

Categorical databases

David I. Spivak

`dspivak@math.mit.edu`
Mathematics Department
Massachusetts Institute of Technology

Presented on 2017/03/29
at the Broad Institute

Purpose of the talk

- We live in a world of interacting entities.
 - Humans interact with machines.
 - Companies interact with companies.
 - Databases interact with applications.
 - And so on, ad infinitum.
- Each entity has its own internal language optimized for operating in some context.
- Interaction between entities is mediated by an interaction between their languages.
- Communication is the successful transfer of information through interaction.
- We need to find a formal underpinning for meaningful communication.

Databases as languages

- A database schema serves as a language, optimized for the operation of some entity in some context.
- But today, entities and contexts change more and more frequently.
- We need to transfer information not only to others, but to our later, different selves.

Data management: how to change the form?

We should think of data management as one simple idea: a translation of information from one form to another.

- An **ETL process** translates from normalized form to warehouse form.
- A **data migration** translates between databases, from one schematic form to another.
- A **query** against a database translates from all-purpose form to specific-purpose (bite-size) form.
- An **update** does not change the schematic form, but still procedurally translates information.

Category theory for managing change of form

There is an fundamental connection between databases and categories.

I propose that:

- Category theory can simplify how we think about and use databases.
- We can clearly see all the working parts and how they fit together.
- Powerful theorems can be brought to bear on classical DB problems.

The pros and cons of relational databases

- Relational databases are reliable, scalable, and popular.
- They are provably reliable to the extent that they strictly adhere to the underlying mathematics.
- Make a distinction between
 - the “relational database” system you know and use, vs.
 - the relational model, as a mathematical foundation for this system.

You're not really using the relational model.

- Current implementations have departed from the strict relational formalism:
 - Tables may not be relational (duplicates, e.g from a query).
 - Nulls (and labeled nulls) are commonly used.
 - Updates are outside the model.
- The theory of relations (150 years old) is not adequate to mathematically describe modern DBMS.
- The relational model is either clumsy or completely absent in describing:
 - Foreign keys,
 - Updates,
 - Schema mappings and data migration,
 - Distributed databases, etc.
- Databases have been intuitively moving toward what's best described with a more modern mathematical foundation.

Category theory gives better description

- Category theory (CT) does a better job of describing what's already being done in DBMS.
 - Functional dependencies, foreign keys, key generation.
 - Non-relational tables (e.g. duplicates in a query).
 - Labeled nulls and semi-structured data.
- CT offers guidance for all sorts of information hand-off:
 - Foreign key as a handoff from one table to another.
 - Query as a handoff from general schema to specific schema.
 - ETL as a handoff from normalized schema to unnormalized schema.
 - Data migration as information handoff from one schema to another.
 - Update as information handoff from a schema to itself.
 - Applications as information handoff between programming language and database.

What is category theory?

- Category theory is the mathematics of information handoff.
 - That is, CT is about integrity of relationship under change of form.
- Since its invention in the early 1940s, category theory has revolutionized math.
- It's like set theory and logic, except more about semantics and designed to build bridges.
- Category theory has been proposed as a new foundation for mathematics (to replace set theory).
- It was invented to build bridges between disparate branches of math by distilling the essence of mathematical structure.

Branching out

- Category theory naturally fosters connections between disparate fields.
- It has branched out of math and into physics, linguistics, and materials science.
- It has had much success in the theory of programming languages.
 - Haskell is based in category theory and is gaining popularity.
 - The pure category-theoretic concept of *monads* has vastly extended the reach of functional programming.
- Can category theory improve how we think about databases?

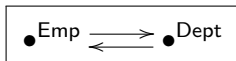
The basic similarity between databases and categories

- The connection between databases and categories is simple and strong.
- Reason: categories and database schemas do the same thing.
 - A schema gives a framework for modeling a situation;
 - Tables
 - Attributes
 - This is precisely what a category does.
 - Objects
 - Arrows.
 - They both model how entities within a given context interact.

Today's talk: Categorical model of databases

- There are many possible category-theoretic models of databases.
- The power of category theory is that these will all be comparable and interoperable.
 - Reason: category theory is based on behavior, not implementation.
 - When two models are describing similar behavior, they can be categorically compared.
- Today's talk will be about one particular model, which we'll call the functorial data model.
- The functorial data model is what you get when you demand:

Schema = Finite Category.



