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".

Category Theory

- A category $\mathcal{C}$ consists of
  - a set of *objects*, $\mathsf{Ob}(\mathcal{C})$
  - forall $X, Y \in \mathsf{Ob}(\mathcal{C})$, a set $\mathcal{C}(X, Y)$ of *morphisms* a.k.a *arrows*
  - forall $X \in \mathsf{Ob}(\mathcal{C})$, a morphism $id \in \mathcal{C}(X, X)$
  - forall $X, Y, Z \in \mathsf{Ob}(\mathcal{C})$, a function $\circ : \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) \to \mathcal{C}(X, Z)$ s.t.

$$f \circ id = f \qquad id \circ f = f \qquad\qquad (f \circ g) \circ h = f \circ (g \circ h)$$

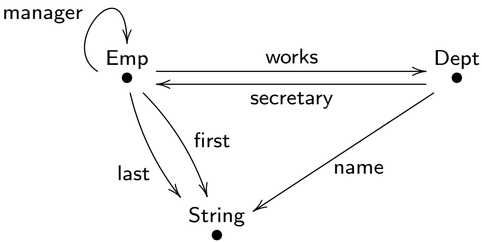
- The category $\mathbf{Set}$ has sets as objects and functions as arrows, and the "category" $\mathbf{Haskell}$ has types as objects and programs as arrows.

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

$$F(id) = id \qquad F(f \circ g) = F(f) \circ F(g)$$

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

Schemas and Instances



$$[\text{manager.works}] = [\text{works}] \qquad [\text{secretary.works}] = []$$

Emp				
ID	**mgr**	**works**	**first**	**last**
101	103	q10	Al	Akin
102	102	x02	Bob	Bo
103	103	q10	Carl	Cork

Dept		
ID	**sec**	**name**
q10	101	CS
x02	102	Math

String
ID
Al
Bob
. . .

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 $[\![S]\!]$.
  - Ob($[\![S]\!]$) is the nodes of S.
  - Forall nodes X, Y, $[\![S]\!](X, Y)$ 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 $[\![S]\!] \to [\![T]\!]$.
  - It can be defined as an equation-preserving function:

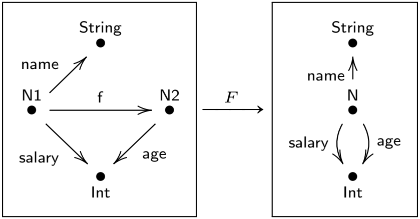
$$nodes(S) \to nodes(T) \qquad edges(S) \to paths(T).$$

- An S-instance is a functor $[\![S]\!] \to \mathbf{Set}$.
  - 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** $F : S \to T$ is an equation-preserving function:

$$nodes(S) \to nodes(T) \qquad edges(S) \to paths(T)$$



$$F(\text{Int}) = \text{Int} \qquad F(\text{String}) = \text{String}$$

$$F(\text{N1}) = \text{N} \qquad F(\text{N2}) = \text{N}$$

$$F(\text{name}) = [\text{name}] \qquad F(\text{age}) = [\text{age}] \qquad F(\text{salary}) = [\text{salary}]$$

$$F(\text{f}) = []$$

Functorial Data Migration

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

- $\Delta_F\colon T$-inst $\to S$-inst (like project)

$$\begin{tikzcd} S \arrow[r, "F"] \arrow[rr, bend right, "\Delta_F(I) \ := \ I \circ F"'] & T \arrow[r, "I"] & \mathbf{Set} \end{tikzcd}$$

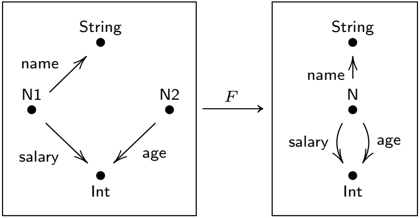
- $\Pi_F\colon S$-inst $\to T$-inst (right adjoint to Δ_F; like join)

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

- $\Sigma_F\colon S$-inst $\to T$-inst (left adjoint to Δ_F; like outer union then merge)

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

Δ (Project)



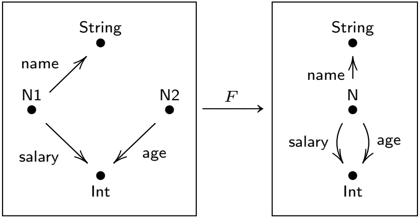
N1		
ID	**name**	**salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2	
ID	**age**
4	20
5	20
6	30

