A Functorial Query Language

Ryan Wisnesky, David Spivak

Department of Mathematics
Massachusetts Institute of Technology

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

Presented at Boston Haskell
April 16, 2014

Outline

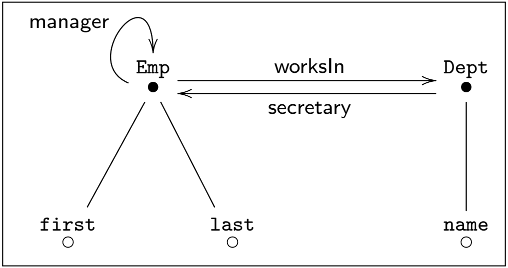
- Introduction to FQL.
    - FQL is a database query language based on *category theory*.
    - But, there will be no category theory in this talk.

- How to program FQL using Haskell.
    - FQL provides an *alternative semantics* for Haskell programs.
    - If you can program Haskell, you can program FQL.

- Demo of the FQL IDE.
    - Project webpage: categoricaldata.net/fql.html

Introduction to FQL

- In FQL, a database schema is a special kind of entity-relationship (ER) diagram.

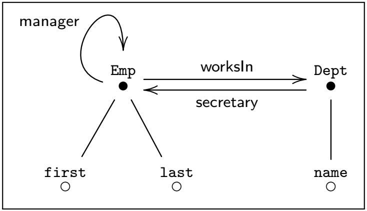


`Emp.manager.worksIn` = `Emp.worksIn` `Dept.secretary.worksIn` = `Dept`

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

Introduction to FQL



$$\mathtt{Emp.manager.worksIn} = \mathtt{Emp.worksIn} \qquad \mathtt{Dept.secretary.worksIn} = \mathtt{Dept}$$

- Each black node represents an entity set (of IDs).
- Each directed edge represents a foreign key.
- Each open circle represent an attribute.
- Data integrity constraints are path equalities.
- Data is stored as tables in the obvious way.

Why FQL?

- FQL is a language for manipulating the schemas and instances just defined.

- But you can also manipulate such schemas and instances using SQL.

- We assert that, because of its categorical roots, FQL is a better language for doing so.
  - FQL is “database at a time”, not “table at a time”.
  - FQL operations necessarily respect constraints.
  - Unlike SQL, FQL is expressive enough to be used for information integration (see papers).
  - Parts of FQL can run on SQL, and vice versa.

FQL Basics

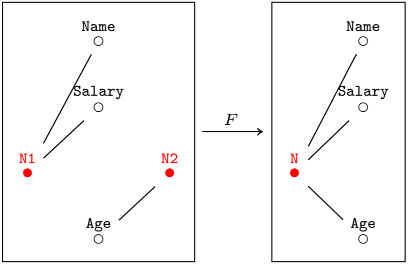
- A **schema mapping** $F : S \to T$ is a constraint-respecting mapping:

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

  and it induces three **data migration** operations:

  - $\Delta_F : T$-inst $\to S$-inst (like projection)
  - $\Sigma_F : S$-inst $\to T$-inst (like union)
  - $\Pi_F : S$-inst $\to T$-inst (like join)

∆ (Project)



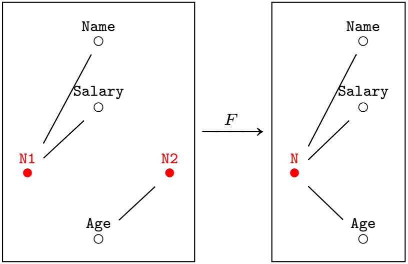
N1		
ID	**Name**	**Salary**
1	Bob	\$250
2	Sue	\$300
3	Alice	\$100

N2	
ID	**Age**
1	20
2	20
3	30

$\xleftarrow{\Delta_F}$

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

Π (Join)



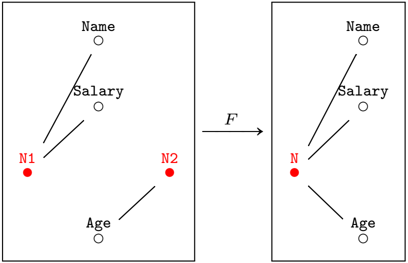
N1		
ID	**Name**	**Salary**
1	Bob	$250
2	Sue	$300
3	Alice	$100

N2	
ID	**Age**
1	20
2	20
3	30

$\xrightarrow{\Pi_F}$

N			
ID	**Name**	**Age**	**Salary**
1	Alice	20	$100
2	Alice	20	$100
3	Alice	30	$100
4	Bob	20	$250
5	Bob	20	$250
6	Bob	30	$250
7	Sue	20	$300
8	Sue	20	$300
9	Sue	30	$300

Σ (Union)