The category of categories

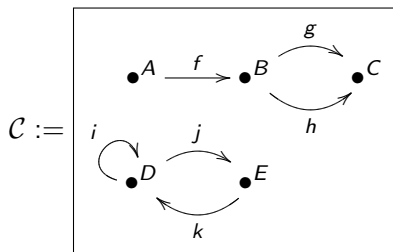
- Any finite category acts as a database schema.
 - It models the interactions between tables, via columns.
 - The semantics of this category is captured by an instance of the database.
 - Each instance describes the relationship between rows from interacting tables.
- But there is also a *category of categories*.
 - It models the interactions between database schemas.
 - The semantics of this category is understood as data migration, querying, etc.
 - Each data migration describes the relationship between instances of interacting schemas.

Rest of the talk

- Lay out the basic idea of categories and that of databases, and show the tight connection between them.
- Discuss querying, schema evolution, and data migration.
- Develop a connection to programming language theory.
- Understand RDF in these terms.

What is a category?

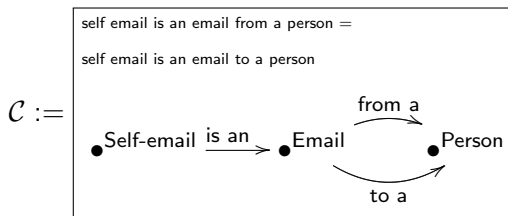
- Idea: A category models entities of a certain sort and the relationships between them.



- Think of it like a graph: the nodes are entities and the arrows are relationships.
- Some paths can be declared equivalent to others
 - Example: declare that $j; k \simeq i; i$ and $f; g \simeq f; h$.

Example

- How could one interpret this kind of abstraction?
- Here's a category with English-labeled nodes and arrows:



- Such “business rules” can be encoded into the category.

What is the essence of structure?

- If mathematics is the art of getting organized, what organizes math?
- After thousands of years, people realized that there were some essential features in common throughout much of math.
- These are objects, arrows, paths, and path equivalence.
- Or: things, tasks, processes, and “sameness of outcome”.
- Or: primary keys, foreign keys, paths of FKs, and path equations.
- Let's give the definition.

Definition of a category I: Constituents

A *category* $\mathcal{C}$ consists of the following constituents:

- ① A set $\mathbf{Ob}(\mathcal{C})$, called *the set of objects of $\mathcal{C}$* .
 - (These will be tables.)
 - Objects $x \in \mathbf{Ob}(\mathcal{C})$ is often written as $\bullet^x$.
- ② A set $\mathbf{Arr}(\mathcal{C})$, called *the set of arrows of $\mathcal{C}$* , and two functions

$$src, tgt: \mathbf{Arr}(\mathcal{C}) \rightarrow \mathbf{Ob}(\mathcal{C}),$$

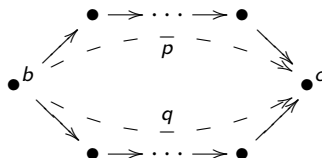
assigning to each arrow its *source* and its *target* object, respectively.

- (Arrows will be foreign keys from “source” table to “target” table.)
 - An arrow $f \in \mathbf{Arr}(\mathcal{C})$ is often written $\bullet^x \xrightarrow{f} \bullet^y$, where $x = src(f), y = tgt(f)$.
 - We define a *path in $\mathcal{C}$* to be a finite “head-to-tail” sequence of arrows in $\mathcal{C}$, e.g. $\bullet^x \xrightarrow{f} \bullet^y \xrightarrow{g} \bullet^z$.
- ③ An notion of equivalence for paths, denoted $\simeq$.

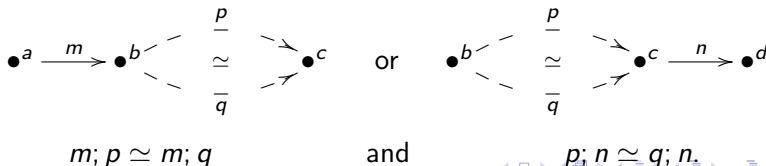
Definition of a category II: Rules

These constituents must satisfy the following requirements:

- ① If $p \simeq q$ are equivalent paths then the sources agree: $\text{src}(p) = \text{src}(q)$.
- ② If $p \simeq q$ are equivalent paths then the targets agree: $\text{tgt}(p) = \text{tgt}(q)$.
- ③ Suppose we have two paths (of any lengths) $b \rightarrow c$:



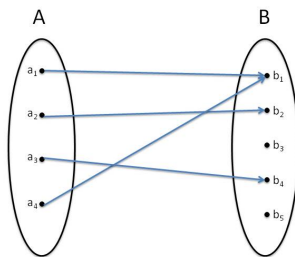
If $p \simeq q$ then for any extensions



What does equivalence of paths mean?

