.2.1.** *Suppose $i\colon \mathcal{C} \hookrightarrow \mathcal{D}$ is a fully faithful morphism and $a\colon i^*\delta \to \gamma$ is a progressive change. The $i$-view of the progressive $a$-update,*

$$i^*P(i, a)\colon i^*\delta \to i^*\delta(i, a),$$

*is just $a$. That is, $i^*\delta(i, a) \cong \gamma$ and $i^*P(i, a) \cong a$.*

*With $i$ as above, if $b\colon \beta \to i^*\delta$ is a regressive change, then the $i$-view of the regressive $b$-update*

$$i^*R(i, b)\colon i^*\delta(i, b) \to i^*\delta,$$

*is just $b$. That is, $i^*\delta(i, b) \cong \beta$ and $i^*R(i, b) \cong b$.*

*Proof.* The functor $i^*$, being both a left and a right adjoint, commutes with colimits and limits. Hence if we apply it to Diagram (9) we obtain a new push-out square, and if we apply it to Diagram (10) we obtain a new pull-back square. Furthermore, by Lemma 3.1.1(4), the top map of the new push-out square is an isomorphism and the left-hand map is $a$, so the right-hand map is $a$ as well. By the same reasoning, Lemma 3.1.1(5) implies that the right-hand map of the new pull-back diagram is an isomorphism and the bottom map is $b$, so the top map is $b$ as well.

□

Definition 3.1.2 and Proposition 3.2.1 tell us that when a user makes a change to his or her view of the database, it must be classified as either progressive (usually inserting) or regressive (usually deleting) – these two events must be handled differently. As the whole database is updated to reflect that change, the user's view remains consistant.

3.3. **Basic views.** In Section 3.1 we described data-entry views on a database $\mathcal{D}$–**Set**, which we obtain via any fully faithful morphism $i\colon \mathcal{C} \to \mathcal{D}$. More generally a view is an established "query." This notion can be quite broad and utilize algorithms outside the database system itself, such as aggregation of data. In this section we look at something in between these two extremes, roughly those views that can be obtained by a declared protocol of projections, joins, and unions.

**Definition 3.3.1.** Let $\mathcal{C}$ be a database schema. A *basic view setup of $\mathcal{C}$* is a sequence of functors of the form

$$\mathcal{C} \xleftarrow{F} \mathcal{D} \xrightarrow{G} \mathcal{E} \xrightarrow{H} \mathcal{F}.$$

The *basic view functor* for this setup is $H_!G_*F^*\colon \mathcal{C}$–**Set** $\to \mathcal{F}$–**Set**. For a database state $\gamma\colon \mathcal{C} \to$ **Set**, we call $H_!G_*F^*(\delta)$ the *view of* $\gamma$ under this setup.

*Remark* 3.3.2. View functors are sometimes called *polynomial functors* in mathematics. These are functors of the form $F^*G_*H_!$. They are closed under composition (see [GK, Corollary 1.1.4]). Therefore, the sequence of upper-star, lower-star, and lower-shriek functors in Definition 3.3.1 is of no importance.

Every construction in Section 2.5 was a basic view. In other words, joins, unions, images, fixed points, and orbits can all be considered basic views. More basically, one can just take a subschema $F\colon \mathcal{D} \subset \mathcal{C}$ as a basic view. Even this is a significant generalization of data-entry views (for which $F$ was assumed fully faithful). For example in Section 2.4 we discussed the data migration functors for a subcategory that was not full and saw interesting results.

The functoriality of this construction may be useful in considering the "view update problem," in which one hopes to determine what kinds of updates on a view should have what kinds of effects on the database. For more on this, see [P2].

## 4. Updates and their global effects

In Section 3 we discussed what it means for a user to make a change within his or her "data-entry" view and then update the database accordingly. We also showed in Proposition 3.2.1 that such updates show up to the user as consistent with his or her changes. But what happens to the rest of the database? It may be that local changes have rippling effects. For example, we show in this section that if a user adds a row, updates, and then deletes that row, this whole transaction may leave the database in a different state – traces of the inserted row remain. See Example 4.1.4. There is much to be understood here, and this section will barely scratch the surface; see Section 6.3.

### 4.1. **Effects of an update.**

**Definition 4.1.1.** Let $\mathcal{D}$ and $\mathcal{E}$ be database schemas and let $F\colon \mathcal{E} \to \mathcal{D}$ a morphism. Given a morphism $g\colon \delta \to \delta'$ of $\mathcal{D}$-sets, we refer to the morphism $F^*(g)\colon F^*\delta \to F^*\delta'$ as the *effect of $g$ via $F$* (or sometimes by abuse of notation, the *effect of $g$ on $\mathcal{E}$*). If $F^*(g)$ is an isomorphism then we say that *$g$ has no effect on $\mathcal{E}$*.

A database administrator wants to know how updates will affect various parts of the database. As a trivial example, we can show that tables that are "upstream" of a progressive update are unchanged by it and tables that are "downstream" of a regressive update are unchanged by it.

