Categorical Databases

Patrick Schultz, David Spivak
MIT

Ryan Wisnesky
Categorical Informatics

and others

November 2017
Introduction

- 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.
- 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.
AQL Value Proposition

- AQL implements this talk in software.
  - catinf.com

- The AQL “execution engine” is an automated theorem prover.
  - High-assurance: 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 “real-world pilot projects”.
A category $C$ consists of
- a set of objects, $\text{Ob}(C)$
- for all $X, Y \in \text{Ob}(C)$, a set $C(X, Y)$ of morphisms a.k.a arrows
- for all $X \in \text{Ob}(C)$, a morphism $id \in C(X, X)$
- for all $X, Y, Z \in \text{Ob}(C)$, a function $\circ : C(Y, Z) \times C(X, Y) \to C(X, Z)$ s.t.
  \[
  f \circ id = f \quad id \circ f = f \quad (f \circ g) \circ h = f \circ (g \circ h)
  \]

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 $F : C \to D$ between categories $C, D$ consists of
- a function $\text{Ob}(C) \to \text{Ob}(D)$
- for all $X, Y \in \text{Ob}(C)$, a function $C(X, Y) \to D(F(X), F(Y))$ s.t.
  \[
  F(id) = id 
  F(f \circ g) = F(f) \circ F(g)
  \]

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] = []
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 $\lbrack S \rbrack$.
  - $\text{Ob}(\lbrack S \rbrack)$ is the nodes of $S$.
  - For all nodes $X, Y$, $\lbrack S \rbrack(X, Y)$ 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 $\lbrack S \rbrack \to \lbrack T \rbrack$.
  - 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 $\lbrack S \rbrack \to \text{Set}$.
  - 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 $F: S \rightarrow T$ induces three data migration functors:

- $\Delta_F: T\text{-inst} \rightarrow S\text{-inst}$ (like project)
  
  $$
  S \xrightarrow{F} T \xrightarrow{I} \text{Set} \\
  \Delta_F(I) := I \circ F
  $$

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

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

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

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

\[ F \]

\[ \Delta F \]

\begin{tabular}{|c|c|c|}
\hline
ID & name & salary  \\
\hline
1  & Alice & $100 \ \\
2  & Bob    & $250 \ \\
3  & Sue    & $300 \ \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
ID & age  \\
\hline
4  & 20   \ \\
5  & 20   \ \\
6  & 30   \ \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|}
\hline
ID & name & salary & age  \\
\hline
a  & Alice & $100    & 20   \ \\
b  & Bob    & $250    & 20   \ \\
c  & Sue    & $300    & 30   \ \\
\hline
\end{tabular}
Π (Product)

\[ \Pi (\text{Product}) \]

\[ \begin{array}{ccc}
\text{N1} & \text{N2} & \Pi_{F}
\end{array} \]

\[ \begin{array}{|c|c|c|}
\hline
\text{ID} & \text{name} & \text{salary} \\
\hline
1 & Alice & $100 \\
2 & Bob & $250 \\
3 & Sue & $300 \\
\hline
\end{array} \]

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

\[ \begin{array}{|c|c|c|c|}
\hline
\text{ID} & \text{name} & \text{salary} & \text{age} \\
\hline
a & Alice & $100 & 20 \\
b & Alice & $100 & 20 \\
c & Alice & $100 & 30 \\
d & Bob & $250 & 20 \\
e & Bob & $250 & 20 \\
f & Bob & $250 & 30 \\
g & Sue & $300 & 20 \\
h & Sue & $300 & 20 \\
i & Sue & $300 & 30 \\
\hline
\end{array} \]
\( \Sigma \) (Outer Union)

\[ \begin{align*}
\Sigma & \quad F \\
\end{align*} \]

\begin{tabular}{|c|c|c|}
\hline
\textbf{N1} & \textbf{ID} & \textbf{Name} \\
\hline
1 & Alice & $100 \\
2 & Bob & $250 \\
3 & Sue & $300 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\textbf{ID} & \textbf{Age} \\
\hline
4 & 20 \\
5 & 20 \\
6 & 30 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{N2} & \textbf{ID} & \textbf{Name} & \textbf{Age} \\
\hline
a & Alice & $100 & null_1 \\
b & Bob & $250 & null_2 \\
c & Sue & $300 & null_3 \\
d & null_4 & null_5 & 20 \\
e & null_6 & null_7 & 20 \\
f & null_8 & null_9 & 30 \\
\hline
\end{tabular}
Unit of $\Sigma_F \rightarrow \Delta_F$

<table>
<thead>
<tr>
<th>N1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Name</td>
<td>Salary</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Age</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\Sigma_F \rightarrow \eta \rightarrow \Delta_F$