- Arrows represent foreign keys.
- A path $p: \bullet^a \rightarrow \bullet^b$ represents “following foreign keys” from table a to table b .
- Following a path p , we can take any record in table a and return a record in table b .
- We declare two paths $p, q: \bullet^a \rightarrow \bullet^b$ equivalent if they should return the same record in b for any record in a .
- In typical DB practices, equivalent paths are avoided by cutting one of the paths.
 - This is considered good design.
 - However, it often causes pain in ones neck.
 - Category theory has this concept built in.

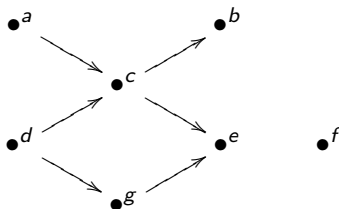
The category of Sets



- Above we see two sets and a function between them. We would denote this categorically by $\bullet^A \xrightarrow{f} \bullet^B$
 - The objects of **Set** represent sets.
 - The arrows in **Set** represent functions.
 - A path represents a sequence of composable functions.
 - Two paths are equivalent if their compositions are the same.

A totally different category: an ordered set

- A ordered set is a set S together with a notion of $\leq$, satisfying
 - $a \leq a$ for all $a \in S$, and
 - if $a \leq b$ and $b \leq c$, then $a \leq c$.
- Given some ordered set S , we can build a corresponding category $\mathcal{S}$:
 - $\mathbf{Ob}(\mathcal{S}) = S$,
 - One arrow $a \rightarrow b$ if $a \leq b$
 - No arrows $a \rightarrow b$ if $a \not\leq b$.
 - All pairs of paths (having same source and target) are equivalent.
- "Hasse diagram":



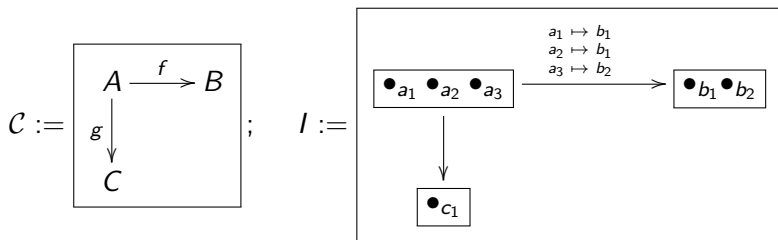
- Think "permissions": $a \leq c$ means a has fewer accessors than b .

Functors: mappings between categories

- One way to think of a category is as a directed graph, where certain paths have been declared equivalent.
- A functor is a graph mapping that is required to respect equivalence of paths.
- **Definition:** A functor $F: \mathcal{C} \rightarrow \mathcal{D}$ consists of
 - a function $\mathbf{Ob}(\mathcal{C}) \rightarrow \mathbf{Ob}(\mathcal{D})$ and
 - a function $\mathbf{Arr}(\mathcal{C}) \rightarrow \mathbf{Path}(\mathcal{D})$,such that F
 - respects sources and targets,
 - respects equivalences of paths.

Functors to **Set**

- A category $\mathcal{C}$ is a system of objects and arrows, and an equivalence relation on its paths.
- A functor $\mathcal{C} \rightarrow \mathcal{D}$ is a mapping that preserves these structures.
- **Set** is the category whose objects are sets, whose arrows are functions, and where paths are equivalent if they compose to the same function.
- If $\mathcal{C}$ is the category on the left below, then a functor $I: \mathcal{C} \rightarrow \mathbf{Set}$ might look like this:



What is a database?

- A database consists of a bunch of tables and relationships between them.
- The rows of a table are called “records” or “tuples.”
- The columns are called “attributes.”
- An attribute may be “pure data” or may be a “key.”
 - A table may have “foreign key columns” that link it to other tables.
 - A foreign key of table A links into the primary key of table B .
- A schema may have “business rules.”

Foreign Keys and business rules

- Example:

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
q10	Sales	101
x02	Production	102

- Note the Id (primary key) columns and the foreign key columns.
 - Id column could just be internal “row numbers” or could be typed.
 - “Row numbers” (i.e. pointers) are not part of the relational model but they are naturally part of the categorical model.
- Perhaps we should enforce certain integrity constraints (business rules):
 - The manager of an employee E must be in the same department as E ,
 - The secretary of a department D must be in D .