$\xleftarrow{\Delta_F}$

N			
ID	**name**	**salary**	**age**
a	Alice	$100	20
b	Bob	$250	20
c	Sue	$300	30

Π (Product)



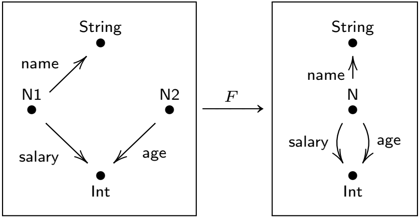
N1		
ID	**name**	**salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2	
ID	**age**
4	20
5	20
6	30

$\xrightarrow{\Pi_F}$

N			
ID	**name**	**salary**	**age**
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

Σ (Outer Union)



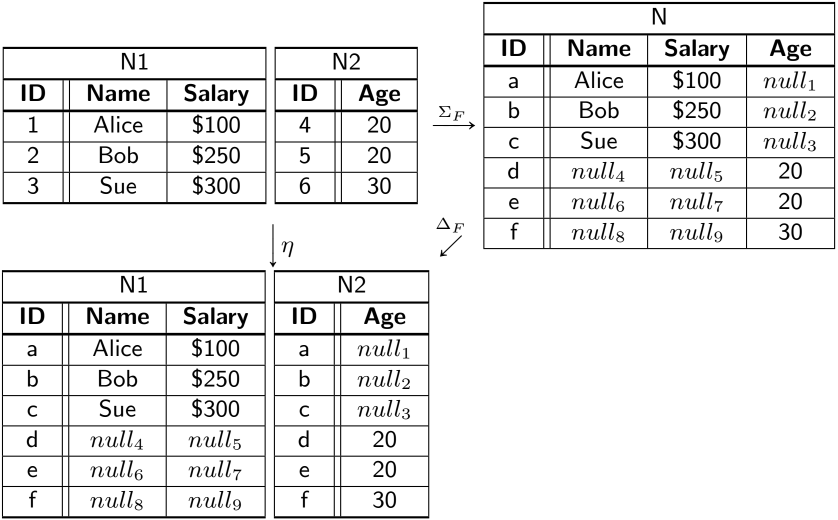
N1		
ID	**Name**	**Salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2	
ID	**Age**
4	20
5	20
6	30

$\xrightarrow{\Sigma_F}$

N			
ID	**Name**	**Salary**	**Age**
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

Unit of $\Sigma_F \dashv \Delta_F$



N1		
ID	**Name**	**Salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

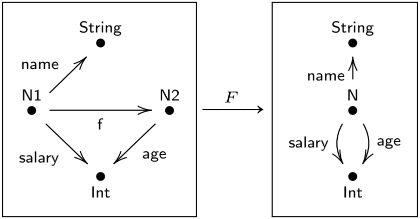
N2	
ID	**Age**
4	20
5	20
6	30

N			
ID	**Name**	**Salary**	**Age**
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

N1		
ID	**Name**	**Salary**
a	Alice	$100
b	Bob	$250
c	Sue	$300
d	$null_4$	$null_5$
e	$null_6$	$null_7$
f	$null_8$	$null_9$

N2	
ID	**Age**
a	$null_1$
b	$null_2$
c	$null_3$
d	20
e	20
f	30

A Foreign Key



N1			
ID	**name**	**salary**	**f**
1	Alice	$100	4
2	Bob	$250	5
3	Sue	$300	6

N2	
ID	**age**
4	20
5	20
6	30

$$\xleftarrow{\Delta_F}$$

$$\xrightarrow{\Pi_F,\Sigma_F}$$

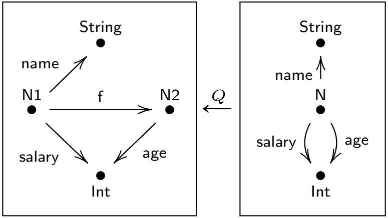
N			
ID	**name**	**salary**	**age**
a	Alice	$100	20
b	Bob	$250	20
c	Sue	$300	30

Queries

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

$$eval_Q \cong \Delta_G \circ \Pi_F \qquad coeval_Q \cong \Delta_F \circ \Pi_G$$

These can be specified using comprehension notation similar to SQL.