<table>
<thead>
<tr>
<th>N1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Name</td>
<td>Salary</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>null_4</td>
<td>null_5</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>null_6</td>
<td>null_7</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>null_8</td>
<td>null_9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Age</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>null_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>null_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>null_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Foreign Key

\[ F \]

\[ \Delta_F \]

\[ \Pi_F, \Sigma_F \]

<table>
<thead>
<tr>
<th>N1</th>
<th></th>
<th>N2</th>
<th></th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
<td>salary</td>
<td>f</td>
<td>ID</td>
</tr>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Queries

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

$$\text{eval}_Q \cong \Delta_G \circ \Pi_F \quad \text{coeval}_Q \cong \Delta_F \circ \Pi_G$$

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

\[
\begin{array}{c|c|c|c}
\text{ID} & \text{name} & \text{salary} & f \\
1 & Alice & $100 & 4 \\
2 & Bob & $250 & 5 \\
3 & Sue & $300 & 6 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c}
\text{ID} & \text{name} & \text{salary} & \text{age} \\
a & Alice & $100 & 20 \\
b & Bob & $250 & 20 \\
c & Sue & $300 & 30 \\
\end{array}
\]
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.

\[
\begin{align*}
\text{Emp} &\quad \text{Dept} &\quad \text{String} \\
\text{first} &\quad \text{last} &\to \text{Emp} &\to \text{String} \\
\text{name} &\to \text{Dept} &\to \text{String} \\
\text{works} &\to \text{Emp} &\to \text{Dept} \\
\text{mgr} &\to \text{Emp} &\to \text{Emp} \\
\text{secr} &\to \text{Dept} &\to \text{Emp} \\
\forall e : \text{Emp}. \, \text{works}(\text{manager}(e)) = \text{works}(e) \\
\forall d : \text{Dept}. \, \text{works}(\text{secretary}(d)) = d
\end{align*}
\]

- 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 \( S \).

\[
\begin{align*}
101 &\quad 102 &\quad 103 : \text{Emp} \\
q10 &\quad x02 : \text{Dept} \\
\text{mgr}(101) &\equiv 103 \\
\text{works}(101) &\equiv q10 \\
\ldots
\end{align*}
\]

- Treating instances as theories allows instances that are infinite or inconsistent (e.g., Alice=Bob).
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.
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 \( \alpha = \epsilon \circ \varphi \).

\[
\begin{array}{c}
A \xrightarrow{\alpha} I \\
\downarrow \varphi \\
E
\end{array}
\begin{array}{c}
\downarrow \epsilon \\
I
\end{array}
\]

- 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.:

\[
\begin{array}{ccc}
| & p & | \\
\downarrow & & \downarrow \\
| & m & | \\
\downarrow & & \downarrow \\
\downarrow & & \downarrow \\
\bullet & \bullet & \bullet \\
\end{array}
\]

- The category of schemas has all pushouts.
- For every schema \( S \), the category \( S\)-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:

$$\Sigma_{G_1}(I_1) \xleftarrow{\Sigma_{G_1}(h_1)} \left( \Sigma_{G_1 \circ F_1}(I) = \Sigma_{G_2 \circ F_2}(I) \right) \xrightarrow{\Sigma_{G_2}(h_2)} \Sigma_{G_2}(I_2)$$

as the integrated $T$-instance.
Schema Integration
## Data Integration

### Observation Table

<table>
<thead>
<tr>
<th>ID</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ID</th>
<th>BP</th>
<th>Wt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Person Table

<table>
<thead>
<tr>
<th>ID</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender ID</th>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Type Table

<table>
<thead>
<tr>
<th>ID</th>
<th>BP</th>
<th>Wt</th>
<th>HR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Method Table

<table>
<thead>
<tr>
<th>ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method ID</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>f</th>
<th>g1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation Type ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender ID</th>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ID</th>
<th>BP</th>
<th>Wt</th>
<th>HR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>f</th>
<th>g1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender ID</th>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ID</th>
<th>BP</th>
<th>Wt</th>
<th>HR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>f</th>
<th>g1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender ID</th>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ID</th>
<th>BP</th>
<th>Wt</th>
<th>HR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>f</th>
<th>g1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender ID</th>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ID</th>
<th>BP</th>
<th>Wt</th>
<th>HR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>f</th>
<th>g1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

    Employee: Set Int  
    manager: Set (Int × Int)

    - e: Employee is not a judgment.
    - There is a term ∈: Int × Set Int → Bool.

