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.

\[
\begin{align*}
\text{Emp} & \xrightarrow{\text{manager}} \text{Emp} \\
\text{worksIn} & \xrightarrow{\text{secretary}} \text{Dept}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
\text{ID} & \text{mgr} & \text{works} & \text{first} & \text{last} \\
101 & 103 & q10 & Al & Akin \\
102 & 102 & x02 & Bob & Bo \\
103 & 103 & q10 & Carl & Cork \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\text{ID} & \text{sec} & \text{name} \\
q10 & 102 & CS \\
x02 & 101 & Math \\
\hline
\end{array}
\]

Emp.manager.worksIn = Emp.worksIn  \quad \text{Dept.secretary.worksIn} = \text{Dept}
Introduction to FQL

Emp \rightarrow \text{manager} \downarrow \downarrow \text{Dept}

Emp \text{.worksIn} \quad \text{Emp.worksIn} \quad \text{Emp.worksIn Dept}

Dept \text{.secretary.worksIn} = \text{Dept}

- Each black node represents an entity set (of IDs).
- Each directed edge represents a foreign key.
- Each open circle represents 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.
A schema mapping $F : S \rightarrow T$ is a constraint-respecting mapping:

$$\text{nodes}(S) \rightarrow \text{nodes}(T) \quad \text{edges}(S) \rightarrow \text{paths}(T)$$

and it induces three data migration operations:

- $\Delta_F : T\text{-inst} \rightarrow S\text{-inst}$ (like projection)
- $\Sigma_F : S\text{-inst} \rightarrow T\text{-inst}$ (like union)
- $\Pi_F : S\text{-inst} \rightarrow T\text{-inst}$ (like join)
\( \Delta \) (Project)

\[
\begin{array}{c|c|c}
\text{ID} & \text{Name} & \text{Salary} \\
1 & Bob & $250 \\
2 & Sue & $300 \\
3 & Alice & $100 \\
\end{array}
\quad
\begin{array}{c|c}
\text{ID} & \text{Age} \\
1 & 20 \\
2 & 20 \\
3 & 30 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{ID} & \text{Name} & \text{Age} & \text{Salary} \\
1 & Bob & 20 & $250 \\
2 & Sue & 20 & $300 \\
3 & Alice & 30 & $100 \\
\end{array}
\]
Π (Join)

\[ \pi \]

\[ \pi (\text{Join}) \]

\[ \pi \]

\[ F \]

\[ \pi_F \]

\[ N \]

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>20</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>Alice</td>
<td>20</td>
<td>$100</td>
</tr>
<tr>
<td>3</td>
<td>Alice</td>
<td>30</td>
<td>$100</td>
</tr>
<tr>
<td>4</td>
<td>Bob</td>
<td>20</td>
<td>$250</td>
</tr>
<tr>
<td>5</td>
<td>Bob</td>
<td>20</td>
<td>$250</td>
</tr>
<tr>
<td>6</td>
<td>Bob</td>
<td>30</td>
<td>$250</td>
</tr>
<tr>
<td>7</td>
<td>Sue</td>
<td>20</td>
<td>$300</td>
</tr>
<tr>
<td>8</td>
<td>Sue</td>
<td>20</td>
<td>$300</td>
</tr>
<tr>
<td>9</td>
<td>Sue</td>
<td>30</td>
<td>$300</td>
</tr>
</tbody>
</table>
\( \Sigma \) (Union)

\[\begin{array}{c|c|c}
\text{ID} & \text{Name} & \text{Salary} \\
0 & 0 & 0 \\
1 & Bob & $250 \\
2 & Sue & $300 \\
3 & Alice & $100 \\
4 & null & null \\
5 & null & 20 \\
6 & null & 30 \\
\end{array} \]

\[\begin{array}{c|c|c|c}
\text{ID} & \text{Name} & \text{Age} & \text{Salary} \\
0 & 0 & 0 & 0 \\
1 & Bob & 20 & $250 \\
2 & Sue & 20 & $300 \\
3 & Alice & 20 & $100 \\
4 & null & 20 & null \\
5 & null & 30 & null \\
6 & null & 30 & null \\
\end{array} \]
Foreign keys

![Diagram showing foreign keys and relational operations]