**Definition 4.1.2.** Let $\mathcal{D}$ be a schema and let $i\colon \mathcal{C} \to \mathcal{D}$ and $j\colon \mathcal{E} \to \mathcal{D}$ be morphisms. We say that $\mathcal{E}$ is *upstream of* $\mathcal{C}$ if, for no objects $c \in \mathrm{Ob}(\mathcal{C})$ and $e \in \mathrm{Ob}(\mathcal{E})$ does there does exist a morphism $i(c) \to j(e)$ in $\mathcal{D}$; we say that $\mathcal{E}$ is *downstream of* $\mathcal{C}$ if, for no objects $c \in \mathrm{Ob}(\mathcal{C}), e \in \mathrm{Ob}(\mathcal{E})$ does there is exist a morphism $j(e) \to i(c)$ in $\mathcal{D}$.

**Proposition 4.1.3.** *Let $\mathcal{C} \xrightarrow{i} \mathcal{D} \xleftarrow{j} \mathcal{E}$ be morphisms. If $P\colon \delta \to \delta'$ is a progressive update made by $i$, and $\mathcal{E}$ is upstream of $\mathcal{C}$, then $P$ has no effect on $\mathcal{E}$. Similarly, if $R\colon \delta' \to \delta$ is a regressive update made by $i$, and $\mathcal{E}$ is downstream of $\mathcal{C}$, then $R$ has no effect on $\mathcal{E}$.*

*Proof.* Recall that a progressive change made by $i$ is the right-hand map in some push-out diagram

$$\begin{array}{ccc} i_!i^*\delta & \longrightarrow & \delta \\ \downarrow & \ulcorner & \downarrow P \\ i_!\gamma & \longrightarrow & \delta' \end{array}$$

and a regressive change made by $i$ is the left map in some pull-back diagram

$$\begin{array}{ccc} \delta' & \longrightarrow & i_*\beta \\ {\scriptstyle R}\downarrow & \lrcorner & \downarrow \\ \delta & \longrightarrow & i_*i^*\delta. \end{array}$$

Since $j^*$ is both a left and a right adjoint, we can apply it to either of these and the result will (respectively) be a push-out and a pull-back. The two cases are similar, so we focus on the progressive change.

To show that $j^*P$ is an isomorphism it suffices to show that the map $j^*i_!i^*\delta \to j^*i_!\gamma$ is an isomorphism; in fact it will be the isomorphism $\emptyset \to \emptyset$ because $j^*i_!\phi = \emptyset$ for all $\phi \in \mathcal{C}$–**Set**. To see this, choose an object $e \in \mathrm{Ob}(\mathcal{E})$. Then by Definition 2.1.3

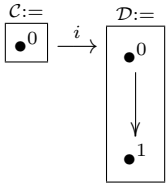
$$j^*i_!\phi(e) = i_!\phi(j(e)) = \operatorname*{colim}_{(i\downarrow j(e))} \Big(\phi \circ \pi_i(j(e))\Big).$$

But since $\mathcal{E}$ is upstream of $\mathcal{C}$, the category $(i \downarrow j(e))$ is empty, so this colimit is $\emptyset$. Thus the map in question is a morphism between initial states, and the only such morphism is an isomorphism.

For the regressive change, everything is appropriately dual. We will find the indexing category $(j(e) \downarrow i)$ for the relevant limit to be empty, and our map will reduce to the unique morphism between terminal states, which is an isomorphism. □

Proposition 4.1.3 offers a glimpse into the kinds of facts that can be proven about the effects of updates. It shows that certain parts of a database are unaffected by certain changes in other parts, but in general updates can have very interesting and hard-to-predict effects. These should be studied further. In the following example we show an update made in one part of the database which *does* affect other parts of the database. We also show that one cannot always "undo" a transaction at the local level.

*Example* 4.1.4. Let $\mathcal{C} := [0]$ denote the terminal category, let $\mathcal{D} := [1]$ denote the free arrow category, and let $i\colon \mathcal{C} \to \mathcal{D}$ denote the inclusion of the first vertex. In pictures:




What happens to a database state $\delta\colon \mathcal{D} \to \mathbf{Set}$ when the user of $\mathcal{C}$ views $\delta$, adds a row, updates $\mathcal{D}$, deletes that row, and then updates $\mathcal{D}$ with that deletion? We calculate the result of such an "undoing" procedure here.

Suppose that $A_0 \to A_1$ is a function of sets, considered as a state $\delta\colon \mathcal{D} \to \mathbf{Set}$. Then $i^*\delta$ is just the set $A_0$. Given a set $X$, considered as a state $\mathcal{C} \to \mathbf{Set}$, one

calculates the push-forwards as

$$i_*X = \boxed{\begin{array}{c} X \\ \downarrow \\ \{\star\} \end{array}} \qquad \text{and} \qquad i_!X = \boxed{\begin{array}{c} X \\ \downarrow \\ X \end{array}}$$

We now walk through the "undoing" procedure outlined above, beginning with our database state $\delta = \boxed{A_0 \to A_1}$. The user of $\mathcal{C}$ views it as $i^*(\delta) = \boxed{A_0}$. To add rows is to find an injection $a\colon A_0 \to B_0$. To update $\mathcal{D}$ is to perform the push-out $\delta \leftarrow i_!i^*\delta \to i_!(B_0)$, more explicitly the push-out of $\mathcal{D}$-sets

$$\begin{array}{ccccc} \boxed{\begin{array}{c} A_0 \\ \downarrow \\ A_1 \end{array}} & = & \boxed{\begin{array}{c} A_0 \\ \downarrow \\ A_0 \end{array}} & \to & \boxed{\begin{array}{c} B_0 \\ \downarrow \\ B_0, \end{array}} \\ \leftarrow & & \to & & \end{array}$$

which is $\delta(i,a) = \boxed{B_0 \longrightarrow A_1 \amalg_{A_0} B_0}$. This roughly means "the new rows added to $A_0$ will be freely added to $A_1$ as well."

