Functional Query Languages with Categorical Types

Ryan Wisnesky

November 2013

Introduction

- My dissertation concerns **functional query languages** – simply typed λ-calculi (STLC) extended with operations for data processing.

- Differences from functional programming languages:
  - Purely functional and total
  - Data processing operations chosen for efficiency
  - Optimization by cost-guided search through equivalent programs

- Traditional examples: Nested Relational Calculus, SQL/PSM
- NoSQL examples: Data Parallel Haskell, Links, LINQ, Jaql-Pig [MapReduce]

Outline

- Functional query languages with **categorical types** can do useful things that traditional functional query languages can't.
- By adding **a type of propositions** to STLC, we obtain a query calculus that is both higher-order and unbounded.
- By adding **identity types** to the STLC, we obtain a language where data integrity constraints can be expressed as types.
- By adding **types of categories** to STLC, we obtain a query language for a proposed successor to the relational model.

Chapter 1: Generalizing Codd's Theorem

- Adding a **type of propositions** to the STLC yields higher-order logic (HOL).
  - We prove that every hereditarily domain independent HOL program can be translated into the nested relational calculus (NRC).

- Why is this useful?
  - We obtain a query calculus that is **higher-order** (useful for complex objects) and has **unbounded comprehension** (useful for negation).

- Related work:

	Higher-order	First-order
Bounded	NRC (Wong)	RC (Codd)
Unbounded	HOL (this talk)	Set theory (Abiteboul)

Relational Calculus and Algebra

- A **relational calculus** expression is a first-order comprehension over relations:

$$\{ x_1, \ldots, x_n \mid FOL(x_1, \ldots, x_n) \}$$

- Projection: $\{ x \mid \exists y.R(x, y) \}$
- Cartesian product: $\{ x, y \mid R(x) \wedge R(y) \}$
- Composition: $\{ x, z \mid \exists y.R_1(x, y) \wedge R_2(y, z) \}$

- A **relational algebra** expression consists of $\sigma, \pi, \times, \cup, -$
- Composition: $\pi_{0,3}(\sigma_{1=2}(R_1 \times R_2))$
- Conjunctive queries: $\pi(\sigma(R_1 \times \ldots \times R_n))$

Codd's Theorem Example

- We will translate

$$\{ x \mid \forall y R(x,y) \} \quad = \quad \{ x \mid \neg\exists y \neg R(x,y) \}$$

- to relational algebra by constructing the **active domain** $adom$:

$$adom := \pi_1(R) \cup \pi_2(R)$$

$$\neg R(x,y) := adom \times adom - R$$

$$\exists y \neg R(x,y) := \pi_1 \ (adom \times adom - R)$$

$$\neg\exists y \neg R(x,y) := adom - \pi_1 \ (adom \times adom - R)$$

- The above query is independent of the quantification domain.
- When a query is not **domain independent**, the translation will change its semantics:

$$\{ x, y \mid \neg R(x,y) \} = dom \times dom - R \quad \neq \quad adom \times adom - R$$

Higher-order Logic and Nested Relational Calculus

- HOL and NRC types:

$$t ::= D \mid 1 \mid t \times t \mid t \to \texttt{prop} \mid \texttt{prop}$$

- Terms of HOL (= STLC + equality):

$$e ::= x \mid \lambda x : t.e \mid ee \mid () \mid (e, e) \mid e.1 \mid e.2 \mid e = e$$

- Terms of NRC + power set:

$$e ::= x \mid \mathit{for}\ x : t\ \mathit{in}\ e\ \mathit{where}\ e.\ \mathit{return}\ e \mid () \mid (e, e) \mid e.1 \mid e.2 \mid e = e$$

$$\mid \mathcal{P}e \mid \emptyset \mid \{e\} \mid e \cup e$$

- Key difference: HOL has **unbounded** comprehension with λ, NRC has **bounded** quantification with for.

