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 small category and an instance is a set-valued functor on it. We show that morphisms of schemas induce three "data migration functors", which translate instances from one schema to the other in canonical ways. These functors parameterize projections, unions, and joins over all tables simultaneously and can be used in place of conjunctive and disjunctive queries. We also show how to connect a database and a functional programming language by introducing a functorial connection between the schema and the category of types for that language. We begin the paper with a multitude of examples to motivate the definitions, and near the end we provide a dictionary whereby one can translate database concepts into category-theoretic concepts and vice-versa.

CONTENTS

1. INTRODUCTION

This paper has two main goals. The first goal is to present a straightforward category-theoretic model of databases under which every theorem about small categories becomes a theorem about databases. To do so, we will present a category **Sch** of database schemas, which has three important features:

- the category **Sch** is equivalent to **Cat**, the category of small categories,
- the category **Sch** is a faithful model for real-life database schemas, and
- the category **Sch** serves as a foundation upon which high-level database concepts rest easily and harmoniously.

The second goal is to apply this category-theoretic formulation to provide new data migration functors, so that for any translation of schemas $F: \mathcal{C} \to \mathcal{D}$, one can transport instances on the source schema $\mathcal{C}$ to instances on the target schema $\mathcal{D}$ and vice versa, with provable "round-trip" properties. For example, homomorphisms of instances are preserved under all migration functors. While these migration functors do not appear to have been discussed in database literature, their analogues

This project was supported by ONR grant N000141010841.

are well-known in modern programming languages theory, e.g. the theory of dependent types ([Mar]), and polynomial data types ([Kan]). This is part of a deeper connection between database schemas and *kinds* (structured collections of types, see [SH], [Ham]) in programming languages. See also Section 3.6.

An increasing number of researchers in an increasing variety of disciplines are finding that categories and functors offer high-quality models, which simplify and unify their respective fields.[1] The quality of a model should be judged by its efficiency as a proxy or interface—that is, by the ease with which an expert can work with only an understanding of the model, and in so doing successfully operate the thing itself. Our goal in this paper is to provide a high-quality model of databases.

Other category-theoretic models of databases have been presented in the past ([JoM],[BCW],[JRW],[IP],[PS],[DC],[Tui]). Almost all of them used the more expressive notion of sketch where we have used categories. The additional expressivity came at a cost that can be cast in terms of our two goals for this paper. First, the previous models were more complex and this may have created a barrier to wide-spread understanding and adoption. Second, morphisms of sketches do not generally induce the sorts of data migration functors that morphisms of categories do.

It is our hope that the present model is simple enough that anyone who has an elementary understanding of categories (i.e. who knows the definition of category, functor, and natural transformation) will, without too much difficulty, be able to understand the basic idea of our formulation: database schemas as categories, database instances as functors. Moreover, we will provide a dictionary (see Section 3.7, Table 1) whereby the main results and definitions in this paper will simply correspond to results or definitions of standard category theory; this way, the reader can rely on tried and true sources to explain the more technical ideas presented here. Moreover, one may hope to leverage existing mathematical theory to their own database issues through this connection.

Before outlining the plan of the paper in Section 1.2 we will give a short introduction to the fundamental idea on which the paper rests, and provide a corresponding "categorical normal form" for databases, in Section 1.1.

1.1. **Categorical normal form.** A database schema may contain hundreds of tables and foreign keys. Each foreign key links one table to another, and each sequence of foreign keys $T_1 \to T_2 \to \cdots \to T_n$ results in a function f from the set of records in T_1 to the set of records in T_n. It is common that two different foreign key paths, both connecting table T_1 to table T_n, may exist; and they may or may not define the same mapping on the level of records. For example, an operational "landline" phone is assigned a phone number whose area code corresponds to the

[1] Aside from mathematics, in which category-theoretic language and theorems are indispensable in modern algebra, geometry, and topology, category theory has been successful in: programming language theory [Mog],[Pie]; physics [BS],[DI]; materials science [SGWB],[GSB]; and biology [Ros],[EV].

region in which the physical phone is located. Thus we have two paths $OLP \to R$:
(1)

$$\begin{tikzcd}
 & \overset{N}{\boxed{\text{phoneNum}}} \arrow[r, "\text{has}"] \arrow[dr, phantom, "\simeq"] & \overset{C}{\boxed{\text{areaCode}}} \arrow[d, "\text{correspondsTo}"] \\
\overset{OLP}{\boxed{\text{operationalLandlinePhone}}} \arrow[ur, "\text{assigned}"] \arrow[r, "\text{is}"'] & \overset{P}{\boxed{\text{physicalPhone}}} \arrow[r, "\text{locatedIn}"'] & \overset{R}{\boxed{\text{region}}}
\end{tikzcd}$$

and the data architect for this schema knows whether or not these two paths should always produce the same mapping. In (1), the two paths $OLP \to R$ do produce the same mapping, and the $\simeq$ sign is intended to record that fact. For contrast, if we replace operationalLandlinePhone with operationalMobilePhone (OMP), the two paths $OMP \to R$ would not produce the same mapping, because a cellphone need not be currently located in the region indicated by its area code. Thus we would get a similar but different diagram,

(2)

$$\begin{tikzcd}
 & \overset{N}{\boxed{\text{phoneNum}}} \arrow[r, "\text{has}"] & \overset{C}{\boxed{\text{areaCode}}} \arrow[d, "\text{correspondsTo}"] \\
\overset{OMP}{\boxed{\text{operationalMobilePhone}}} \arrow[ur, "\text{assigned}"] \arrow[r, "\text{is}"'] & \overset{P}{\boxed{\text{physicalPhone}}} \arrow[r, "\text{locatedIn}"'] & \overset{R}{\boxed{\text{region}}}
\end{tikzcd}$$

We are emphasizing here that the notion of *path equivalence* is an important and naturally-arising integrity constraint, which provides a crucial clue into the intended semantics of the schema. Its enforcement is often left to the application layer, but it should actually be included as part of the schema ([JD]). Including path equivalence information in the database schema has three main advantages:

- it permits the inclusion of "hot" query columns without redundancy,
- it provides an important check for creating schema mappings, and
- it promotes healthy schema evolution.

A *category* (in the mathematical sense) is roughly a graph with one additional bit of expressive power: the ability to declare two paths equivalent. We now have the desired connection between database schemas and categories: Tables in a schema are specified by vertices (or as we have drawn them in Diagram (1), by boxes); columns are specified by arrows; and functional equivalence of foreign key paths are specified by the category-theoretic notion of path equivalence (indicated by the $\simeq$ symbol). The categorical definition of schema will be presented rigorously in Section 3.

The above collection of ideas leads us to the following normal form for databases.

Definition 1.1.1. A database is in *categorical normal form* if

- every table t has a single primary key column 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 exists some *target table* t' such that the value in each cell of column c refers to some row-id of t'. We denote

this relationship by

$$c\colon t \to t';$$

- in particular, if some column d of t consists of "pure data" (such as strings or integers), then its target table t' is simply a 1-column table (i.e. a controlled vocabulary, containing at least the active domain of column d), and we still write $d\colon t \to t'$; and
- when there are two paths p, q through the database from table t to table u (denoted $p\colon t \to u$, $q\colon t \to u$) and it is known to the schema designer that p and q should correspond to the same mapping of row-ids, then this path equivalence must be declared as part of the schema. We denote this path equivalence by

$$p \simeq q.$$

1.2. **Plan of the paper.** We begin in Section 2 by giving a variety of examples, which illustrate the virtues of the category-theoretic approach. These include conceptual clarity, simplified data migration, updatable views, interoperability with RDF data, and a close connection between data and program. In Section 3 we will give the precise definitions of categorical schemas and translations, and show that the category thereof, denoted **Sch**, is equivalent to **Cat**, the category of small categories. In this section we will also define the category of database instances on a given schema. In Section 4 we will define the data migration functors associated to a translation and begin to wrap up our tour by returning to the examples from Section 2 for a more in-depth treatment. We finish this work in Section 5, where we discuss data types and filtering.

1.3. **Terminology and notation.** One obstacle to writing this paper is a certain overlap in terminology between databases and categories: the word "object" is commonly used in both contexts. While object databases are interesting and perhaps relevant to some of the ideas presented here, we will not discuss them at all, hence keeping the namespace clear for categorical terminology. **Unless otherwise specified, the word *object* will always be intended in the category-theoretic sense.**

Say we have maps $A \xrightarrow{f} B \xrightarrow{g} C$; we may denote their composite $A \to C$ in one of two ways, depending on what seems more readable. The first is called "diagrammatic order" and is written as $f;g$. The second is called "classical order" and is written as $g \circ f$. We may sometimes choose not to write a symbol between f and g, and in that case we use diagrammatic order $fg\colon A \to C$.

1.4. **Acknowledgments.** The present paper is a total revamping of [Sp1], which, in an attempt to accommodate two disjoint communities (mathematicians and database experts), ended up as a sprawling and somewhat incoherent document. My thanks go to the referees and to Bob Harper for pointing me toward a streamlined presentation. I would also like to thank Peter Gates, Dave Balaban, John Launchbury, and Greg Morrisett for many useful conversations. I appreciate very much the encouragement of Jack Morava, especially in the form his vision ([Mor]) of how ideas from this paper may be useful for exposing patterns in pure mathematics. Special thanks also go to Scott Morrison for coding many of the ideas from this paper into working form, available online for demonstration or open-source participation at http://categoricaldata.net/.

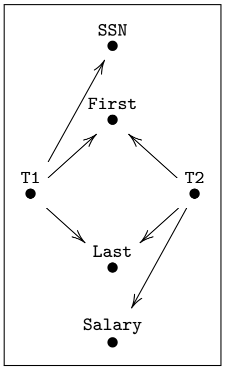
2. Virtues by Example

In what follows we illustrate the merits of the category-theoretic approach by way of several examples. One thing to note is that each of these features flows naturally from our compact mathematical definitions of schemas and translations. These definitions will be given in Section 3.

2.1. **Conceptual clarity.** In a categorical schemas (Definition 3.2.6), every table is a vertex and every column is an arrow. An arrow $\overset{\mathtt{T}}{\bullet}\xrightarrow{c}\overset{\mathtt{U}}{\bullet}$ represents a column of table `T`, with target table `U`, i.e. a foreign key constraint declaring that each cell in column c refers to a row-id in table `U`. We draw a box around our system of vertices and arrows, and the result is a categorical representation of the schema[2].

Example 2.1.1. The following picture represents a schema S with six tables, two of which are multi-column "fact tables" and four of which are 1-column "leaf" tables.

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



The fact table `T1` has three columns (pointing to `SSN`, `First`, `Last`), in addition to its ID column; the fact table `T2` also has three non-ID columns (pointing to `First`, `Last`, `Salary`). The leaf tables, `SSN`, `First`, `Last`, and `Salary` have only ID columns, as is seen by the fact that no arrows emanate from them.

As a set of tables, an instance on schema $\mathcal{C}$ may look something like this:

(4)

T1			
ID	**SSN**	**First**	**Last**
T1-001	115-234	Bob	Smith
T1-002	122-988	Sue	Smith
T1-003	198-877	Alice	Jones

T2			
ID	**First**	**Last**	**Salary**
T2-A101	Alice	Jones	$100
T2-A102	Sam	Miller	$150
T2-A104	Sue	Smith	$300
T2-A110	Carl	Pratt	$200

[2] A system of vertices and arrows of this sort is called a graph. A graph can be considered as a kind of category (a so-called *free category*) in which no path equivalences have been declared.

(5)

SSN
ID
115-234
118-334
122-988
198-877
342-164

First
ID
Adam
Alice
Bob
Carl
Sam
Sue

Last
ID
Jones
Miller
Pratt
Richards
Smith

Salary
ID
\$100
\$150
\$200
\$250
\$300

The thing to recognize here is that each column header c of `T1` (respectively, of `T2`) points to some target table, in such a way that every cell in column c refers to a row-id in that target table.[3] The leaf tables serve as controlled vocabularies for the fact tables.

