Databases are categories

David I. Spivak

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

Presented on 2010/11/03

Purpose of the talk

The purpose of this talk is:

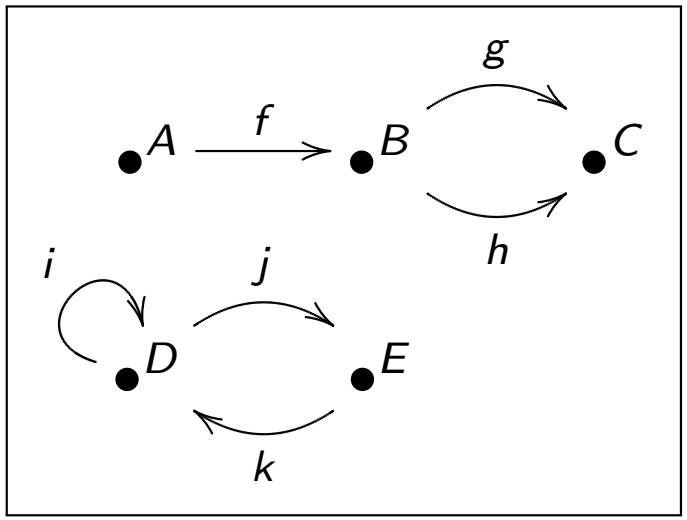
- To make a clear connection between databases and categories.
- To show the advantages brought by this connection:
  - conceptual clarity,
  - inclusion of RDF and semi-structured data in the model,
  - ability to define functorial data migration and schema evolution,
  - good understanding of views and queries,
  - integration with PL.
- To discuss possible applications to data provenance.

Basic idea

- A database schema is a category $\mathcal{C}$.
- A state on $\mathcal{C}$ is a functor $\delta\colon \mathcal{C} \longrightarrow \mathbf{Set}$.
- The power of this idea is in its simplicity.
  - The model is clear and easy to learn.
  - Its simplicity invites flexibility and expansion.
  - It is self-contained — much of database theory fits inside without ad-hoc extensions.

Categories

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

$\mathcal{C} :=$ 

- Think of it like a graph: the nodes are objects and the arrows are relationships.
- Some paths can be equated with others (example: $j.k = i^3$).

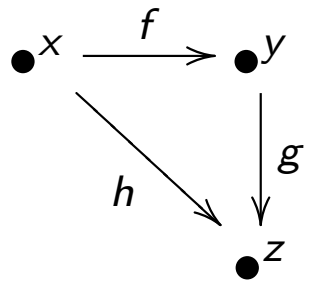
Formal definition of a category I: data

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

1. A set $\mathbf{Ob}(\mathcal{C})$, called *the set of objects of* $\mathcal{C}$.
   Objects $x \in \mathbf{Ob}(\mathcal{C})$ may be written as $\bullet^x$ or simply as x.
2. For each $x, y \in \mathbf{Ob}(\mathcal{C})$ a set $\mathbf{Arr}_{\mathcal{C}}(x, y)$, called *the set of arrows in* $\mathcal{C}$ *from* x *to* y.
   An element of $\mathbf{Arr}_{\mathcal{C}}(x, y)$ may be written $f : x \to y$ or $\bullet^x \xrightarrow{f} \bullet^y$.
3. For each $x, y, z \in \mathbf{Ob}(\mathcal{C})$ a *composition law*

$$\mathbf{Arr}_{\mathcal{C}}(x, y) \times \mathbf{Arr}_{\mathcal{C}}(y, z) \to \mathbf{Arr}_{\mathcal{C}}(x, z).$$

   We write $f; g = h$ to denote that following arrow f then arrow g is the same as following arrow h:



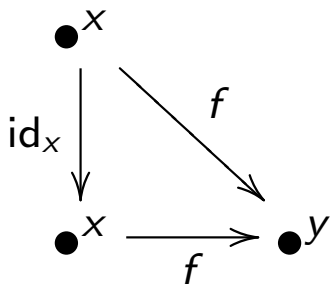
Formal definition of a category II: rules

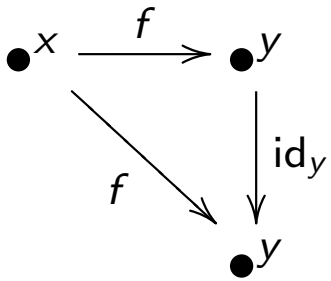
These data must satisfy the following requirements:

1. For every object $y \in \mathbf{Ob}(\mathcal{C})$ there is an "identity arrow"

$$\mathrm{id}_y : y \longrightarrow y$$

in $\mathbf{Arr}_{\mathcal{C}}(y, y)$ such that for any $f : x \longrightarrow y$, the equations $\mathrm{id}_x ; f = f$ and $f ; \mathrm{id}_y = f$ hold:





2. Composition is *associative*. That is, given

$$\bullet^{w} \xrightarrow{f} \bullet^{x} \xrightarrow{g} \bullet^{y} \xrightarrow{h} \bullet^{z},$$

the following equation holds:

$$f ; (g ; h) = (f ; g) ; h$$

The category of Sets

- Arguably the most important category is the category of Sets.
- I'll denote it **Set**.
- An object in **Set** is a set, like $\emptyset, \{1, 2, 7\}$, or $\mathbb{N}$.
- An arrow in **Set** is a total function.
  - For example there are 8 functions (arrows) from $\{x, y\}$ to $\{1, 2, 7\}$.
- The composition law simply takes two "composable" functions

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

and spits out their "composition" $f; g \colon A \to C$.