HOL and NRC examples

- HOL abbreviations:

$$true := () = () \qquad \ldots$$

- Singleton set of e:

$$\lambda x : t.x = e \;\; (HOL) \qquad \{e\} \; (NRC)$$

- Empty set of type t:

$$\lambda x : t.false \;\; (HOL) \qquad \emptyset \; (NRC)$$

- Universal set of type t

$\lambda x : t.true \;\; (HOL)$ no NRC term - not domain independent

Translating HOL → NRC

- Basic idea of translation: bound all λs by active domain query.

$$\lambda x : t.e$$

$$\Rightarrow$$

$$\textit{for } x : t \textit{ in adom where } e. \textit{ return } x$$

- $adom$ is an NRC expression that computes the active domain.

Results

- Proving the correctness of the translation requires a lot of category theory.
- I could only prove the theorem for **hereditarily** domain independent programs.
  - My proof fails for this HOL program:

$$(\emptyset, \lambda x : t.true).1$$

  - Yet the translation is still correct.
- Mechanized the results in Coq.

Outline

- We study three types for functional query languages:
  - `Prop`, a type of propositions
  - `Id`, a type of identities
  - `Cat`, a type of categories

Chapter 2: Reifying Constraints as Identity Types

- Adding **identity types** to the STLC yields a language where data integrity constraints can be expressed as types.
  - We prove that the chase optimization procedure is sound in this language.
- Why is this useful?
  - A compiler can optimize queries by examining types.
- Identity types express equality of two terms:

$$t ::= 1 \mid t \times t \mid t \to t \mid e = e$$

$$e ::= x \mid \lambda x : t.e \mid \ldots \mid \texttt{refl}\ e : e = e$$

- Practical programming with identity types usually requires other dependent types as well (c.f., Coq, Agda, etc).

Motivation for constraints as types

- This query returns tuples (d, a) where a acted in a movie directed by d

$$\texttt{for}\ (m_1 \in \mathit{Movies})\ (m_2 \in \mathit{Movies})$$
$$\texttt{s.t.}\ m_1.\text{title} = m_2.\text{title}$$
$$\texttt{return}\ (m_1.\text{director}, m_2.\text{actor})$$

- Only when *Movies* satisfies the functional dependency title → director is the above query is equivalent to

$$\texttt{for}\ (m \in \mathit{Movies})$$
$$\texttt{return}\ (m.\text{director}, m.\text{actor})$$

- Goal: express constraints as identity types to enable this kind of **type-directed** optimization.

Embedded Dependencies (EDs)

- A functional dependency title → director means that if two Movies tuples agree on the title of a movie, they also agree on the director of that movie:

$$\texttt{forall}\ (x \in \text{Movies})\ (y \in \text{Movies})$$
$$\texttt{s.t.}\ x.\text{title} = y.\text{title},$$
$$\texttt{exists}\ \ -$$
$$\texttt{s.t.}\ x.\text{director} = y.\text{director}$$

- Constraints expressible in this ∀∃ form are called **embedded dependencies** (EDs).
  - By using the `exists` clause, EDs can express join decompositions, foreign keys, inclusions, etc.
- The **chase** procedure re-writes relational queries using EDs.

EDs as equalities

- An ED d:

$$\texttt{forall}\ v_1 \in R_i, \ldots\ \texttt{s.t.}\ P(v_1, \ldots),$$

$$\texttt{exists}\ u_1 \in R_k, \ldots\ \texttt{s.t.}\ P'(v_1, \ldots, u_1, \ldots)$$

  can be expressed as an equation between two comprehensions, $\mathit{front}(d)$ and $\mathit{back}(d)$:

$$\begin{array}{rcl}
\mathit{front}(d) & = & \mathit{back}(d) \\
\texttt{for}\ v_1 \in R_i, \ldots & & \texttt{for}\ v_1 \in R_i, \ldots, u_1 \in R_k, \ldots \\
\texttt{s.t.}\ P(v_1, \ldots) & & \texttt{s.t.}\ P(v_1, \ldots) \wedge P'(v_1, \ldots, u_1, \ldots) \\
\texttt{return}\ (v_1, \ldots) & & \texttt{return}\ (v_1, \ldots)
\end{array}$$

- **Key idea**: to express an ED d in a language with identity types, we use $\mathit{front}(d) = \mathit{back}(d)$.