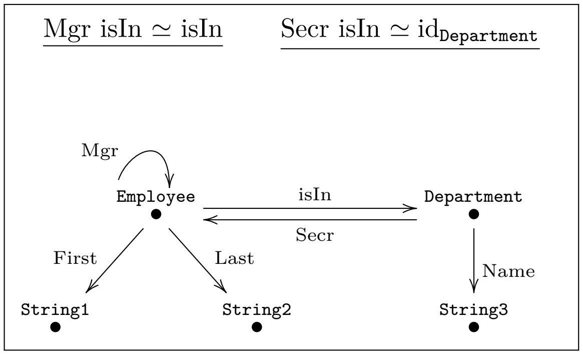
Notation 2.1.2. In Example 2.1.1, we wrote out all four leaf tables (Display 5). In the future we will not generally write out leaf tables for space reasons. In fact, any table that does not add explanatory power to a given example may be left out from our displays.

Example 2.1.3. In this example we present a schema $\mathcal{C}$ that includes path equivalences, and hence takes advantage of the full expressivity of categories. We imagine a company with employees and departments; every employee is in a department, every employee has a manager employee, and every department has a secretary employee. Using path equivalences, we enforce the following facts:

- the manager of an employee is in the same department as that employee, and
- the secretary of a department is in that department.

These facts are recorded as equations at the top of the following diagram:

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



As a set of tables, an instance on $\mathcal{C}$ may look something like this:

[3] In the case that c is the ID column of `T1`, the target table to which c points is `T1`, and each cell in column c is a row-id in `T1` that refers to itself.

Employee

ID	**First**	**Last**	**Mgr**	**isIn**
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Alan	Turing	103	q10

Department

ID	**Name**	**Secr**
q10	Sales	101
x02	Production	102

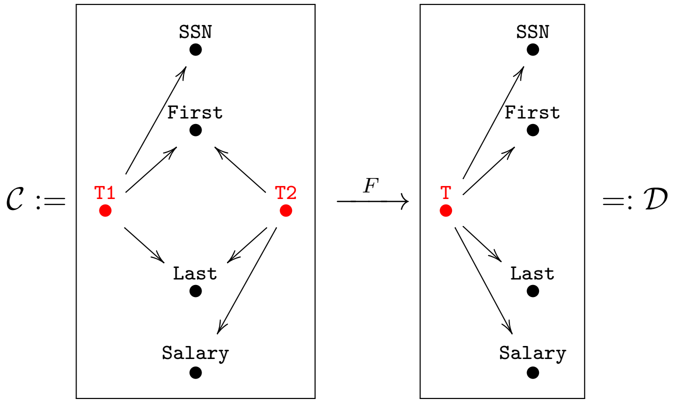
It is no coincidence that there are a total of six non-ID columns in the two tables and a total of six arrows in the schema $\mathcal{C}$. The equations can be checked on these cells; for example we can check the first equation on row-id 101:

$$101.\text{Manager}.\text{isIn} = 103.\text{isIn} = \text{q10} \qquad \text{and} \qquad 101.\text{isIn} = \text{q10}.$$

An instance I on $\mathcal{C}$ is only valid if the two equations hold for every row in I.

2.2. **Simplified data migration.** A translation $F \colon \mathcal{C} \to \mathcal{D}$ of schemas (Definition 3.3.1) is a mapping that takes vertices in $\mathcal{C}$ to vertices in $\mathcal{D}$ and arrows in $\mathcal{C}$ to *paths* in $\mathcal{D}$; in so doing, it must respect arrow sources, arrow targets, and path equivalences. We will be using the following translation $F \colon \mathcal{C} \to \mathcal{D}$ throughout Section 2.2.

(7)



The mapping F is drawn as suggestively as possible. In the future, we will rely on this "power of suggestion" to indicate the translations, but this time we will be explicit. Each of the four leaf vertices, `SSN`, `First`, `Last`, and `Salary` in $\mathcal{C}$ is mapped to the vertex in $\mathcal{D}$ of the same label. The two other vertices in $\mathcal{C}$, namely `T1` and `T2`, are mapped to vertex `T` in $\mathcal{D}$. Since translations must respect arrow sources and targets, there is no additional choice about where the arrows in $\mathcal{C}$ are sent; for example F sends the arrow `T1` → `First` in $\mathcal{C}$ to the arrow `T` → `First` in $\mathcal{D}$.

We have now specified a translation F from schema $\mathcal{C}$ to schema $\mathcal{D}$, pictured in Diagram 7. Springing forth from this translation are three data migration functors, which we will discuss in turn in Examples 2.2.1, 2.2.3, and 2.2.5. They migrate instance data on $\mathcal{D}$ to instance data on $\mathcal{C}$ and vice versa. Instances on $\mathcal{C}$ (respectively on $\mathcal{D}$) form a category, which we denote by $\mathcal{C}$–**Inst** (respectively by $\mathcal{D}$–**Inst**); they are defined in Definition 3.5.1. We have the following chart, the jargon of which will be introduced shortly:

Data migration functors induced by a translation $F\colon \mathcal{C} \to \mathcal{D}$			
Name	**Symbol**	**Long symbol**	**Idea of definition**
Pullback	Δ_F	$\Delta_F\colon \mathcal{D}$–**Inst** $\to \mathcal{C}$–**Inst**	Composition with F
Right Pushforward	Π_F	$\Pi_F\colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**	Right adjoint to Δ_F
Left Pushforward	Σ_F	$\Sigma_F\colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**	Left adjoint to Δ_F

Everything in the chart will be defined in Section 4. For now we give three examples. In each, we will be starting with the translation $F\colon \mathcal{C} \to \mathcal{D}$, given above in Diagram (7).

Example 2.2.1 (Pullback). Let $F\colon \mathcal{C} \to \mathcal{D}$ be the translation given in Diagram (7). In this example, we explore the data migration functor $\Delta_F\colon \mathcal{D}$–**Inst** $\to \mathcal{C}$–**Inst**[4] by applying it to a $\mathcal{D}$-instance J. Note that even though our translation F points forwards (from $\mathcal{C}$ to $\mathcal{D}$), our migration functor Δ_F points "backwards" (from $\mathcal{D}$-instances to $\mathcal{C}$-instances). We will see why it works that way, but first we bring the discussion down to earth by working with a particular $\mathcal{D}$-instance.

Consider the instance J, on schema $\mathcal{D}$, defined by the table

(8) J:=

`T`				
ID	**SSN**	**First**	**Last**	**Salary**
XF667	115-234	Bob	Smith	\$250
XF891	122-988	Sue	Smith	\$300
XF221	198-877	Alice	Jones	\$100

and having the four leaf tables from Example 2.1.1, Display (5). Pulling back along the translation F, we are supposed to get an instance $\Delta_F(J)$ on schema $\mathcal{C}$, which we must describe. But the description is easy: $\Delta_F(J)$ splits up the columns of table `T` according to the translation F. The four leaf tables will be exactly the same as above (i.e. the same as in Example 2.1.1 (5)), and the two fact tables will be something like[5]

(9)

`T1`			
ID	**SSN**	**First**	**Last**
XF667T1	115-234	Bob	Smith
XF891T1	122-988	Sue	Smith
XF221T1	198-877	Alice	Jones

`T2`			
ID	**First**	**Last**	**Salary**
A21	Alice	Jones	\$100
A67	Bob	Smith	\$250
A91	Sue	Smith	\$300

The fact that `T1` and `T2` are simply projections of `T` is a result of our choice of translation F.

Remark 2.2.2. We have seen that **the pullback functor Δ_F**, which arises naturally for any translation F between schemas, **automatically produces projections**.

In the next two examples, we will explore the right and left pushforward migration functors induced by the translation $F\colon \mathcal{C} \to \mathcal{D}$ given in Diagram (7). These functors, denoted Π_F and Σ_F, send $\mathcal{C}$-instances to $\mathcal{D}$-instances. Thus we start with the instance I (which was presented in Example 2.1.1) and explain its pushforwards $\Pi_F(I)$ and $\Sigma_F(I)$ below in Examples 2.2.3 and 2.2.5, respectively.

[4] We have not defined Δ_F yet; this will be done in Section 4.1.

[5] There may be choice in the naming convention for row-ids.

Example 2.2.3 (Right Pushforward). Let $F\colon \mathcal{C} \to \mathcal{D}$ be the translation given in Diagram (7). In this example, we explore the data migration functor $\Pi_F\colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**[6] by applying it to the $\mathcal{C}$-instance I shown in Displays (4) and (5). Note that our migration functor Π_F points in the same direction as F: it takes $\mathcal{C}$-instances to $\mathcal{D}$-instances. We now describe the $\mathcal{D}$-instance $\Pi_F(I)$, which has four leaf tables $\Pi_F(I)(\mathtt{SSN})$, etc., and one fact table $\Pi_F(I)(\mathtt{T})$.

The four leaf tables of $\Pi_F(I)$ will be as in Display (5). The fact table of $\Pi_F(I)$ will be the join of T1 and T2:

T

ID	**SSN**	**First**	**Last**	**Salary**
T1-002T2-A104	122-988	Sue	Smith	$300
T1-003T2-A101	198-877	Alice	Jones	$100

Remark 2.2.4. We have seen that **the right pushforward functor** Π_F, which arises naturally for any translation F between schemas, **automatically produces joins**.

Example 2.2.5 (Left Pushforward). Let $F\colon \mathcal{C} \to \mathcal{D}$ be the translation given in Diagram (7). In this example, we explore the data migration functor $\Sigma_F\colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**[7] by applying it to the $\mathcal{C}$-instance I shown in Displays (4) and (5). Note that our migration functor Σ_F points in the same direction as F: it takes $\mathcal{C}$-instances to $\mathcal{D}$-instances. We now describe the $\mathcal{D}$-instance $\Sigma_F(I)$, which has four leaf tables $\Sigma_F(I)(\mathtt{SSN})$, etc., and one fact table $\Sigma_F(I)(\mathtt{T})$.

Instead of being a join, as in the case of $\Pi_F(I)$ above, the fact table T in instance $\Sigma_F(I)$ will be the union of T1 and T2. One may wonder then how the category theoretic construction will deal with the fact that records in T1 do not have salary information and the records in T2 do not have SSN information. The answer is that the respective cells are *Skolemized*. In other words, the universal answer is to simply add a brand new "variable" wherever one is needed in and downstream of T. Thus in instance $\Sigma_F(I)$, table T looks like this:

T

ID	**SSN**	**First**	**Last**	**Salary**
T1-001	115-234	Bob	Smith	T1-001.Salary
T1-002	122-988	Sue	Smith	T1-002.Salary
T1-003	198-877	Alice	Jones	T1-003.Salary
T2-A101	T2-A101.SSN	Alice	Jones	$100
T2-A102	T2-A102.SSN	Sam	Miller	$150
T2-A104	T2-A104.SSN	Sue	Smith	$300
T2-A110	T2-A110.SSN	Carl	Pratt	$200

The Skolem variables (such as T1-001.Salary) can be equated with actual values later. They can also be equated with each other; for example we may know that T1-001.Salary=T2-002.Salary, without knowing the value of these salaries.

Remark 2.2.6. We have seen that **the left pushforward functor** Σ_F, which arises naturally for any translation F between schemas, **automatically produces unions and automatically Skolemizes unknown values**.

[6] We have not defined Π_F yet; this will be done in Section 4.2.

[7] We have not defined Σ_F yet; this will be done in Section 4.3.