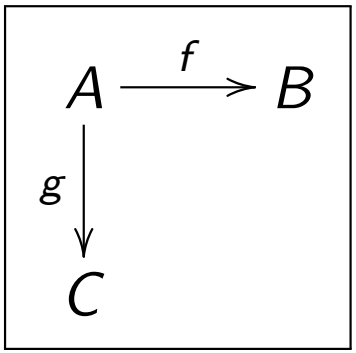
Functors: mappings between categories

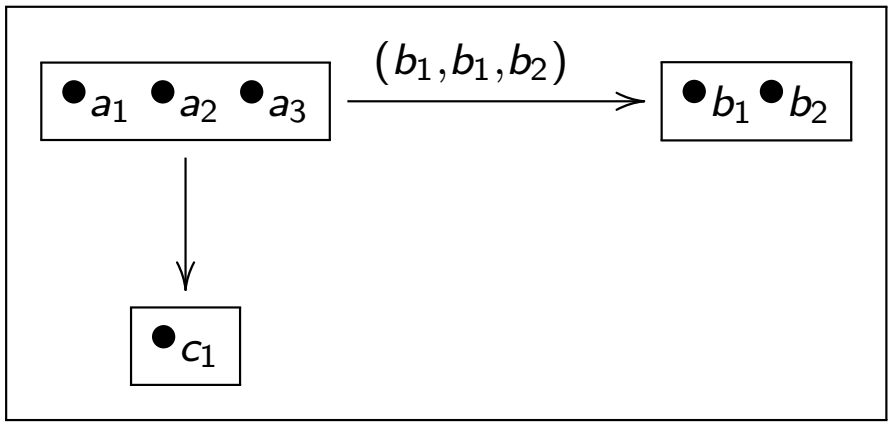
- One way to think of a category is as a directed graph, where certain paths have been declared equivalent.
- A functor is a graph morphism that is required to respect these equivalences.
- Definition: A functor $F : \mathcal{C} \to \mathcal{D}$ consists of
  - A function $\mathbf{Ob}(F) : \mathbf{Ob}(\mathcal{C})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

  that respect
  - the source and target of every arrow,
  - the identity arrow of every node, and
  - the composition law.
- Note that the composition of functors is a functor.

Recap

- A category $\mathcal{C}$ is a system of objects and arrows, and a composition law.
- A functor $\mathcal{C} \to \mathcal{D}$ is a mapping that preserves these structures.
- **Set** is the category whose objects are sets, whose arrows are (total) functions, and whose composition law is the specification of how functions compose.
- If $\mathcal{C}$ is the category on the left below, then a functor $\delta : \mathcal{C} \to \mathbf{Set}$ might look like this:

$\mathcal{C} :=$ 

; $\delta =$ 

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."
- A column 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.
  - Luckily, they are a part of every implementation (I have been told), so this model fits better with practice than the relational model does.
- 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
a
b
⋮
z
aa
ab
⋮

Integer
0
1
⋮
9
10
11
⋮

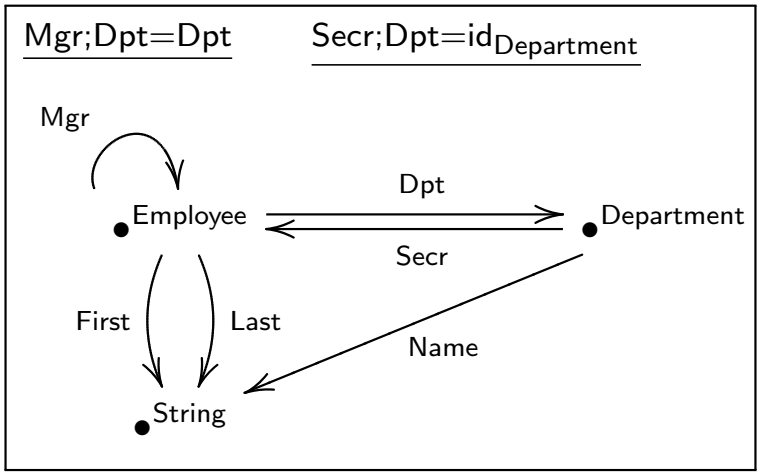
- So even data columns can be considered as foreign keys (to respective 1-column tables).
- Conclusion: each column in a table is a key – one primary, the rest foreign.

Example again

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

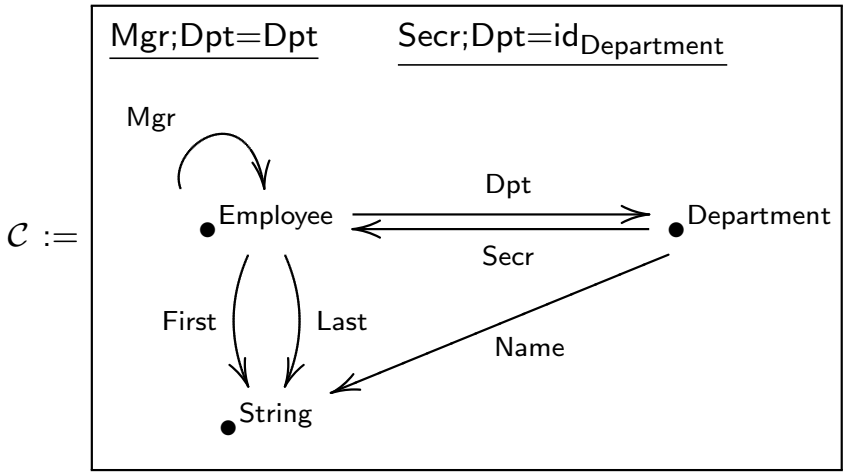
Department		
Id	**Name**	**Secr**
q10	Sales	101
x02	Production	102

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!

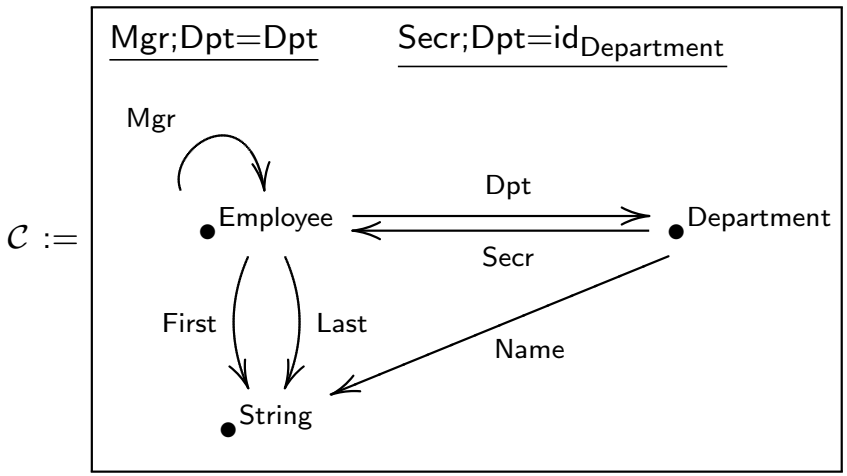


$\mathcal{C} :=$

  - Each object x in $\mathcal{C}$ is a table (Employee, Departments, String);
  - each arrow $x \longrightarrow y$ is a column of table x.
- Id column of a table corresponds to the identity arrow of that object.
- Declaring business rules (e.g. Mgr;Dpt=Dpt) is declaring the composition law.