Example ED as equality

$$\texttt{forall}\ (x \in \text{Movies})\ (y \in \text{Movies})$$
$$\texttt{s.t.}\ x.\text{title} = y.\text{title},$$
$$\texttt{exists}\ -$$
$$\texttt{s.t.}\ x.\text{director} = y.\text{director}$$

=

$$\texttt{for}\ (x \in \text{Movies})\ (y \in \text{Movies})$$
$$\texttt{s.t.}\ x.\text{title} = y.\text{title},$$
$$\texttt{return}\ (x, y)$$

$$=$$

$$\texttt{for}\ (x \in \text{Movies})\ (y \in \text{Movies})$$
$$\texttt{s.t.}\ x.\text{title} = y.\text{title} \land x.\text{director} = y.\text{director},$$
$$\texttt{return}\ (x, y)$$

Results

- The chase is sound for STLC + EDs as identity types.
  - Our paper proof follows (Popa, Tannen), but also holds for other kinds of structured sets, e.g., with probability annotations.

- In a dependently typed language like Coq, where types are first-class objects, programmers can manipulate data integrity constraints directly:

```
Definition q (I: set Movie) (pf: d I) := ...

Definition I : set Movies := ...
Theorem d_holds_on_I : d I := ...

Definition q_on_I := q I d_hold_on_I.
```

Outline

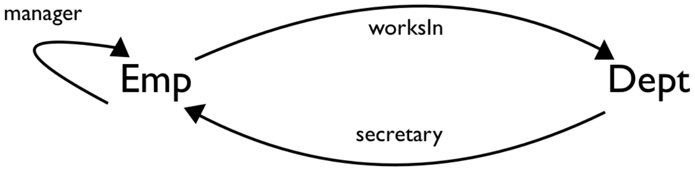
- We study three types for functional query languages:
  - `Prop`, a type of propositions
  - `Id`, a type of identities
  - `Cat`, a type of categories

Chapter 3: A Functorial Query Language

- Adding **types of categories** to the STLC yields a schema mapping language for the functorial data model (FDM).
  - We define FQL, a functional query language for the FDM, and compile it to SQL/PSM.
- The FDM (Spivak) is a proposed successor to the relational model, based on categorical foundations.
  - Naturally bag, ID, and graph based - unlike the relational model.
  - Many relational results still apply.
- Why is my work useful?
  - This works provides a practical deployment platform for the FDM (SQL), and establishes connections between the FDM and the relational model.

Functorial Schemas and Instances

- In the FDM (Spivak), database schemas are **finitely presented categories**. For example:



Emp.manager.worksIn = Emp.worksIn

Emp		
Emp	**manager**	**worksIn**
Alice	Chris	CS
Bob	Bob	Math
Chris	Chris	CS

Dept	
Dept	**secretary**
Math	Bob
CS	Alice

Functorial Data Migration

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

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

- A schema mapping $F : S \to T$ induces three **adjoint data migration functors**:
  - $\Delta_F : T - inst \to S - inst$ (like projection and selection)
  - $\Sigma_F : S - inst \to T - inst$ (like union)
  - $\Pi_F : S - inst \to T - inst$ (like join)

- Functorial data migrations have a powerful normal form:

$$\Sigma_F \circ \Pi_{F'} \circ \Delta_{F''}$$

FQL

- The category of schemas and mappings is cartesian closed.
  - The FDM's natural query language is the STLC + categories.