2.3. **Updatable views and linked multi-views.** Suppose we have a translation $F\colon \mathcal{C} \to \mathcal{D}$. In this case we can consider $\mathcal{D}$ as a view on $\mathcal{C}$ and consider $\mathcal{C}$ a view on $\mathcal{D}$. Unlike the classical version of views, our definition allows for arbitrarily many foreign keys between view tables; indeed, both $\mathcal{C}$ and $\mathcal{D}$ can be arbitrary schemas. Typical relational databases management systems such as SQL do not support "linked multi-views", i.e. multiple view tables with foreign keys between them. For our data migration functors Π_F, Σ_F and Δ_F, this is no problem.

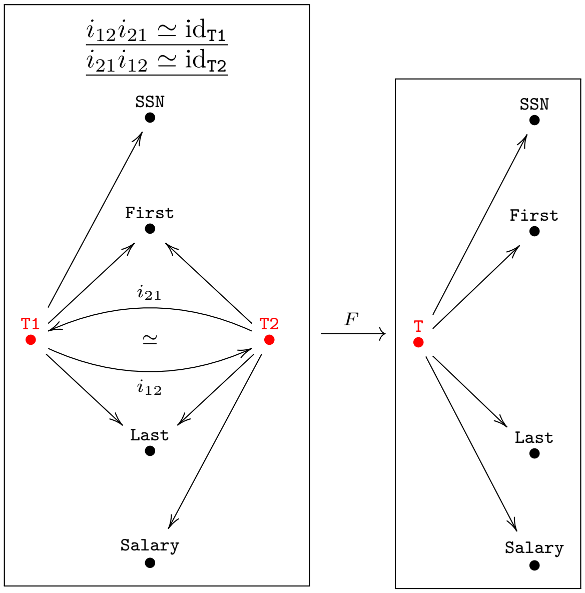
In fact, by the very nature of these three migration functors (i.e. by definition of the fact that they are functors), we have access to powerful theorems relating updates of $\mathcal{C}$-instances to updates of $\mathcal{D}$-instances. For example, given an instance I on $\mathcal{D}$ whose Δ_F-view is the instance $J = \Delta_F(I)$ on $\mathcal{C}$, and given an update $J \to J'$ on $\mathcal{C}$, there is a unique update $I \to \Pi_F J'$, of instances on $\mathcal{D}$. A similar result holds for Σ_F in place of Π_F: these facts follow from the fact that Π (respectively Σ_F) is "adjoint" to Δ_F. See Section 4.

The view update problem is often phrased as asking that "the round trips are equivalences," ([BCFGP]), which for us amounts to the composites $\Delta_F\Pi_F$ and $\Pi_F\Delta_F$ (respectively $\Sigma_F\Delta_F$ and $\Delta_F\Sigma_F$) being isomorphisms. This will only happe in case F is an equivalence of categories. However, our data migration adjunctions provide view updates in more general circumstances, and these have provable formal properties (e.g. Σ_F and Δ_F commute with inserts and Π_F and Δ_F commute with deletes). But of course the best formal properties occur when F is an equivalence.

The following example shows two things. First, it gives an example of a linked multi-view (foreign keys between views). Second, the translation F is an equivalence of categories (a fact which relies essentially on the fact that $\mathcal{C}$ has path equivalences declared), and so the data migration functors Δ, Π, and Σ are also equivalences—they exhibit no information loss.

Example 2.3.1. Consider the two schemas drawn here:

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


$i_{12}i_{21} \simeq \mathrm{id}_{\mathtt{T1}}$
$i_{21}i_{12} \simeq \mathrm{id}_{\mathtt{T2}}$
SSN
First
i_{21}
T1
$\simeq$
T2
i_{12}
Last
Salary
F
T
SSN
First
Last
Salary

$=: \mathcal{D}$

The arrows i_{12} and i_{21}, which are declared to be mutually inverse, ensure that the data which can be captured by schema $\mathcal{C}$ is equivalent to that which can be captured by schema $\mathcal{D}$. The translation F sends i_{12} and i_{21} to the trivial path $\mathrm{id}_{\mathtt{T}}$ on `T` (see Definitions 3.2.1 and 3.3.1, and Example 3.3.3).

If table `T` is as in Example 2.2.1 (8), then its pullback under Δ_F to an instance on $\mathcal{C}$ is similar to (9), but with additional columns i_{12} and i_{21} (because our schema has additional arrows i_{12} and i_{21}):

(11)

T1				
ID	**SSN**	**First**	**Last**	i_{12}
XF667T1	115-234	Bob	Smith	A67
XF891T1	122-988	Sue	Smith	A91
XF221T1	198-877	Alice	Jones	A21

T2				
ID	**First**	**Last**	**Salary**	i_{21}
A21	Alice	Jones	\$100	XF221T1
A67	Bob	Smith	\$250	XF667T1
A91	Sue	Smith	\$300	XF891T1

The foreign key columns i_{12} and i_{21} on the $\mathcal{C}$-view keep track of the data necessary for successful round-tripping. An update to a $\mathcal{D}$-instance will yield a corresponding update to a $\mathcal{C}$-instance and vice versa. The fact that F is an *equivalence of categories* implies that Σ_F, Π_F, and Δ_F are also equivalences of categories, and roundtrip isomorphisms will hold for all possible updates.

2.4. **Interoperability with RDF data.** The *Resource Descriptive Framework (RDF)* is the semantic web standard data format [KC]. The basic idea is to encode all facts in terms of basic

(Subject Predicate Object)

triples, such as (Bob hasMother Sue). There are papers devoted to understanding the transformation from relational databases to RDF triple stores, and vice versa ([ASX],[KT]). In this section we will assume a basic familiarity with the jargon of that field, such as *URI* (uniform resource identifier).

Category-theoretically, the formulation of RDF triple stores is quite simple. Given a schema $\mathcal{C}$, a triple store over $\mathcal{C}$ is a category $\mathcal{S}$ (representing the triples) and a functor $\pi\colon \mathcal{S} \to \mathcal{C}$ (representing their types). The objects in $\mathcal{S}$ are URIs; the arrows in $\mathcal{S}$ are triples

$$\overset{\mathtt{Subject}}{\bullet} \xrightarrow{\text{Predicate}} \overset{\mathtt{Object}}{\bullet}$$

Given an object $c \in \mathcal{C}$ in the schema, the inverse-image $\pi^{-1}(c) \subseteq \mathcal{S}$ consists of all URIs of type c. Given an arrow $f\colon c \to c'$ in $\mathcal{C}$, the inverse image is the f-relation between $\pi^{-1}(c)$ and $\pi^{-1}(d)$.[8]

There is a basic category-theoretic operation that converts a relational database instance into an RDF triple store (and a straightforward inverse as well, converting an RDF triple store into a relational database instance). It is called the *Grothendieck construction.* Consider for example the instance I from Example 2.1.3 Display (6):

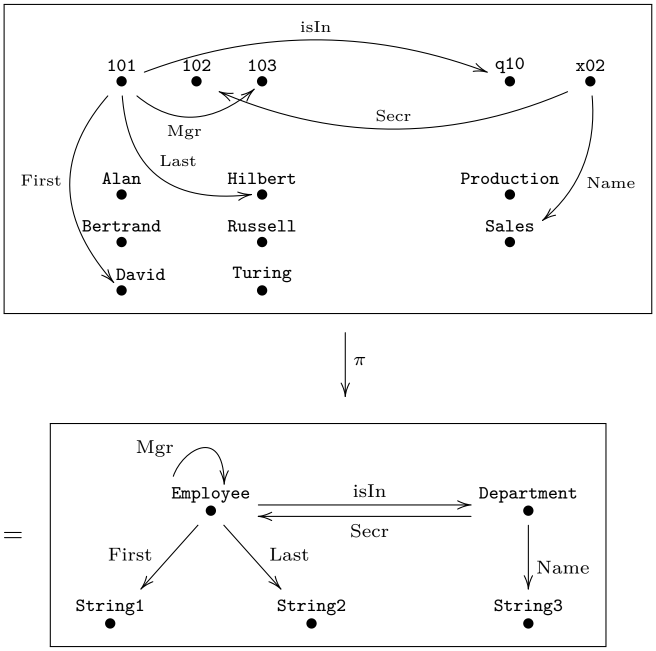
Employee				
ID	**First**	**Last**	**Mgr**	**isIn**
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Alan	Turing	103	q10

Department		
ID	**Name**	**Secr**
q10	Sales	101
x02	Production	102

[8] This relation can be functional or inverse functional, as dictated by the RDF schema; the subject can be understood category-theoretically by so-called "lifting constraints" (see [Sp4]).

Taking the Grothendieck construction yields the following triple store $\mathcal{S} = Gr(I)$, where each arrow designates a RDF triple, as above:

(12) $\mathcal{S} =$



$\mathcal{C} =$

In Display (12), ten arrows have been left out of the picture of S, (e.g. the arrow $\overset{\texttt{102}}{\bullet} \xrightarrow{\text{Last}} \overset{\texttt{Russell}}{\bullet}$ is not pictured) for readability reasons. The point is that the RDF triple store associated to instance I is nicely represented using the standard Grothendieck construction.

2.5. **Close connection between data and program.** Currently, there is an "impedance mismatch" between databases and programming languages; their respective formulations and underlying models do not cohere as well as they should ([CI]). Whereas the programming languages (PL) community has embraced category theory for the conceptual clarity and expressive power it brings, most database theorists tend to concentrate on practical considerations, such as speed, reliability, and scalability. The importance of databases in the modern world cannot be overstated, and yet in order for databases to reach their full potential, better theoretical integration with applications must be developed.

As stated in the Introduction (Section 1), the first goal of this paper is to present a straightforward model of databases under which every theorem about small categories becomes a theorem about databases. Thus the favorite category of PL theorists, namely the category **Type** of types and terms (for some fixed λ-calculus, see [Awo, Section 6.5]), is a kind of infinite database schema: its tables correspond to types and its foreign key columns correspond to terms. Of course, unlike real-world databases in which tables model real-world entities and their relationships (such as people and their heights), the schema **Type** models mathematical entities and their relationships (such as integers and their factorials). However, these ideas clearly live in the same platonic realm, so to speak, and this notion is expressed by saying that both database schemas and **Type** can be considered as categories and related by functors.

This leads to nice integration between data and program. For example many spreadsheet capabilities, such summing up the values in two columns to get the value in a third, can be included at the schema level. At this point in the paper, we do not yet have the necessary machinery to show exactly how that should work (see Section 5.1), but in the following diagram one can see the schematic presentation of the relevant subcategory of **Type**:

$$P := \begin{tikzcd}\texttt{(Int,Int)} \bullet \arrow[r, bend left, "\text{outl}"] \arrow[r, "+"] \arrow[r, bend right, "\text{outr}"'] & \bullet \texttt{Int}\end{tikzcd} \tag{13}$$

The point is to simultaneously see two different things within this one diagram (like an optical illusion). The first thing to see in Diagram (13) is a database schema. In schema P, we have two tables:

(Int,Int)			
ID	**outl**	**outr**	**+**
P0c0	0	0	0
P1c0	1	0	1
P1c1	1	1	2
P0c1	0	1	1
P2c0	2	0	2
P2c1	2	1	3
P2c2	2	2	4
P1c2	1	2	3
P0c2	0	2	2
⋮	⋮	⋮	⋮

Int
ID
0
1
2
3
4
⋮

The second thing to see in Diagram (13) is a subcategory $P \subseteq$ **Type**, i.e. a close connection to standard PL theory. The same category P is viewed extensionally in the context of databases and intentionally in the context of programs. This dual citizenship of categories makes category theory a good candidate for solving the impedance mismatch between databases and programming languages.

3. Definitions

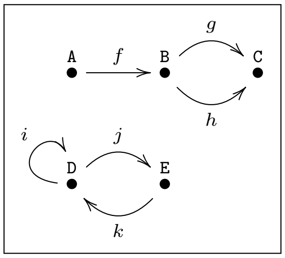
In this section, our main goal is to define a category of schemas and translations, and to show that it is equivalent to **Cat**, the category of small categories. Along the way we will define the category of instances on a given schema. Finally, we will give a dictionary that one can use to translate between database concepts (e.g. found in [EN]) and category-theoretic concepts (e.g. found in [Mac]).