Now what happens when we delete those new rows and again update? To delete the new rows is simply to reconsider the injection $b\colon A_0 \to A_1$. To update $\mathcal{D}$ with that deletion is to perform the pull-back $\delta' \to i_*i^*\delta \leftarrow i_*(A_0)$, more explicitly the pull-back of $\mathcal{D}$-sets

$$\begin{array}{ccccc} \boxed{\begin{array}{c} B_0 \\ \downarrow \\ A_1 \amalg_{A_0} B_0 \end{array}} & = & \boxed{\begin{array}{c} B_0 \\ \downarrow \\ \{\star\} \end{array}} & \leftarrow & \boxed{\begin{array}{c} A_0 \\ \downarrow \\ \{\star\}, \end{array}} \\ \to & & \leftarrow & & \end{array}$$

which is $\delta(i,b) = \boxed{A_0 \longrightarrow A_1 \amalg_{A_0} B_0}$. This is the final result of the "undoing" procedure.

Under this process, then, we have not changed the domain, but we have added new rows to the codomain. The upshot is that inserting and then deleting rows locally may have unexpected changes on the global state. "Once information is added to the system, it cannot be easily destroyed."

There are two morphisms $[0] \to [1]$, and we have calculated what happens when you update an insertion and then update a deletion for one of these two morphisms. The reader is invited to try the same exercise for the other morphism $[0] \to [1]$.

## 5. Databases and toposes

A single topos is a generalization of all of set theory; in particular the category **Set** is the most basic topos. The idea is that a set is a static thing, whereas in a topos one has "variable sets" – the topos governs how those sets vary. Every topos $\mathcal{E}$ has an internal language which supports the lambda-calculus; in other words every topos is a cartesian-closed category. It has one additional property, the existence of a "subobject classifier" $\Omega$ – for **Set** this is the object $\{0,1\}$ – subobjects of any $X \in \mathrm{Ob}(\mathcal{E})$ are characterized by maps $X \to \Omega$. Thus a topos is the setting for higher-order predicate logic. In particular, this logic is not necessarily boolean.

Toposes were invented by Grothendieck (see e.g. [GV]) for use in proving the Weil conjectures (a result in the intersection of number theory and algebraic geometry).

Their definition was extended by Lawvere, and the topic has been widely studied. A good reference is [B].

5.1. **The topos of states.** For any category $\mathcal{C}$, the category $\mathcal{C}$–**Set** of functors $\mathcal{C} \to \mathbf{Set}$ is a topos. Thus every database schema has an associated *topos of states*. There is a notion of morphisms of toposes, and these are tightly linked to data migration functors.

**Definition 5.1.1.** A *topos* (or *elementary topos*) is a category $\mathcal{E}$ which is Cartesian-closed (meaning it is closed under products and exponentials) and which has a sub-object classifier $\Omega \in \mathrm{Ob}(\mathcal{E})$. That means that for any object $X \in \mathrm{Ob}(\mathcal{E})$ there is a natural one-to-one correspondence between the set of sub-objects of $X$ (monomorphisms $X' \hookrightarrow X$) and the set of morphisms $X \to \Omega$.

A *geometric morphism* $\mathcal{E} \to \mathcal{E}'$ is an adjunction $\mathcal{E}' \underset{R}{\overset{L}{\rightleftarrows}} \mathcal{E}$ (where $L$ left adjoint and $R$ right adjoint) and where $L$ preserves finite limits. An *essential geometric morphism* $\mathcal{E} \to \mathcal{E}'$ is a geometric morphism $(L, R)$ such that the left adjoint $L$ is also the right adjoint of something. Often, a geometric morphism $(L, R)$ is denoted $(f^*, f_*)$ and if it is an essential geometric morphism then the left adjoint of $f^*$ is denoted $f_!$.

In case it was not clear, the category of database states on a given schema is a topos, and our three data migration functors are the three aspects of essential geometric morphisms, as we make precise in the following proposition.

