This talk describes a new algebraic (purely equational) way to formalize databases and migrate data based on category theory. Category theory was designed to migrate theorems from one area of mathematics to another, so it is a very natural language with which to describe migrating data from one schema to another. Research has culminated in an open-source prototype ETL and data migration tool, CQL (Categorical Query Language), available at categoricaldata.net.

Outline:
- Review of basic category theory.
- Introduction to CQL.
- CQL demo.
- Optional: additional CQL constructions.
- Extra slides: How CQL instances model the simply-typed λ-calculus.
Motivation / Background

- CQL is a ‘category-theoretic’ SQL, used as an ETL tool.
  - Users define schemas and mappings, which induce data transformations.
- CQL schema mappings must preserve data integrity constraints, requiring the use of an automated theorem prover at compile time.
  - CQL catches mistakes at compile time that existing ETL / data migration tools catch at runtime – if at all.
- Some projects using CQL:
  - NIST - several projects.
  - DARPA BRASS project.
  - Empower Retirement.
  - Stanford Chemistry Department.
  - Uber/Tinkerpop
  - and more
Category Theory

- A category \( \mathcal{C} \) consists of
  - a set of objects, \( \text{Ob}(\mathcal{C}) \)
  - for all \( X, Y \in \text{Ob}(\mathcal{C}) \), a set \( \mathcal{C}(X, Y) \) of morphisms a.k.a arrows
  - for all \( X \in \text{Ob}(\mathcal{C}) \), a morphism \( \text{id} \in \mathcal{C}(X, X) \)
  - for all \( X, Y, Z \in \text{Ob}(\mathcal{C}) \), 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” \( \text{Haskell} \) 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

\[ \text{[manager.works]} = \text{[works]} \quad \text{[secretary.works]} = [] \]

<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>

<table>
<thead>
<tr>
<th>ID</th>
<th>sec</th>
<th>name</th>
</tr>
</thead>
<tbody>
<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>ID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
A CQL 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:

$$nodes(S) \rightarrow nodes(T) \quad edges(S) \rightarrow paths(T)$$

\[
\begin{align*}
F(\text{Int}) &= \text{Int} & F(\text{String}) &= \text{String} \\
F(\text{N1}) &= N & F(\text{N2}) &= N \\
F(\text{name}) &= [\text{name}] & F(\text{age}) &= [\text{age}] & F(\text{salary}) &= [\text{salary}] \\
F(f) &= []
\end{align*}
\]
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)

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

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

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

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

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

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

<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>

<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>

<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>
\( \Sigma \) (Outer Union)

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

<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>

\( \Sigma F \)

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>null_1</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
<td>null_2</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
<td>null_3</td>
</tr>
<tr>
<td>d</td>
<td>null_4</td>
<td>null_5</td>
<td>20</td>
</tr>
<tr>
<td>e</td>
<td>null_6</td>
<td>null_7</td>
<td>20</td>
</tr>
<tr>
<td>f</td>
<td>null_8</td>
<td>null_9</td>
<td>30</td>
</tr>
</tbody>
</table>
Unit of $\Sigma_F \rightarrow \Delta_F$

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

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

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

\[F\]

\[\Delta F_{\Pi F, \Sigma F}\]

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 $G : T \rightarrow X$.

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

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

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
<td>salary</td>
</tr>
<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>
<tr>
<td>ID</td>
<td>age</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>name</td>
<td>salary</td>
</tr>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
</tr>
</tbody>
</table>
CQL implements $\Delta$, $\Sigma$, $\Pi$, and more in software.

- catinf.com
Interlude - Additional Constructions

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

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

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

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

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

- Data migration triplets of the form

\[ \Sigma_F \circ \Pi_G \circ \Delta_H \]

can be expressed using (difference-free) 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.

- For arbitrary \( F \), \( \Sigma_F \) can be implemented using canonical/deterministic chase (fire all active triggers across all rules at once.)
Find the name of every manager’s department:

CQL
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)

```
<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>
```

```
<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 $\alpha = \epsilon \circ \varphi$.

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

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

\[
\begin{array}{c}
p \\
\downarrow \\
\bullet \\
\downarrow \\
f' \\
\downarrow \\
\bullet \\
\downarrow \\
f \\
\downarrow \\
\bullet \\
\downarrow \\
g \\
\downarrow \\
\bullet \\
\downarrow \\
g' \\
\downarrow \\
\bullet \\
\downarrow \\
m \\
\downarrow \\
\bullet \\
\downarrow \\
q \\
\end{array}
\]

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 \textit{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 $F_1, F_2$:

$$S_1 \leftarrow S \rightarrow S_2$$

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

$$S_1 \rightarrow T \leftarrow S_2$$

- **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:

$$\Sigma G_1(I_1) \leftarrow (\Sigma G_1 \circ F_1(I) = \Sigma G_2 \circ F_2(I)) \rightarrow \Sigma G_2(I_2)$$

as the integrated $T$-instance.
Schema Integration

Observation

Person → Type

Observation

Person → Method

Person → Gender

Person → Gender

26 / 38
# Data Integration

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>f g</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>BP</td>
</tr>
<tr>
<td>m2</td>
<td>BP</td>
</tr>
<tr>
<td>m3</td>
<td>Wt</td>
</tr>
<tr>
<td>m4</td>
<td>Wt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>f g</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>f g</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person ID</th>
<th>Gender ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>F</td>
<td>BP</td>
</tr>
<tr>
<td>Paul</td>
<td>M</td>
<td>Wt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>f g</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method ID</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>BP</td>
</tr>
<tr>
<td>m2</td>
<td>BP</td>
</tr>
<tr>
<td>m3</td>
<td>HR</td>
</tr>
<tr>
<td>m4</td>
<td>Wt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>f g</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Person ID</th>
<th>Gender ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>F</td>
<td>BP</td>
</tr>
<tr>
<td>Paul</td>
<td>M</td>
<td>Wt</td>
</tr>
<tr>
<td>Peter</td>
<td>null4</td>
<td>HR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation ID</th>
<th>Person ID</th>
<th>Type ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>Peter</td>
<td>BP</td>
</tr>
<tr>
<td>06</td>
<td>Paul</td>
<td>HR</td>
</tr>
<tr>
<td>07</td>
<td>Peter</td>
<td>Wt</td>
</tr>
</tbody>
</table>
In practice, rather than providing entire schema mappings and instance transforms to define pushouts, it is easier to provide equivalence relations and use quotients. In CQL:

```cql
schema T = S1 + S2 /
    S1_Observation = S2.Observation
    S1_Person = S2_Patient
    S1_ObsType = S2_Type
    S1_f = S2_f
    S1_g = S2_g1.S2_g2

instance J = sigma F1 I1 + sigma F2 I2 /
    Peter = Pete
    BloodPressure = BP
    Wt = BodyWeight
```
Conclusion

- We described a new algebraic (equational) 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 CQL, an open-source project, available at catinf.com. Collaborators welcome!
- CQL is only one example of a language I’ve developed that includes strong static reasoning principles; others include
  - HIL
  - Hoare Type Theory (Coq RDBMS, etc)
Patrick Schultz, Ryan Wisnesky. Algebraic Data Integration. (JFP-PlanBig 2017)


CQL 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
    $$\pi_1 : \text{Set} \ (\text{Int} \times \text{String}) \rightarrow \text{Set} \ \text{Int}$$

- CQL
  - Schemas are type theories over a base type theory
    $$\text{Dept, name}: \text{Dept} \rightarrow \text{String}$$
  - 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
    $$\Delta_{\text{Dept}} : (\text{Dept, name}: \text{Dept} \rightarrow \text{String})\text{-inst} \rightarrow (\text{Dept})\text{-inst}$$
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** $f, g$:
  \[ \begin{align*}
    id_t : t \to t & \quad ()_t : t \to 1 \\
    \pi^1_{s,t} : s \times t \to s & \quad \pi^2_{s,t} : s \times t \to t \\
    eval_{s,t} : t^s \times s \to t \\
    f : s \to u & \quad g : u \to t \\
    g \circ f : s \to t \\
    f : s \to t & \quad g : s \to u \\
    (f, g) : s \to t \times u \\
    f : s \times u \to t \\
    \lambda f : s \to t^u
  \end{align*} \]

- **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 $f : t \to t'$, you get a data mapping $[f] : [t] \to [t']$.
  - 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] ----> [b]
Programming CQL 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>p</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>q</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
t = \begin{array}{c|c|c|c|c|c|c}
\hline
& a & & b & & \\hline
ID & f & & ID & & \\hline
p & q & & q & & \\hline
r & t & & t & & \hline
\end{array}
\]

\[
()_t \text{ sends } p, q, r, t \rightarrow x
\]

\[
(\cdot)_t : t \rightarrow 1
\]

\[
= 1
\]
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:

\[
\begin{array}{c|c}
    \text{a} & \text{b} \\
    \hline
    \text{ID} & \text{f} \\
    1 & 3 \\
    2 & 3 \\
\end{array}
\times
\begin{array}{c|c|c}
    \text{a} & \text{b} \\
    \hline
    \text{ID} & \text{f} & \text{ID} \\
    \text{ID} & \text{ID} & \text{ID} \\
    3 & 4 & 3 \\
    a & c & c \\
    b & c & d \\
\end{array}
= 
\begin{array}{c|c|c|c|c}
    \text{a} & \text{b} \\
    \hline
    \text{ID} & \text{f} & \text{ID} & \text{ID} & \text{ID} \\
    (1,a) & (3,c) & (1,b) & (3,c) & (3,c) \\
    (2,a) & (3,c) & (2,b) & (3,c) & (4,c) \\
\end{array}
\]

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 $s \rightarrow t$:

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>f</td>
<td>ID</td>
<td>f</td>
<td>ID</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

Defining $eval$ and $\lambda$ are left to the reader.