3.1. Some references.

Throughout this section, we will assume the reader has familiarity with the fundamental notions of category theory: objects, morphisms, and commutative diagrams within a category; as well as categories, functors, and natural transformations. There are many good references on category theory, including [LS], [Sic], [Pie], [BW1], [Awo], and [Mac]; the first and second are suited for general audiences, the third and fourth are suited for computer scientists, and the fifth and sixth are suited for mathematicians (in each class the first reference is easier than the second). One may also see [SK] for a different perspective.

3.2. **Graphs, Paths, Schemas, and Instances.** A graph (sometimes called a directed multi-graph) is a collection of vertices and arrows, looking something like this:

(14)



This is one graph with two connected components; it has five vertices and six arrows.

Definition 3.2.1. A *graph* G is a sequence $G = (A, V, src, tgt)$, where A and V are sets (respectively called *the set of arrows* and *the set of vertices* of G), and $src\colon A \to V$ and $tgt\colon A \to V$ are functions (respectively called *the source function* and *the target function* for G). If $a \in A$ is an arrow with source $src(a) = v$ and target $tgt(a) = w$, we draw it as

$$v \xrightarrow{a} w.$$

Definition 3.2.2. Let $G = (A, V, src, tgt)$ be a graph. A *path of length* n in G, denoted $p \in \mathrm{Path}_G^{(n)}$ is a head-to-tail sequence

$$p = (v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \xrightarrow{a_3} \ldots \xrightarrow{a_n} v_n) \tag{15}$$

of arrows in G. In particular, $\mathrm{Path}_G^{(1)} = A$ and $\mathrm{Path}_G^{(0)} = V$; we refer to the path of length 0 on vertex v as the *trivial path on* v and denote it by id_v. We denote by Path_G the set of all paths on G,

$$\mathrm{Path}_G := \bigcup_{n \in \mathbb{N}} \mathrm{Path}_G^{(n)}.$$

Every path $p \in \mathrm{Path}_G$ has a source vertex and a target vertex, and we may abuse notation and write $src, tgt\colon \mathrm{Path}_G \to V$. If p is a path with $src(p) = v$ and $tgt(p) = w$, we may denote it by $p\colon v \to w$. Given two vertices $v, w \in V$, we write $\mathrm{Path}_G(v, w)$ to denote the set of all paths $p\colon v \to w$.

There is a composition operation on paths. Given a path $p\colon v \to w$ and $q\colon w \to x$, we define the composition, denoted $pq\colon v \to x$ in the obvious way. In particular, if p (resp. r) is the trivial path on vertex v (resp. vertex w) then for any path $q\colon v \to w$, we have $pq = q$ (resp. $qr = q$). Thus, for clarity, we may always denote a path as beginning with a trivial path on its source vertex; e.g. the path p from Diagram (15) may be denoted $p = \mathrm{id}_{v_0} a_1 a_2 \cdots a_n$.

Example 3.2.3. In Diagram (14), there are no paths from A to D, one path (f) from A to B, two paths (fg and fh) from A to C, and infinitely many paths $\{i^{p_1}(jk)^{q_1} \cdots i^{p_n}(jk)^{q_n} \mid n, p_1, q_1, \ldots, p_n, q_n \in \mathbb{N}\})$ from D to D.

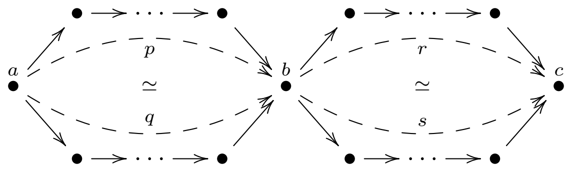
We now define the notion of categorical equivalence relation on the set of paths of a graph. Such an equivalence relation (in addition to being reflexive, symmetric, and transitive) has two sorts of additional properties: equivalent paths have the same source and target, and the composition of equivalent paths with other equivalent paths must yield equivalent paths. Formally we have Definition 3.2.4.

Definition 3.2.4. Let $G = (A, V, src, tgt)$ be a graph. A *categorical path equivalence relation* (or *CPER*) on G is an equivalence relation $\simeq$ on Path_G that has the following properties:

(1) If $p \simeq q$ then $src(p) = src(q)$.
(2) If $p \simeq q$ then $tgt(p) = tgt(q)$.
(3) Suppose $p, q \colon b \to c$ are paths, and $m \colon a \to b$ is an arrow. If $p \simeq q$ then $mp \simeq mq$.
(4) Suppose $p, q \colon a \to b$ are paths, and $n \colon b \to c$ is an arrow. If $p \simeq q$ then $pn \simeq qn$.

Lemma 3.2.5. *Suppose that G is a graph and $\simeq$ is a CPER on G. Suppose $p \simeq q \colon a \to b$ and $r \simeq s \colon b \to c$. Then $pr \simeq qs$.*

Proof. The picture to have in mind is this:

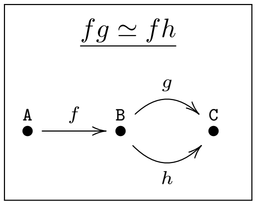


Applying condition (3) from Definition 3.2.4 to each arrow in path p, it follows by induction that $pr \simeq ps$. Applying condition (4) to each arrow in path s, it follows similarly that $ps \simeq qs$. Because $\simeq$ is an equivalence relation, it follows that $pr \simeq qs$. □

Definition 3.2.6. A *categorical schema* $\mathcal{C}$ consists of a pair $\mathcal{C} := (G, \simeq)$ where G is a graph and $\simeq$ is a categorical path equivalence relation on G. We sometimes refer to a categorical schema as simply a *schema*.

Example 3.2.7. Consider the schema, i.e. the graph together with the indicated equivalence[9], pictured in the box below:

$\mathcal{C} :=$



This schema models, for example, the phenomenon of sending an email to oneself. Suppose we populate B with emails, C with people, g and h with the sender and receiver fields, respectively. Then for fg to equal fh we must have that senders equal receivers on the image of f, and thus the subset of self-emails is a perfect fit for `A`. See example 3.2.9.

[9] More precisely, consider the graph with the categorical equivalence relation generated by the set $\{fg = fh\}$.

More on the subject of categorical schemas, including a picture of a schema and an associated set of tables, can be found in Example 2.1.3.

In the following, we will define what it means to be an instance of a categorical schema $\mathcal{C}$. We consider the case in which our instances are set-models of $\mathcal{C}$, but the same idea works in much more generality (see Definition 3.5.1).

Definition 3.2.8. Let $\mathcal{C} := (G, \simeq)$ be a categorical schema, where $G = (A, V, src, tgt)$. An *instance on* $\mathcal{C}$, denoted I, consists of the following

1. For every vertex $v \in V$, a set $I(v)$.
2. For every arrow $a \colon v \to v'$ in A, a function $I(a) \colon I(v) \to I(v')$
3. For every path equivalence $p \simeq q$ a guarantee that the equation $I(p) = I(q)$ holds.[10]

Example 3.2.9. We now return to Example 3.2.7, and write down a sample instance I for schema $\mathcal{C} = (G, \simeq)$.

A	
ID	f
SEm1207	Em1207
SEm1210	Em1210
SEm1211	Em1211

B		
ID	g	h
Em1206	Bob	Sue
Em1207	Carl	Carl
Em1208	Sue	Martha
Em1209	Chris	Bob
Em1210	Chris	Chris
Em1211	Julia	Julia
Em1212	Martha	Chris

C
ID
Bob
Carl
Chris
Julia
Martha
Sue

For each vertex v in G, the set $I(v)$ is given by the set of rows in the corresponding table (e.g. $I(\mathtt{A}) = \{$SEm1207, SEm1210, SEm1211$\}$). For each arrow $a \colon v \to w$ in G the function $I(a) \colon I(v) \to I(w)$ is also evident as a column in the table. For example, $I(g) \colon I(\mathtt{B}) \to I(\mathtt{C})$ sends Em1206 to Bob, etc. Finally, the path equivalence $fg = fh$ is borne out in the fact that for every row-id in table A, following f then g returns the same result as following f then h.

3.3. **Translations.** A translation is a mapping from one categorical schema to another. Vertices are sent to vertices, arrows are sent to paths, and all path equivalences are preserved. More precisely, we have the following definition.

Definition 3.3.1. Let $G = (A_G, V_G, src_G, tgt_G)$ and $H = (A_H, V_H, src_H, tgt_H)$ be graphs (see Definition 3.2.1), and let $\mathcal{C} = (G, \simeq_{\mathcal{C}})$ and $\mathcal{D} = (H, \simeq_{\mathcal{D}})$ be categorical schemas. A *translation* F *from* $\mathcal{C}$ *to* $\mathcal{D}$, denoted $F \colon \mathcal{C} \to \mathcal{D}$ consists of the following constituents:

1. a function $V_F \colon V_G \to V_H$, and
2. a function $A_F \colon A_G \to \text{Path}_H$

subject to the following conditions:

[10] Once I is defined on arrows, as it is in item (2), we can extend it to paths in the obvious way: if $p = a_1 a_2 \cdots a_n$, then the function $I(p)$ is the composition $I(p) = I(a_1)I(a_2) \cdots I(a_n)$.

(a) the function A_F preserves sources and targets; in other words, the following diagrams of sets commute:

$$\begin{tikzcd} A_G \arrow[r, "A_F"] \arrow[d, "src_G"'] & \mathrm{Path}_H \arrow[d, "src_H"] \\ V_G \arrow[r, "V_F"'] & V_H \end{tikzcd} \qquad \begin{tikzcd} A_G \arrow[r, "A_F"] \arrow[d, "tgt_G"'] & \mathrm{Path}_H \arrow[d, "tgt_H"] \\ V_G \arrow[r, "V_F"'] & V_H \end{tikzcd}$$

(b) the function A_F preserves path equivalences.[11] Precisely, suppose we are given lengths $m, n \in \mathbb{N}$ and paths $p = id_{v_0} f_1 f_2 \cdots f_m$ and $q = id_{v_0} g_1 g_2 \cdots g_n$ in G. Let $v_0' = V_F(v_0)$ and for each $i \leq m$ (resp. $j \leq n$), let $f_i' = A_F(f_i)$ (resp. $g_j' = A_F(g_j)$), and let $p' = \mathrm{id}_{v_0'} f_1' f_2' \cdots f_m'$ (resp. $q' = \mathrm{id}_{v_0'} g_1' g_2' \cdots g_n'$). If $p \simeq_{\mathcal{C}} q$ then $p' \simeq_{\mathcal{D}} q'$.

Two translations $F, F' \colon \mathcal{C} \to \mathcal{D}$ are considered identical if they agree on vertices (i.e. $V_F = V_{F'}$) and if, for every arrow f in $\mathcal{C}$, there is a path equivalence

$$A_F(f) \simeq_{\mathcal{D}} A_{F'}(f).$$

In the following two examples we will reconsider translations discussed in Section 2.

Example 3.3.2. Recall the mapping F given in Diagram (7). The schemas $\mathcal{C}$ and $\mathcal{D}$ are just graphs in the sense that there are no declared path equivalences in either of them. The mapping F sends vertices in $\mathcal{C}$ to vertices in $\mathcal{D}$ and arrows in $\mathcal{C}$ to arrows in $\mathcal{D}$. Since an arrow is a particular sort of path, and since there are no path equivalences to be preserved, $F\colon \mathcal{C} \to \mathcal{D}$ is indeed a translation.