- AQL enriches (schemas, queries and instances) with programs.
  - Collections are types

    Employee: Type  
    manager: Employee → Employee

    - e: Employee is a judgment.
    - There is not a term ∈: Employee × Type → 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
    \[
    \pi_1 : \text{Set} (\text{Int} \times \text{String}) \rightarrow \text{Set Int}
    \]

- **AQL**
  - Schemas are type theories over a base type theory
    \[
    \text{Dept}, \text{name} : \text{Dept} \rightarrow \text{String}
    \]
  - Instances are term models (initial algebras) of theories
    \[
    d_1, d_2 : \text{Dept}, \text{name}(d_1) = \text{CS}, \text{name}(d_2) = \text{Math}
    \]
  - Data migrations are functors
    \[
    \Delta_{\text{Dept}} : (\text{Dept, name: Dept} \rightarrow \text{String})\text{-inst} \rightarrow (\text{Dept} \text{-inst}
    \]
Part 2

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

- **Equations**:
  \[
  \begin{align*}
  id \circ f &= f \\
  f \circ id &= f \\
  f \circ (g \circ h) &= (f \circ g) \circ h \\
  () \circ f &= () \\
  \pi^1 \circ (f, g) &= f \\
  \pi^2 \circ (f, g) &= g \\
  (\pi^1 \circ f, \pi^2 \circ f) &= f \\
  eval \circ (\lambda f \circ \pi^1, \pi^2) &= f \\
  \lambda(eval \circ (f \circ \pi^1, \pi^2)) &= f
  \end{align*}
  \]
Programming AQL in CAML

- 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:

```
a  f  b
```

a  f  b

---

32 / 36
The unit instance 1 has one row per table:

\[
\begin{array}{c|c}
\text{a} & \text{b} \\
\hline
\text{ID} & \text{f} \\
\hline
x & x \\
\end{array}
\]

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

\[
\begin{array}{c|c|c|c}
\text{p} & \text{q} & \text{r} & \text{t} \\
\hline
\text{ID} & \text{f} & \text{ID} & \text{ID} \\
\hline
\text{q} & \text{x} & \text{t} & \text{x} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{a} & \text{b} & \text{a} & \text{b} \\
\hline
\text{ID} & \text{f} & \text{ID} & \text{ID} \\
\hline
\text{x} & \text{x} & \text{x} & \text{x} \\
\end{array}
\]

\( p, q, r, t \xrightarrow{()_t} x \)
Programming AQL in CAML: Products

- Products $s \times t$ are computed row-by-row, with evident projections $\pi^1 : s \times t \rightarrow s$ and $\pi^2 : s \times t \rightarrow t$. For example:

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

$\times$

$\begin{array}{c|c} a & b \\ \hline f & c \\ \hline d \\ \end{array}$

- Given data mappings $f : s \rightarrow t$ and $g : s \rightarrow u$, how to define $(f, g) : s \rightarrow t \times u$ is left to the reader.
  - hint: try it on $\pi^1$ and $\pi^2$ and verify that $(\pi^1, \pi^2) = id$. 

}\end{document}
Exponentials \( t^s \) are given by finding all data mappings \( s \rightarrow t \):

\[
\begin{array}{c|c}
\text{a} & \text{b} \\
\hline
\text{ID} & \text{f} \\
\hline
1 & 3 \\
2 & 3 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
\text{a} & \text{b} \\
\hline
\text{ID} & \text{f} \\
\hline
3 & a \\
4 & c \\
\end{array}
\quad =
\begin{array}{c|c}
\text{b} & \\
\hline
\text{ID} & \\
\hline
3 & c \\
4 & d \\
\end{array}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d )</td>
</tr>
<tr>
<td>1</td>
<td>( b, 2 \mapsto a, 3 \mapsto c, 4 \mapsto d )</td>
</tr>
<tr>
<td>1</td>
<td>( a, 2 \mapsto a, 3 \mapsto c, 4 \mapsto d )</td>
</tr>
<tr>
<td>1</td>
<td>( b, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d )</td>
</tr>
<tr>
<td>1</td>
<td>( a, 2 \mapsto b, 3 \mapsto d, 4 \mapsto c )</td>
</tr>
<tr>
<td>1</td>
<td>( b, 2 \mapsto a, 3 \mapsto d, 4 \mapsto c )</td>
</tr>
<tr>
<td>1</td>
<td>( a, 2 \mapsto a, 3 \mapsto d, 4 \mapsto c )</td>
</tr>
<tr>
<td>1</td>
<td>( b, 2 \mapsto b, 3 \mapsto d, 4 \mapsto c )</td>
</tr>
</tbody>
</table>

Defining \( \text{eval} \) and \( \lambda \) are left to the reader.
Concussion

- 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 “real-world pilot projects”.