<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>Bob</td>
<td>$250</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>$300</td>
</tr>
<tr>
<td>3</td>
<td>Alice</td>
<td>$100</td>
</tr>
</tbody>
</table>
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.
  - \((\Sigma_F, \Delta_F)\) and \((\Delta_F, \Pi_F)\) are adjoint functors.
By Haskell, I mean the simply-typed $\lambda$-calculus (STLC):

- **Types $t$:**
  $$t ::= 0 \mid 1 \mid t + t \mid t \times t \mid t \rightarrow t$$

- **Expressions $e$:**
  $$e ::= v \mid \lambda v : t. e \mid ee \mid () \mid \text{fst } e \mid \text{snd } e \mid (e, e) \mid \bot \mid \text{inl } e \mid \text{inr } e \mid (e + e)$$

- **Equations:**
  $$\text{fst}(e, f) = e \quad \text{snd}(e, f) = f \quad (\lambda v : t.e)f = e[v \mapsto f] \quad \ldots$$

- **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 $[e] : [\Gamma] \rightarrow [t]$.
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:

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\bullet & \bullet & \bullet
\end{array} + \begin{array}{cc}
\text{d} & \text{e} \\
\bullet & \bullet
\end{array} = \begin{array}{ccc}
\text{inl} \ a & \text{inl} \ b & \text{inl} \ c \\
\bullet & \bullet & \bullet \\
\text{inr} \ d & \text{inr} \ e \\
\bullet & \bullet
\end{array}
\]

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

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\bullet & \bullet & \bullet
\end{array} \times \begin{array}{cc}
\text{d} & \text{e} \\
\bullet & \bullet
\end{array} = \begin{array}{ccc}
(a, d) & (b, d) & (c, d) \\
\bullet & \bullet & \bullet \\
(a, e) & (b, e) & (b, e) \\
\bullet & \bullet & \bullet
\end{array}
\]
Function types, $t \rightarrow t'$ are given by exponentiation:

```
(a\to d, b\to d, c\to d)   (a\to e, b\to d, c\to d)
  \bullet

(a\to d, b\to e, c\to d)   (a\to d, b\to d, c\to e)
  \bullet

(a\to e, b\to e, c\to d)   (a\to d, b\to e, c\to e)
  \bullet

(a\to e, b\to d, c\to e)   (a\to e, b\to e, c\to e)
  \bullet
```
Constant types, corresponding to user defined types in Haskell, are simply schemas:

\[ \text{Emp} \xrightarrow{\text{worksIn}} \text{Dept} \]

- The operations $\times, +, \rightarrow$ 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 \( \bot : 0 \to a \):

  \[
  \begin{array}{c}
  \text{manager} \\
  \downarrow \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Emp} \\
  \text{worksIn} \\
  \text{Dept} \\
  \text{secretary}
  \end{array}
  \]

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

  \[
  \begin{array}{c}
  \text{manager} \\
  () \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Emp} \\
  \text{worksIn} \\
  \text{Dept} \\
  \text{secretary}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{TT}
  \end{array}
  \]
In Haskell, we have \( \text{inl} :: a \rightarrow a + b \) and \( \text{inr} :: b \rightarrow a + b \).

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\bullet & \bullet & \bullet \\
\end{array}
\begin{array}{cc}
\text{d} & \text{e} \\
\bullet & \bullet \\
\end{array}
\xrightarrow{\text{inl,inr}}
\begin{array}{ccc}
\text{inl a} & \text{inl b} & \text{inl c} \\
\bullet & \bullet & \bullet \\
\end{array}
\begin{array}{cc}
\text{inr d} & \text{inr e} \\
\bullet & \bullet \\
\end{array}
\]

In Haskell, we have \( \text{fst} :: a \times b \rightarrow a \) and \( \text{snd} :: a \times b \rightarrow b \).

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\bullet & \bullet & \bullet \\
\end{array}
\begin{array}{cc}
\text{d} & \text{e} \\
\bullet & \bullet \\
\end{array}
\xleftarrow{\text{fst,snd}}
\begin{array}{ccc}
(a,d) & (b,d) & (c,d) \\
\bullet & \bullet & \bullet \\
\end{array}
\begin{array}{cc}
(a,e) & (b,e) & (c,e) \\
\bullet & \bullet & \bullet \\
\end{array}
\]
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,
  \[
  fst(a, b) = a \quad snd(a, b) = b \quad \ldots
  \]
- And the FQL operations work correctly with foreign keys.
- Hence, FQL mappings are a model of the STLC.
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.
By Haskell, I mean the simply-typed \(\lambda\)-calculus (STLC):

