Categorical Databases

Patrick Schultz, David Spivak
MIT

Ryan Wisnesky
Categorical Informatics

and others

October 2017
This talk describes a new algebraic (purely equational) way to formalize databases based on category theory.

Category theory was designed to migrate theorems from one area of mathematics to another, but researchers at MIT developed a way to use it to migrate data from one schema to another.

Research has culminated in an open-source prototype ETL and data integration tool, AQL (Algebraic Query Language), available at categoricaldata.net/aql.html. (These slides are also there.)

Goal: Categorical databases needs you – needs a community – to grow. Please, talk to us and to each other and get involved.

Outline:

- Review of basic category theory.
- Introduction to AQL.
- AQL demo.
- Optional interlude: additional AQL constructions.
- How AQL instances model the simply-typed λ-calculus.
A category $C$ consists of
- a set of objects, $\text{Ob}(C)$
- for all $X, Y \in \text{Ob}(C)$, a set $\mathcal{C}(X, Y)$ of morphisms a.k.a arrows
- for all $X \in \text{Ob}(C)$, a morphism $\text{id} \in \mathcal{C}(X, X)$
- for all $X, Y, Z \in \text{Ob}(C)$, a function $\circ : \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) \to \mathcal{C}(X, Z)$ s.t.

$$f \circ \text{id} = f \quad \text{id} \circ f = f \quad (f \circ g) \circ h = f \circ (g \circ h)$$

The category $\text{Set}$ has sets as objects and functions as arrows, and the “category” Haskell has types as objects and programs as arrows.

A functor $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}, \mathcal{D}$ consists of
- a function $\text{Ob}(\mathcal{C}) \to \text{Ob}(\mathcal{D})$
- for all $X, Y \in \text{Ob}(\mathcal{C})$, a function $\mathcal{C}(X, Y) \to \mathcal{D}(F(X), F(Y))$ s.t.

$$F(\text{id}) = \text{id} \quad F(f \circ g) = F(f) \circ F(g)$$

The functor $\mathcal{P} : \text{Set} \to \text{Set}$ takes each set to its power set, and the functor $\text{List} : \text{Haskell} \to \text{Haskell}$ takes each type $t$ to the type $\text{List } t$. 
Schemas and Instances

(manager.works) = (works)  (secretary.works) = []

<table>
<thead>
<tr>
<th>Emp</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>mgr</td>
<td>works</td>
<td>first</td>
<td>last</td>
</tr>
<tr>
<td>101</td>
<td>103</td>
<td>q10</td>
<td>Al</td>
<td>Akin</td>
</tr>
<tr>
<td>102</td>
<td>102</td>
<td>x02</td>
<td>Bob</td>
<td>Bo</td>
</tr>
<tr>
<td>103</td>
<td>103</td>
<td>q10</td>
<td>Carl</td>
<td>Cork</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>sec</td>
<td>name</td>
</tr>
<tr>
<td>q10</td>
<td>101</td>
<td>CS</td>
</tr>
<tr>
<td>x02</td>
<td>102</td>
<td>Math</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>Al</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
An AQL Schema: Code

entities
    Emp
    Dept

foreign keys
    manager : Emp -> Emp
    works : Emp -> Dept
    secretary : Dept -> Emp

attributes
    first last : Emp -> string
    name : Dept -> string

path equations
    manager.works = works
    secretary.works = Department
Categorical Semantics of Schemas and Instances

- The meaning of a schema $S$ is a category $\mathbb{[S]}$.
  - $\text{Ob}(\mathbb{[S]})$ is the nodes of $S$.
  - For all nodes $X, Y$, $\mathbb{[S]}(X, Y)$ is the set of finite paths $X \to Y$, modulo the path equivalences in $S$.
  - Path equivalence in $S$ may not be decidable! ("the word problem")
- A morphism of schemas (a "schema mapping") $S \to T$ is a functor $\mathbb{[S]} \to \mathbb{[T]}$.
  - It can be defined as an equation-preserving function:
    $$\text{nodes}(S) \to \text{nodes}(T) \quad \text{edges}(S) \to \text{paths}(T).$$
- An $S$-instance is a functor $\mathbb{[S]} \to \text{Set}$.
  - It can be defined as a set of tables, one per node in $S$ and one column per edge in $S$, satisfying the path equivalences in $S$.
- A morphism of $S$-instances $I \to J$ (a "data mapping") is a natural transformation $I \to J$.
  - Instances on $S$ and their mappings form a category, written $S$-inst.
Schema Mappings