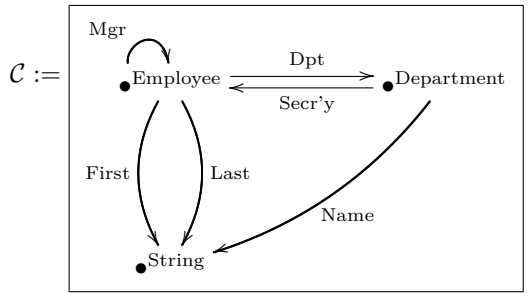
**Proposition 5.1.2.** *Let $\mathcal{C}$ be a schema. The category $\mathcal{C}$–**Set** of states on $\mathcal{C}$ is a topos. Given a morphism of schemas $F\colon \mathcal{C} \to \mathcal{D}$, the adjunction*

$$\mathcal{D}\text{–}\mathbf{Set} \underset{F_*}{\overset{F^*}{\rightleftarrows}} \mathcal{C}\text{–}\mathbf{Set}$$

*of Definition 2.1.2 is an essential geometric morphism, in which the left adjoint of $F^*$ is $F_!$ as in Definition 2.1.3.*

Not all toposes are of the form $\mathcal{C}$–**Set**; those that are (for some $\mathcal{C}$) are called *presheaf toposes* (technically $\mathcal{C}$–**Set** $= \mathbf{Pre}(\mathcal{C}^{\mathrm{op}})$). Presheaf toposes are generally not boolean, i.e. the substates of any database state do not form a boolean algebra. However, they do form a Heyting algebra, which means they have the familiar notions of subobject, implication, union, and intersection. The sense in which they are not boolean is that, while a compliment $\gamma^c$ is defined for each subobject $\gamma \subset \delta$, one does not necessarily have $\gamma \cup \gamma^c = \delta$. Another way to say this is that the compliment of $\gamma^c$ is not necessarily $\gamma$. In general we have $\gamma \subset (\gamma^c)^c$. See [B, Volume 3, Section 1.2].

5.2. **Slice toposes.** As mentioned in 1.1.4, database states in $\mathcal{C}$–**Set** are untyped. For instance, recall Example 1.1.3:




Even though we called one object $\bullet^{\text{String}}$, nothing really guaranteed that a state $\gamma\colon \mathcal{C} \to \mathbf{Set}$ sends that object to a set of strings.

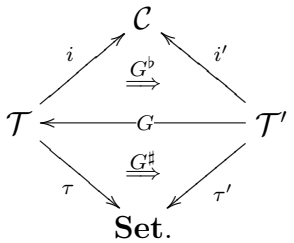
We can correct this using slice toposes. Before continuing, we define slice toposes. We will state some basic facts about them below in Proposition 5.2.4, and Lemmas5.2.7, 5.2.8 and 5.2.9.

**Definition 5.2.1.** Suppose that $\mathcal{C}$ is a category and $c \in \mathrm{Ob}(\mathcal{C})$ is an object. The category $(\mathrm{id}_{\mathcal{C}} \downarrow c)$ is also denoted $\mathcal{C}_{/c}$ and called the *slice of* $\mathcal{C}$ *over* $c$. An object in $\mathcal{C}_{/c}$ is a map $x \to c$ in $\mathcal{C}$ and a morphism in $\mathcal{C}_{/c}$ is a commutative triangle (see Section 2.1). If $\mathcal{C}$ is a topos, then we say that the category $\mathcal{C}_{/c}$ is a *slice topos*; in particular this category will always be a topos (see [MM, IV.7.1]).

We now explain what slice toposes have to do with typing. Suppose that certain columns in a database are supposed to be "typed," meaning that the values are all to be within some fixed set. Let $\mathcal{C}$ be a database schema and let $i\colon \mathcal{T} \subset \mathcal{C}$ denote a subcategory of objects to be typed. To type this subcategory is to fix some functor $\tau\colon \mathcal{T} \to \mathbf{Set}$. If $\mathcal{T}$ is just a set of objects (a discrete category), then $\tau$ assigns a set of possible values to each object in $\mathcal{T}$. Now a database state on $\mathcal{C}$ that is typed by $\tau$ consists of a $\mathcal{C}$-set $F$ together with a map $i^*F \to \tau$. We make this precise.

**Definition 5.2.2.** Let $\mathcal{C}$ be a database schema. A *type signature on* $\mathcal{C}$ consists of a sequence $(\mathcal{T}, i, \tau)$ where $i\colon \mathcal{T} \to \mathcal{C}$ is a functor and $\tau\colon \mathcal{T} \to \mathbf{Set}$ is a functor.

A *morphism* $(\mathcal{T}, i, \tau) \to (\mathcal{T}', i', \tau')$ *of type signatures* consists of a functor $G\colon \mathcal{T}' \to \mathcal{T}$ and two natural transformations, $G^\flat\colon i \circ G \to i'$ and $G^\sharp\colon \tau \circ G \to \tau'$ as in the diagram

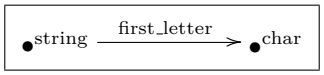



In practice, $G^\flat$ will often be the identity transformation (i.e. in practice the top triangle will often commute, $i' = i \circ G$).

Let $\tau = (\mathcal{T}, i, \tau)$ be a type signature on $\mathcal{C}$. The *category of* $\tau$*-typed states on* $\mathcal{C}$ is the slice topos $\mathcal{C}$–$\mathbf{Set}_{/i_*\tau}$ for which an object is a $\mathcal{C}$-set $F\colon \mathcal{C} \to \mathbf{Set}$ together with a natural transformation $F \to i_*\tau$ and a morphism is a commutative diagram.