- Schemas T ($\mathcal{T}$ = finitely presented categories)

$$T ::= 1 \mid T \times T \mid T \to T \mid \mathcal{T}$$

- Mappings F ($\mathcal{F}$ = schema mappings)

$$F ::= x \mid \lambda x : T.F \mid FF \mid () \mid (F, F) \mid F.1 \mid F.2 \mid \mathcal{F}$$

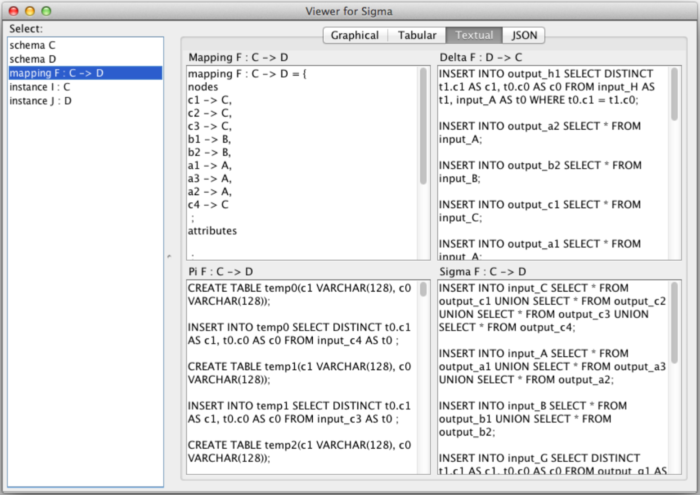
- T-Instances I ($\mathcal{I}$ = given database tables)

$$I ::= 1 \mid I \times I \mid I \to \texttt{prop} \mid \texttt{prop} \mid \Delta_F I \mid \Sigma_F I \mid \Pi_F I \mid \mathcal{I}$$

- T-Homomorphisms H

$$H ::= x \mid \lambda x : I.H \mid HH \mid () \mid (H, H) \mid H.1 \mid H.2 \mid H = H$$

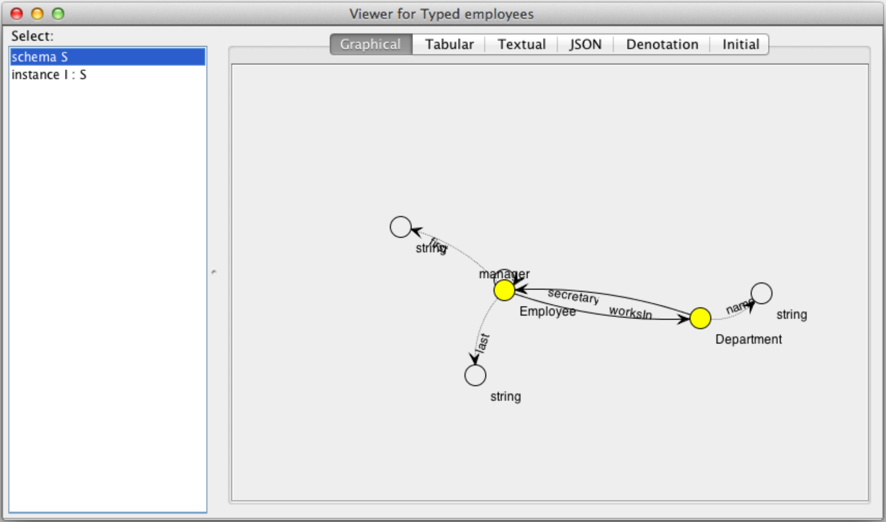
FQL Tutorial



FQL Schema Example

```
schema S = { nodes Employee, Department;

 attributes
  name  : Department -> string,
  first : Employee -> string,
  last  : Employee -> string;

 arrows
  manager   : Employee -> Employee,
  worksIn   : Employee -> Department,
  secretary : Department -> Employee;

 equations
  Employee.manager.worksIn = Employee.worksIn,
  Department.secretary.worksIn = Department,
  Employee.manager.manager = Employee.manager;
}
```

