The Functorial Data Model

Patrick Schultz, David Spivak, **Ryan Wisnesky**

Department of Mathematics
Massachusetts Institute of Technology

{schultzp, dspivak, **wisnesky**}@math.mit.edu

Foundational Methods in Computer Science
June 6, 2015

Outline

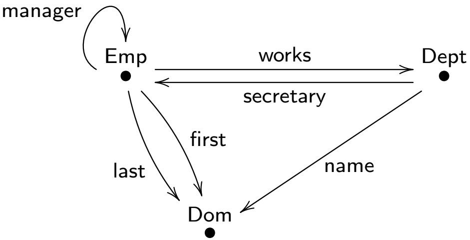
- The functorial data model (my name) originated with Rosebrugh et al. in the late 1990s.
  - Schemas are categories, instances are set-valued functors.
  - Spivak proposes using it to solve information integration problems.
- I will describe:
  - Rosebrugh's original model (the FDM)
  - How to use the FDM for information integration
  - Extending the FDM towards SQL (FQL)
  - Extending the FDM towards functional programming (FPQL)
  - Conjectures
- Sponsored by:
  - ONR grant N000141310260
  - AFOSR grant FA9550-14-1-0031

Category theory

- A category $\mathcal{C}$ consists of
  - a set of *objects*
  - for all objects X, Y a set $\mathcal{C}(X, Y)$ of *arrows*
  - for all objects X an arrow $id \in \mathcal{C}(X, X)$
  - for all objects X, Y, Z a function $\circ \colon \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) \to \mathcal{C}(X, Z)$
  - such that $f \circ id = id$ and $id \circ f = f$ and $(f \circ g) \circ h = f \circ (g \circ h)$
- A functor $F : \mathcal{C} \to \mathcal{D}$ is a function taking objects in $\mathcal{C}$ to objects in $\mathcal{D}$ and arrows $f \colon X \to Y$ in $\mathcal{C}$ to arrows $F(f) \colon F(X) \to F(Y)$ in $\mathcal{D}$ such that $F(id) = id$ and $F(f \circ g) = F(f) \circ F(g)$.

- A category presentation $\mathcal{C}$ consists of
  - a set of *nodes*
  - for all nodes X, Y a set $\mathcal{C}(X, Y)$ of *edges*
  - a set of path equations
- A functor presentation $F \colon \mathcal{C} \to \mathcal{D}$ is a function taking nodes in $\mathcal{C}$ to nodes in $\mathcal{D}$ and edges $f \colon X \to Y$ in $\mathcal{C}$ to paths $F(f) \colon F(X) \to F(Y)$ in $\mathcal{D}$ such that $\mathcal{C} \vdash p = q$ implies $\mathcal{D} \vdash F(p) = F(q)$.

The Functorial Data Model



Emp.manager.works = Emp.works

Dept.secretary.works = Dept

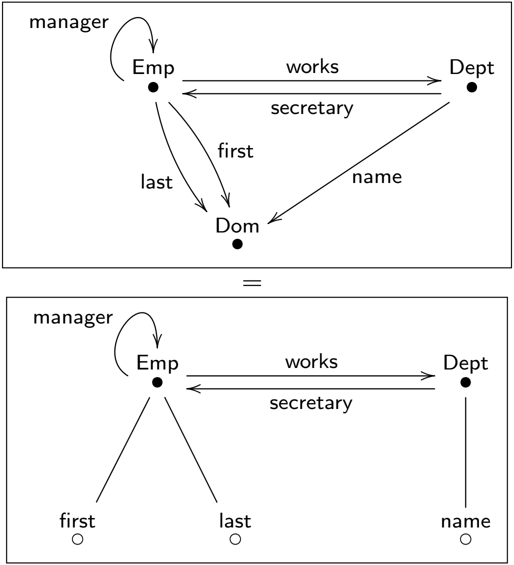
Emp				
ID	**mgr**	**works**	**first**	**last**
101	103	q10	Al	Akin
102	102	x02	Bob	Bo
103	103	q10	Carl	Cork