A schema mapping \( F : S \rightarrow T \) is an equation-preserving function:

\[
\text{nodes}(S) \rightarrow \text{nodes}(T) \quad \text{edges}(S) \rightarrow \text{paths}(T)
\]

\[
F(\text{Int}) = \text{Int} \quad F(\text{String}) = \text{String} \\
F(\text{N1}) = \text{N} \quad F(\text{N2}) = \text{N} \\
F(\text{name}) = [\text{name}] \quad F(\text{age}) = [\text{age}] \quad F(\text{salary}) = [\text{salary}] \\
F(f) = []
\]
Functorial Data Migration

A schema mapping $F: S \to T$ induces three data migration functors:

- $\Delta_F: T\text{-inst} \to S\text{-inst}$ (like project)

\[
\begin{array}{ccc}
S & \xrightarrow{F} & T \\
\downarrow{\Delta_F(I)} & & \downarrow{I} \\
\text{Set} & & \text{Set}
\end{array}
\]

\[\Delta_F(I) := I \circ F\]

- $\Pi_F: S\text{-inst} \to T\text{-inst}$ (right adjoint to $\Delta_F$; like join)

\[\forall I, J. \ S\text{-inst}(\Delta_F(I), J) \cong T\text{-inst}(I, \Pi_F(J))\]

- $\Sigma_F: S\text{-inst} \to T\text{-inst}$ (left adjoint to $\Delta_F$; like outer union then merge)

\[\forall I, J. \ S\text{-inst}(J, \Delta_F(I)) \cong T\text{-inst}(\Sigma_F(J), I)\]
\[ \Delta \text{ (Project)} \]

\[ \begin{array}{|c|c|c|} \hline \text{ID} & \text{name} & \text{salary} \\ \hline 1 & Alice & $100 \\ 2 & Bob & $250 \\ 3 & Sue & $300 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline \text{ID} & \text{age} \\ \hline 4 & 20 \\ 5 & 20 \\ 6 & 30 \\ \hline \end{array} \quad \begin{array}{|c|c|c|c|} \hline \text{ID} & \text{name} & \text{salary} & \text{age} \\ \hline a & Alice & $100 & 20 \\ b & Bob & $250 & 20 \\ c & Sue & $300 & 30 \\ \hline \end{array} \]
Π (Product)

\[ N \]

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>Alice</td>
<td>$100</td>
<td>30</td>
</tr>
<tr>
<td>d</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>e</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>f</td>
<td>Bob</td>
<td>$250</td>
<td>30</td>
</tr>
<tr>
<td>g</td>
<td>Sue</td>
<td>$300</td>
<td>20</td>
</tr>
<tr>
<td>h</td>
<td>Sue</td>
<td>$300</td>
<td>20</td>
</tr>
<tr>
<td>i</td>
<td>Sue</td>
<td>$300</td>
<td>30</td>
</tr>
</tbody>
</table>

N1

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
</tr>
</tbody>
</table>

N2

\[ F \]

\[ \Pi F \]
Σ (Outer Union)

\[\Sigma\] (Outer Union)

\[
\begin{array}{c}
\text{N1} \\
\begin{array}{c|c|c}
\text{ID} & \text{Name} & \text{Salary} \\
1 & Alice & $100 \\
2 & Bob & $250 \\
3 & Sue & $300 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{N2} \\
\begin{array}{c|c|c}
\text{ID} & \text{Age} \\
4 & 20 \\
5 & 20 \\
6 & 30 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{N} \\
\begin{array}{c|c|c|c}
\text{ID} & \text{Name} & \text{Salary} & \text{Age} \\
\text{a} & Alice & $100 & \text{null}_1 \\
\text{b} & Bob & $250 & \text{null}_2 \\
\text{c} & Sue & $300 & \text{null}_3 \\
\text{d} & \text{null}_4 & \text{null}_5 & 20 \\
\text{e} & \text{null}_6 & \text{null}_7 & 20 \\
\text{f} & \text{null}_8 & \text{null}_9 & 30 \\
\end{array}
\end{array}
\]
Unit of $\Sigma_F \rightarrow \Delta_F$