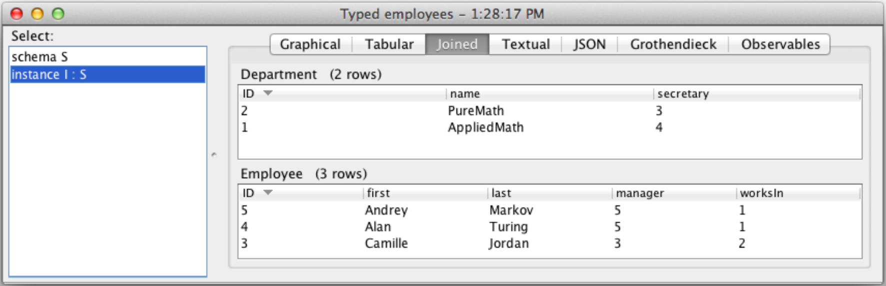
FQL Schema Viewer Example



FQL Instance Example

```
instance I : S = {
 nodes
  Employee -> {101, 102, 103},
  Department -> {q10, x02};

 attributes
  first -> {(101, Alan), (102, Camille), (103, Andrey)},
  last  -> {(101, Turing), (102, Jordan), (103, Markov)},
  name  -> {(q10, AppliedMath), (x02, PureMath)};

 arrows
  manager -> {(101, 103), (102, 102), (103, 103)},
  worksIn -> {(101, q10), (102, x02), (103, q10)},
  secretary -> {(q10, 101), (x02, 102)};
}
```