Data columns as foreign keys

- Theoretically we can consider a data-type as a 1-column table.
- Examples:

String	
Id	
a	
b	
.	
.	
.	
z	
aa	
ab	
.	
.	
.	

Integer	
Id	
0	
1	
.	
.	
.	
9	
10	
11	
.	
.	
.	

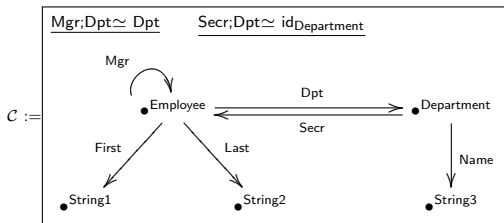
- So even data columns can be considered as foreign keys (to respective 1-column tables).
- Conclusion: each column in a table is a key – one primary, the rest foreign.

Example again

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

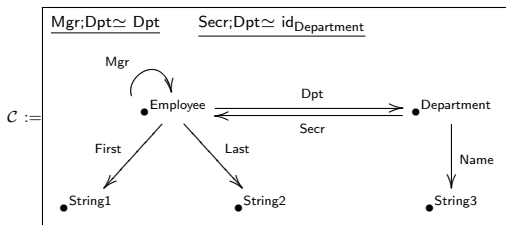
Department		
Id	Name	Secr
q10	Sales	101
x02	Production	102

String	
Id	
a	
b	
.	
.	
z	
aa	
ab	
.	
.	
.	



Database schema as a category

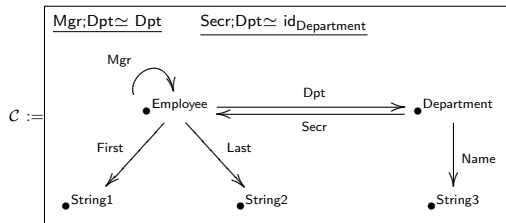
- A database schema is a system of tables linked by foreign keys.
- This is just a category!



- Each object x in $\mathcal{C}$ is a table (Employee, Departments, String);
- each arrow $x \rightarrow y$ is a column of table x .
- Id column of a table corresponds to the trivial path on that object.
- Declaring business rules (e.g. $\text{Mgr}; \text{Dpt} \simeq \text{Dpt}$) is declaring the path equivalence.

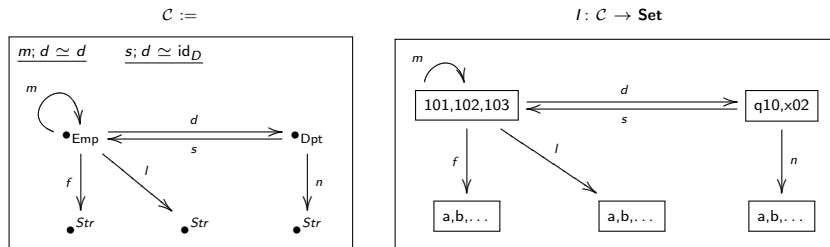
Schema=Category, Instance=Set-valued functor

- Let $\mathcal{C}$ be the following category



- A functor $I : \mathcal{C} \rightarrow \mathbf{Set}$ consists of
 - A set for each object of $\mathcal{C}$ and
 - a function for each arrow of $\mathcal{C}$, such that
 - the declared equations hold.
- In other words, I fills the schema with compatible data.
- Categorical databases could also be called *functional databases*.

Data as a set-valued functor



- A category $\mathcal{C}$ is a schema. An object $x \in \mathbf{Ob}(\mathcal{C})$ is a table.
- A functor $I: \mathcal{C} \rightarrow \mathbf{Set}$ fills the tables with compatible data.
- For each table x , the set $I(x)$ is its set of rows.
- The path equivalences in $\mathcal{C}$ are enforced by I as business rules.

Summary

- The connection between categories and databases is simple.
- A schema is a custom category.
- Functors $I: \mathcal{C} \rightarrow \mathbf{Set}$ are instances.
- What about functors $F: \mathcal{C} \rightarrow \mathcal{D}$ between schemas?

Changes

- We've discussed the situation as though static: a single schema and a single instance.
- Next we'll discuss changes.
- Changing the schema (schema mappings).
 - This topic covers data migration, queries, updates, and ETL in a unified way.
- Managing change: provenance.