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
</tr>
</tbody>
</table>

$\Sigma_F$

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>a</td>
<td>Alice</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
</tr>
<tr>
<td>d</td>
<td>null4</td>
</tr>
<tr>
<td>e</td>
<td>null6</td>
</tr>
<tr>
<td>f</td>
<td>null8</td>
</tr>
</tbody>
</table>

$\Delta_F$

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>a</td>
<td>Alice</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
</tr>
<tr>
<td>d</td>
<td>null4</td>
</tr>
<tr>
<td>e</td>
<td>null6</td>
</tr>
<tr>
<td>f</td>
<td>null8</td>
</tr>
</tbody>
</table>
A Foreign Key

### N1

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td>6</td>
</tr>
</tbody>
</table>

### N2

<table>
<thead>
<tr>
<th>ID</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

### N

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
<td>30</td>
</tr>
</tbody>
</table>
Queries

A query $Q : S \rightarrow T$ is a schema $X$ and mappings $F : S \rightarrow X$ and $G : T \rightarrow X$.

$$eval_Q \equiv \Delta_G \circ \Pi_F \quad coeval_Q \equiv \Delta_F \circ \Pi_G$$

These can be specified using comprehension notation similar to SQL.

N1 -> select n1.name as name, n1.salary as salary
    from N as n1

N2 -> select n2.age as age
    from N as n2

f -> {n2 -> n1}
A Foreign Key

\[ Q \]

### N1

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td>6</td>
</tr>
</tbody>
</table>

### N2

\[ eval_Q \]

<table>
<thead>
<tr>
<th>ID</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

\[ coeval_Q \]

### N

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
<td>30</td>
</tr>
</tbody>
</table>
• AQL implements $\Delta, \Sigma, \Pi$, and more in software.
  • catinf.com
• The AQL “execution engine” is an automated theorem prover.
  • Value proposition: AQL catches mistakes at compile time that existing ETL / data integration tools catch at runtime – if at all.
  • Data import and export by JDBC-SQL and CSV.
• We are looking for collaborators for a “real-world pilot project”.
Interlude - Additional Constructions

- What is “algebraic” here?
- AQL vs SQL.
- Pivot.
- Non-equational data integrity constraints.
- Data integration via pushouts.
- AQL vs comprehension calculi.
Why “Algebraic”? 

- A schema can be identified with an algebraic (equational) theory.
  
  \[
  \begin{align*}
  \text{Emp} & : \text{Type} \\
  \text{first} & : \text{Emp} \to \text{String} \\
  \text{last} & : \text{Emp} \to \text{String} \\
  \text{name} & : \text{Dept} \to \text{String} \\
  \text{works} & : \text{Emp} \to \text{Dept} \\
  \text{mgr} & : \text{Emp} \to \text{Emp} \\
  \text{secr} & : \text{Dept} \to \text{Emp} \\
  \forall e : \text{Emp}. \text{works}(\text{manager}(e)) = \text{works}(e) \\
  \forall d : \text{Dept}. \text{works}(\text{secretary}(d)) = d
  \end{align*}
  \]

- This perspective makes it easy to add functions such as \(+ : \text{Int}, \text{Int} \to \text{Int}\) to a schema. See *Algebraic Databases*.