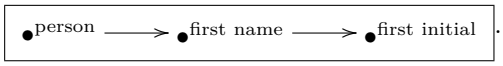
Note that if $\mathcal{T} = \emptyset$ is the empty category then for any schema $\mathcal{C}$ there is a unique choice of $i$ and $\tau$ and that $\mathcal{T}$-typed states on $\mathcal{C}$ are just states on $\mathcal{C}$ in the original sense of Definition 1.1.1.

*Example* 5.2.3. Suppose that $\mathcal{T}$ is the category




and that $\tau\colon \mathcal{T} \to \mathbf{Set}$ sends $\bullet^{\text{char}}$ to the 26 lower-case english letters, $\bullet^{\text{string}}$ to the set of all strings, and that first_letter sends a string to its first letter.

Suppose that $\mathcal{C}$ is the category




Then a $\mathcal{C}$-set does not have to enforce that first names are strings or that first initials are characters. However, if we use the appropriate functor $i\colon \mathcal{T} \to \mathcal{C}$ then the category of $\tau$-typed states on $\mathcal{C}$ will be what we want. Note that a typed state $\gamma \in \mathcal{C}\text{–}\mathbf{Set}_{/i_*\tau}$ *does not* have to contain virtual tables of all strings or all characters – $\gamma(\bullet^{\text{first name}})$ may just be the set of first names of persons in $\gamma(\bullet^{\text{person}})$.

Typing of this sort can be useful in creating "calculated fields"; for example here we have "calculated" the first character of a person's first name using the functionality of the underlying typing system $\mathcal{T}$. If we allow our schemas to be limit sketches (see [B]) rather than mere categories, then we can encode more complex calculations, like the sum across many columns of a given table. However, this will not be detailed in the present paper.

**Proposition 5.2.4.** *A morphism of type signatures $g\colon (\mathcal{T}, i, \tau) \to (\mathcal{T}', i', \tau')$ induces an essential geometric morphism of slice toposes*

$$\mathcal{C}\text{–}\mathbf{Set}_{/i'_*\tau'} \underset{g_*}{\overset{g^*}{\rightleftarrows}} \mathcal{C}\text{–}\mathbf{Set}_{/i_*\tau}.$$

*Proof.* Let $g = (G, G^\flat, G^\sharp)\colon (\mathcal{T}, i, \tau) \to (\mathcal{T}', i', \tau')$ be the morphism of type signatures. We consider the transformation $G^\sharp$ as a map $G^*\tau \to \tau'$ or equivalently $G^\sharp\colon \tau \to G_*\tau'$. Applying $i_*$, we get a morphism $i_*\tau \xrightarrow{i_*G^\sharp} i_*G_*\tau'$. Finally, we compose this map with $G^\flat(\tau)\colon i_*G_*\tau' \to i'_*\tau'$ to get a morphism of $\mathcal{C}$-sets

$$i_*\tau \to i'_*\tau'. \tag{15}$$

The result follows because a morphism of $\mathcal{C}$-sets induces a geometric morphism of slice toposes by Lemma 5.2.8 below. The left adjoint to $g^*$, which we may denote by $g_!\colon \mathcal{C}\text{–}\mathbf{Set}_{/i_*\tau} \to \mathcal{C}\text{–}\mathbf{Set}_{/i'_*\tau'}$, is simply given by composition with map (15). □

*Example* 5.2.5. Let me give one quick example of how morphisms of type signatures can be useful. In early 2010, I was discussing my research on databases with some employees of a large corporation. On entering the building, one mentioned that their ID cards no longer sufficed to unlock the doors – we had to wait for someone inside to open them – and that this was a funny story. It turned out that the data-type for ID numbers was a "short-int," an integer less than 32,768, but that this corporation now had more than 33,000 employees. So the system had stopped working.

This was quite strange to me. I asked whether the difficulty in changing data types to allow for longer integers was simply a storage issue or something more fundamental. They said that it was more fundamental, and in fact that my question showed how little I knew about the horrors of using databases in the real world!

Definition 5.2.2 shows that with category theory, changing type signatures is straightforward. Suppose that $(\mathcal{T}, i, \tau)$ was the old type signature, where $\tau(t) = [-32767..32767]$ for some $t \in \mathcal{T}$. Then let $\mathcal{T}' = \mathcal{T}$, let $G = \mathrm{id}_{\mathcal{T}}$, let $G^\flat$ be the identity transformation, let $\tau'(x) = \tau(x)$ for all $x \neq t$, and let $\tau'(t) = [-32767..65534]$. Here $G^\sharp \colon \tau \to \tau'$ the obvious map (identity on all $x$ except $t$, where it the inclusion of sets $[-32767..32767] \hookrightarrow [-32767..65534]$).

Using Proposition 5.2.4 one can easily transform the database on the old type signature to a database on the new type signature with no loss. All the old IDs would simply be considered as elements of a larger set of possibilities.

One can think of a change of type signatures as a kind of "refactoring"; see [F].

*Remark* 5.2.6. The relational model for databases is based on the logical notion of relations. Given sets $A, B$, and $C$, a relation on them is a subset $R \subset A \times B \times C$. Given a table, each attribute has a set of possible values, and the table is a relation on these sets. In this remark we will say quickly how this connects with the model of databases presented in this paper.