Example 3.3.3. Recall the mapping F given in Example 2.3.1, Diagram (10). In this setup, $\mathcal{C}$ has declared path equivalences and $\mathcal{D}$ does not; however $\mathcal{D}$ still has a categorical path equivalence relation $\simeq_{\mathcal{D}}$ on it, the minimal reflexive relation. The mapping F on vertices (V_F) is self-explanatory; the only arrows on which A_F is not self-explanatory are i_{12} and i_{21}, both of which are sent to the trivial path $\mathrm{id}_{\mathtt{T}}$ on vertex $\mathtt{T}$.

Because $V_F(\mathtt{T1}) = V_F(\mathtt{T2}) = \mathtt{T}$, it is clear that A_F preserves sources and targets. The path equivalence $i_{12}i_{21} = \mathrm{id}_{\mathtt{T1}}$ and $i_{21}i_{12} = \mathrm{id}_{\mathtt{T2}}$ are preserved because $A_F(i_{12}) = A_F(i_{21}) = \mathrm{id}_{\mathtt{T}}$, and the concatenation of a trivial path with any path p yields p.

3.4. **The equivalence Sch $\simeq$ Cat.** We assume familiarity with categories and functors, and in particular the category **Cat** of small categories and functors (a list of references is given in Section 3.1). In this section we will define the category **Sch** and show it is equivalent to **Cat**. It is this result that justifies our advertisement in the introduction that "every theorem about small categories becomes a theorem about databases".

Definition 3.4.1. Recall the notions of categorical schemas and translations from Definitions 3.2.6 and 3.3.1. The *category of categorical schemas*, denoted **Sch**, is the category whose objects are categorical schemas and whose morphisms are translations.

[11] This is easier to understand conceptually than to write down.

Construction 3.4.2 (From schema to category)**.** We will define a functor $L\colon \mathbf{Sch} \to \mathbf{Cat}$. Let $\mathcal{C} = (G, \simeq_{\mathcal{C}})$ be a categorical schema, where $G = (A, V, src, tgt)$. Define $\mathcal{C}'$ to be the free category with objects V generated by arrows A. Define $L(\mathcal{C}) \in \mathbf{Cat}$ to be the category defined as the quotient of $\mathcal{C}'$ by the equivalence relation $\simeq_{\mathcal{C}}$ (see [Mac, Section 2.8]). This defines L on objects of $\mathbf{Sch}$.

Given a translation $F\colon \mathcal{C} \to \mathcal{D}$, there is an induced functor on free categories $F'\colon \mathcal{C}' \to \mathcal{D}'$, sending each generator $f \in A$ to the morphism in $\mathcal{D}'$ defined as the composite of the path $A_F(f)$. The preservation of path equivalence ensures that F' descends to a functor $L(F)\colon L(\mathcal{C}) \to L(\mathcal{D})$ on quotient categories. This defines L on morphisms in $\mathbf{Sch}$. It is clear that L preserves composition, so it is a functor.

Construction 3.4.3 (From category to schema)**.** We will define a functor $R\colon \mathbf{Cat} \to \mathbf{Sch}$. Let $\mathcal{C}$ be a small category with object set $\mathrm{Ob}(\mathcal{C})$, morphism set $\mathrm{Mor}(\mathcal{C})$, source and target functions $s, t\colon \mathrm{Mor}(\mathcal{C}) \to \mathrm{Ob}(\mathcal{C})$, and composition law $\circ\colon \mathrm{Mor}(\mathcal{C}) \times_{\mathrm{Ob}(\mathcal{C})} \mathrm{Mor}(\mathcal{C}) \to \mathrm{Mor}(\mathcal{C})$. Let $R(\mathcal{C}) = (G, \simeq)$ where G is the graph

$$G = (\mathrm{Mor}(\mathcal{C}), \mathrm{Ob}(\mathcal{C}), s, t)$$

and with $\simeq$ defined as follows: for all $f, g \in \mathrm{Mor}(\mathcal{C})$ with $t(f) = s(g)$ we put

$$fg \simeq (g \circ f). \tag{16}$$

This defines R on objects of $\mathbf{Cat}$.

A functor $F\colon \mathcal{C} \to \mathcal{D}$ induces a translation $R(F)\colon R(\mathcal{C}) \to R(\mathcal{D})$, because vertices are sent to vertices, arrows are sent to arrows, and path equivalence is preserved by (16) and the fact that F preserves the composition law. This defines R on morphisms in $\mathbf{Cat}$. It is clear that R preserves compositions, so it is a functor.

Theorem 3.4.4. *The functors*

$$L\colon \mathbf{Sch} \rightleftarrows \mathbf{Cat}\colon R$$

are mutually inverse equivalences of categories.

Sketch of proof. It is clear that there is a natural isomorphism $\epsilon\colon \mathrm{id}_{\mathbf{Cat}} \to L \circ R$; i.e. for any category $\mathcal{C}$, there is an isomorphism $\mathcal{C} \cong L(R(\mathcal{C}))$. Thus the functor L is essentially surjective. We first show that L is fully faithful.

Choose schemas X and Y, and suppose $X = (A_X, V_X, src_X, tgt_X)$; we must show that the function $L_1\colon \mathrm{Hom}_{\mathbf{Sch}}(X, Y) \to \mathrm{Hom}_{\mathbf{Cat}}(LX, LY)$ is a bijection. It is clearly injective. To show that it is surjective, choose a functor $G\colon LX \to LY$; we will define a translation $F\colon X \to Y$ with $L_1(F) = G$. Define F on vertices of X as G is defined on objects of LX. Define F on arrows of X via the function $A_X \to \mathrm{Path}_X \to \mathrm{Mor}(LX) \xrightarrow{G} \mathrm{Mor}(LY)$, and choose a representative for its equivalence class from Path_{LY} (note that any two choices result in the same translation: see Definition 3.3.1). Two equivalent paths in X compose to the same element of $\mathrm{Mor}(LX)$, so F preserves path equivalence. This defines F, completing the proof that L an equivalence of categories.

By similar reasoning one proves that R is fully faithful, and concludes that it is inverse to L.

□

3.5. **The category of instances on a schema.** Given Theorem 3.4.4, the compound notion of categorical schemas and translations is equivalent to that of categories and functors. In the remainder of the paper, we elide the difference between **Sch** and **Cat**, using nomenclature from each interchangeably.

One sees easily (c.f. Definition 3.2.8) that an instance I on a schema $\mathcal{C}$ is the same thing as a functor $\mathcal{C} \to \mathbf{Set}$, where **Set** is the category of sets. Thus we have a ready-made concept of morphisms between instances: natural transformations of functors. This is an established notion in database literature, often called a *homomorphism* of instances (see e.g. [DNR]).

Definition 3.5.1. Let $\mathcal{C}$ be a schema and let $I, J\colon \mathcal{C} \to \mathbf{Set}$ be instances on $\mathcal{C}$. A *morphism m from I to J*, denoted $m\colon I \to J$, is simply a natural transformation between these functors. We define *the category of instances on* $\mathcal{C}$, denoted $\mathcal{C}\text{–}\mathbf{Inst}$, to be the category of instances and morphisms, as above.

More generally, let $\mathbb{S}$ denote any category; we define *the category of* $\mathbb{S}$*-valued instances on* $\mathcal{C}$, denoted $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}} := \mathbb{S}^{\mathcal{C}}$, to be the category whose objects are functors $\mathcal{C} \to \mathbb{S}$ and whose morphisms are natural transformations. We refer to $\mathbb{S}$ as the *value category* in this setup.

Remark 3.5.2. It appears that programming language theorists do not include homomorphisms between instances in their conception of database instances as elements of a type, preferring instead to work with just the *set* $\mathrm{Ob}(\mathcal{C}\text{–}\mathbf{Inst})$ of instances on a schema $\mathcal{C}$. Doing so makes it easier to define aggregate functions, such as sums and counts; see e.g. [LT].

In the rest of the paper, we will generally work with $\mathcal{C}\text{–}\mathbf{Inst}$, the category of **Set**-valued instances. However, most of the results go through more generally for $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$, provided that $\mathbb{S}$ is complete and cocomplete (i.e. has all small limits and all small colimits). Obviously, given a functor $\mathbb{S} \to \mathbb{S}'$ there is an induced functor $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}'}$, so the choice of value-category can be changed without much cost.

Example 3.5.3. Given a schema $\mathcal{C}$, there are many categories $\mathbb{S}$, other than $\mathbb{S} = \mathbf{Set}$, for which one might be interested in $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$. For example, given a lambda calculus, the associated category $\mathbb{S} = \mathbf{Type}$ of types and terms is a good choice ([Awo, Section 6.5]). One can also use the category **Fin** of finite sets, **cpo** of complete partial orders, **Cat** of small categories, or **Top** of topological spaces.

The choice of value-category is based on how one chooses to view the collection of rows in each table. We usually consider this collection to be a set, but for example one can imagine instead a *topological space* of rows, and in this case each column would consist of a continuous map from one space to another.

Toposes, invented by Grothendieck and Verdier [GV] and extended by Lawvere [**?**], are categories that mimic the category of sets in several important ways (see [MM]). In Proposition 3.5.4, we show that the instance categories are often toposes.

Proposition 3.5.4. *If* $\mathbb{S} = \mathbf{Set}$ *then for any schema* $\mathcal{C} \in \mathbf{Sch}$*, the category* $\mathcal{C}\text{–}\mathbf{Inst} = \mathcal{C}\text{–}\mathbf{Inst}_{\mathbf{Set}}$ *is a topos. If* $\mathcal{C}$ *is a finite category then for any topos* $\mathbb{S}$ *(e.g.* $\mathbb{S} = \mathbf{Fin}$*, the category of finite sets), the category* $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$ *is a topos. If* $\mathbb{S}$ *is complete (resp. cocomplete) then* $\mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$ *is also complete (resp. cocomplete).*

Proof. The first pair of claims are [JoP, A.2.1.3]. The second pair of claims are found in [Bor1, Theorem 2.15.2] and [Awo, Proposition 8.8], respectively.

□

Given a database instance I, updates on I include deletion of rows, insertion of rows, splitting (one row becoming two), and merging (two rows becoming one). In fact, we classify insertions and merges together as *progressive updates* and we classify deletions and splits together as *regressive updates*. Then every update can be considered as a regressive update followed by a progressive update.

Definition 3.5.5. Let $\mathcal{C}$ be a schema and $I \in \mathcal{C}$–**Inst** an instance. A *progressive update on I* consists of an instance J and a natural transformation $p\colon I \to J$. A *regressive update on I* consists of an instance J and a natural transformation $r\colon J \to I$. That is, a regressive update is just a progressive update in reverse. An *update* is a finite sequence of progressive and regressive updates.

Proposition 3.5.6. *Let $\mathcal{C}$ be a schema and I an instance on $\mathcal{C}$. Any update on I can be obtained as a single regressive update followed by a single progressive update.*

Proof. The composition of two progressive (resp. regressive) updates is a progressive (resp. regressive) update. Hence any update on $I_0 := I$ can be written as a diagram D in $\mathcal{C}$–**Inst**:

$$\begin{tikzcd}
& I_{01} \arrow[dl, "r_1"'] \arrow[dr, "p_1"] & & I_{12} \arrow[dl, "r_2"'] \arrow[dr, "p_2"] & & I_{23} \arrow[dl, "r_2"'] & \cdots & I_{n_1,n} \arrow[dr, "p_n"] & \\
I_0 & & I_1 & & I_2 & & \cdots & & I_n
\end{tikzcd}$$

But the limit of this diagram (which can be taken, if one wishes, by taking fiber products such as $I_{j-1,j} \to I_j \leftarrow I_{j,j+1}$, and repeating $\frac{n^2-n}{2}$ times), is what we need:

$$I_0 \leftarrow \lim D \to I_n.$$

□

Remark 3.5.7. Another way to understand deletes is via filtering—one filters out all rows of a certain form. Filtering will be discussed in Section 5.3.