- **Types** \(t\):
  \[
  t ::= 0 \mid 1 \mid t + t \mid t \times t \mid t \rightarrow t
  \]

- **Expressions** \(e\):
  \[
  e ::= v \mid \lambda v : t. e \mid ee \mid () \mid fst e \mid snd e \mid (e, e) \mid \bot \mid \text{inl } e \mid \text{inr } e \mid (e + e)
  \]

- **Equations:**
  \[
  \text{fst}(e, f) = e \quad \text{snd}(e, f) = f \quad (\lambda v : t.e)f = e[v \mapsto f] \quad \ldots
  \]

**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

\[
[e] : [\Gamma] \rightarrow [t]
\]
Programming FQL Instances using Haskell

- Let $S$ be the schema

  ![Diagram of a schema with nodes labeled a, f, and b connected by an arrow from a to f to b.]

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

  ![Table showing a ID f and b ID with no content.]

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

  ![Table showing a ID f with 1 and b ID with 1.]
Programming FQL Instances using Haskell

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

- Product types $t \times t'$ are given by joining:
Programming FQL Instances using Haskell

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

$$
\begin{array}{c|c}
\text{a} & \text{ID} \\
\hline
1 & 3 \\
2 & 3 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
\text{b} & \text{ID} \\
\hline
3 & 4 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c|c|c}
\text{a} & \text{ID} & \text{f} \\
\hline
1 & a & b \\
2 & b & c \\
& c & d \\
\end{array}
\quad = \quad
\begin{array}{c|c|c|c}
\text{b} & \text{ID} \\
\hline
3 & c & 4 \\
\end{array}
\quad 
$$
Constant instances, corresponding to user defined types in Haskell, are simply instances:

<table>
<thead>
<tr>
<th>a</th>
<th>ID</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \bot :: a \). In FQL, we have a homomorphism \( \bot : 0 \to a \):

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

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

\[
\begin{array}{c|c}
\hline
a & () \\
ID & f \\
p & q \\
r & t \\
\hline
\end{array}
\quad
\quad
\begin{array}{c|c}
\hline
b & () \\
ID & q \\
q & t \\
\hline
\end{array}
\quad
\begin{array}{c|c}
\hline
a & () \\
ID & 1 \\
1 & 1 \\
\hline
\end{array}
\quad
\begin{array}{c|c}
\hline
b & () \\
ID & 1 \\
\hline
\end{array}
\]
Programming FQL Homomorphisms using Haskell

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

\[
\begin{array}{ccc}
| a | & | b | & + \\
| ID | f | | ID | f |
\end{array}
\quad
\begin{array}{cc}
| a | & | b |
| ID | f | | ID | f |
| a | | c |
| b | | c |
| c | | d |
\end{array}
\]

- As before, \( \text{fst} : a \times b \to a \) and \( \text{snd} : a \times b \to b \)

\[
\begin{array}{ccc}
| a | & | b | & \times \\
| ID | f | | ID |
\end{array}
\quad
\begin{array}{cc}
| a | & | b |
| ID | f | | ID |
| a | | c |
| b | | c |
| c | | d |
\end{array}
\]

\[
\begin{array}{ccc}
| a | & | b | & \to \\
| ID | f | | ID |
\end{array}
\quad
\begin{array}{cc}
| a | & | b |
| (1,a) | | (3,c) |
| (1,b) | | (3,c) |
| (2,a) | | (3,c) |
| (2,b) | | (3,c) |
\end{array}
\]

\[
\begin{array}{ccc}
| a | & | b | & \to \\
| ID | f | | ID |
\end{array}
\quad
\begin{array}{cc}
| a | & | b |
| (3,c) | | (3,d) |
| (4,c) | | (4,d) |
\end{array}
\]
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 $\Delta$, $\Sigma$, $\Pi$ data migration operations in a nice way, e.g.,

\[
\Sigma_F(I + J) = \Sigma_F(I) + \Sigma_F(J) \quad \Pi_F(I \times J) = \Pi_F(I) \times \Pi_F(I)
\]
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, +, ×, → for schemas and instances, and the data migration operations Δ, Σ, Π.

- To the extent possible, all operations are implemented with SQL:
  - 0, 1, +, ×, Δ, Π 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 $\Delta, \Sigma, \Pi$ 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