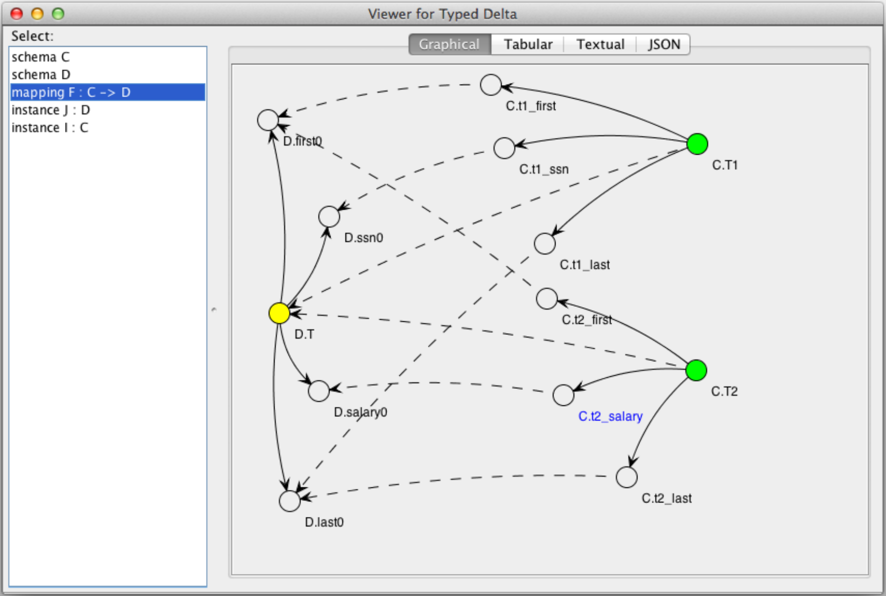
FQL Instance Viewer



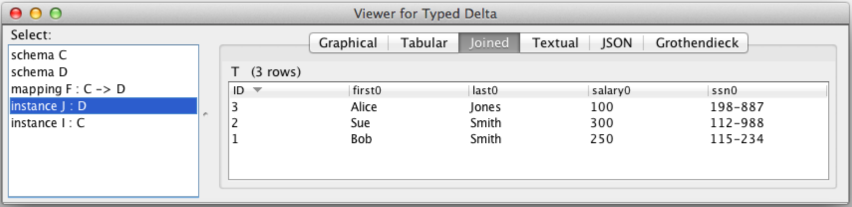
FQL Mapping Example

```
schema C = {
nodes T1, T2;
attributes
t1_ssn:T1->string,t1_first:T1->string,t1_last:T1->string,
t2_first:T2->string,t2_last:T2->string,t2_salary:T2->int;}

schema D = {
nodes T;
attributes
 ssn0  : T -> string, first0  : T -> string,
 last0: T -> string, salary0 : T -> int; }

mapping F : C -> D = {
nodes T1 -> T, T2 -> T;
attributes
t1_ssn->ssn0, t1_first->first0, t1_last->last0,
t2_last->last0, t2_salary->salary0, t2_first->first0; }
```

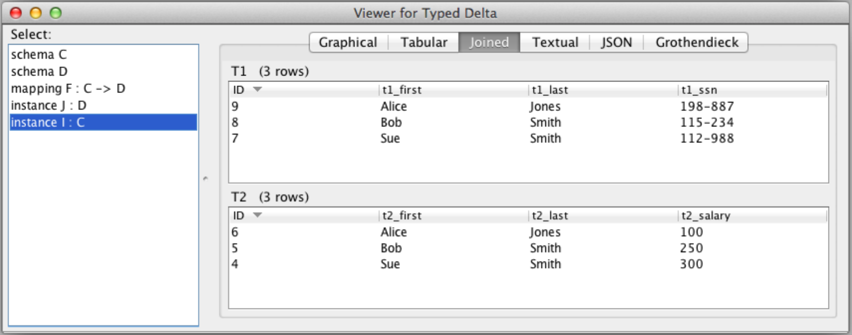
FQL Schema Mapping Viewer Example



Delta (Project and Select)



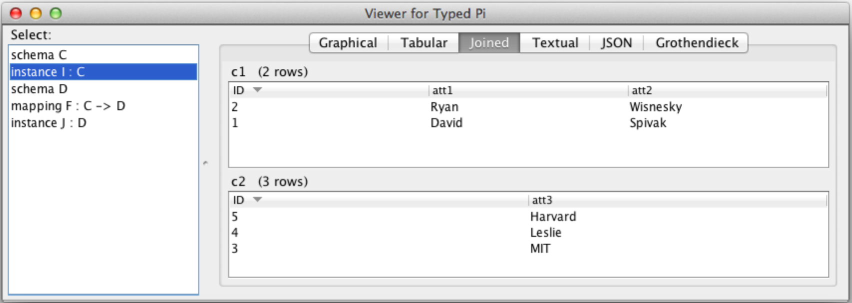
ID	first0	last0	salary0	ssn0
3	Alice	Jones	100	198-887
2	Sue	Smith	300	112-988
1	Bob	Smith	250	115-234



ID	t1_first	t1_last	t1_ssn
9	Alice	Jones	198-887
8	Bob	Smith	115-234
7	Sue	Smith	112-988

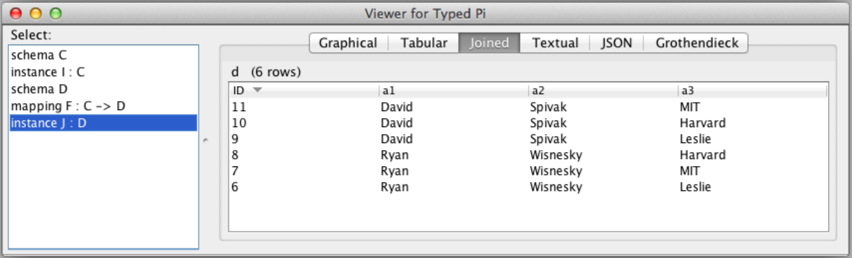
ID	t2_first	t2_last	t2_salary
6	Alice	Jones	100
5	Bob	Smith	250
4	Sue	Smith	300

Pi (Product)