- An \(S\)-instance can be identified with the initial algebra of an algebraic theory extending \(S\).
  
  \[
  \begin{align*}
  101 & \quad 102 \quad 103 : \text{Emp} \\
  q10 & \quad x02 : \text{Dept}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{mgr}(101) & = 103 \\
  \text{works}(101) & = q10
  \end{align*}
  \]

- Treating instances as theories allows instances that are infinite or inconsistent (e.g., Alice=Bob).
AQL vs SQL

- Data migration triplets of the form

\[ \Sigma F \circ \Pi G \circ \Delta H \]

can be expressed using relational algebra and keygen, provided:
- \( F \) is a discrete op-fibration (ensures union compatibility).
- \( G \) is surjective on attributes (ensures domain independence).
- All categories are finite (ensures computability).

- The difference-free fragment of relational algebra can be expressed using such triplets. See *Relational Foundations*.

- Such triplets can be written in “foreign-key aware” SQL-ish syntax.
Select-From-Where Syntax

Find the name of every manager’s department:

AQL
select e.manager.works.name
from Emp as e

SQL
select d.name
from Emp as e1, Emp as e2, Dept as d
where e1.manager = e2.ID and
e2.works = d.ID
Pivot (Instance ↔ Schema)

Emp

<table>
<thead>
<tr>
<th>ID</th>
<th>mgr</th>
<th>works</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>103</td>
<td>q10</td>
<td>Al</td>
<td>Akin</td>
</tr>
<tr>
<td>102</td>
<td>102</td>
<td>x02</td>
<td>Bob</td>
<td>Bo</td>
</tr>
<tr>
<td>103</td>
<td>103</td>
<td>q10</td>
<td>Carl</td>
<td>Cork</td>
</tr>
</tbody>
</table>

Dept

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>q10</td>
<td>CS</td>
</tr>
<tr>
<td>x02</td>
<td>Math</td>
</tr>
</tbody>
</table>
Richer Constraints

- Not all data integrity constraints are equational (e.g., keys).
- A data mapping $\varphi : A \rightarrow E$ defines a constraint: instance $I$ satisfies $\varphi$ if for every $\alpha : A \rightarrow I$ there exists an $\epsilon : E \rightarrow I$ s.t $\alpha = \epsilon \circ \varphi$.

$$
\begin{array}{c}
A \xrightarrow{\alpha} I \\
\varphi \downarrow \\
E
\end{array}
$$

- Most constraints used in practice can be captured the above way. E.g.,

$$
\forall d_1, d_2 : \text{Dept. } \text{name}(d_1) = \text{name}(d_2) \rightarrow d_1 = d_2
$$

is captured as

$$
A(\text{Dept}) = \{d_1, d_2\} \quad A(\text{name})(d_1) = A(\text{name})(d_2)
$$

$$
E(\text{Dept}) = \{d\} \quad \varphi(d_1) = \varphi(d_2) = d
$$

- See Database Queries and Constraints via Lifting Problems and Algebraic Model Management.
Pushouts

- A pushout of \( p, q \) is \( f, g \) s.t. for every \( f', g' \) there is a unique \( m \) s.t.:

\[
\begin{array}{ccc}
p & \rightarrow & q \\
\downarrow \quad & & \downarrow \\
\bullet & \rightarrow & \bullet
\end{array}
\]

\[
\begin{array}{ccc}
f & \rightarrow & g \\
\downarrow \quad & & \downarrow \\
\bullet & \rightarrow & \bullet
\end{array}
\]

\[
\begin{array}{ccc}
f' & \rightarrow & g' \\
\downarrow \quad & & \downarrow \\
\bullet & \rightarrow & \bullet
\end{array}
\]

\[
\begin{array}{ccc}
m & \downarrow \\
\downarrow \quad & & \downarrow \\
\bullet & \rightarrow & \bullet
\end{array}
\]

- The category of schemas has all pushouts.
- For every schema \( S \), the category \( S\text{-inst} \) has all pushouts.
- Pushouts of schemas, instances, and \( \Sigma \) are used together to integrate data - see *Algebraic Data Integration*.
Using Pushouts for Data Integration