Schema=Category, Data=Functor

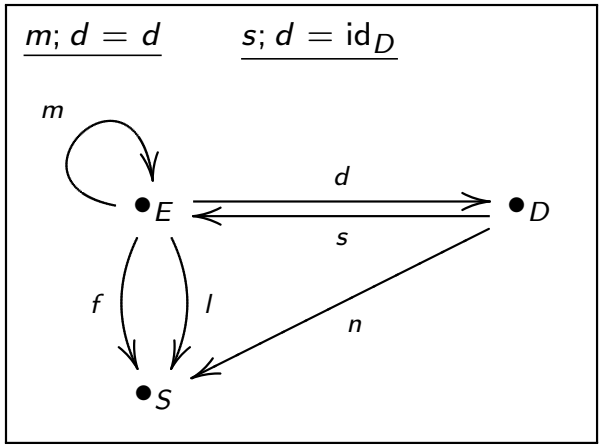
- Let $\mathcal{C}$ be the category



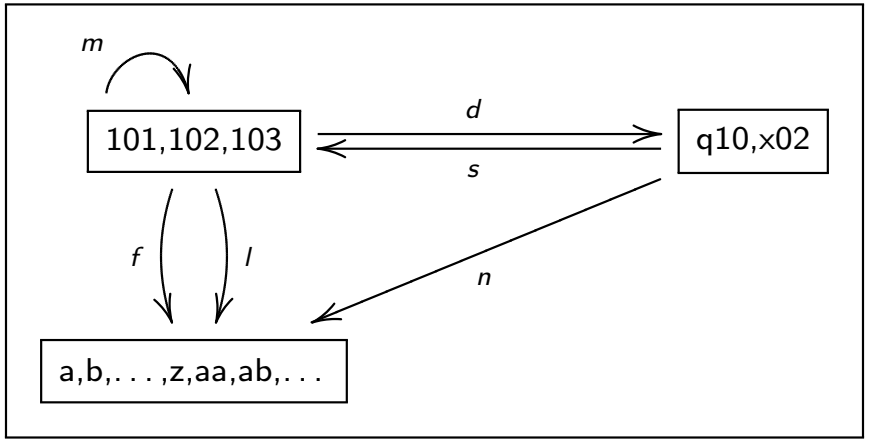
- A functor $\delta : \mathcal{C} \to \mathbf{Set}$ consists of
  - A set for each object of $\mathcal{C}$ and
  - a function for each arrow of $\mathcal{C}$, such that
  - the declared equations hold.
- In other words, δ fills the schema with data.

Data as a set-valued functor

$\mathcal{C} :=$



$\delta : \mathcal{C} \to \mathbf{Set}$



.

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

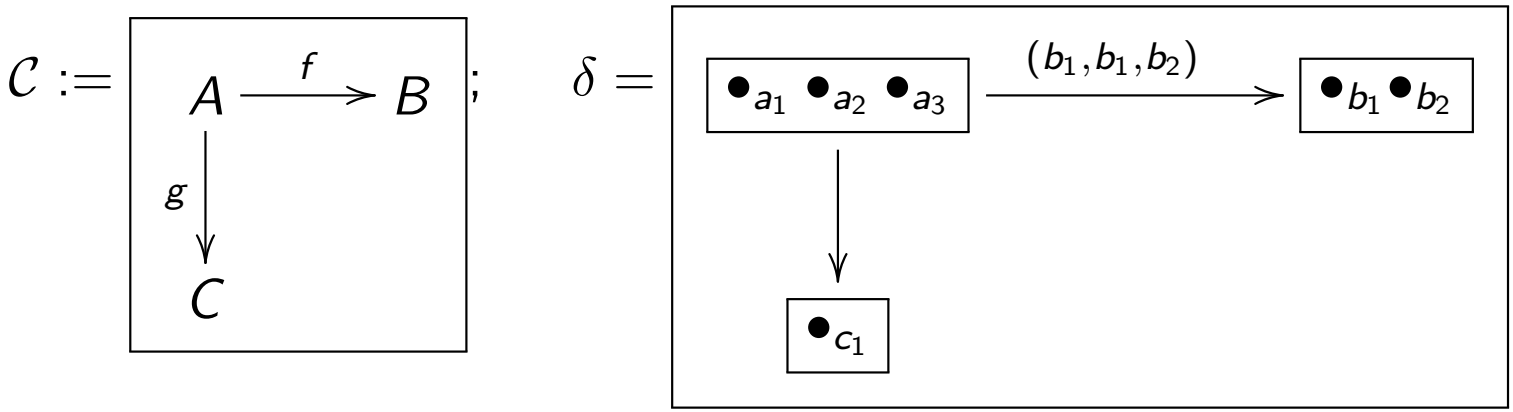
What does this model do for you?

The above ideas are hopefully clear, but what are the advantages?

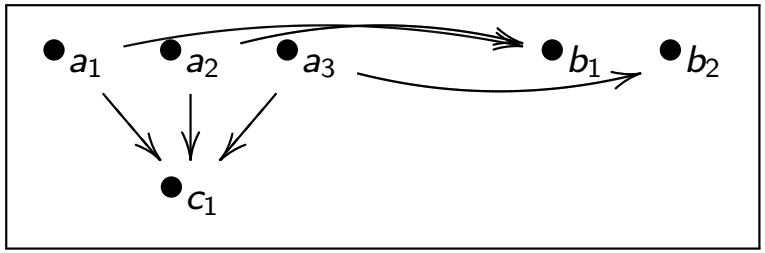
- Conceptual clarity is nice; the model is straightforward.
- A simple model is a flexible model.
- Outline of the rest of the talk:
  - The Grothendieck construction gives the conversion from relational to RDF.
  - The RDF picture allows us to model semi-structured (or what appears to RDB theory as incomplete, non-atomic, or bad) data.
  - Functors between schemas give rise to data migration and views.
  - Integration with PL can also be accomplished with functors.
  - These ideas may help in formalizing data provenance.