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

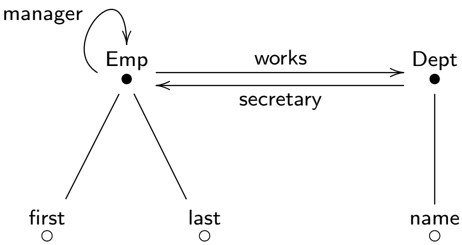
Dom
ID
Al
Akin
Bob
Bo
Carl
Cork
CS
Math

Convention

- Omit Dom table, and draw edges $\bullet \rightarrow_f \bullet_{Dom}$ as $\bullet - \circ_f$:



The Functorial Data Model (abbreviated)



Emp.manager.works = Emp.works   Dept.secretary.works = Dept

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	102	CS
x02	101	Math

Functorial Data Migration

- A functor $F\colon S \to T$ is a constraint-respecting mapping:

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

  and it induces three adjoint data migration functors:

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

$$S \xrightarrow{F} T \xrightarrow{I} \mathbf{Set}$$

$$\Delta_F(I) := I \circ F$$

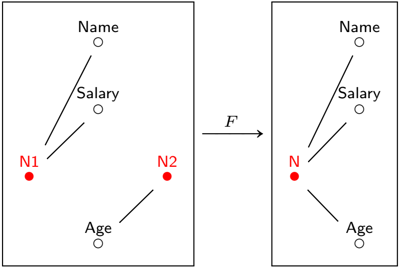
  - $\Pi_F\colon S\text{-inst} \to T\text{-inst}$ (like join)

$$\Delta_F \dashv \Pi_F$$

  - $\Sigma_F\colon S\text{-inst} \to T\text{-inst}$ (like outer disjoint union then quotient)

$$\Sigma_F \dashv \Delta_F$$

Δ (Project)



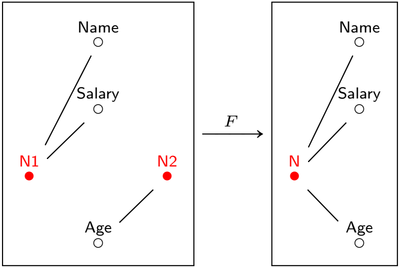
N1		
ID	**Name**	**Salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

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 (Join)



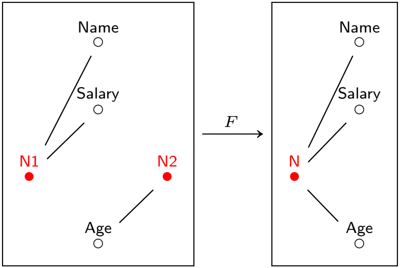
N1		
ID	**Name**	**Salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

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 (Union)



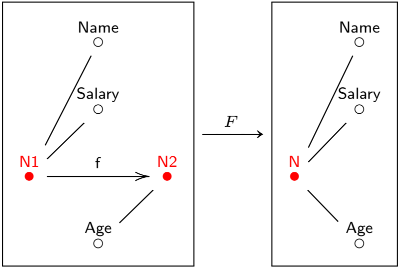
N1		
ID	**Name**	**Salary**
1	Alice	$100
2	Bob	$250
3	Sue	$300