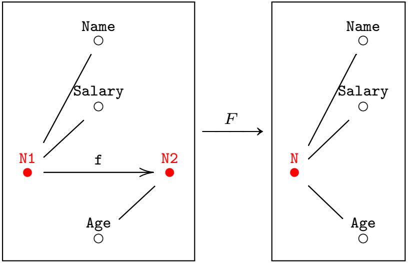
N1		
ID	**Name**	**Salary**
1	Bob	$250
2	Sue	$300
3	Alice	$100

N2	
ID	**Age**
1	20
2	20
3	30

$\xrightarrow{\Sigma_F}$

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

Foreign keys



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

N2	
ID	**Age**
1	20
2	20
3	30

$$\xleftarrow{\Delta_F}$$

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

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

FQL Summary

- FQL provides a “database at a time” query language for certain kinds of relational databases.

- For the categorically inclined, roughly:
  - Schemas are finitely-presented categories.
  - Schema mappings are functors.
  - Instances are functors to the category of sets.
  - The instances on any schema form a category.
  - (Σ_F, Δ_F) and (Δ_F, Π_F) are adjoint functors.

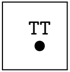
Programming FQL Schemas and Mappings using Haskell

- By Haskell, I mean the the simply-typed λ-calculus (STLC):
  - Types t:
  $$t ::= 0 \mid 1 \mid t + t \mid t \times t \mid t \to t$$
  - Expressions e:
  $$e ::= v \mid \lambda v : t.e \mid ee \mid () \mid fst\ e \mid snd\ e \mid (e, e) \mid \bot \mid inl\ e \mid inr\ e \mid (e + e)$$
  - Equations:
  $$fst(e, f) = e \qquad snd(e, f) = f \qquad (\lambda v : t.e)f = e[v \mapsto f] \qquad \ldots$$
- Theorem: FQL schemas and mappings are a model of the STLC.
  - Given an STLC type t, you get an FQL schema $[t]$.
  - Given an STLC term $\Gamma \vdash e : t$, you get an FQL schema mapping
  $$[e] : [\Gamma] \to [t]$$

Programming FQL Schemas using Haskell

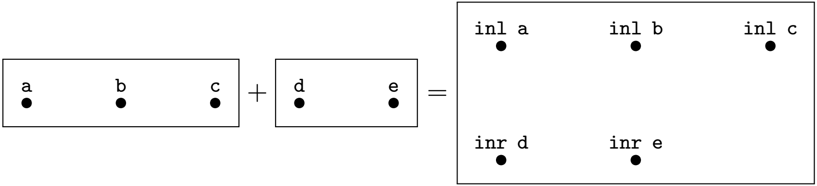
- The empty type, 0, (in Haskell, `data Empty = `), becomes a schema with no nodes:

- The unit type, 1, (in Haskell, `data Unit = TT`), becomes a schema with one node:

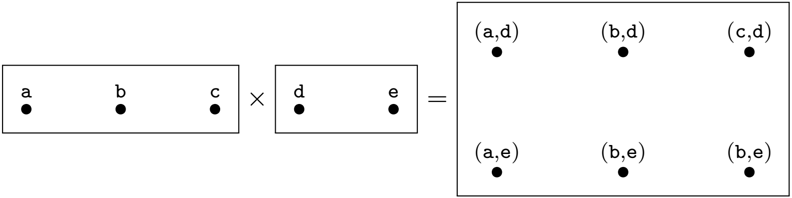


Programming FQL Schemas using Haskell

- Sum types, $t + t'$, (in Haskell, `Either t t'`), are given by addition:

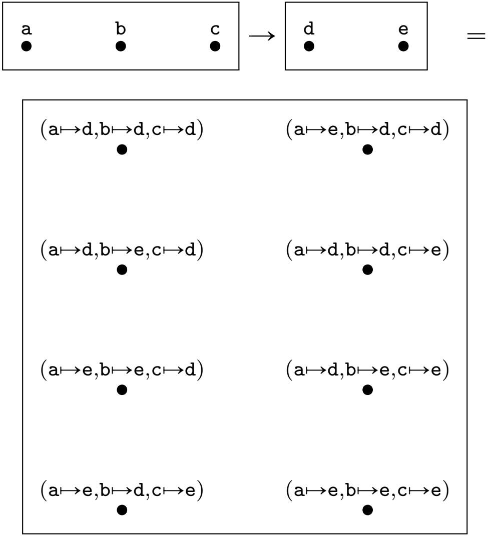


- Product types, $t \times t'$, (in Haskell, `(t,t')`), are given by multiplication:



Programming FQL Schemas using Haskell

- Function types, $t \to t'$ are given by exponentiation:



a b c → d e =

(a↦d,b↦d,c↦d)

(a↦e,b↦d,c↦d)

(a↦d,b↦e,c↦d)

(a↦d,b↦d,c↦e)

(a↦e,b↦e,c↦d)

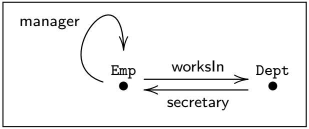
(a↦d,b↦e,c↦e)

(a↦e,b↦d,c↦e)

(a↦e,b↦e,c↦e)

Programming FQL Schemas using Haskell

- Constant types, corresponding to user defined types in Haskell, are simply schemas:

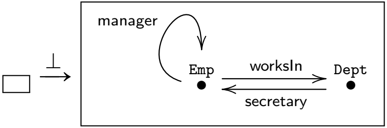


- The operations ×, +, → behave correctly with respect to foreign keys.