The Grothendieck construction

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



  - $Gr(\delta)$ is also known as *the category of elements of* δ:



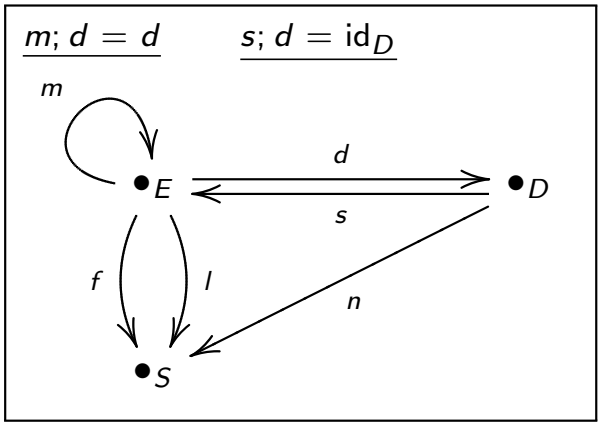
Grothendieck construction applied to database states

- Suppose given the following state, considered as $\delta\colon \mathcal{C} \to \mathbf{Set}$

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

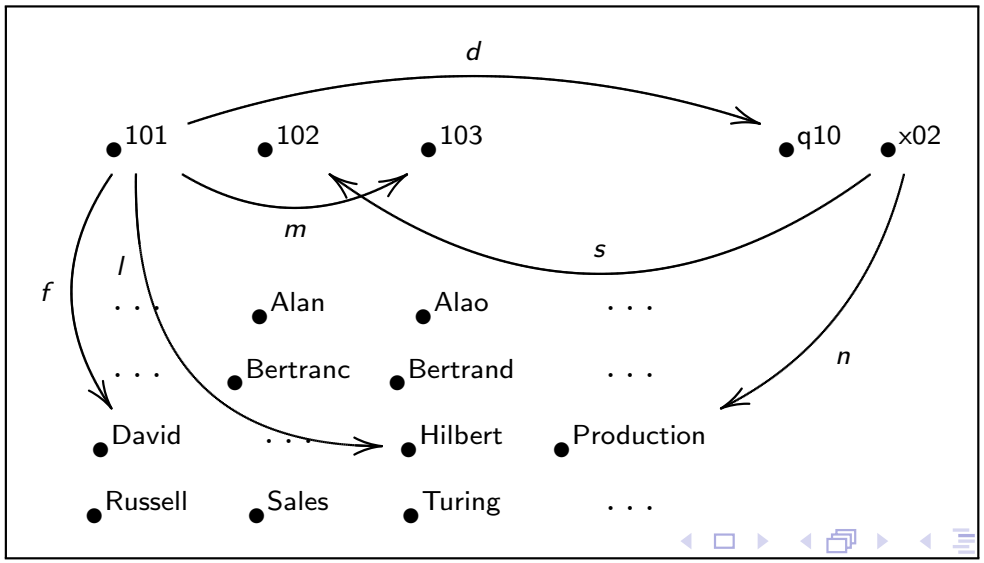
Department		
Id	**Name**	**Secr'y**
q10	Sales	101
x02	Production	102

$\mathcal{C} =$



Here is $Gr(\delta)$, the category of elements of δ:

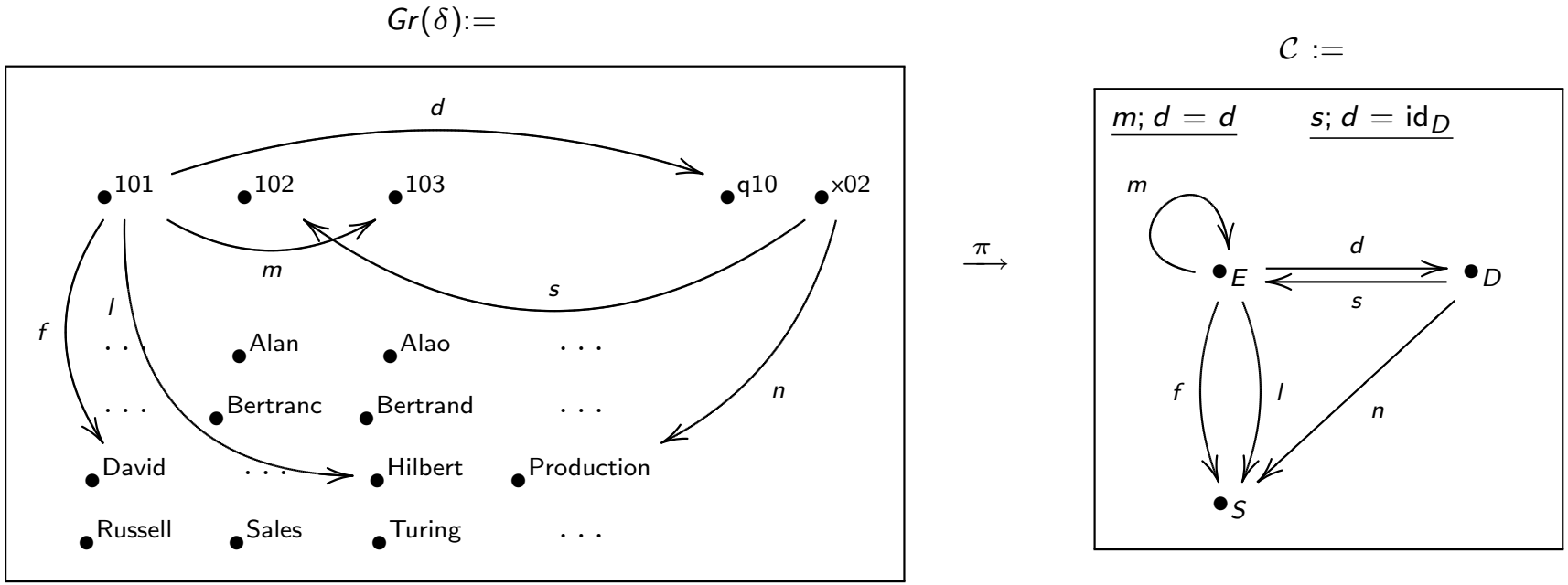
$Gr(\delta) =$



A different perspective on data

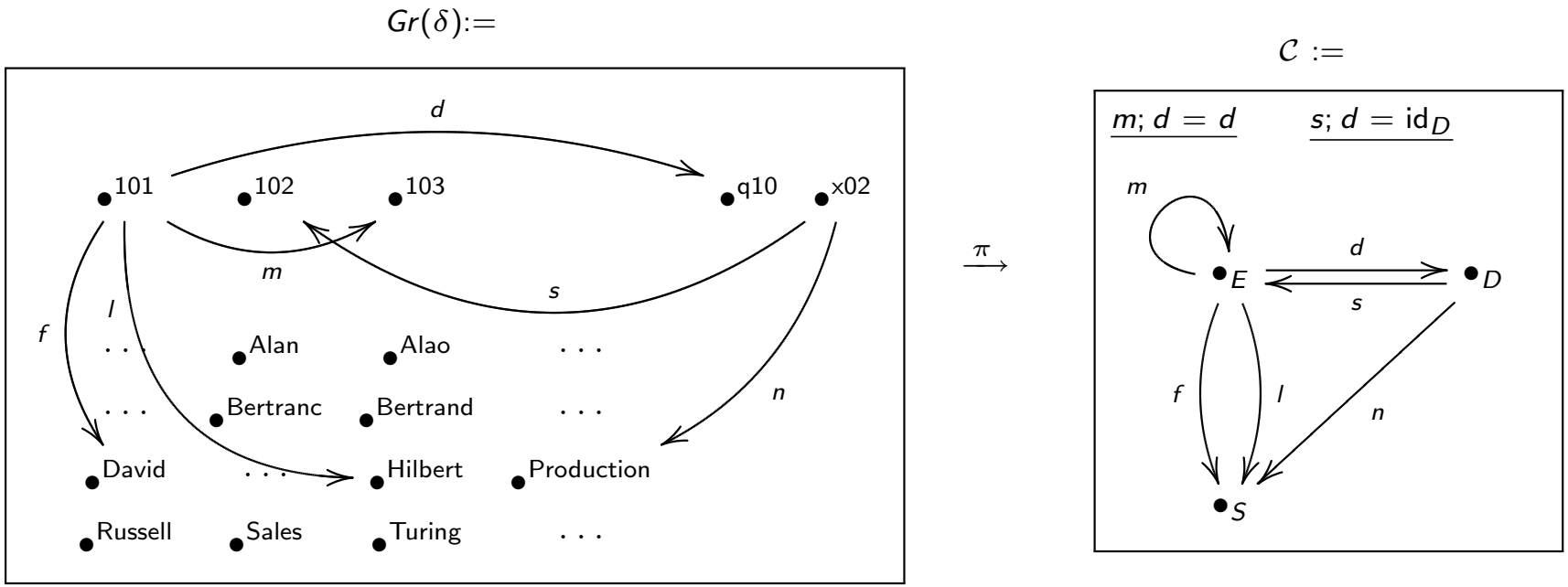
In fact, the Grothendieck construction of $\delta\colon \mathcal{C} \to \mathbf{Set}$ always yields not only a category $Gr(\delta)$ but a functor