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

Foreign keys



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

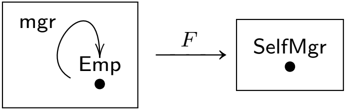
N2	
ID	**Age**
4	20
5	20
6	30

$\xleftarrow{\Delta_F}$

$\xrightarrow{\Pi_F, \Sigma_F}$

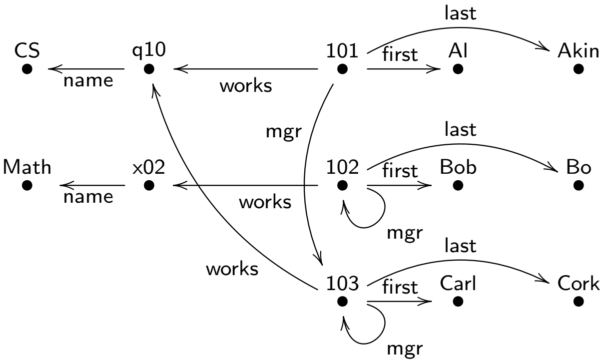
N			
ID	**Name**	**Salary**	**Age**
a	Alice	$100	20
b	Bob	$250	20
c	Sue	$300	30

Self-managers



- Δ_F will copy SelfMgr into Mgr, and put the identity into mgr.
- Π_F will migrate into SelfMgr those Emps who are their own mgr.
- Σ_F will migrate into SelfMgr representatives of the "management groups" of Emp, i.e. equivalence classes of Emps modulo the equivalence relation generated by mgr.
  - Adjoints are only unique up to isomorphism; hence, there are many Σ_F functors; each will choose a different representative.

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

Evaluation of the functorial data model

- Positives:
  - The category of categories is bi-cartesian closed (model of the STLC).
  - For each category C, the category C-inst is a topos (model of HOL).
  - Data integrity constraints (path equations) are built-in to schemas.
  - Data migration functors transform entire instances.
  - The FDM is expressive enough for many information integration tasks.
  - Easy to pivot.

- Negatives:
  - Data integrity constraints (in schemas) are limited to path equalities.
  - Data migrations lack analog of set-difference.
  - No aggregation.
  - Data migration functors are hard to program directly.
  - Instance isomorphism is too coarse for many integration tasks.
  - Many problems about finitely-presented categories are semi-computable:
    - Path equivalence (required to check functors are constraint-respecting).
    - Generating a category from a presentation (hence the category of finitely-presented categories is not cartesian closed).

The Attribute Problem

N			
ID	**Name**	**Age**	**Salary**
1	Alice	20	$100
2	Bob	20	$250
3	Sue	30	$300

$\cong$ (good)

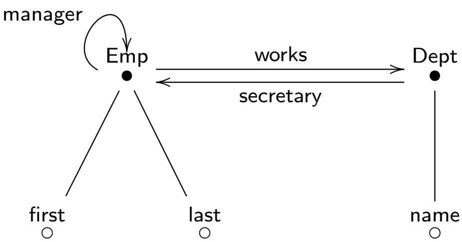
N			
ID	**Name**	**Age**	**Salary**
4	Alice	20	$100
5	Bob	20	$250
6	Sue	30	$300

$\cong$ (bad)

N			
ID	**Name**	**Age**	**Salary**
1	Amy	20	$100
2	Bill	20	$250
3	Susan	30	$300

Solving the Attribute Problem

- Mark certain edges to leaf nodes as "attributes".
  - In this extension, a schema is a category C, a discrete category C_0, and a functor $C_0 \to C$. Instances and migrations also generalize.
  - Schemas become special ER (entity-relationship) diagrams.
  - The FDM takes C_0 to be empty.
  - The example schema below, which was an abbreviation in the FDM, is a bona-fide schema in this extension: attributes are first, last, and name.



Solved Attribute Problem

N			
ID	**Name**	**Age**	**Salary**
1	Alice	20	$100
2	Bob	20	$250
3	Sue	30	$300

$\cong$ (good)

N			
ID	**Name**	**Age**	**Salary**
4	Alice	20	$100
5	Bob	20	$250
6	Sue	30	$300

$\ncong$ (good)