Given a table whose set of attributes (columns) is $A := \{A_1, A_2, \ldots, A_n\}$, consider the terminal morphism out of $A$, denoted $t \colon A \to [0]$. Suppose each element of $A$ represents a type, i.e. a set of elements; we use $\tau$ to designate this typing and write $\tau(A_i) := B_i \in \mathbf{Set}$ for each $1 \leq i \leq n$. We can consider $(A, t, \tau)$ as a type signature on $[0]$; the category $[0]$–$\mathbf{Set}_{/\tau}$ of $\tau$-typed states is the topos of functions $X \to A_1 \times A_2 \times \cdots \times A_n$.

In any topos, it is interesting to look at the lattice $\mathbf{Sub}(1)$ of subobjects of the terminal object. In the setting of $[0]$–$\mathbf{Set}_{/\tau}$, this lattice is the usual boolean algebra of relations $X \hookrightarrow A_1 \times A_2 \times \cdots \times A_n$ on $A$. The dynamics of this lattice (in terms of products, unions, and complements) agrees with the classical "relational algebra," and projection and selection can be obtained using the geometric morphisms presented in Proposition 5.2.4.

Some additional information about slice toposes is in order. Most importantly for our discussion of typing we have the following lemma.

**Lemma 5.2.7.** *Suppose that $\mathcal{C}$ is a category and $\tau \colon \mathcal{C} \to \mathbf{Set}$ is a functor whose category of elements is denoted $\mathcal{G} := \int \tau$. Then there is an equivalence of toposes*

$$\mathcal{C}\text{–}\mathbf{Set}_{/\tau} \simeq \mathcal{G}\text{–}\mathbf{Set};$$

*i.e. the slice topos of $\tau$-typed states on $\mathcal{C}$ is equivalent the topos of states on $\mathcal{G}$ .*

*Proof.* This is [Joh, A.1.1.7].

□

The point then is that typing a database schema does not change how data is managed at a basic level. We can convert a typed schema into a much bigger untyped schema and vice versa. In other words, all the same methods will work regardless of typing.

**Lemma 5.2.8.** *Let $\mathcal{C}$ be a database schema, let $\phi, \psi$ be $\mathcal{C}$-sets, and let $f\colon \phi \to \psi$ be a morphism. Then there is an induced essential geometric morphism of toposes*

$$\mathcal{C}\text{–}\mathbf{Set}_{/\psi} \underset{f_*}{\overset{f^*}{\rightleftarrows}} \mathcal{C}\text{–}\mathbf{Set}_{/\phi}$$

*where the left adjoint of $f^*$ sends $g\colon \gamma \to \phi$ to $f \circ g\colon \gamma \to \psi$.*

*Proof.* This is [MM, IV.7.2].

□

**Lemma 5.2.9.** *Let $F\colon \mathcal{C} \to \mathcal{D}$ be a morphism of database schemas. For any $\mathcal{C}$-set $\gamma$ there is a geometric morphism of slice toposes*

$$\mathcal{D}\text{–}\mathbf{Set}_{/F_*\gamma} \rightleftarrows \mathcal{C}\text{–}\mathbf{Set}_{/\gamma},$$

*and for any $\mathcal{D}$-set $\delta \in \mathcal{D}$–**Set** there is a geometric morphism of slice toposes*

$$\mathcal{D}\text{–}\mathbf{Set}_{/\delta} \rightleftarrows \mathcal{C}\text{–}\mathbf{Set}_{/F^*\delta}.$$

*Proof.* For the first adjunction, one easily checks the isomorphism

$$\mathrm{Hom}_{\mathcal{C}\text{–}\mathbf{Set}_{/\gamma}}((F^*x \to \gamma), (y \to \gamma)) \cong \mathrm{Hom}_{\mathcal{D}\text{–}\mathbf{Set}_{/F^*\gamma}}((x \to F_*\gamma), (F_*y \to F_*\gamma)).$$

For the second adjunction, we need to define the left adjoint as follows. Given a map $g\colon y \to F^*\delta$, let $F_+(y) \to \delta$ be the left-hand map in the pull-back square

$$\begin{array}{ccc} F_+y & \longrightarrow & F_*y \\ \downarrow & \lrcorner & \downarrow{\scriptstyle F_*g} \\ \delta & \xrightarrow[\eta]{} & F_*F^*\delta \end{array}$$

where the bottom map is the unit. Now one checks the isomorphism

$$\mathrm{Hom}_{\mathcal{C}\text{–}\mathbf{Set}_{/F^*\delta}}((F^*x \to F^*\delta), (y, \to F^*\delta)) \cong \mathrm{Hom}_{\mathcal{D}\text{–}\mathbf{Set}_{/\delta}}((x \to \delta), (F_+y \to \delta)).$$

□

5.3. **The "ability topos".** Let $\mathcal{C}$ be a category (database schema) and $F\colon \mathcal{C} \to \mathbf{Set}$ a functor (database state). Then we consider $F$ to be giving us a set of "examples" for every object (type) in $\mathcal{C}$ and a way to transform an example of type $c$ into an example of type $c'$ for every morphism $f\colon c \to c'$ in $\mathcal{C}$. This has been the idea throughout the paper thus far.