$$\pi\colon Gr(\delta) \to \mathcal{C}.$$



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

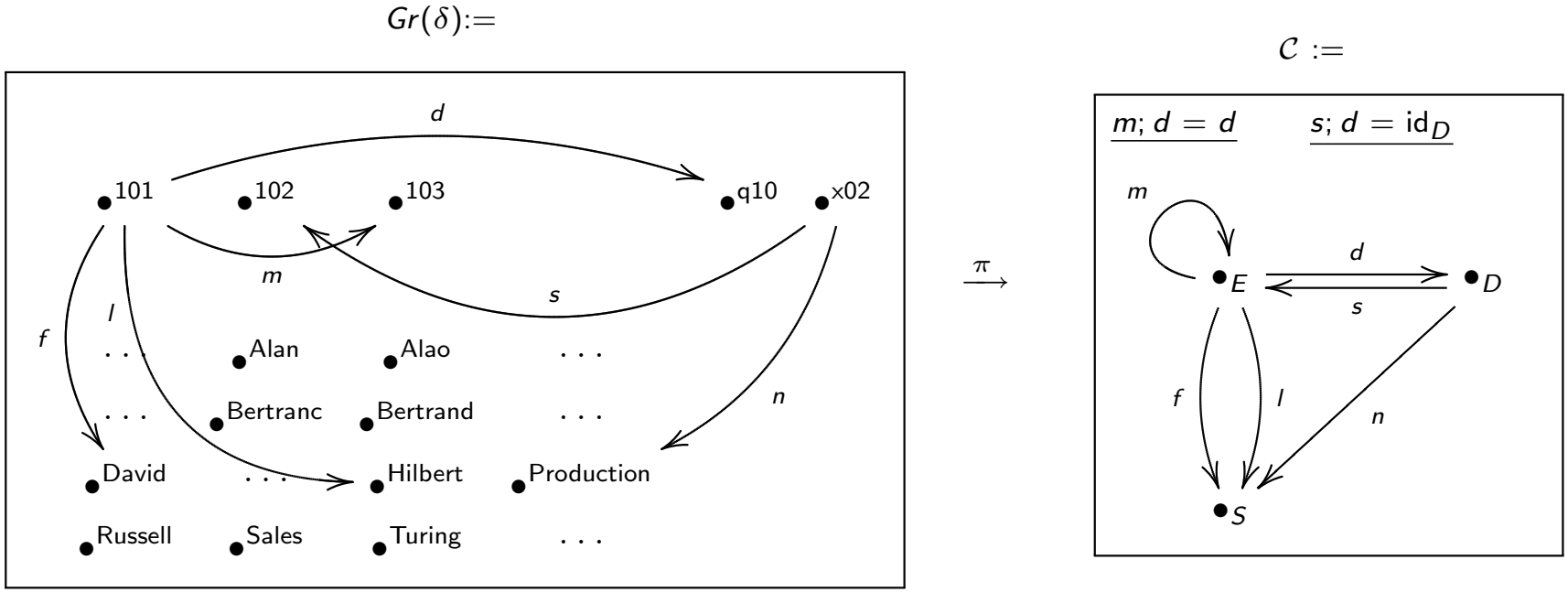
RDF schema and stores



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

Relaxing into the RDF perspective

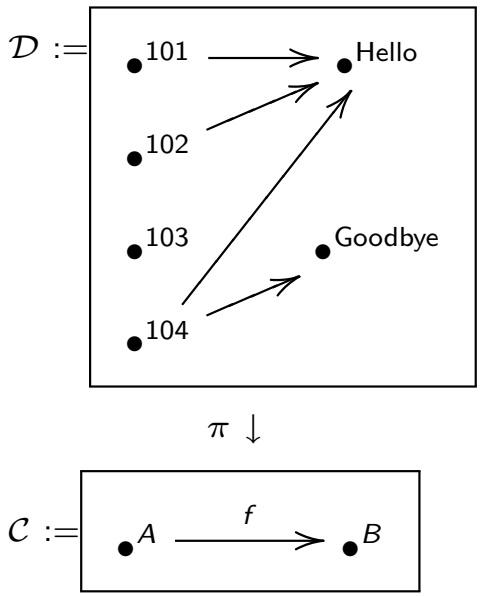
- If $\mathcal{C}$ is a schema, a functor $\delta\colon \mathcal{C} \to \mathbf{Set}$ may be too inflexible.
  - This is the well-known issue with relational databases.
- What properties does the associated functor $\pi\colon Gr(\delta) \to \mathcal{C}$ have?



- π is a "discrete op-fibration."
- What if we relax that requirement?

Allowing for semi-structured data

- We can think of any functor $\pi : \mathcal{D} \to \mathcal{C}$ as a "semi-state" on $\mathcal{C}$.
- Such a functor π can encode incomplete, non-atomic, or bad data.



- Row 103 has no data in the f cell, and row 104 has too much.
- Bad data (data not conforming to declared composition laws) can also occur in a functor $\pi : \mathcal{D} \to \mathcal{C}$.
- Any semi-state on $\mathcal{C}$ can be functorially "corrected" to a state if necessary.
- Conclusion: category theory flexibly deals with problems – the model gracefully extends.

States and migration

- Given a schema $\mathcal{C}$, the category of states on $\mathcal{C}$ is denoted $\mathcal{C}$–**Set**.
  - The objects of $\mathcal{C}$–**Set** are functors $\delta : \mathcal{C} \to$ **Set**.
  - The morphisms are natural transformations.
  - $\mathcal{C}$–**Set** is a topos; it has an internal language and logic supporting the typed lambda calculus.
- A morphism between schemas is a functor $F : \mathcal{C} \to \mathcal{D}$.
- Given such a morphism, we want to be able to canonically transfer states on $\mathcal{C}$ to states on $\mathcal{D}$ and vice versa.
- That means, given $F : \mathcal{C} \to \mathcal{D}$ we want functors

$$\mathcal{C}\text{–}\mathbf{Set} \to \mathcal{D}\text{–}\mathbf{Set}$$

and

$$\mathcal{D}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\mathbf{Set}.$$

The "easy" migration functor, Δ

- Given a morphism of schemas (i.e. a functor)

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

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

Given $\delta : \mathcal{D} \to \mathbf{Set}$ $\quad$ $\mathcal{C} \xrightarrow{F} \mathcal{D} \xrightarrow{\delta} \mathbf{Set}$, $F;\delta$ $\quad$ get $F;\delta : \mathcal{C} \to \mathbf{Set}$

- Thus we have a functor $\Delta_F : \mathcal{D}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\mathbf{Set}$.
- Δ_F basically operates by "re-indexing." Using it, one can duplicate or drop tables or columns.

The "harder" migration functors, Σ and Π

Given a morphism of schemas (i.e. a functor) $F : \mathcal{C} \to \mathcal{D}$,

- the functor $\Delta_F : \mathcal{D}$–**Set** $\to \mathcal{C}$–**Set** has two adjoints:
  - a left adjoint $\Sigma_F : \mathcal{C}$–**Set** $\to \mathcal{D}$–**Set**, and
  - a right adjoint $\Pi_F : \mathcal{C}$–**Set** $\to \mathcal{D}$–**Set**,

$$\mathcal{C}\text{–}\mathbf{Set} \underset{\Delta_F}{\overset{\Sigma_F}{\rightleftarrows}} \mathcal{D}\text{–}\mathbf{Set} \underset{\Pi_F}{\overset{\Delta_F}{\rightleftarrows}} \mathcal{C}\text{–}\mathbf{Set}$$

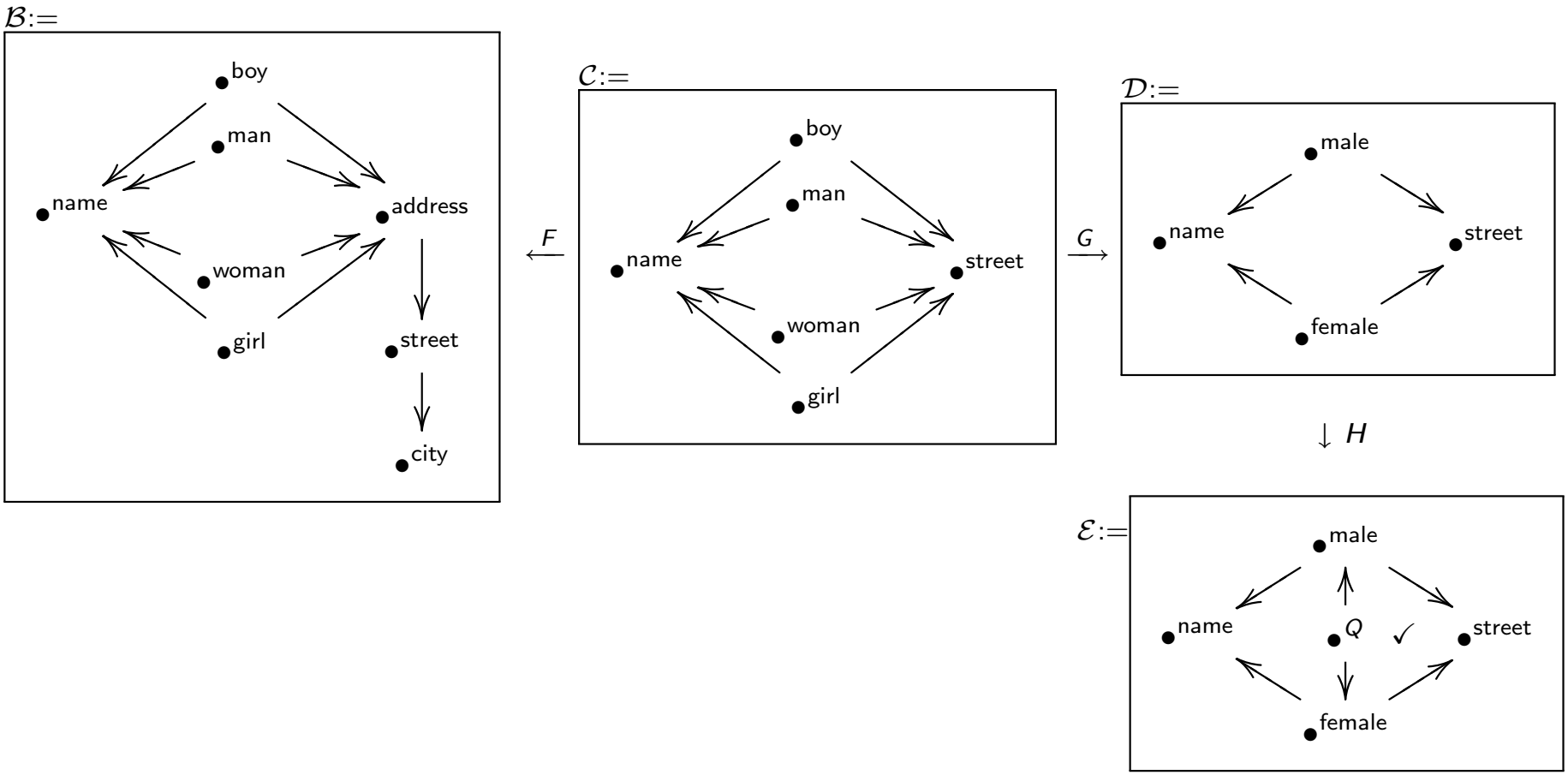
- Thus, given a morphism F of schemas, these three functors,