N			
ID	**Name**	**Age**	**Salary**
1	Amy	20	$100
2	Bill	20	$250
3	Susan	30	$300

FQL - A Functorial Query Language

- The "schemas as ER diagrams" extension to the functorial data model is the basis of FQL.
  - Open-source, graphical IDE available at categoricaldata.net/fql.html.
- FQL translates data migrations of the form

$$\Sigma_F \circ \Pi_G \circ \Delta_H$$

  into SQL and vice versa. Caveats:
  - F must be a discrete op-fibration (ensures union compatibility).
  - G must be a surjection on attributes (ensures domain independence).
  - All categories must be finite (ensures computability).
  - FQL $\mapsto$ SPCU+idgen (sets)
    SPCU (bags) $\mapsto$ FQL, SPCU (sets) $\mapsto$ FQL+squash
    selection equality conjunctive and between variables only.
- Theorem: FQL queries are closed under composition.

FQL Demo

FQL evaluation

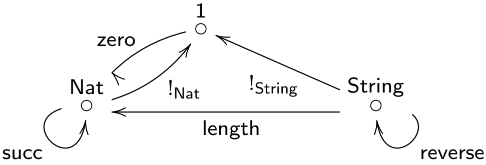
- Positives:
  - Attributes.
  - Running on SQL enables interoperability and execution speed.
  - Better Σ semantics than TGD-only systems (e.g., Clio).
- Negatives:
  - No selection by constants.
  - Relies on fresh ID generation.
  - Cannot change type of data during migration.
  - Attributes not nullable.
- Apply type-theory to FQL to overcome negatives.

FPQL - a functorial programming and query language

- FPQL extends FQL schemas to include edges between attributes.
  - A typing Γ is a category with terminal object.
  - A schema S on typing Γ is a category extending Γ in a special way.
  - An instance I on schema S is a category extending S in a special way.
- Design decision: treat all categories as finitely-presented, and use monoidal Knuth-Bendix to reduce paths.
- FPQL instances are deductive databases, not extensional ones.
  - FPQL allows inconsistent and infinite databases, if desired.
  - FPQL cannot be implemented with SQL, but can borrow implementation techniques from SQL.

Typings

- A typing is a category with terminal object 1:



$$\text{reverse.reverse} = id \qquad \text{length} = \text{reverse.length}$$

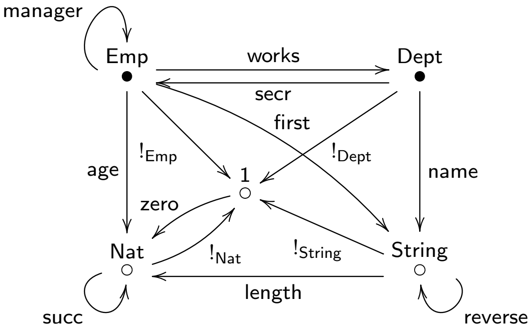
- Implicitly includes, for all well-typed edges e:

$$id_1 = !_1 \qquad (e\colon t \to 1) = !_t \qquad (e\colon t \to t').!_{t'} = !_t$$

- Objects are *types*, arrows are *functions*.

Schemas

- A schema over a typing Γ is a category extending Γ with
  - New objects, called *entities*.
  - New arrows from entities to entities, called *foreign keys*.
  - New arrows from entities to types, called *attributes*.
  - New equations.



$$\text{manager.works} = \text{works} \qquad \text{secr.works} = id$$

Instances

- An instance over a schema S is a category extending S with
  - New edges from 1, called *variables*, such as

$$\text{bill}\colon 1 \to \text{Emp} \qquad \text{infinity}\colon 1 \to \text{Nat}$$

  - New equations, such as

$$\text{bill.age} = \text{zero} \qquad \text{bill.works.secr.manager} = \text{bill} \qquad \text{bill.manager} = \text{bill}$$

- Tabular view of instances:

Emp				
ID	**manager**	**works**	**age**	**first**
bill	bill	bill.works	zero	bill.first
bill.works.secr	bill	bill.works	bill.works.secr.age	bill.works.secr.first