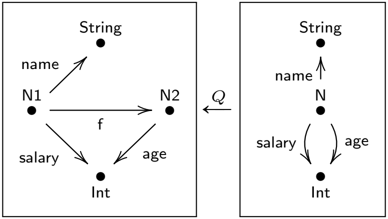


```
N1 -> select n1.name as name, n1.salary as salary
  from N as n1

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

f -> {n2 -> n1}
```

A Foreign Key



N1			
ID	**name**	**salary**	**f**
1	Alice	$100	4
2	Bob	$250	5
3	Sue	$300	6

N2	
ID	**age**
4	20
5	20
6	30

$\xleftarrow{eval_Q}$

$\xrightarrow{coeval_Q}$

N			
ID	**name**	**salary**	**age**
a	Alice	$100	20
b	Bob	$250	20
c	Sue	$300	30

AQL Demo

- AQL implements Δ, Σ, Π, 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.

$$\text{Emp Dept String} : \text{Type} \qquad \text{first last} : \text{Emp} \to \text{String} \qquad \text{name} : \text{Dept} \to \text{String}$$

$$\text{works} : \text{Emp} \to \text{Dept} \qquad \text{mgr} : \text{Emp} \to \text{Emp} \qquad \text{secr} : \text{Dept} \to \text{Emp}$$

$$\forall e : \text{Emp}.\ \text{works}(\text{manager}(e)) = \text{works}(e) \qquad \forall d : \text{Dept}.\ \text{works}(\text{secretary}(d)) = d$$

- 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.

$$\text{101 102 103} : \text{Emp} \qquad \text{q10 x02} : \text{Dept}$$

$$\text{mgr}(101) = 103 \qquad \text{works}(101) = \text{q10} \qquad \ldots$$

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

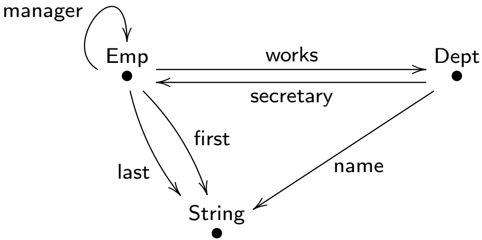
AQL vs SQL

- 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.

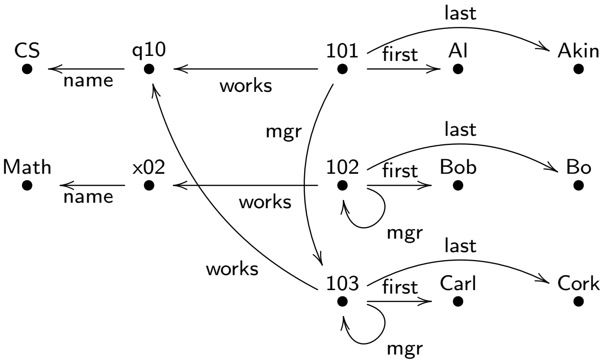
Select-From-Where Syntax



Find the name of every manager's department:

```
AQL                             SQL
select e.manager.works.name     select d.name
from Emp as e                   from Emp as e1, Emp as e2, Dept as d
                                where e1.manager = e2.ID and
                                      e2.works = d.ID
```

Pivot (Instance ⇔ Schema)



Emp				
ID	**mgr**	**works**	**first**	**last**
101	103	q10	Al	Akin
102	102	x02	Bob	Bo
103	103	q10	Carl	Cork

Dept	
ID	**name**
q10	CS
x02	Math

Richer Constraints

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

$$\begin{tikzcd} A \arrow[r, "\alpha"] \arrow[d, "\varphi"'] & I \\ E \arrow[ur, dashed, "\epsilon"'] & \end{tikzcd}$$

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

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

  is captured as

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

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

- 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.:

$$\begin{tikzcd}
& \bullet \arrow[dl, "p"'] \arrow[dr, "q"] & \\
\bullet \arrow[dr, "f"] \arrow[ddr, bend right, "f'"'] & & \bullet \arrow[dl, "g"'] \arrow[ddl, bend left, "g'"] \\
& \bullet \arrow[d, dashed, "m"] & \\
& \bullet &
\end{tikzcd}$$

- The category of schemas has all pushouts.
- For every schema S, the category S-inst has all pushouts.
- Pushouts of schemas, instances, and Σ 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:

$$S_1 \overset{F_1}{\leftarrow} S \overset{F_2}{\rightarrow} S_2$$

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

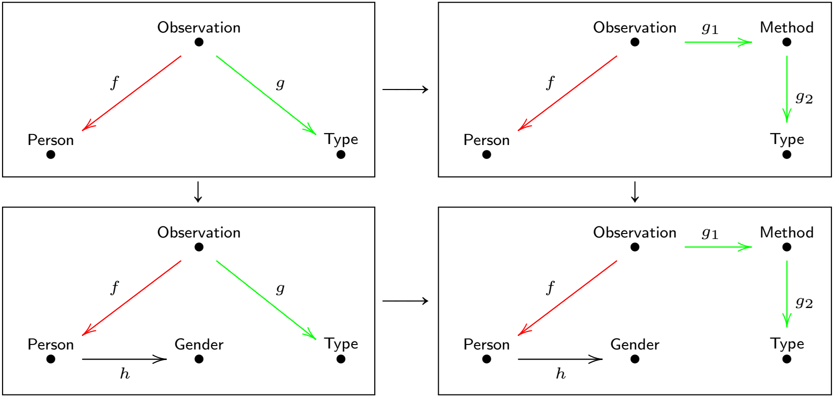
$$S_1 \overset{G_1}{\rightarrow} T \overset{G_2}{\leftarrow} S_2$$

- 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:

$$\Sigma_{G_1}(I_1) \overset{\Sigma_{G_1}(h_1)}{\longleftarrow} \big(\Sigma_{G_1 \circ F_1}(I) = \Sigma_{G_2 \circ F_2}(I)\big) \overset{\Sigma_{G_2}(h_2)}{\longrightarrow} \Sigma_{G_2}(I_2)$$

  as the integrated T-instance.

Schema Integration



Data Integration

Observation

ID	f	g

Person

ID
p

Type

ID
BP
Wt

→

Gender

ID
F
M

Type

ID
BP
Wt
HR

Observation

ID	f	g
o_5	Peter	BP
o_6	Paul	HR
o_7	Peter	Wt

Person

ID	h
Paul	M
Peter	M

↓ ↓

Method

ID	g2
m_1	BP
m_2	BP
m_3	Wt
m_4	Wt

Type

ID
BP
Wt

Observation

ID	f	g1
o_1	Pete	m_1
o_2	Pete	m_2
o_3	Jane	m_3
o_4	Jane	m_1

Person

ID
Jane
Pete

→

Method

ID	g2
$null_1$	BP
$null_2$	Wt
$null_3$	HR
m_1	BP
m_2	BP
m_3	Wt
m_4	Wt

Observation

ID	f	g1
o_1	Peter	m_1
o_2	Peter	m_2
o_3	Jane	m_3
o_4	Jane	m_1
o_5	Peter	$null_1$
o_6	Paul	$null_2$
o_7	Peter	$null_3$

Gender

ID
F
M
$null_4$

Type

ID
BP
Wt
HR

Person

ID	h
Jane	$null_4$
Paul	M
Peter	M

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*

$$\mathsf{Employee} \colon \mathsf{Set\ Int} \quad \mathsf{manager} \colon \mathsf{Set}\ (\mathsf{Int} \times \mathsf{Int})$$

  - $\mathsf{e} \colon \mathsf{Employee}$ *is not* a judgment.
  - There *is* a term $\in \colon \mathsf{Int} \times \mathsf{Set\ Int} \to \mathsf{Bool}$.
- AQL enriches (schemas, queries and instances) with programs.
  - Collections are *types*

$$\mathsf{Employee} \colon \mathsf{Type} \quad \mathsf{manager} \colon \mathsf{Employee} \to \mathsf{Employee}$$

  - $\mathsf{e} \colon \mathsf{Employee}$ *is* a judgment.
  - There *is not* a term $\in \colon \mathsf{Employee} \times \mathsf{Type} \to \mathsf{Bool}$.
- 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

$$\mathsf{Set}\ (\mathsf{Int} \times \mathsf{String})$$

  - Instances are terms

$$\{(1, \mathsf{CS})\} \cup \{(2, \mathsf{Math})\}$$

  - Data migrations are functions

$$\pi_1 : \mathsf{Set}\ (\mathsf{Int} \times \mathsf{String}) \to \mathsf{Set\ Int}$$

- AQL
  - Schemas are type theories over a base type theory

$$\mathsf{Dept},\ \ \mathsf{name} : \mathsf{Dept} \to \mathsf{String}$$

  - Instances are term models (initial algebras) of theories

$$\mathsf{d}_1, \mathsf{d}_2 : \mathsf{Dept},\ \ \mathsf{name}(\mathsf{d}_1) = \mathsf{CS},\ \ \mathsf{name}(\mathsf{d}_2) = \mathsf{Math}$$

  - Data migrations are functors

$$\Delta_{\mathsf{Dept}} : (\mathsf{Dept}, \mathsf{name} : \mathsf{Dept} \to \mathsf{String})\text{-}\mathsf{inst} \ \to\ (\mathsf{Dept})\text{-}\mathsf{inst}$$