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

allow one to move data back and forth between $\mathcal{C}$ and $\mathcal{D}$ in canonical ways.

Views as polynomial functors

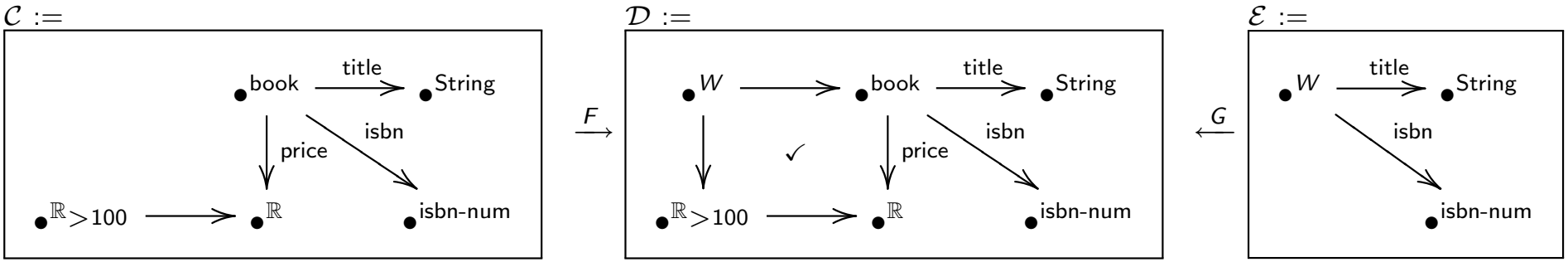
These functors can be arbitrarily composed to create views.



Given a state $\gamma\colon \mathcal{B} \to \mathbf{Set}$, what is $\Pi_H \circ \Sigma_G \circ \Delta_F(\gamma)\colon \mathcal{E} \to \mathbf{Set}$?

A simple "SELECT" query

SELECT title, isbn
FROM book
WHERE price > 100



$\Delta_G \circ \Pi_F$ is the appropriate view.
For any $\delta : \mathcal{C} \to \mathbf{Set}$, we materialize the view as $\Delta_G \circ \Pi_F(\delta)$.

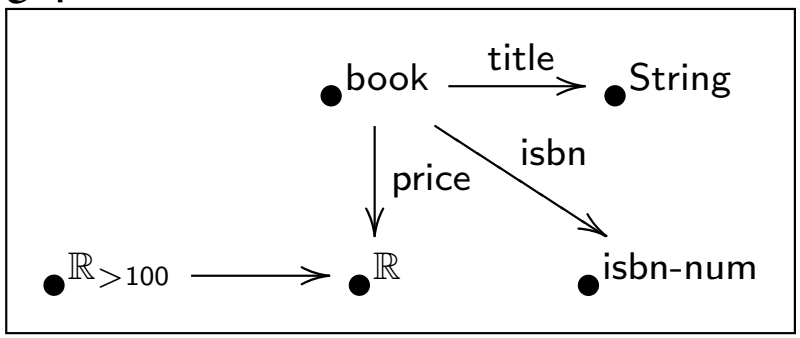
Interfacing between schemas

- We are often interested in taking data from one enterprise model $\mathcal{C}$ and transferring it to another enterprise model $\mathcal{D}$.
- Such transfers can also be accomplished using polynomial functors.
- Functors $\mathcal{C} \to \mathcal{D}$ allow us not only to transform data but also translate queries.
- Also, we might need to perform calculations such as concatenation, addition, comparison, conversion of units, etc. in order to interface these schemas.
- For this we'll need an underlying typing category.

Incorporating data types and functions

- In the example:

$\mathcal{C} :=$



  how do we know that $\bullet^{\text{String}}$, $\bullet^{\mathbb{R}}$, and $\bullet^{\mathbb{R}_{>100}}$ are what they say they are?

- That is, given $\delta : \mathcal{C} \to \mathbf{Set}$, how do we specify that $\delta(\bullet^{\mathbb{R}}) \in \mathbf{Set}$ is some pre-defined data type like Float.

Power of category theory: connection is easy

- Let **Hask** denote a category of types and functions that has been implemented on a computer and for which (at least theoretically) there exists a functor V : **Hask** $\longrightarrow$ **Set**.
  - Think of **Hask** as all Haskell data types and the definable functions between them, as well as all new types that could possibly be output by modules.
- Now **Hask** begins to look like a schema and V a "canonical state."
- Since database schemas are categories and **Hask** is a category, we can integrate the two.

