Functional Query Languages with Categorical Types

Ryan Wisnesky

November 2013
Introduction

▶ My dissertation concerns **functional query languages** – simply typed $\lambda$-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]
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:
  
<table>
<thead>
<tr>
<th></th>
<th>Higher-order</th>
<th>First-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded</td>
<td>NRC (Wong)</td>
<td>RC (Codd)</td>
</tr>
<tr>
<td>Unbounded</td>
<td>HOL (this talk)</td>
<td>Set theory (Abiteboul)</td>
</tr>
</tbody>
</table>
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: \( \{ x, y \mid R(x) \land R(y) \} \)
- Composition: \( \{ x, z \mid \exists y. R_1(x, y) \land R_2(y, z) \} \)

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

\[ \{ x \mid \forall y R(x, y) \} = \{ x \mid \neg \exists y \neg R(x, y) \} \]

- to relational algebra by constructing the active domain \( adom \):

\[
adom := \pi_1(R) \cup \pi_2(R) \\
\neg R(x, y) := \text{adom} \times \text{adom} - R \\
\exists y \neg R(x, y) := \pi_1(\text{adom} \times \text{adom} - R) \\
\neg \exists y \neg R(x, y) := \text{adom} - \pi_1(\text{adom} \times \text{adom} - R)
\]

- The above query is independent of the quantification domain.

- When a query is not \textbf{domain independent}, the translation will change its semantics:

\[
\{ x, y \mid \neg R(x, y) \} = \text{dom} \times \text{dom} - R \neq \text{adom} \times \text{adom} - R
\]
Higher-order Logic and Nested Relational Calculus

- **HOL and NRC types:**
  \[ t ::= D \mid 1 \mid t \times t \mid t \to \text{prop} \mid \text{prop} \]

- **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:**
  \[ e ::= x \mid \text{for } x : t \text{ in } e \text{ where } e. \text{ return } e \mid () \mid (e, e) \mid e.1 \mid e.2 \mid e = e \]
  \[ \mid \mathcal{P}e \mid \emptyset \mid \{e\} \mid e \cup e \]

- **Key difference:** HOL has **unbounded** comprehension with \( \lambda \), NRC has **bounded** quantification with \( \text{for} \).
HOL and NRC examples

- HOL abbreviations:

\[ true := () = () \ldots \]

- Singleton set of \( e \):

\[ \lambda x : t. x = e \ (HOL) \quad \{ e \} \ (NRC) \]

- Empty set of type \( t \):

\[ \lambda x : t. \text{false} \ (HOL) \quad \emptyset \ (NRC) \]

- Universal set of type \( t \)

\[ \lambda x : t. \text{true} \ (HOL) \quad \text{no NRC term - not domain independent} \]
Basic idea of translation: bound all $\lambda$s by active domain query.

$$\lambda x : t.e$$

$$\Rightarrow$$

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

$adom$ 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:

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

    - Yet the translation is still correct.

- Mechanized the results in Coq.
We study three types for functional query languages:

- Prop, a type of propositions
- Id, a type of identities
- Cat, a type of categories
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:

\[
t ::= 1 \mid t \times t \mid t \rightarrow t \mid e = e
\]

\[
e ::= x \mid \lambda x : t.e \mid \ldots \mid \text{refl } e : e = e
\]

- 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 \(d\)

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

- Only when \(\text{Movies}\) satisfies the functional dependency \(\text{title} \rightarrow \text{director}\) is the above query is equivalent to

\[
\text{for } (m \in \text{Movies}) \\
\text{return } (m.\text{director}, m.\text{actor})
\]

- Goal: express constraints as identity types to enable this kind of \textbf{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:

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

- 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 $d$:

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

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

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

  $\text{front}(d) = \text{back}(d)$

  for $v_1 \in R_i, \ldots$

  s.t. $P(v_1, \ldots)$

  return $(v_1, \ldots)$

  for $v_1 \in R_i, \ldots, u_1 \in R_k, \ldots$

  s.t. $P(v_1, \ldots) \land P'(v_1, \ldots, u_1, \ldots)$

  return $(v_1, \ldots)$

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

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

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

\[
\text{for } (x \in \text{Movies}) (y \in \text{Movies}) \\
\text{s.t. } x.\text{title} = y.\text{title} \land x.\text{director} = y.\text{director}, \\
\text{return } (x, y)
\]
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 $q (I : \text{set Movie}) (pf : d I) := \ldots$

Definition $I : \text{set Movies} := \ldots$
Theorem $d\_\text{holds\_on\_I} : d I := \ldots$

Definition $q\_\text{on\_I} := q I d\_\text{hold\_on\_I}$. 
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:

<table>
<thead>
<tr>
<th>Emp</th>
<th>manager</th>
<th>worksIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Chris</td>
<td>CS</td>
</tr>
<tr>
<td>Bob</td>
<td>Bob</td>
<td>Math</td>
</tr>
<tr>
<td>Chris</td>
<td>Chris</td>
<td>CS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>secretary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math</td>
<td>Bob</td>
</tr>
<tr>
<td>CS</td>
<td>Alice</td>
</tr>
</tbody>
</table>

Emp.manager.worksIn = Emp.worksIn
Functorial Data Migration

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

\[
\text{nodes}(S) \to \text{nodes}(T) \quad \text{edges}(S) \to \text{paths}(T)
\]

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

- Functorial data migrations have a powerful normal form:

\[
\Sigma_F \circ \Pi_F' \circ \Delta_F''
\]
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)
  \[
  T ::= 1 \mid T \times T \mid T \rightarrow T \mid \mathcal{T}
  \]

- 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)
  \[
  I ::= 1 \mid I \times I \mid I \rightarrow \text{prop} \mid \text{prop} \mid \Delta_F I \mid \Sigma_F I \mid \Pi_F I \mid \mathcal{I}
  \]

- \( T \)-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 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
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)
Pi (Product)

---

<table>
<thead>
<tr>
<th>ID</th>
<th>att1</th>
<th>att2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Ryan</td>
<td>Wisnesky</td>
</tr>
<tr>
<td>1</td>
<td>David</td>
<td>Spivak</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>att3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Harvard</td>
</tr>
<tr>
<td>4</td>
<td>Leslie</td>
</tr>
<tr>
<td>3</td>
<td>MIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>David</td>
<td>Spivak</td>
<td>MIT</td>
</tr>
<tr>
<td>10</td>
<td>David</td>
<td>Spivak</td>
<td>Harvard</td>
</tr>
<tr>
<td>9</td>
<td>David</td>
<td>Spivak</td>
<td>Leslie</td>
</tr>
<tr>
<td>8</td>
<td>Ryan</td>
<td>Wisnesky</td>
<td>Harvard</td>
</tr>
<tr>
<td>7</td>
<td>Ryan</td>
<td>Wisnesky</td>
<td>MIT</td>
</tr>
<tr>
<td>6</td>
<td>Ryan</td>
<td>Wisnesky</td>
<td>Leslie</td>
</tr>
</tbody>
</table>
Sigma (Union)

```
<table>
<thead>
<tr>
<th>ID</th>
<th>g1</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>g2</th>
<th>h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>g3</th>
<th>h3</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>g1</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>g2</th>
<th>h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>g3</th>
<th>h3</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>18</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>30</td>
<td>36</td>
<td>23</td>
</tr>
<tr>
<td>29</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td>28</td>
<td>36</td>
<td>22</td>
</tr>
<tr>
<td>27</td>
<td>34</td>
<td>20</td>
</tr>
<tr>
<td>26</td>
<td>35</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>19</td>
</tr>
</tbody>
</table>
```
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 (⊆ 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.