3.6. **Grothendieck construction.** In Section 2.4 we showed how to convert an instance $I\colon \mathcal{C} \to$ **Set** to a new category $Gr(I)$, called the *Grothendieck construction* or *category of elements* of I. This construction models the conversion from relational to RDF forms of data. There is a reverse construction that is described in [Sp4, Proposition 2.3.9].

We note here that there is a more general Grothendieck construction that may be useful in the context of federated databases. In programming languages theory, one may hear of a category of *kinds*, each object of which is itself a category of types. Here, each kind is analogous to a schema, and each type in that kind is analogous to a table in that schema. Given a category of kinds, we can "throw all their types together" by applying the generalized Grothendieck construction. This is akin to taking a federated database (i.e. a schema of related schemas) and merging them all into a single grand schema. One can apply this construction at the data level as well, merging all the instances into an instance over the single grand schema.

The version of the Grothendieck construction given in Section 2.4 is for functors $I\colon \mathcal{C} \to$ **Set**. Each object $c \in \mathrm{Ob}(\mathcal{C})$ corresponds to a table whose set of rows is $I(c)$. One can find a description of this construction in [MM, Section 1.5]. The version of

Table 1. Dictionary between database terminology and category theory terminology.

Dictionary between DB and CT terminology	
Database concept	**Category-theory concept**
Database schema, $\mathcal{C}$	Category, $\mathcal{C}$
Table $T \in \mathcal{C}$	Object $T \in \mathrm{Ob}(\mathcal{C})$
Column f of T	Outgoing morphism, $f\colon T \to ?$
Foreign key column f of T pointing to U	Morphism $f\colon T \to U$
Sequence of foreign keys	Composition of morphisms
Primary key column ID of T	Identity arrow, $\mathrm{id}_T\colon T \to T$
Controlled vocabulary (i.e. one-column table D)	Object D without outgoing morphisms (except $\mathrm{id}_D\colon D \to D$)
Foreign key path equivalence in $\mathcal{C}$	Commutative diagram in $\mathcal{C}$
Instance I on schema $\mathcal{C}$	Functor $I\colon \mathcal{C} \to \mathbf{Set}$ (or $I\colon \mathcal{C} \to \mathbb{S}$ for some other "nice" category $\mathbb{S}$)
Conversion of Relational to RDF	Grothendieck construction
Insertion update $u\colon I_0 \to I_1$	Natural transformation $u\colon I_0 \to I_1$
Deletion update $u\colon I_0 \to I_1$	Natural transformation $u\colon I_1 \to I_0$
Schema mapping $\mathcal{C} \to \mathcal{D}$	Functor $F\colon \mathcal{D} \to \mathcal{C}$
Basic ETL process $\mathcal{C}$–$\mathbf{Inst} \to \mathcal{D}$–$\mathbf{Inst}$	Pullback functor, often denoted Δ_F or $F^*\colon \mathcal{C}$–$\mathbf{Set} \to \mathcal{D}$–$\mathbf{Set}$

the Grothendieck construction in which federated schemas are combined into one big schema is for functors $D\colon \mathcal{C} \to \mathbf{Cat}$. Each object $c \in \mathrm{Ob}(\mathcal{C})$ corresponds to a database whose category of tables is $D(c)$. One can find a description of this construction in [JoP, B.1.3.1].

3.7. **Dictionary.** Our hope is that this paper will serve as a dictionary, whereby results from category theory literature can be imported directly into database theory. In Section 4 we will see such a result: translations between schemas provide data migration functors that have useful and provable properties. In Table 1 we gather some of the foundational links between databases and categories, as presented throughout the paper.

4. Data migration functors

Given schemas $\mathcal{C}$ and $\mathcal{D}$, a data migration functor is tasked with transforming any $\mathcal{C}$-instance I into some $\mathcal{D}$-instance J (or vice versa). Moreover, it must do so *in a natural way*, meaning that progressive (resp. regressive) updates on I must result in progressive (resp. regressive) updates on J. Data migration functors were concretely exemplified in Section 2.2.

In this section we will start with a translation between schemas $F\colon \mathcal{C} \to \mathcal{D}$. Recall from Definition 3.3.1 that this is simply a mapping from vertices in $\mathcal{C}$ to vertices in $\mathcal{D}$ and arrows in $\mathcal{C}$ to paths in $\mathcal{D}$, respecting path equivalence. Any translation F generates three data migration functors. These will be denoted

$$\Sigma_F\colon \mathcal{C}\text{–}\mathbf{Inst} \to \mathcal{D}\text{–}\mathbf{Inst} \qquad \Delta_F\colon \mathcal{D}\text{–}\mathbf{Inst} \to \mathcal{C}\text{–}\mathbf{Inst} \qquad \Pi_F\colon \mathcal{C}\text{–}\mathbf{Inst} \to \mathcal{D}\text{–}\mathbf{Inst}.$$

One may notice that Δ_F seems to go backwards—the direction opposite to that of F. Although this may seem counter-intuitive, in fact Δ_F is the simplest of the three data migration functors and the most straightforward to describe.

Before we do so, let us quickly discuss value categories. Recall from Definition 3.5.1 that for any category $\mathbb{S}$, we have a category $\mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$ of $\mathcal{C}$-instances valued in $\mathbb{S}$, i.e. the category of functors $\mathcal{C} \to \mathbb{S}$. The migration functor $\Delta_F\colon \mathcal{D}$–$\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$ exists regardless of ones choice of $\mathbb{S}$. For Σ_F to exist, $\mathbb{S}$ must be cocomplete, and for $\Pi_{\mathbb{S}}$ to exist, $\mathbb{S}$ must be complete. To fix ideas, most readers should simply take $\mathbb{S} = \mathbf{Set}$, unless they are compelled to do otherwise. This is the case where the rows of each table form a set.

4.1. **The pullback data migration functor Δ.** Suppose we have a translation $F\colon \mathcal{C} \to \mathcal{D}$. Given a $\mathcal{D}$-instance, $I \in \mathcal{D}$–$\mathbf{Inst}_{\mathbb{S}}$, we need to transform it to a $\mathcal{C}$-instance in a natural way. But this is simple, because $I\colon \mathcal{D} \to \mathbb{S}$ is a functor and the composition of functors is a functor, so the composite

$$\mathcal{C} \xrightarrow{F} \mathcal{D} \xrightarrow{I} \mathbb{S},$$

is an object of $\mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$, as desired. Similarly, a natural transformation $m\colon I \to J$ is whiskered with F to yield a natural transformation $(m \circ F)\colon (I \circ F) \longrightarrow (J \circ F)$. Thus we have defined a functor

$$\Delta_F\colon \mathcal{D}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}, \qquad \Delta_F(-) := (- \circ F)$$

The slogan is “Δ_F is given by composition with F.”

We have now defined the pullback functor $\Delta_F\colon \mathcal{D}$–$\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$. It was explicitly discussed in Example 2.2.1. Roughly, it can accommodate: renaming tables, renaming columns, deleting tables, projecting out columns, duplicating tables, and duplicating columns.

4.2. **The right pushforward data migration functor Π.** Suppose we have a translation $F\colon \mathcal{C} \to \mathcal{D}$. Given a $\mathcal{C}$-instance, $I \in \mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$, we need to transform it to a $\mathcal{D}$-instance in a natural way. We will do so by using the right adjoint of Δ_F; however, to do this we will need to assume that $\mathbb{S}$ is complete (i.e. that $\mathbb{S}$ has small limits). Note that $\mathbb{S} = \mathbf{Set}$ is complete.

Proposition 4.2.1. *Let $F\colon \mathcal{C} \to \mathcal{D}$ be a functor, and let $\mathbb{S}$ be a complete category. Then the functor $\Delta_F\colon \mathcal{D}$–$\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$ has a right adjoint, which we denote by*

$$\Pi_F\colon \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{D}\text{–}\mathbf{Inst}_{\mathbb{S}}.$$

Proof. This is [Mac, Corollary X.3.2].

□

We have now defined the right pushforward functor $\Pi_F\colon \mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}} \to \mathcal{D}$–$\mathbf{Inst}_{\mathbb{S}}$. It was explicitly discussed in Example 2.2.3. Roughly, it can accommodate: renaming tables, renaming columns, and joining tables. To see this, one applies the “pointwise formula” for right Kan extensions, e.g. as given in [Mac, Theorem X.3.1]; a more explicit formulation is given in [Sp3].

4.3. **The left-pushforward data migration functor Σ.** Suppose we have a translation $F\colon \mathcal{C} \to \mathcal{D}$. Given a $\mathcal{C}$-instance, $I \in \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$, we need to transform it to a $\mathcal{D}$-instance in a natural way. We will do so by using the left adjoint of Δ_F; however, to do this we will need to assume that $\mathbb{S}$ is cocomplete (i.e.that $\mathbb{S}$ has small colimits). Note that $\mathbb{S} = \mathbf{Set}$ is cocomplete.

Proposition 4.3.1. *Let $F\colon \mathcal{C} \to \mathcal{D}$ be a functor, and let $\mathbb{S}$ be a cocomplete category. Then the functor $\Delta_F\colon \mathcal{D}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$ has a left adjoint, which we denote by*

$$\Sigma_F\colon \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{D}\text{–}\mathbf{Inst}_{\mathbb{S}}.$$

Proof. This follows from [Bor1, Theorem 3.7.2] and [BW2, Proposition 9.11]. □

We have now defined the left pushforward functor $\Sigma_F\colon \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}} \to \mathcal{D}\text{–}\mathbf{Inst}_{\mathbb{S}}$. It was explicitly discussed in Example 2.2.5. Roughly, it can accommodate: renaming tables, renaming columns, taking the union of tables, and creating Skolem variables. To see this, one applies the "pointwise formula" for left Kan extensions, e.g. as given in [Bor1, Theorem 3.7.2]; a more explicit formulation is given in [Sp3].

5. Data types and filtering

In this section we will formulate the typing relationship that holds between abstract data and its representation. Until now, we have been considering data as simply a collection of interconnected elements—an instance in the sense of Definition 3.2.8 keeps track of various sets of abstract elements, segregated into tables and connected together in precise ways. However, in reality, each such element is represented in its table by way of a datatype, such as strings or integers. Semantically, each cell in a given column $c\colon t \to t'$ of a table t should have the same datatype, namely they should all have the datatype of the target table, t'.

However, datatypes not only give a uniform method for displaying each data element, but they can also carry a notion of value. For example, salaries are numbers that can be added together to give meaningful invariants. It is for this reason that we must find a connection between database formalism and programming language formalism, as was discussed in Section 2.5. Category theory provides such a connection: both database schemas and programming languages form categories, and these categories can be related by functors.

Below we will explain these concepts, in particular how to attach datatypes from a programming language to tables in a categorical schema. To do so, we will make use of the concepts in Section 4. After defining type signatures on schemas we will proceed to define morphisms of type signatures, which will enable us to filter data. For example, to filter all names that start with the letter R, we might pull back along an inclusion $\{\mathrm{R}\} \to \mathrm{Str}$, where Str is the set of strings.

5.1. **Assigning data types via natural transformation.** We begin with two examples to motivate the definition.

Example 5.1.1. What is meant mathematically by the phrase "a set of integers"? Consider that it is a set labeled X, together with a function $f\colon X \to \mathbb{Z}$. Allowing our set of integers to change, we get different sets labeled X and different functions labeled f, but the set $\mathbb{Z}$ of integers is unchanged. From the categorical perspective we can understand "a set X of integers" in a couple different ways:

(1) as a database instance $I\colon \mathcal{C} \to \mathbf{Set}$ on the schema

$$\mathcal{C} := \boxed{\begin{tikzcd} \overset{\texttt{X}}{\bullet} \arrow[r, "f"] & \overset{\mathbb{Z}}{\bullet} \end{tikzcd}},$$

such that the image on $\overset{\mathbb{Z}}{\bullet}$ is fixed as $I(\overset{\mathbb{Z}}{\bullet}) = \mathbb{Z}$; or

(2) as a database instance $J\colon \mathcal{D} \to \mathbf{Set}$ on the schema

$$\mathcal{D} := \boxed{\overset{\texttt{X}}{\bullet}}$$

equipped with a natural transformation $f\colon J \to \{\mathbb{Z}\}$ (where $\{\mathbb{Z}\}$ is shorthand for the functor $\mathcal{D} \to \mathbf{Set}$ given by $D(\overset{\texttt{X}}{\bullet}) = \mathbb{Z}$).