Example

- Lets make a category $\mathcal{B}$ and a functor F to take just what we need from **Hask** to make sense of $\mathcal{C}$.

$$\mathcal{B} := \boxed{\begin{array}{ccc} & & \bullet^{\text{String}} \\ \bullet^{\mathbb{R}_{>100}} & \longrightarrow & \bullet^{\mathbb{R}} \end{array}} \xrightarrow{F} \textbf{Hask}$$

$F(\bullet^{\text{String}}) = \text{char}(40)$, $F(\bullet^{\mathbb{R}}) = \text{Float}$, $F(\bullet^{\mathbb{R}_{>100}}) = $ some new type.

- There is also an obvious functor

$$\mathcal{B} = \boxed{\begin{array}{ccc} & & \bullet^{\text{String}} \\ \bullet^{\mathbb{R}_{>100}} & \longrightarrow & \bullet^{\mathbb{R}} \end{array}} \xrightarrow{G} \boxed{\begin{array}{ccccc} \bullet^{\text{book}} & \xrightarrow{\text{title}} & \bullet^{\text{String}} & & \\ \downarrow{\scriptstyle \text{price}} & \searrow{\scriptstyle \text{isbn}} & & & \\ \bullet^{\mathbb{R}_{>100}} & \longrightarrow & \bullet^{\mathbb{R}} & & \bullet^{\text{isbn-num}} \end{array}} = \mathcal{C}$$

- We are interested in functors $\delta : \mathcal{C} \to \textbf{Set}$ equipped with a map $\Delta_G \delta \to \Delta_F V$.

Typing in general

- If we need to enforce data types, our schema $\mathcal{C}$ will more than just a category,
- It will be a category including a fragment $\mathcal{B}$ of **Hask**:

$$\textbf{Hask} \xleftarrow{F} \mathcal{B} \xrightarrow{G} \mathcal{C}.$$

- And we won't interested in any old state $\delta : \mathcal{C} \to \textbf{Set}$ but only those with a map $\Delta_G \delta \to \Delta_F V$, where $V : \textbf{Hask} \to \textbf{Set}$ is as above.
- By definition of adjunction, that's just

$$\delta \to \Pi_G \Delta_F(V),$$

and $\Pi_G \Delta_F(V)$ is some huge fixed state on $\mathcal{C}$ that encodes our typing requirements.

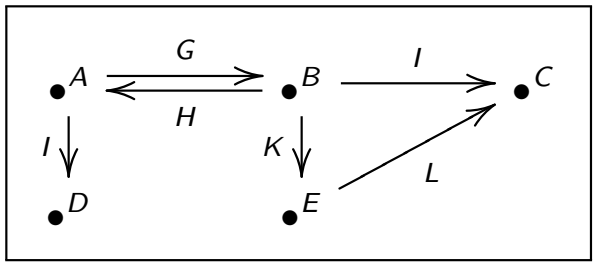
- So the category of typed states is $\mathcal{C}$–$\textbf{Set}_{/\Pi_G \Delta_F(V)}$. This is a topos.

Parallelizing or federating data

- Given $F\colon \mathcal{C} \to \mathcal{D}$, we can import data from $\mathcal{D}$ to $\mathcal{C}$ via Δ_F.
- Given $\epsilon\colon \mathcal{D} \to \mathbf{Set}$ and $\delta\colon \mathcal{C} \to \mathbf{Set}$, we can hold $\mathcal{D}$'s data with $\mathcal{C}$'s,

$$\delta \amalg \Delta_F(\epsilon) \qquad \in \mathcal{C}\text{–}\mathbf{Set}.$$

- But we'll always know which data came from where.
- Works just as well with a network of databases.



- We have a bunch of databases with some overlapping ideas (understood using functors), and data can be traded around.

Provenance: a future history

- Around Y2.01K, future histories of provenance were all the rage.
- In a 2010 talk by David Spivak, who knew next to nothing about provenance, he claimed the future history to be thus:
  - It was realized that there would never be one world-view.
  - Everyone had their own, and these were not necessarily converging.
  - But people and businesses did save energy through trading information.
  - Thus, people and enterprises coalesced into a network of databases.
  - Each node was someone's schema and each edge was a functor.
  - Now, data could be shared between people and businesses via migration functors, and it was always clear where that data originated.
  - The entire structure (network + "sheaf of data" = cloud) captured much of data provenance.

Going further...

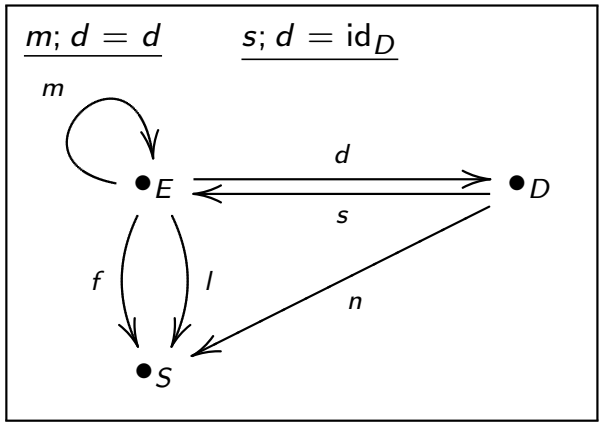
- If each node stores its own data, we'd always know where it comes from.
- But there is more to provenance than this.
- For one thing, we not only want to know where data comes from, but when it was created and how it was calculated.
  - Scientists should be able to "zoom in" on each other's findings.
  - The "data reduction process" of synthesizing large amounts of data must be recorded in the schema so that others can understand.
  - This may be doable with the above category-theoretic approach, but it must studied in earnest.
- The point is, the above model of "data migration functors" is precise, formal, and repeatable.
- One hopes that the functoriality leads to a natural querying interface on the whole.

Summary

- 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



- I discussed how one can use this connection to facilitate:
  - formalizing views as polynomial functors;
  - merging database and PL theory;
  - merging relational and RDF outlooks (NoSQL);
  - formalizing federation and provenance.
- The main point is that this is a self-contained, unified approach.
- With a clear formalism of databases, one might be able to apply proof assistants in exciting new ways (e.g. as query-optimizers).

Thanks for the invitation to speak!