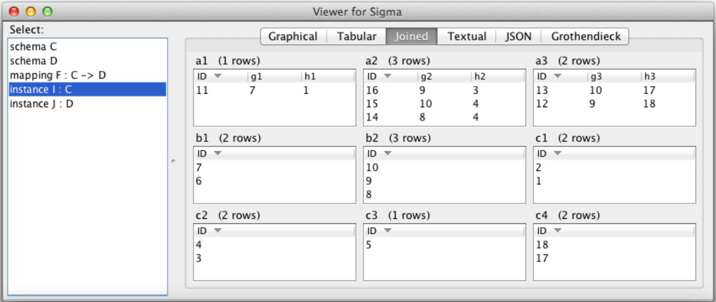
ID	att1	att2
2	Ryan	Wisnesky
1	David	Spivak

ID	att3
5	Harvard
4	Leslie
3	MIT



ID	a1	a2	a3
11	David	Spivak	MIT
10	David	Spivak	Harvard
9	David	Spivak	Leslie
8	Ryan	Wisnesky	Harvard
7	Ryan	Wisnesky	MIT
6	Ryan	Wisnesky	Leslie

Sigma (Union)



a1 (1 rows)

ID	g1	h1
11	7	1

a2 (3 rows)

ID	g2	h2
16	9	3
15	10	4
14	8	4

a3 (2 rows)

ID	g3	h3
13	10	17
12	9	18

b1 (2 rows)

ID
7
6

b2 (3 rows)

ID
10
9
8

c1 (2 rows)

ID
2
1

c2 (2 rows)

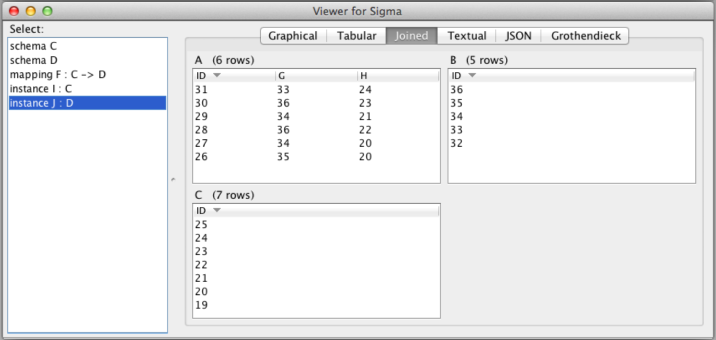
ID
4
3

c3 (1 rows)

ID
5

c4 (2 rows)

ID
18
17



A (6 rows)

ID	G	H
31	33	24
30	36	23
29	34	21
28	36	22
27	34	20
26	35	20

B (5 rows)

ID
36
35
34
33
32

C (7 rows)

ID
25
24
23
22
21
20
19

Recap for FQL

- The functorial data model (FDM) is a proposed categorical alternative to the relational model.
  - Naturally bag, ID, and graph based (unlike the relational model)
- Many relational results still apply:
  - Every conjunctive query under bag semantics is expressible.
  - Unions of conjunctive queries are still a normal form.
- I propose FQL, the first query language for the functorial data model, and demonstrate how to compile it to SQL.
  - Provides a practical deployment platform for the FDM, and connects the FDM to relational database theory.

Conclusion

- Functional query languages with categorical types can do useful things traditional functional query languages cannot:

- STLC + `Prop` (= HOL).
  - Result: a translation to the nested relational calculus.
  - Why: obtain a higher-order, unbounded query calculus.
  - Future work: generalize the soundness proof.

- STLC + `Id` ($\subseteq$ Coq, Agda, NuPrl, etc)
  - Result: soundness of the chase.
  - Why: to optimize/program constrained databases in e.g., Coq.
  - Future work: implement the chase as a Coq plug-in.

- STLC + `Cat` (= FQL)
  - Result: SQL compiler for FQL.
  - Why: connect FQL to database theory.
  - Future work: updates, aggregation, negation.