- Step 1: integrate schemas. Given input schemas $S_1$, $S_2$, an overlap schema $S$, and mappings $F_1, F_2$:

$$S_1 \xleftarrow{F_1} S \xrightarrow{F_2} S_2$$

we propose to use their pushout $T$ as the integrated schema:

$$S_1 \xrightarrow{G_1} T \xleftarrow{G_2} S_2$$

- Step 2: integrate data. Given input $S_1$-instance $I_1$, $S_2$-instance $I_2$, overlap $S$-instance $I$ and data mappings $h_1 : \Sigma_{F_1}(I) \rightarrow I_1$ and $h_2 : \Sigma_{F_2}(I) \rightarrow I_2$, we propose to use the pushout of:

$$\Sigma_{G_1}(I_1) \xleftarrow{(\Sigma_{G_1 \circ F_1}(I) = \Sigma_{G_2 \circ F_2}(I))} \Sigma_{G_2}(I_2)$$

as the integrated $T$-instance.
Schema Integration

```
Person \rightarrow \text{Gender} \rightarrow \text{Type} \rightarrow \text{Observation} \rightarrow \text{Method}
```

```
Person \rightarrow \text{Gender} \rightarrow \text{Type} \rightarrow \text{Observation} \rightarrow \text{Method}
```
Data Integration

Observation ID: f, g
Person ID: p
Type ID: BP, Wt

Gender ID: F, M
Type ID: BP, Wt, HR

Method ID: g2
Type ID: BP, Wt

Observation ID: f, g
Person ID: h
Type ID: BP, Wt, HR

Method ID: g2
Type ID: BP, Wt

Observation ID: f, g1
Person ID: h
Type ID: BP, Wt, HR

Gender ID: F, M
Type ID: BP, Wt, HR

Person ID: h
Type ID: F, M

Method ID: g2
Type ID: BP, Wt

Observation ID: f, g1
Person ID: h
Type ID: BP, Wt, HR

Gender ID: F, M
Type ID: BP, Wt, HR

Person ID: h
Type ID: F, M

null4
AQL vs LINQ

- Treating entity sets as types rather than terms makes AQL a conceptual dual to comprehension calculi (e.g., LINQ). See QINL: Query-Integrated Languages.

- LINQ enriches programs with (schemas, queries and instances).
  - Collections are terms
    
    \[
    \begin{align*}
    \text{Employee} & : \text{Set Int} \\
    \text{manager} & : \text{Set (Int} \times \text{Int)}
    \end{align*}
    \]
  - e: Employee is not a judgment.
  - There is a term \( \in : \text{Int} \times \text{Set Int} \rightarrow \text{Bool} \).

- AQL enriches (schemas, queries and instances) with programs.
  - Collections are types
    
    \[
    \begin{align*}
    \text{Employee} & : \text{Type} \\
    \text{manager} & : \text{Employee} \rightarrow \text{Employee}
    \end{align*}
    \]
  - e: Employee is a judgment.
  - There is not a term \( \in : \text{Employee} \times \text{Type} \rightarrow \text{Bool} \).

- LINQ is more popular, but AQL’s style is common in Coq, Agda, etc.
AQL is “one level up” from LINQ

- **LINQ**
  - Schemas are collection types over a base type theory
    \[
    \text{Set } (\text{Int} \times \text{String})
    \]
  - Instances are terms
    \[
    \{(1, \text{CS})\} \cup \{(2, \text{Math})\}
    \]
  - Data migrations are functions
    \[
    \pi_1 : \text{Set } (\text{Int} \times \text{String}) \to \text{Set Int}
    \]

- **AQL**
  - Schemas are type theories over a base type theory
    \[
    \text{Dept, name: Dept} \to \text{String}
    \]
  - Instances are term models (initial algebras) of theories
    \[
    d_1, d_2 : \text{Dept, name}(d_1) = \text{CS}, \text{name}(d_2) = \text{Math}
    \]
  - Data migrations are functors
    \[
    \Delta_{\text{Dept}} : (\text{Dept, name: Dept} \to \text{String})\text{-inst} \to (\text{Dept})\text{-inst}
    \]
For every schema $S$, $S$-inst models simply-typed $\lambda$-calculus (STLC).
The STLC is the core of typed functional languages ML, Haskell, etc.
We will use the internal language of a cartesian closed category, which is equivalent to the STLC.
Lots of “point-free” functional programming ahead.
The category of schemas and mappings is also cartesian closed - see talk at Boston Haskell.
Categorical Abstract Machine Language (CAML)