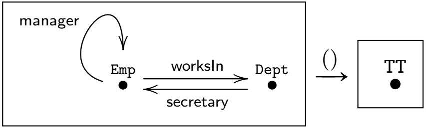
- Hence, STLC types translate to FQL schemas.

Programming FQL Mappings using Haskell

- In Haskell, we have $\bot :: a$. In FQL, we have a mapping $\bot : 0 \to a$:

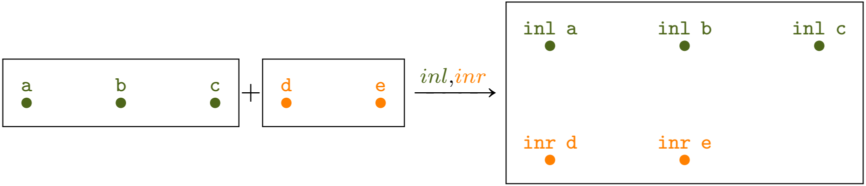


- In Haskell, we have $() :: 1$. In FQL, we have a mapping $() : a \to 1$:

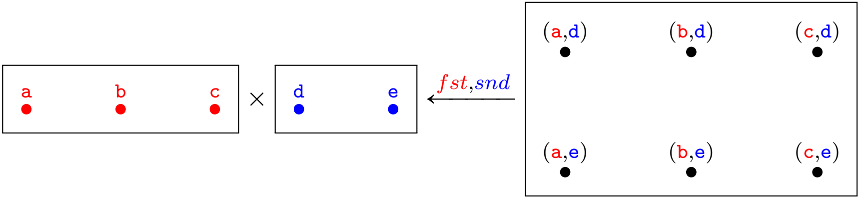


Programming FQL Mappings using Haskell

- In Haskell, we have $inl :: a \to a + b$ and $inr :: b \to a + b$.



- In Haskell, we have $fst :: a \times b \to a$ and $snd :: a \times b \to b$.



Programming FQL Mappings using Haskell

- We can translate the other STLC operations too:
  - If $f :: t \to a$ and $g :: t \to b$, we need $(f, g) :: t \to a \times b$.
    - This is pairing.
  - If $f :: a \to t$ and $g :: b \to t$, we need $(f + g) :: a + b \to t$.
    - This is `case`.
  - If $f :: a \times b \to c$, we need $\Lambda f : a \to (b \to c)$.
    - This is usually called `curry`.
  - We need $ev :: (a \to b) \times b \to a$.
    - This is function application.

- All FQL operations obey the required equations,

$$fst(a, b) = a \qquad snd(a, b) = b \qquad \ldots$$

- And the FQL operations work correctly with foreign keys.
- Hence, FQL mappings are a model of the STLC.

Retrospective

- STLC types and terms, FQL schemas and mappings, and even sets and functions between them, are all *bi-cartesian closed categories*.
- Haskell programmers will eventually encounter category theory, starting with bi-cartesian closed categories.
- That theory can be put to use in other places, namely databases.
- In fact, as we will see next, for every FQL schema S, the category of S-instances is also bi-cartesian closed.

Programming FQL Instances and Morphisms using Haskell

- By Haskell, I mean the the simply-typed λ-calculus (STLC):
  - Types t:

    $$t ::= 0 \mid 1 \mid t + t \mid t \times t \mid t \to t$$

  - Expressions e:

    $$e ::= v \mid \lambda v : t.e \mid ee \mid () \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid (e, e) \mid \perp \mid \mathit{inl}\ e \mid \mathit{inr}\ e \mid (e + e)$$

  - Equations:

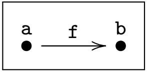
    $$\mathit{fst}(e, f) = e \qquad \mathit{snd}(e, f) = f \qquad (\lambda v : t.e)f = e[v \mapsto f] \qquad \ldots$$