Now, suppose instead of a set of examples for each type, we consider a set of abilities (or methods) for each type. Given an ability $\alpha$ for $c'$, we see that $f$ allows $c$ to inherit or delegate that ability. That is, given an example of type $c$, we can transform it into a unique example of type $c'$ which can then achieve the ability $\alpha$; hence $c$ has achieved something as well: $\alpha \circ f$.

For example, if $c$ represents children, $m$ represents mothers, and $c \to m$ sends every child to his or her mother, then any ability of a mother becomes an ability of her child: one says "ask your mother to do this." In computer science terminology, we would call abilities *methods* and speak of $c$ *delegating* a method from $m$.

Coherently assigning a set of methods to each object in $\mathcal{C}$ amounts to a functor (state) $\alpha\colon \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$. There is again a category of such assignments: the objects are functors $\mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ and the morphisms are natural transformations. It is a topos called *the topos of presheaves on* $\mathcal{C}$ (in accordance with classical mathematical terminology) and denoted $\mathbf{Pre}(\mathcal{C}) = \mathcal{C}^{\mathrm{op}}$–**Set**. The point is that given a database

schema meant to model a certain situation, its opposite schema will also represent something relevant to that situation.

## 6. Future work

In this section we discuss some possible future directions that we or others may be interested in taking the ideas presented in this paper. These directions broadly fall into three categories: extending or modifying the definitions, looking at interesting special cases, and better understanding the behavior of data migration.

6.1. **Extending or modifying the definitions.** The idea of a database presented in this paper can be extended in a variety of ways. One of the more exciting is to exchange **Set** for some more interesting category. Recall that if $\mathcal{C}$ is a database schema modeling some situation, then to exemplify this model is to provide a state $\mathcal{C} \to \mathbf{Set}$. This uses the model to capture a moment in time – but what happens as time progresses? What about letting the model vary over space?

If $X$ is a topological space (e.g. $X = \mathbb{R}$, which can be thought of as a time line) then there is a topos $\mathbf{Shv}(X)$ of sheaves on $X$. It is interesting to consider functors $\mathcal{C} \to \mathbf{Shv}(X)$ rather than $\mathcal{C} \to \mathbf{Set}$. Now, if $X = \{\star\}$ is a single point then $\mathbf{Shv}(X) = \mathbf{Set}$ and so we can recover our original definition of a database state. But more generally, a functor $\mathcal{C} \to \mathbf{Shv}(X)$ models a situation distributed over time or space. This idea ties databases in with Goguen's work on "sheaf semantics for concurrent interacting objects" (see [G]). It may have further applications to issues of data governance.

There are other nice categories which we might consider exchanging for **Set**. For example, a functor $\mathcal{C} \to \mathbf{Vect}$, where **Vect** is the category of real vector spaces is (in mathematics) called a *representation* of $\mathcal{C}$. In this case, the rows of each table could be added together or multiplied by a scalar to get new rows. Alternatively one could consider functors $\mathcal{C} \to \mathbf{Top}$; here the primary id column of each table would display a fixed topological space, say $T$ (like a map of something) and each column would display another space to which $T$ maps. A user could highlight a region of $T$ and watch what region becomes highlighted within each column.

Another direction in which to look is into using sketches rather than categories in Definition 1.1.1. Work in this kind of direction has been done before (see e.g. [JRW]), but with a very different perspective on the matter. Using sketches would allow the database administrator to create product tables, union tables, or anything else that can be described in the UML (Unified Modeling Language) visualizations of database schemas. Much of this can be done using the data migration functors, and an interchange between the migration approach and the sketch approach should be considered. However the difference is roughly that sketches allow a table in the schema to *persist as* (constantly update itself in order to be) the product of other tables, whereas data migration functors allow one to *create* a new table which, upon its creation, is the product of said other tables.

Just as the category of states on a schema is a topos and hence has a logic and a language readymade, the category of states on a sketch-schema would also have a logic and language readymade; see [AF]. The ramifications of these ideas in the world of databases should be explored.

6.2. **Interesting special cases.** One of the most interesting classes of categories is that of monoids – categories with one object. As a database schema, a monoid

can represent a collections of "actions" a user can take. For example in a paint-shop program, there are various things one can do: draw something, erase something, move something, etc. These actions form a monoid, which can act on any "canvass" (including the blank one). This monoid can be represented as a database schema $M$ with one table, and the canvass as a state. Even if this table is much too large to store or view, its semantics will conform to those of categorical databases.

Monoids can similarly be made to model any set of "actions" a user could take (e.g. updates of a database!) The generating transactions for a given system form the columns of a table, and the states that it can act on form the rows of that table. If in a new version of the software we want to add detail to a certain action (i.e. subdivide it into smaller parts) or want to make macros (i.e. agglomerate many actions into one), we can do so in a rigorous way using a morphism of monoids. Then, the data migration functors allow us to transform states and transactions from one version of the software to the other.

While this paper has mainly been about the ability of mathematics to serve the database community, we can also turn the ideas around and have database systems help the mathematics community. There are many categories in mathematics that have finite presentations; each of them can be modeled by a database. For example sheaves on a topological space could be entered as database states. Perhaps Oracle could be set to work finding the cohomology of a given space.