- **Types** $t$:
  
  $$ t ::= 1 \mid t \times t \mid t^t $$

- **Terms** $f, g$:

  $$
  \begin{align*}
  id_t : t & \to t \\
  ()_t : t & \to 1 \\
  \pi^1_{s, t} : s \times t & \to s \\
  \pi^2_{s, t} : s \times t & \to t \\
  eval_{s, t} : t^s \times s & \to t \\
  f : s & \to u \\
  g : u & \to t \\
  f : s & \to t \\
  g : s & \to u \\
  g \circ f : s & \to t \\
  (f, g) : s & \to t \times u \\
  f : s \times u & \to t \\
  \lambda f : s & \to t^u
  \end{align*}
  $$

- **Equations**:

  $$
  \begin{align*}
  id \circ f & = f \\
  f \circ id & = f \\
  f \circ (g \circ h) & = (f \circ g) \circ h \\
  () \circ f & = () \\
  \pi^1 \circ (f, g) & = f \\
  \pi^2 \circ (f, g) & = g \\
  (\pi^1 \circ f, \pi^2 \circ f) & = f \\
  eval \circ (\lambda f \circ \pi^1, \pi^2) & = f \\
  \lambda (eval \circ (f \circ \pi^1, \pi^2)) & = f
  \end{align*}
  $$
For every schema $S$, the category $S$-inst is cartesian closed.
- Given a type $t$, you get an $S$-instance $[t]$.
- Given a term $f : t \rightarrow t'$, you get a data mapping $[f] : [t] \rightarrow [t']$.
- All equations obeyed.

$S$-inst is further a topos (model of higher-order logic / set theory).

We consider the following schema in the examples that follow:
Programming AQL in CAML: Unit

- The unit instance 1 has one row per table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>f</td>
<td>ID</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

- The data mapping $()_t : t \rightarrow 1$ sends every row in $t$ to the only row in 1. For example,

\[
t = \begin{array}{ccc}
\hline
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>f</td>
<td>ID</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>r</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>
\hline
\end{array}
\quad \xrightarrow{()_t} \quad \begin{array}{ccc}
\hline
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>f</td>
<td>ID</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
\hline
\end{array}
\]

\[p, q, r, t \xrightarrow{()_t} x\]
Products $s \times t$ are computed row-by-row, with evident projections $\pi^1 : s \times t \to s$ and $\pi^2 : s \times t \to t$. For example:

\begin{align*}
\begin{array}{|c|c|}
\hline
a & f \\
\hline
1 & 3 \\
2 & 3 \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
b & ID \\
\hline
3 & 4 \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
a & ID \\
\hline
1 & a \\
(1,b) & b \\
(2,a) & c \\
(2,b) & c \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
b & ID \\
\hline
2 & c \\
3 & d \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
a & f \\
\hline
(1,a) & 3 \\
(1,b) & 3 \\
(2,a) & 3 \\
(2,b) & 3 \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
b & ID \\
\hline
3 & 4 \\
3 & d \\
4 & c \\
4 & d \\
\hline
\end{array}
\end{align*}

Given data mappings $f : s \to t$ and $g : s \to u$, how to define $(f, g) : s \to t \times u$ is left to the reader.

- hint: try it on $\pi^1$ and $\pi^2$ and verify that $(\pi^1, \pi^2) = id$. 
Exponentials $t^s$ are given by finding all data mappings $s \rightarrow t$:

- Defining `eval` and `\lambda` are left to the reader.
Concusion

- We described a new “algebraic” approach to databases based on category theory.
  - Schemas are categories, instances are set-valued functors.
  - Three adjoint data migration functors, $\Sigma, \Delta, \Pi$ manipulate data.
  - Instances on a schema model the simply-typed $\lambda$-calculus.
- Our approach is implemented in AQL, an open-source project, available at catinf.com.
- Collaborators welcome!
  - We are looking for a “real-world pilot project”.