Changes in schema

- Suppose in our modeling of a given context, we evolve from schema $\mathcal{C}$ to schema $\mathcal{D}$.
- We should find a functorial connection between them.
- Over time we may have something like

$$\mathcal{C} = \mathcal{C}_0 \xrightarrow{F_0} \mathcal{C}_1 \xrightarrow{F_1} \cdots \xrightarrow{F_{n-1}} \mathcal{C}_n = \mathcal{D}$$

- We want to be able to migrate data from $\mathcal{C}$ to $\mathcal{D}$ and vice versa.
- We want to be able to migrate queries against $\mathcal{C}$ to queries against $\mathcal{D}$ and vice versa.
- And we want this all to work as it “should”.

Composing functors

- Suppose $F: \mathcal{C} \rightarrow \mathcal{D}$ and $G: \mathcal{D} \rightarrow \mathcal{E}$ are functors.
- What is their composition $\mathcal{C} \rightarrow \mathcal{E}$?
 - We have a way to take objects in $\mathcal{C}$ to objects in $\mathcal{E}$,
 - Arrows in $\mathcal{C}$ turn into paths in $\mathcal{D}$ and arrows in $\mathcal{D}$ turn into paths in $\mathcal{E}$.
 - We can concatenate these, thus taking arrows in $\mathcal{C}$ to paths in $\mathcal{E}$.
 - Our rules ensure that the equivalences in $\mathcal{C}$ will be preserved in $\mathcal{E}$.
- Composing functors is going to make migrating data more straightforward.

Changes in data

- Let $\mathcal{C}$ be a schema and let $I, J: \mathcal{C} \rightarrow \mathbf{Set}$ be two instances.
- A *natural transformation* $p: I \rightarrow J$ consists of the following:
 - For each object (table) $T \in \mathbf{Ob}(\mathcal{C})$ we get a map of record sets

$$p_T: I(T) \rightarrow J(T).$$

- For each arrow (foreign key) $f: T \rightarrow T'$, we get data consistency; formally,

$$J(f) \circ p_T = p_{T'} \circ I(f).$$

- A natural transformation $p: I \rightarrow J$ gives provenance:
 - A coherent story for how everything in I was transformed to something in J .

The category of instances

- Given a schema $\mathcal{C}$, the *category of instances* on $\mathcal{C}$ is denoted $\mathcal{C}\text{-Inst}$.
 - The objects of $\mathcal{C}\text{-Inst}$ are functors (instances) $I: \mathcal{C} \rightarrow \mathbf{Set}$.
 - The arrows are natural transformations (provenance).
 - Two provenance paths are equivalent if they result in the same “provenance story”.
- Mathematicians have studied categories like $\mathcal{C}\text{-Inst}$, decades before a connection to databases was known.
 - In particular, the category $\mathcal{C}\text{-Inst}$ is a topos.
 - As such, it has an internal language and logic supporting the *typed lambda calculus*.
 - That means, it works well with the theory of programming languages.

Data migration

- Let $\mathcal{C}$ and $\mathcal{D}$ be different schemas.
- We call a functor between them, $F: \mathcal{C} \rightarrow \mathcal{D}$, a *schema mapping*.
- Given such a mapping, we want to be able to canonically transfer instances on $\mathcal{C}$ to instances on $\mathcal{D}$ and vice versa.
- That means, given $F: \mathcal{C} \rightarrow \mathcal{D}$ we want functors

$$\mathcal{C}\text{-Inst} \rightarrow \mathcal{D}\text{-Inst}$$

and

$$\mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}.$$

What a functor $\mathcal{C}\text{-Inst} \rightarrow \mathcal{D}\text{-Inst}$ means.

A functor $\mathcal{C}\text{-Inst} \rightarrow \mathcal{D}\text{-Inst}$ means:

- **Objects:** To every instance on $\mathcal{C}$ we associate an instance on $\mathcal{D}$.
- **Arrows:** Provenance between $\mathcal{C}$ -instances is converted to provenance between their associated $\mathcal{D}$ -instances.
- **Path equivalences:** Equality of provenance paths is preserved too.