6.3. **The behavior of data migration.** The behavior of data migration should be studied in more detail. In particular the database administrator wants to understand and provably predict the effects of updates and ETL processes. We gave some basic facts about this in Sections 3.2 and 4.1 but much more should be said.

It would also be interesting to study what happens to a database over time as the schema continually changes and the old data is continually imported. The state may be examined like a geological survey, in that certain records may have characteristic "formations" that suggest the "ages" from which they came. We believe that theorems could be easily proven about these formations.

One should also consider the view update problem. Given a view of a database, we ask "when can changes made to that view be updated to the whole database? What properties should such updates have?" It would be interesting to study which views (aside from data-entry views, studied in Section 3.1) can have changes made to them be meaningfully reflected in the database.

Finally, we present the following question. Suppose that $\mathcal{C}$ and $\mathcal{D}$ are schemas (managed by separate entities, say $C$ and $D$) and that $F\colon \mathcal{C} \to \mathcal{D}$ is a morphism that is managed by a third party. It would be interesting to see what inferences $C$ could make about $D$ given data received via the pull-back functor $F^*$, or what inferences $D$ could make about $C$ given data received via the push forward functors $F_!$ and $F_*$. In other words, as one entity watches the information coming from a foreign entity, can it deduce something about the internal structure of that entity? Such results could have interesting applications to security, data mining, and even psychology.

## References

[AF] S. Awodey and H. Forssell, *First-order logical duality.* ePrint available: http://front.math.ucdavis.edu/1008.3145.

[B] F. Borceux, *Handbook of categorical algebra 1., 2., 3.* Encyclopedia of Mathematics and its Applications 50, 51, 52. Cambridge University Press, Cambridge, 1994.

[Ber] Philip A. Bernstein, *Generic model management: A database infrastructure for schema manipulation*, pp. 1–6, Springer Berlin/Heidelberg, 2001.

[Dis] Zinovy Diskin, *Databases as diagram algebras: Specifying queries and views via the graph-based logic of sketches*, Tech. report, Frame Inform Systems, 1996.

[DK] Zinovy Diskin and Boris Kadish, *Algebraic graph-oriented=category-theory-based manifesto of categorizing data base theory*, Tech. report, Frame Inform Systems, 1994.

[F] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.

[G] J. Goguen, *Sheaf Semantics for Concurrent Interacting Objects*, Mathematical Structures in Computer Science, 1992, pp. 159 –191.

[GB] Joseph A. Goguen and Rod M. Burstall, *Institutions: abstract model theory for specification and programming*, J. Assoc. Comput. Mach. **39** (1992), no. 1, 95–146.

[GK] N. Gambino and J. Kock, *Polynomial functors and polynomial monads.* ePrint available: http://arxiv.org/pdf/0906.4931v2

[GV] Grothendieck and Verdier: Théorie des topos et cohomologie étale des schémas (known as SGA4). New York/Berlin: Springer. (Lecture notes in mathematics, 269 – 270)

[Joh] P. Johnstone, *Sketches of an elephant, Volume 1,2.* Oxford logic guides 43, 44. The Clarendon Press, Oxford University Press, Oxford, 2002.

[Joy] A. Joyal, *Catlab*, available online: http://ncatlab.org/joyalscatlab/show/Factorisation+systems

[JRW] Michael Johnson, Robert Rosebrugh, and R. J. Wood, *Entity-relationship-attribute designs and sketches*, Theory Appl. Categ. **10** (2002), 94–112 (electronic).

[M] S. Mac Lane, *Categories for the working mathematician* 2nd edition. Graduate texts in mathematics 5, Springer Verlag, New York, 1998.

[MM] S. Mac Lane and I. Moerdijk, *Sheaves in Geometry and Logic: a first introduction to topos theory*, Universitext. Springer-Verlag, New York, 1994.

[Mak] M. Makkai, *Generalized sketches as a framework for completeness theorems I.*. J. Pure Appl. Algebra 115 (1997), no. 1, 49–79.

[P1] B. Pierce, Basic category theory for computer scientists. Foundations of Computing Series. MIT Press, Cambridge, MA, 1991, pp. 391–407.

[P2] A. Bohannon, J. A. Vaughan, and B. C. Pierce. *Relational Lenses: A Language for Updateable Views.* In Principles of Database Systems (PODS), 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[PS] Frank Piessens and Eric Steegmans, *Categorical data-specifications*, Theory Appl. Categ. **1** (1995), No. 8, 156–173 (electronic).

[RW] Robert Rosebrugh and R. J. Wood, *Relational databases and indexed categories*, Category theory 1991 (Montreal, PQ, 1991), CMS Conf. Proc., vol. 13, Amer. Math. Soc., Providence, RI, 1992.

[S1] D. I. Spivak, *Simplicial databases.* 2009. ePrint available: http://arxiv.org/abs/0904.2012

[S2] D. I. Spivak, *Table manipulation in simplicial databases.* 2010. ePrint available: http://arxiv.org/abs/1003.2682

Department of Mathematics, Massachusetts Institute of Technology, Cambridge MA 02139

*E-mail address*: dspivak@math.mit.edu