Example 5.1.2. Suppose we have a database and that we would like to enforce a mathematical relationship between two columns, say t and d, in a certain table $\texttt{X}$. For example, it might be that column t, say "time spent" (in an integer number of hours) is related to column d, say "debt owed" (as a dollar-figure) by a mathematical function $d = r(t)$, say

$$d(x) = r(t(x)) := \$50 * t(x)$$

for each $x \in X$. Just as in Example 5.1.1, this situation could be categorically represented in a couple ways, but we will focus on only one. Namely, we understand it as the collection of database instances $J\colon \mathcal{D} \to \mathbf{Set}$ on schema $\mathcal{D}$ as exemplified below

$$(17) \qquad \mathcal{D} := \boxed{\begin{array}{c} \underline{d = tr} \\ \begin{tikzcd} \overset{\texttt{X}}{\bullet} \arrow[r, "t"] \arrow[dr, "d"'] & \overset{\texttt{Y}}{\bullet} \arrow[d, "r"] \\ & \overset{\texttt{Z}}{\bullet} \end{tikzcd} \end{array}} \qquad J(\texttt{X}) :=$$

X		
ID	**t**	**d**
CtrX13	4	\$200
CtrX14	7	\$350
CtrX15	2	\$100

The fact that the d has dollar-figure datatype and that $d = \$50 * t$ are enforced by a certain natural transformation, $f\colon J \to P$, where P is a *typing instance*. We will give more details in Example 5.1.7.

Definition 5.1.3. Let $\mathcal{C}$ be a schema and let $P \in \mathrm{Ob}(\mathcal{C}\text{–}\mathbf{Inst})$ be an instance. The category of *P-typed instances on $\mathcal{C}$*, denoted $\mathcal{C}\text{–}\mathbf{Inst}_{/P}$, is defined to be the "slice" category of instances over P (see, e.g. [MM, Categorical Preliminaries]). In other words, a P-typed instance on $\mathcal{C}$ is a pair (I, τ) where I is an instance and $\tau\colon I \to P$ is a natural transformation; and a morphism of P-typed instances is a commutative triangle.

Remark 5.1.4. Given a schema $\mathcal{C}$ we may refer to any instance $P \in \mathrm{Ob}(\mathcal{C}\text{–}\mathbf{Inst})$ as a *typing instance* if our plan is to consider P-typed instances, i.e. the category $\mathcal{C}\text{–}\mathbf{Inst}_{/P}$.

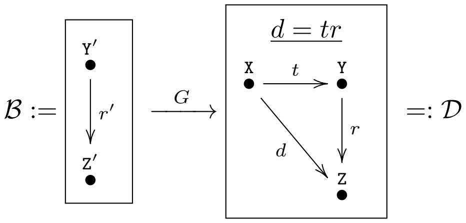
Remark 5.1.5. Fix a schema $\mathcal{C}$ and a category $\mathbb{S}$ and let $\mathcal{E} := \mathcal{C}\text{–}\mathbf{Inst}_{\mathbb{S}}$ denote the category of $\mathbb{S}$-valued instances on $\mathcal{C}$. In practice $\mathcal{E}$ is often a topos (see Proposition 3.5.4). In case it is, then for any instance $P \in \mathrm{Ob}(\mathcal{E})$, the category $\mathcal{E}_{/P}$ of P-typed $\mathbb{S}$-valued instances on $\mathcal{C}$ is again a topos (see [MM, Theorem IV.7.1]).

Construction 5.1.6. Suppose given a category **Type** of types for some programming language and an $\mathbb{S}$-valued functor $V\colon \mathbf{Type} \to \mathbb{S}$ which sends each type to its set (or $\mathbb{S}$-object) of values. We often wish to use a fragment of **Type** to add typing information to our database schema $\mathcal{C}$. If the fragment is given by the functor $\mathcal{B} \xrightarrow{F} \mathbf{Type}$ and $\mathcal{B}$ is associated to the schema via a functor $\mathcal{B} \xrightarrow{G} \mathcal{C}$,

$$\mathbb{S} \xleftarrow{V} \mathbf{Type} \xleftarrow{F} \mathcal{B} \xrightarrow{G} \mathcal{C},$$

then $P := \Pi_G \circ \Delta_{\mathcal{F}}(V)$ is the implied typing instance. We call the sequence $(\mathcal{B}, F, G)$ *the typing auxiliary* in this setup.

Example 5.1.7. We return to Example 5.1.2 with the language from Construction 5.1.6. We will now describe a typing auxiliary. Let $\mathcal{B}$ be one-arrow category drawn below, and let $G\colon \mathcal{B} \to \mathcal{D}$ be the suggested functor



Consider also the functor $F\colon \mathcal{B} \to \mathbf{Type}$ sending Y′ to `Int`, the type of integers, Z′ to `Dollar` the type of dollar figures, and r' to the function that multiplies an integer by 50.

With $V\colon \mathbf{Type} \to \mathbb{S}$ as in Construction 5.1.6, the implied typing instance $P := \Pi_G \circ \Delta_F(V)\colon \mathcal{D} \to \mathbb{S}$ has

$$P(\mathtt{X}) = \mathtt{Int} \times \mathtt{Dol}; \qquad P(\mathtt{Y}) = \mathtt{Int}; \qquad P(\mathtt{Z}) = \mathtt{Dollar},$$

and $P(r)\colon P(\mathtt{Y}) \to P(\mathtt{Z})$ is indeed the multiplication by 50 map.

Now a P-typed instance $\tau\colon I \to P$ is exactly what we want. For each of X, Y, Z it is a set with a map to the given data type, and the naturality of τ ensures the properties described in Example 5.1.2 (i.e. that $d = \$50 * t$ in the table $J(\mathtt{X})$ from Display (17)). In other words, it ensures that for any row in $J(\mathtt{X})$, the value of the cell in column d will be 50 times the value of the cell in column t.

5.2. **Morphisms of type signatures.** Each data migration functor discussed in Section 4 is a kind of tool for schema evolution. As new tables and columns are created and others are discarded, the translation between old schema and new will induce data migration functors that convert seamlessly from old data to new (and vice versa) and from queries against the old schema to queries against the new one (see also [Sp4]).

There is a slightly different kind of schema evolution that comes up often, namely changing data types. For example, if a company surpasses around 32,000 employees, they may need to change the datatype on their `Employee` table from a smallint to a bigint. More complex changes include cutting the price for every share of stock by half, or concatenating a first and a last name pair to form a new field. Importantly, one needs to be able to reason about how queries against today's schema will differ from those against yesterday's. Change-of-types functors are just like data migration functors, and the formal nature of their description allows one to reason about their behavior.

Let $\mathcal{C}$ be a schema and let $\mathcal{E}$ be the topos $\mathcal{C}$–**Inst**. Given a morphism of typing instances $k\colon P \to Q$, there are induced adjunctions

$$\begin{tikzcd}\mathcal{E}_{/P} \arrow[r, shift left, "\widehat{\Sigma}_k"] & \mathcal{E}_{/Q} \arrow[l, shift left, "\widehat{\Delta}_k"]\end{tikzcd} \qquad \begin{tikzcd}\mathcal{E}_{/Q} \arrow[r, shift left, "\widehat{\Delta}_k"] & \mathcal{E}_{/P} \arrow[l, shift left, "\widehat{\Pi}_k"]\end{tikzcd}$$

In other words, k induces an essential geometric morphism of toposes ([MM, Theorem IV.7.2]).

Definition 5.2.1. Let $\mathcal{C}$ be a schema and $k\colon P \to Q$ a morphism of typing instances. We refer to the induced functors

$$\widehat{\Sigma}_k, \widehat{\Pi}_k\colon \mathcal{C}\text{–}\mathbf{Inst}_{/P} \to \mathcal{C}\text{–}\mathbf{Inst}_{/Q} \qquad \widehat{\Delta}_k\colon \mathcal{C}\text{–}\mathbf{Inst}_{/Q} \to \mathcal{C}\text{–}\mathbf{Inst}_{/P}$$

as *type-change functors*. To be more specific, $\widehat{\Sigma}_k$ will be called the *left pushforward type-change functor*, $\widehat{\Pi}_k$ will be called the *right pushforward type-change functor*, and $\widehat{\Delta}_k$ will be called the *pullback type-change functor*.

Example 5.2.2. We return to Example 5.1.2 with $\mathcal{D}$ and P as defined there. Consider the left pushforward type-change functor $\widehat{\Sigma}_k$ in the case that $k\colon P \to Q$ sends a dollar figure x to True if $x \geq \$200$ and False if $x < \$200$. The functor $\widehat{\Sigma}_k$ converts the table on the left to the table on the right below:

$\tau :=$

X		
ID	**t**	**d**
CtrX13	4	\$200
CtrX14	7	\$350
CtrX15	2	\$100

$\widehat{\Sigma}_k(\tau) =$

X		
ID	**t**	**d**
CtrX13	4	True
CtrX14	7	True
CtrX15	2	False

The other type-change functors, $\widehat{\Pi}_k$ and $\widehat{\Delta}_k$ not have useful results in the context of this particular example, but see Examples 5.2.3 and 5.3.1.

The right type-change functor $\widehat{\Pi}$ handles what might be called "group satisfaction." Suppose we have a bunch (P) of people and a set (I) of items are distributed among them $(\tau\colon I \to P)$. If the people are then subdivided into groups $(k\colon P \to Q)$, then we can ask each group $q \in Q$, "how many ways are there for each of your members to offer up one of their items?" (cardinality of $\widehat{\Pi}_k(\tau)^{-1}(q) \subseteq \widehat{\Pi}_k(I)$). For example, if one of the people was handed an empty set of items then his or her group will have no such joint offering $(\widehat{\Pi}_k(\tau)^{-1}(q) = \emptyset)$. We now explain this by example.