- Theorem: For each schema S, the FQL S-instances and S-homomorphisms are a model of the STLC.
  - A database homomorphism is a map of IDs to IDs.
  - Given an STLC type t, you get an FQL S-instance $[t]$.
  - Given an STLC term $\Gamma \vdash e : t$, you get an FQL S-homomorphism

    $$[e] : [\Gamma] \to [t]$$

Programming FQL Instances using Haskell

- Let S be the schema



- The empty type, 0, (in Haskell, `data Empty = `), becomes an S instance with no data:

a	
ID	**f**

b
ID

- The unit type, 1, (in Haskell, `data Unit = TT`), becomes an S instance with one ID per table:

a	
ID	**f**
1	1

b
ID
1

Programming FQL Instances using Haskell

- Sum types $t + t'$ are given by disjoint union:

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**
inl 1	inl 3
inl 2	inl 3
inr a	inr c
inr b	inr c

b
ID
inl 3
inl 4
inr c
inr d

- Product types $t \times t'$ are given by joining:

a	
ID	**f**
1	3
2	3

b
ID
3
4

$\times$

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)

Programming FQL Instances using Haskell

- Function types $t \to t'$ are given by finding all homomorphisms:

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

Programming FQL Instances using Haskell

- Constant instances, corresponding to user defined types in Haskell, are simply instances:

a	
ID	**f**
p	q
r	t

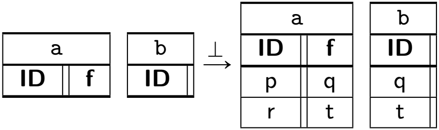
b	
ID	
q	
t	

- The operations $\times, +, \rightarrow$ behave correctly with respect to foreign keys.

- Hence, for every schema S, STLC types translate to S-instances.

Programming FQL Homomorphisms using Haskell

- in Haskell, we have $\perp :: a$. In FQL, we have a homomorphism $\perp : 0 \to a$:



a	
ID	**f**

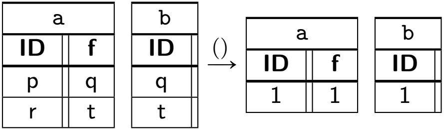
b
ID

$\xrightarrow{\perp}$

a	
ID	**f**
p	q
r	t

b
ID
q
t

- In Haskell, we have $() :: 1$. In FQL, we have a homomorphism $() : a \to 1$:



a	
ID	**f**
p	q
r	t

b
ID
q
t

$\xrightarrow{()}$

a	
ID	**f**
1	1

b
ID
1

Programming FQL Homomorphisms using Haskell

- As before, $inl : a \to a + b$ and $inr : b \to a + b$

a	
ID	**f**
1	3
2	3

b
ID
3
4

$+$

a	
ID	**f**
a	c
b	c

b
ID
c
d

$\xrightarrow{inl, inr}$

a	
ID	**f**
inl 1	inl 3
inl 2	inl 3
inr a	inr c
inr b	inr c

b
ID
inl 3
inl 4
inr c
inr d

- As before, $fst : a \times b \to a$ and $snd : a \times b \to b$

a	
ID	**f**
1	3
2	3

b
ID
3
4

$\times$

a	
ID	**f**
a	c
b	c

b
ID
c
d

$\xleftarrow{fst, snd}$

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)

Retrospective

- The language of FQL instances contains all operations required to be a model of the STLC.

- In fact, at the level of instances, FQL is a model of higher-order logic:

$$\text{types } t ::= \ldots \mid Prop$$

$$\text{expressions } e ::= \ldots \mid e = e$$

- The STLC structure interacts with the Δ, Σ, Π data migration operations in a nice way, e.g,:

$$\Sigma_F(I + J) = \Sigma_F(I) + \Sigma_F(J) \qquad \Pi_F(I \times J) = \Pi_F(I) \times \Pi_F(I)$$

Demo of the FQL IDE

- The FQL IDE is an open-source java application, downloadable at

  categoricaldata.net/fql.html

- It supports all the operations discussed above: $0, 1, +, \times, \rightarrow$ for schemas and instances, and the data migration operations Δ, Σ, Π.
- To the extent possible, all operations are implemented with SQL:
  - $0, 1, +, \times, \Delta, \Pi$ implemented with SQL.
  - Σ_F only implementable with SQL if F has a certain property.
  - → not implementable with SQL.
- Other features:
  - It translates from SQL to FQL.
  - It emits RDF encodings of instances.
  - It comes with many built-in examples.
  - It can be used as a command-line compiler.

Conclusion

- First, we talked about FQL, a functorial query language based on category theory.
  - Schemas are particular ER diagrams, and instances are relational tables.
  - The Δ, Σ, Π operations migrate data from one schema to another.

- FQL contains *two* copies of the STLC: one at the level of schemas and mappings, and one at the level of instances and homomorphisms.
  - Conclusion: Haskell, in the guise of the STLC, occurs in many areas of CS outside of programming.

- Finally, we saw a demo of the FQL IDE.
  - We are looking for collaborators: categoricaldata.net/fql.html