The “easy” migration functor, Δ

- Given a schema mapping (i.e. a functor)

$$F: \mathcal{C} \rightarrow \mathcal{D},$$

we can transform instances on $\mathcal{D}$ to instances on $\mathcal{C}$ as follows:

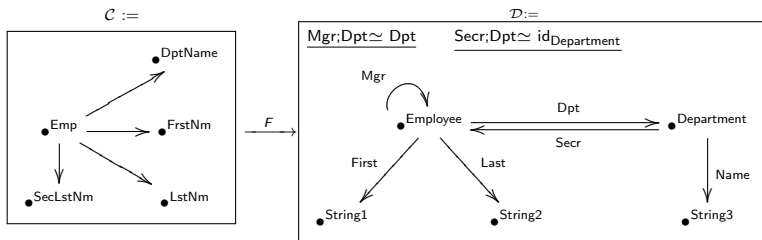
$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F} \mathcal{D} & \xrightarrow{I} \mathbf{Set} \\ & \searrow \scriptstyle F;I & \nearrow \\ & & \end{array}$$

Given $I: \mathcal{D} \rightarrow \mathbf{Set}$ get $F;I: \mathcal{C} \rightarrow \mathbf{Set}$

- This process will preserve provenance.
- Thus we have a functor $\Delta_F: \mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}$.

How Δ_F works

- Consider the schema mapping



- We get $\Delta_F: \mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}$
- Given an instance on $\mathcal{D}$ we get one on $\mathcal{C}$.
- Given an (insert) update on $\mathcal{D}$ we get one on $\mathcal{C}$.

So many kinds of functors..

- Functors in three different contexts.
 - We started with functors as instances, $I: \mathcal{C} \rightarrow \mathbf{Set}$.
 - Then we introduced functors as schema mappings, $F: \mathcal{C} \rightarrow \mathcal{D}$.
 - In the last slide we showed a functor on instance categories

$$\Delta_F: \mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}.$$

- Recall the simple definition of functor we gave at the beginning: it holds in each case.
- Functors provide a powerful and reusable abstraction because of the simplicity of their definition.

Adjoints

- Some functors $\mathcal{X} \rightarrow \mathcal{Y}$ have a “special partner” $\mathcal{Y} \rightarrow \mathcal{X}$ called an *adjoint*.
- What it will mean to us is that we can always “invert” a data migration $\mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}$ in two universal ways.
- These migration functors are “weak inverses” for Δ_F .
 - As such they provide something like updatable views.
- The important thing is to note is that these weren’t made up for databases.
 - They are “canonical” or “universal”, and well-known in mathematics.
 - Coincidentally, they also correspond to well-known data management operations.

The “adjoint” migration functors, Σ and Π

Given a schema mapping (i.e. a functor) $F: \mathcal{C} \rightarrow \mathcal{D}$,

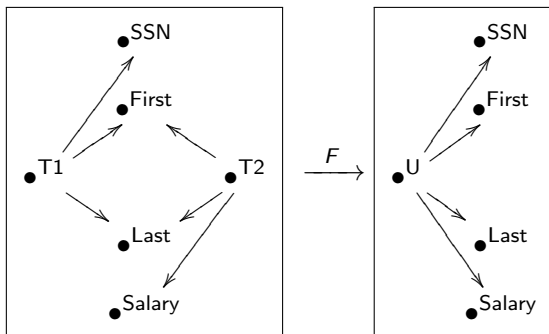
- We have a functor $\Delta_F: \mathcal{D}\text{-Inst} \rightarrow \mathcal{C}\text{-Inst}$ given by composition.
- It has two adjoints:
 - a “sum-oriented” adjoint $\Sigma_F: \mathcal{C}\text{-Inst} \rightarrow \mathcal{D}\text{-Inst}$, and
 - a “product-oriented” adjoint $\Pi_F: \mathcal{C}\text{-Inst} \rightarrow \mathcal{D}\text{-Inst}$.
- Thus, given a schema mapping F , three functors emerge for the instance categories,

$$\Delta_F, \Sigma_F, \text{ and } \Pi_F$$

come with the package.

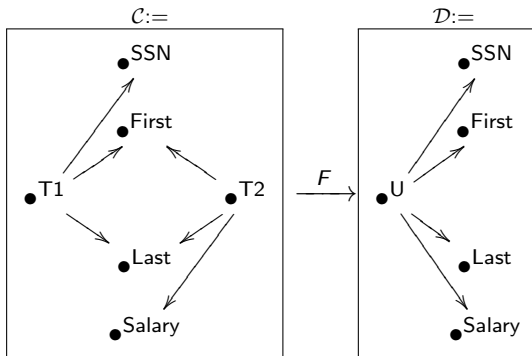
- Roughly, these correspond to project (Δ), union (Σ), and join (Π).
- They allow one to move data back and forth between $\mathcal{C}$ and $\mathcal{D}$ in canonical ways.