Example 5.2.3. Let $\mathbb{S}$ be a complete category and write $\mathcal{C}$–**Inst** instead of $\mathcal{C}$–$\mathbf{Inst}_{\mathbb{S}}$. In this example we explain how, given a morphism of typing instances $k\colon P \to Q$ on a schema $\mathcal{C}$, the type-change functor $\widehat{\Pi}_k\colon \mathcal{C}\text{–}\mathbf{Inst}_{/P} \to \mathcal{C}\text{–}\mathbf{Inst}_{/Q}$ operates on a P-typed instance to return a Q-typed instance by what we called *group satisfaction* above. Suppose we have $\mathcal{C}$ and $\mathcal{B}$ as drawn,

$$\mathcal{B} = \boxed{\overset{\texttt{L}'}{\bullet}}, \qquad \mathcal{C} = \boxed{\overset{\texttt{L}}{\bullet} \xrightarrow{f} \overset{\texttt{M}}{\bullet}}$$

with the functor $G\colon \mathcal{B} \to \mathcal{C}$ given by $\texttt{L}' \mapsto \texttt{L}$ and the functor $P'\colon \mathcal{B} \to \mathbb{S}$ given by $P'(\texttt{L}') = \{1, 2, 3, 4\}$. Let $Q'\colon \mathcal{B} \to \mathbb{S}$ be given by $Q'(\texttt{L}') = \{x, y\}$ and let $k'\colon P' \to Q'$ be the map sending $1, 2 \mapsto x;\ \ 3, 4 \mapsto y$. Finally, let

$$P := \Pi_G(P'), \quad Q = \Pi_G(Q'), \text{ and } k := \Pi_G(k')\colon P \to Q.$$

We are ready to compute the right type-change functor along k on any P-typed instance $I \to P$; we just need to write down some such instance. So, if $I \in \mathcal{C}$–**Inst** is the table on the left then $\widehat{\Pi}_k(I)$ is the table in the middle:

$I :=$

L	
ID	**f**
a	1
b	2
c	1
d	3
e	2
f	4
g	2

$\widehat{\Pi}_k(I) =$

L	
ID	**f**
(a,b)	x
(a,e)	x
(a,g)	x
(c,b)	x
(c,e)	x
(c,g)	x
(d,f)	y

$$\begin{array}{cccc} & \mathrm{b} & & \\ \mathrm{a} & \mathrm{e} & & \\ \mathrm{c} & \mathrm{g} & \mathrm{d} & \mathrm{f} \\ \hline \multicolumn{2}{c}{\underbrace{1 \quad 2}_{x}} & \multicolumn{2}{c}{\underbrace{3 \quad 4}_{y}} \end{array}$$

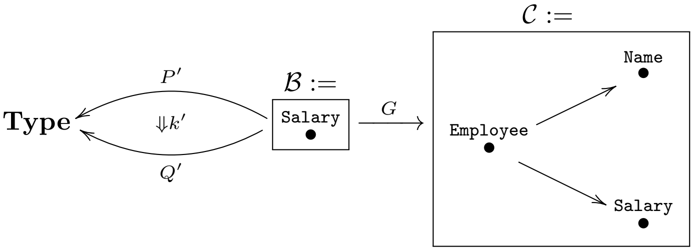
and a sketch of the reasoning is given on the right.

5.3. **Filtering data.** In this section we show how to use pullback type-change functor $\widehat{\Delta}$ to filter data (e.g. answer queries like "return the set of employees whose salary is less than \$100."

In fact, given a morphism of instances $k\colon P \to Q$, the associated pullback type-change functor $\widehat{\Delta}_k$ can either filter or multiply data (or both) depending on the injectivity or surjectivity of k. In this short section we concentrate only on filtering, because it appears to be more useful in practice. We work entirely by example; the definition of $\widehat{Delta}$ is given in Section 5.2 or in the literature ([MM, Section IV.7]).

Example 5.3.1. As advertised above, we show how to filter employees by their salaries, in particular showing only those with salaries less than \$100.

Suppose we have a schema $\mathcal{C}$ and two typing auxiliaries, $(\mathcal{B}, P, G)$ and $(\mathcal{B}, P', G)$, shown below:



Here we want $Q'(\texttt{Salary})$ to be the dollar-figure data type, we want $P'(\texttt{Salary})$ to be the subtype given by requiring that a dollar figure x be less than \$100, and we want $k'\colon P' \to Q'$ to be the inclusion. These typing auxiliaries induce a morphism of typing instances

$$k := \Pi_G(k')\colon P \to Q, \quad \text{where} \quad P := \Pi_G(P') \quad \text{and} \quad Q := \Pi_G(Q').$$

Now suppose that $I \in \mathcal{C}$–**Inst** is the Q-typed $\mathcal{C}$-instance shown to the left below. Then $\widehat{\Pi}_k(I)$ is the P-typed instance to the right below.

$I :=$

Employee		
ID	**Name**	**Salary**
Em101	Smith	\$65
Em102	Juarez	\$120
Em103	Jones	\$105
Em104	Lee	\$90
Em105	Carlsson	\$80

$\widehat{\Pi}_k(I) =$

Employee		
ID	**Name**	**Salary**
Em101	Smith	\$65
Em104	Lee	\$90
Em105	Carlsson	\$80

Thus we see that filtering is simply an application of the same data migration functor story.

5.4. **A normal form for data migration.** We have seen several different forms of data migration functors throughout this paper. One may perform a sequence of data migration functors, e.g. moving data from one schema to another, then changing the data types, and finally filtering the result. In many cases, such combination of data migration functors can be rewritten as a sequence of three: a pullback, a right pushforward, and a left pushforward; see [SW]. [12] This means that there is a normal form for a quite general class of queries. Any combination of such queries can be written in the form $\Delta_F\Pi_G\Sigma_H$ for some F, G, H. This form may not be optimal in terms of speed, but it can serve as a single input format for query optimizers.

References

[Awo] Awodey, S. *Category Theory* 2nd edition. Oxford Logic Guides, 52. Oxford University Press, Oxford 2010. Available online: http://teaguesterling.com/category-theory.pdf. (Accessed 2012/03/08.)

[AW] Awodey, S., Warren M. A. (2009) “Homotopy theoretic models of identity types.” *Math. Proc. Cambridge Philos. Soc.* 146 (1), pp. 45-55.

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

[ASX] Agrawal, R.; Somani, A.; Xu, Y. “Storage and querying of e-commerce data”. Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[BS] Baez, J.; Stay, M. “Physics, topology, logic and computation: a Rosetta Stone.” *New structures for physics*, 95Ð172. Lecture Notes in Phys., 813, Springer, Heidelberg, 2011.

[BW1] Barr, M.; Wells, C. *Category theory for computing science.* Prentice Hall International Series in Computer Science. New York 1990.

[BW2] Barr, M.; Wells, C. *Toposes, Triples, and Theories.* Reprints in Theory and Application of Categories, No. 1. 2005. Available online: http://www.case.edu/artsci/math/wells/pub/pdf/ttt.pdf (Accessed 2012/03/08.)

[Bor1] Borceux, F. *Handbook of categorical algebra 1.* Encyclopedia of Mathematics and its Applications 50, 51, 52. Cambridge University Press, Cambridge, 1994.

[BCW] Baclawski, K., Simovici, D., White, W. “A categorical approach to database semantics” *Mathematical structures in Computer Science* (1994) vol 4., pp. 147–183.

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

[BVP] Bohannon, A.; Vaughan, J.A.; Pierce, B.C. *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.

[CI] Cook, W.R.; Ibrahim, A.H. ÒIntegrating Programming Languages & Databases: WhatÕs the Problem?Ó ODBMS Expert Article. 2005.

[BCFGP] Barbosa, D.M.J.; Cretin, J.; Foster, N.; Greenberg, M.; Pierce, B. “Matching Lenses: Alignment and View update”. Department of Computer and Information science, Technical reports. 2010.

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

[DI] Döring, A.; Isham, C. J. “A topos foundation for theories of physics. I. Formal languages for physics.” *J. Math. Phys.* 49 (2008), no. 5, 053515.

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

[12] An earlier version of this paper included a false claim here. It stemmed from the assumption that the results of [GK] could be translated to this context and that therefore any sequence of data migration functors could be rewritten as a sequence of only three. This does not seem to be the case.

[DNR] Deutsch, A.; Nash, A.; Remmel, J. "The chase revisited." PODS 2008, pp. 149–158.

[DC] Diskin, Z.; Cadish, B. "Databases as Graphical Algebras: Algebraic Graph-Based Approach to Data Modeling and Database Design". Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.3655. (Accessed 2012/03/21.)

[EN] Emasri, R.; Navathe, S. *Database systems: models, languages, design, and application programming*, Edition 6. Pearson, 2011.

[EV] Ehresmann, A.C.; Vanbremeersch, J.P., *Memory Evolutive Systems; hierarchy, emergence, cognition* Elsevier Science, 2007.

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

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

[GB] Goguen, J.A.; Burstall, R.M. *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

[GSB] Giesa, T.; Spivak, D.I.; Buehler, M.J. "Reoccurring Patterns in Hierarchical Protein Materials and Music: The Power of Analogies" (2011) *BioNanoScience.* Volume 1, Number 4, 153-161, DOI: 10.1007/s12668-011-0022-5.

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

[Ham] Hamana, M. "Polymorphic abstract syntax via Grothendieck construction." Lecture notes in computer science, 2011, Vol 6604, 381-395.

[IP] Islam, A., Phoa, W. "Categorical models of relational databases I: fibrational fomulation, schema integration". *Lecture Notes in Comput. Sci.,* 789, Springer, Berlin, 1994.

[JD] Johnson, M.; Dampney, C.N.G. "On the value of commutative diagrams in information modeling." AMAST '93 Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology. Springer-Verlag, 1993.

[JoM] Johnson, M. "On Category Theory as a (meta) Ontology for Information Systems Research" *Proceedings of the international conference on Formal Ontology in INformation Systems*, 2001.

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

[Joy] Joyal, A. *Catlab*, available online: http://ncatlab.org/joyalscatlab/show/Factorisation+systems (Accessed 2012/03/08.)

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

[Kan] Kanda, Akira. "Data types as initial algebras: A unification of Scottery and ADJery," Foundations of Computer Science, IEEE Annual Symposium on, pp. 221-230, 19th Annual Symposium on Foundations of Computer Science (FOCS 1978), 1978.

[KC] Klyne G.; Carroll J.J. (Editors). "Resource Description Framework (RDF): Concepts and Abstract Syntax" *W3C Recommendation* 2004/02/10. http://www.w3.org/TR/rdf-concepts/. (Accessed 2012/03/08.)

[KT] Korotkiy, M.; Top, J. *From relational data to RDFS models.* Lecture Notes in Computer Science 2004, Vol 3140.

[LS] Lawvere, F.W.; Schanuel, S.H. *Conceptual Mathematics. A First Introduction to Categories.* Second edition. Cambridge University Press, Cambridge (2009).

[LT] Lellahi, K.; Tannen, V. "A calculus for collections and aggregates." Proceeding CTCS '97 Proceedings of the 7th International Conference on Category Theory and Computer Science. Springer-Verlag, 1997.

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

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

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

[Mar] Martin-Löf, P. *Intuitionistic type theory.* 1984.

[Mog] Moggi, E. “A category-theoretic account of program modules.” *Category theory and computer science (Manchester, 1989),* 101Ð117, Lecture Notes in Comput. Sci., 389, Springer, Berlin, 1989.

[Mor] Morava, J. “Theories of anything”, ePrint available: http://arxiv.org/abs/1202.0684v1

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

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

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

[Ros] Rosen, R. “The representation of biological systems for the stand-point of the theory of categories.” (1958) *Bull. Math. Biophys.* 20, pp. 317 – 342.

[Sic] Sica, G. (ed.) “What is category theory?”. Polimetrica S.a.s. Milan, Italy (2006).

[Sp1] Spivak, D.I.*Simplicial databases*. 2009. ePrint available: http://arxiv.org/abs/0904.2012

[Sp2] Spivak, D.I. *Table manipulation in simplicial databases*. 2010. Available online: http://arxiv.org/abs/1003.2682

[Sp3] Spivak, D.I. “Functorial data migration”. (2010) ePrint available: http://arxiv.org/abs/1009.1166 (Accessed 2012/03/08.)

[Sp4] Spivak, D.I. “Queries and constraints via lifting problems”. (2012) Available online: http://arxiv.org/abs/1202.2591 (Accessed 2012/03/08.)

[SK] Spivak, D.I., Kent, R. “Ologs: a category-theoretic foundation for knowledge representation” (2012) *PLoS ONE* 7(1): e24274. doi:10.1371/journal.pone.0024274.

[SGWB] Spivak, D.I.; Giesa, T.; Wood, E.; Buehler, M.J. (2011) “Category Theoretic Analysis of Hierarchical Protein Materials and Social Networks.” PLoS ONE 6(9): e23911. doi:10.1371/journal.pone.0023911.

[SH] Stone, C.; Harper, R. “Decidable type equivalence in a language with singleton kinds”. Principles of programming languages conference, 1999.

[SW] Spivak, D.I., Wisnesky, R. “On the relational foundations of functorial data migration”. Submitted to *PODS 2013*. ePrint available: http://arxiv.org/abs/1212.5303

[Tui] Tuijn, C. “Data modeling from a categorical perspective” PhD. thesis. University of Antwerpen, 1994.

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

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