Part 2

- For every schema S, S-inst models simply-typed λ-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:

$$id_t : t \to t \qquad ()_t : t \to 1 \qquad \pi^1_{s,t} : s \times t \to s \qquad \pi^2_{s,t} : s \times t \to t$$

$$eval_{s,t} : t^s \times s \to t \qquad \frac{f : s \to u \qquad g : u \to t}{g \circ f : s \to t} \qquad \frac{f : s \to t \qquad g : s \to u}{(f, g) : s \to t \times u}$$

$$\frac{f : s \times u \to t}{\lambda f : s \to t^u}$$

- Equations:

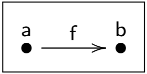
$$id \circ f = f \qquad f \circ id = f \qquad f \circ (g \circ h) = (f \circ g) \circ h \qquad () \circ f = ()$$

$$\pi^1 \circ (f, g) = f \qquad \pi^2 \circ (f, g) = g \qquad (\pi^1 \circ f, \pi^2 \circ f) = f$$

$$eval \circ (\lambda f \circ \pi^1, \pi^2) = f \qquad \lambda(eval \circ (f \circ \pi^1, \pi^2)) = f$$

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 $f : t \to t'$, 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:



Programming AQL in CAML: Unit

- The unit instance 1 has one row per table:

a	
ID	**f**
x	x

b
ID
x

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

$$t = \begin{array}{|c||c|}\hline \multicolumn{2}{|c|}{\text{a}} \\ \hline \textbf{ID} & \textbf{f} \\ \hline \text{p} & \text{q} \\ \hline \text{r} & \text{t} \\ \hline \end{array} \quad \begin{array}{|c||}\hline \text{b} \\ \hline \textbf{ID} \\ \hline \text{q} \\ \hline \text{t} \\ \hline \end{array} \xrightarrow{()_t} \begin{array}{|c||c|}\hline \multicolumn{2}{|c|}{\text{a}} \\ \hline \textbf{ID} & \textbf{f} \\ \hline \text{x} & \text{x} \\ \hline \end{array} \quad \begin{array}{|c||}\hline \text{b} \\ \hline \textbf{ID} \\ \hline \text{x} \\ \hline \end{array} = 1$$

$$\text{p}, \text{q}, \text{r}, \text{t} \xrightarrow{()_t} \text{x}$$

Programming AQL in CAML: Products

- 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:

a	
ID	**f**
1	3
2	3

b
ID
3
4

×

a	
ID	**f**
a	c
b	c

b
ID
c
d

=

a	
ID	**f**
(1,a)	(3,c)
(1,b)	(3,c)
(2,a)	(3,c)
(2,b)	(3,c)

b
ID
(3,c)
(3,d)
(4,c)
(4,d)

- 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 π^1 and π^2 and verify that $(\pi^1, \pi^2) = id$.

Programming AQL in CAML: Exponentials

- Exponentials t^s are given by finding all data mappings $s \to t$:

a	
ID	**f**
1	3
2	3

b
ID
3
4

$\to$

a	
ID	**f**
a	c
b	c

b
ID
c
d

$=$

a	
ID	**f**
$1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d$	$3 \mapsto c, 4 \mapsto d$
$1 \mapsto b, 2 \mapsto a, 3 \mapsto c, 4 \mapsto d$	$3 \mapsto c, 4 \mapsto d$
$1 \mapsto a, 2 \mapsto a, 3 \mapsto c, 4 \mapsto d$	$3 \mapsto c, 4 \mapsto d$
$1 \mapsto b, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d$	$3 \mapsto c, 4 \mapsto d$
$1 \mapsto a, 2 \mapsto b, 3 \mapsto d, 4 \mapsto c$	$3 \mapsto d, 4 \mapsto c$
$1 \mapsto b, 2 \mapsto a, 3 \mapsto d, 4 \mapsto c$	$3 \mapsto d, 4 \mapsto c$
$1 \mapsto a, 2 \mapsto a, 3 \mapsto d, 4 \mapsto c$	$3 \mapsto d, 4 \mapsto c$
$1 \mapsto b, 2 \mapsto b, 3 \mapsto d, 4 \mapsto c$	$3 \mapsto d, 4 \mapsto c$

b
ID
$3 \mapsto c, 4 \mapsto c$
$3 \mapsto c, 4 \mapsto d$
$3 \mapsto d, 4 \mapsto c$
$3 \mapsto d, 4 \mapsto d$

- Defining $eval$ and λ 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, Σ, Δ, Π manipulate data.
    - Instances on a schema model the simply-typed λ-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”.