The “product-oriented” push-forward Π_F makes joins



- Given any instance $I: \mathcal{C} \rightarrow \mathbf{Set}$, get an instance $\Pi_F(I): \mathcal{D} \rightarrow \mathbf{Set}$.
- The rows in table $\bullet^U$ will be the join of the rows in $\bullet^{T1}$ and $\bullet^{T2}$ over $\bullet^{First}$ and $\bullet^{Last}$.

The “sum-oriented” push-forward Σ_F makes unions



- Given any instance $I: \mathcal{C} \rightarrow \mathbf{Set}$, get an instance $\Sigma_F(I): \mathcal{D} \rightarrow \mathbf{Set}$.
- The rows in table $\bullet^U$ will be the union of the rows in $\bullet^{T1}$ and $\bullet^{T2}$.
- It will automatically use labeled nulls for the unknown cells.

Views

- These functors can be arbitrarily composed to create views.
- We can think of any series of functors

$$\mathcal{C}_1 \xleftarrow{F_1} \mathcal{D}_1 \xrightarrow{G_1} \mathcal{E}_1 \xrightarrow{H_1} \mathcal{C}_2 \xleftarrow{F_2} \mathcal{D}_2 \xrightarrow{G_2} \dots \xrightarrow{H_{n-1}} \mathcal{C}_n$$

as a view.

- The view is the functor

$$V := \Sigma_{H_{n-1}} \circ \dots \circ \Pi_{G_1} \circ \Delta_{F_1} : \mathcal{C}_1\text{-Inst} \rightarrow \mathcal{C}_n\text{-Inst}.$$

- We can export data from $\mathcal{C}_1$ into $\mathcal{C}_n$ through V .
- Note that $\mathcal{C}_n$ is a schema: not just one table, but possibly many, with foreign keys.
- It's no problem to create views that have foreign keys (unsupported in DBMS).

FQL, a functorial query language

- All of the above has been implemented in an open source tool.
 - It is called FQL, and you can get it at:
<http://categoricaldata.net/fql.html>
 - This is thanks to Ryan Wisnesky, a postdoc at MIT.
- In the FQL programming language, one can do the following:
 - Enter categorical schemas (nodes, arrows, equations).
 - Enter schema mappings, i.e. functors (nodes to nodes, etc.).
 - Enter instances, i.e. functors $\mathcal{C} \rightarrow \mathbf{Set}$.
 - Use Δ, Σ, Π to move data between schemas and perform queries.
- FQL also has tools for translating to and from SQL and RDF.

Summary of functorial data migration

- By connecting different schemas graphically, we create data migration functors.
- These can be composed to move data from one schema to another.
- But this is not just enterprise-level data migration.
- The same idea covers:
 - Queries,
 - Data warehousing,
 - Updates.

How's the time?

Shall we skip to the summary now, or keep going with programming languages and RDF?

Connection to Programming Languages

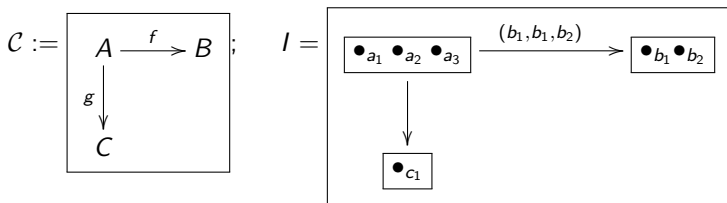
- Functional programming languages are based on category theory.
- The datatypes in a programming language form a category.
- Think of each type (integers, etc.) as a table:
 - Each element of that type is a row.
 - Each program that eats that type is a column.
- We've seen that database schemas are custom categories.
- The whole point of category theory is to allow us to connect different categories.
- Upshot: it is straightforward to categorically connect programming languages and databases.

Structured vs. unstructured data

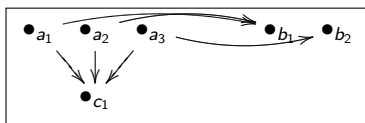
- Think of RDF as the worst kind of data.
- It's easiest to get because it takes no effort to create.
- Everyone's talking about schema-less databases.
- My colleague: "we don't need schema-less, we need schema-more!"
 - In other words, lower the bar to creating tiny custom schemas.
 - Make it easy to translate between them.
- Category theory can field the creation and interconnection of an abundance of custom schemas.
 - Once you've lost the structure of data, it's as easy to parse as a .jpeg.
 - If you keep even a modicum around, you can do much better.
- But let's say you want to think about RDF category-theoretically.