Dept		
ID	**secr**	**name**
bill.works	bill.works.secr	bill.works.name

FPQL Example

```
Nat: type
zero: Nat
succ: Nat -> Nat

String: type
reverse:
  String -> String
length:
  String -> Nat

eq1: reverse.reverse
           = String

eq2: reverse.length
           = length
```

```
S = schema {

 nodes
Emp, Dept;

 edges
age     : Emp ->Nat,
first   : Emp ->String,
name    : Dept->String,
works   : Emp ->Dept,
secr    : Dept->Emp,
manager: Emp ->Emp;

 equations
manager.works = works,
secr.works = Dept;

 }
```

```
I = instance {

 variables
bill : Emp,
infinity : Nat;

 equations
bill.age = zero,
bill.works
  .secr.manager
    = bill;

} : S
```

Data Migration in FPQL

- When S and T are schemas on typing Γ, a schema morphism $F\colon S \to T$ is a constraint-respecting mapping

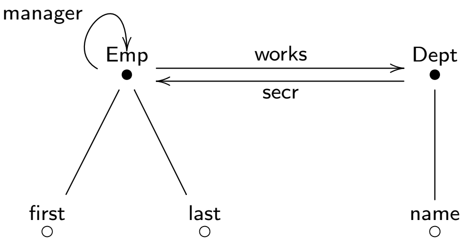
$$nodes(S) \to nodes(T) \qquad edges(S) \to paths(T)$$

  that is the identity on Γ.
- Σ_F is defined to be *substitution* along F:

$$v\colon 1 \to X \ \in I \quad \text{implies} \quad v\colon 1 \to F(X) \ \in \Sigma_F(I)$$

- $\Sigma_F \dashv \Delta_F \dashv \Pi_F$
- Migrations $\Sigma_F \circ \Delta_G \circ \Pi_F$, where F is a discrete op-fibration, are closed under composition, and can be written in SQL-like syntax.

Flower Syntax in FPQL



FPQL

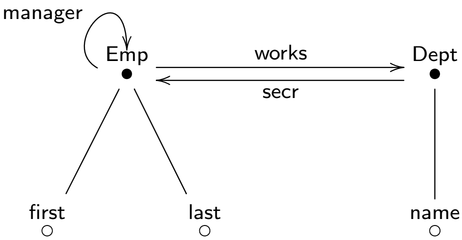
```
select e.first
from Emp as e
where e.manager.manager = e
```

SQL

```
select e.first
from Emp as e, Emp as f
where e.manager = f.ID and
      f.manager = e.ID
```

Uber-Flower Syntax in FPQL

- Set everyone's manager to their manager's manager:




```
EmpQuery = {
 from
  Emp as e
 attributes
  first = e.first
  last = e.last
 edges
  manager =
   {e=e.manager.manager}:EmpQuery
  works =
   {d=e.works}:DeptQuery
} : Emp
```

```
DeptQuery = {
 from
  Dept as d
 attributes
  name = d.name
 edges
  secr = {e=d.secr}:EmpQuery
} : Dept
```

Evaluation of FPQL

- Positives
  - Flower syntax
  - Can change type of data
  - Nullable attributes
  - Typings allow functional programming
  - Σ is extremely cheap
- Negatives
  - No special support for cartesian closed typings (λ-calculi)
  - Categories of instances on a fixed schema are not cartesian closed
  - Cannot run on SQL

Conjectures

- An embedded dependency (ED) is a lifting problem.
- The chase is a left Kan extension.
- Σ_F, Δ_F and Δ_F, Π_F are reverse data exchanges.
- For every data migration $F\colon S \to T$, there exists an X such that F can be implemented by chasing a set of EDs over $S + T + X$.

Conclusion

- Initial success using FPQL with NIST
- Deep connections between the FDM and the relational model
- Looking for collaborators

- Future work:
  - Restrict typings to a particular cartesian closed category, e.g., Java
  - FQL flowers : SQL flowers as ? : EDs
  - Aggregation
  - Generating mappings from matchings
  - Entity-resolution
  - Algorithms