The Grothendieck construction

- Let $\mathcal{C}$ be a category and let $I: \mathcal{C} \rightarrow \mathbf{Set}$ be a functor.
- We can convert I into a category $Gr(I)$ in a canonical way:
 - Example:



- $Gr(I)$ is also known as *the category of elements of I* :

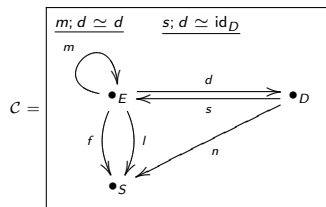


Grothendieck construction applied to database instances

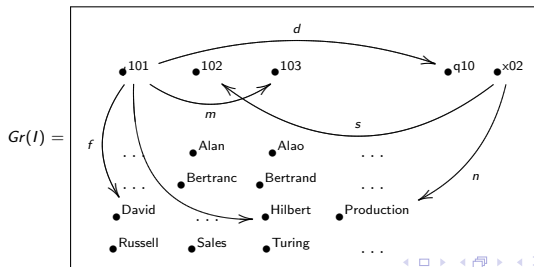
- Suppose given the following instance, considered as $I: \mathcal{C} \rightarrow \mathbf{Set}$

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



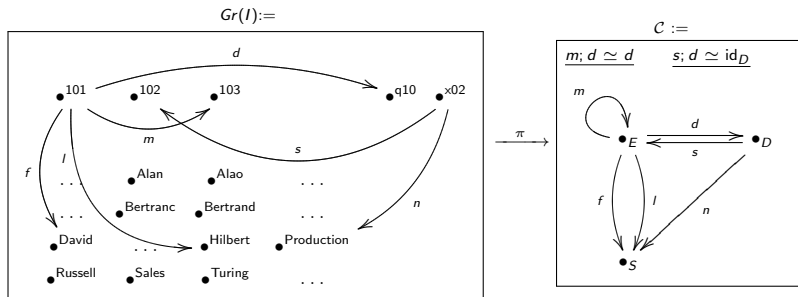
Here is $Gr(I)$, the category of elements of I :



A different perspective on data

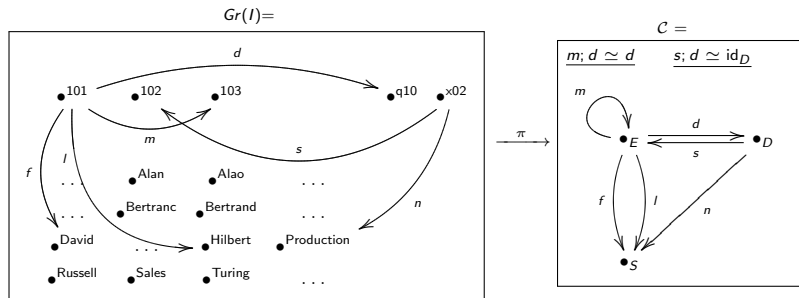
In fact, the Grothendieck construction of $I: \mathcal{C} \rightarrow \mathbf{Set}$ always yields not only a category $Gr(I)$ but a functor

$$\pi: Gr(I) \rightarrow \mathcal{C}.$$



The fiber over (inverse image of) every object $X \in \mathcal{C}$ is a set of objects $\pi^{-1}(X) \subseteq Gr(I)$. That set is $I(X)$.

OWL schema and RDF data



- The relation to RDF triples is clear: each arrow $f: x \rightarrow y$ in $Gr(I)$ is a triple with subject x , predicate f , and object y .
- For example (101 department q10), (x02 name Production), etc..
- $\mathcal{C}$ is an OWL schema and $Gr(I)$ is an RDF triple store.
- SPARQL queries (graph patterns) are easily expressible in this model.

Summary

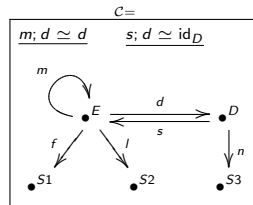
- There's a well-known connection between relational databases and RDF.
- This connection is born out in a most natural way with category theory.
- The model gracefully extends – what should work works.

Summary of the talk

- I hope the connection between databases and categories is clear.

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
q10	Sales	101
x02	Production	102



- Queries are a form of data migration; a unified approach.
- Category theory allows us to change perspectives on information.
 - Without a good theory of how different perspectives interoperate, there's endless duplication of effort.
 - Math can help information management evolve to handle 21st century challenges.

Thanks for the invitation to speak!