Algebraic Databases

Patrick Schultz, David Spivak, Christina Vasilakopoulou
Department of Mathematics
Massachusetts Institute of Technology

Ryan Wisnesky
Categorical Informatics

NEPLS
October 7, 2016

$C_i$
In the early 2010s, Spivak proposed using the functorial data model (FDM) to solve data migration problems.

- Schemas are categories, instances are set-valued functors.
- A precursor to the FDM was proposed by Rosebrugh in the early 2000s.

It turns out the FDM can be understood as algebraic specification.

Talk goal: describe the FDM using algebraic terminology.

The FDM is being commercialized in a data integration tool by an MIT spin-out company. Both the lab and the company are looking for collaborators. (catinf.com)

Sponsored by ONR grant N000141310260, AFOSR grant FA9550-14-1-0031, NIST SBIR grants 70NANB15H290 and 70NANB16H178, and NSF I-Corps grant 1611699.
A **signature** $\text{Sig}$ consists of a set of **sorts** and a set of **function symbols**

$$f : s_1 \times \ldots \times s_n \to s$$

A **context** $\Gamma$ over signature $\text{Sig}$ assigns variables to sorts

$$v_1 : s_1, \ldots, v_k : s_k$$

A **term** of sort $s$ in context $\Gamma$ is either

- a variable $v$, if $v : s \in \Gamma$
- a function application $f(t_1, \ldots, t_n)$, if $f : s_1 \times \ldots \times s_n \to s$ and each $t_i$ is a term of sort $s_i$

We write $t[v \mapsto t']$ for the substitution of term $t'$ for variable $v$ in term $t$. 
Let $\text{Sig}$ be a signature. A (universally quantified) equation over $\text{Sig}$ is a formula $\Gamma. t = t' : s$, where $t, t'$ have sort $s$ in context $\Gamma$. A set of equations $\text{Th}$ is a theory. The entailment relation $\text{Th} \models$ between equations is defined by inference rules

\[
\begin{align*}
\Gamma. t = t : s & \quad \Gamma. t' = t : s \\
\Gamma. t = t' : s & \quad \Gamma. t' = t'' : s \\
\Gamma. t = t' : s & \quad \Gamma. e = e' : s \\
\Gamma, v : s'. t = t' & \quad v \notin \Gamma \\
\Gamma, v : s'. t = t' : s & \quad \Gamma. t[v \mapsto e] = t'[v \mapsto e'] : s'
\end{align*}
\]
Review of Algebraic Theories (Theory Morphisms)

- A morphism of signatures $F : \text{Sig}_1 \to \text{Sig}_2$ consists of
  - a function from sorts in $\text{Sig}_1$ to sorts in $\text{Sig}_2$
  - a function from symbols $f : s_1 \times \ldots \times s_n \to s$ in $\text{Sig}_1$ to (open) terms $F(s_1) \times \ldots \times F(s_n) \to F(s)$ in $\text{Sig}_2$

- A morphism of theories $F : \text{Th}_1 \to \text{Th}_2$ is a morphism of signatures that preserves provable equality of terms

$$\text{Th}_1 \vdash v_1 : s_1, \ldots, v_n : s_n. \ t_1 = t_2 : s$$

implies

$$\text{Th}_2 \vdash v_1 : F(s_1), \ldots, v_n : F(s_n). \ F(t_1) = F(t_2) : F(s)$$
An algebra $A$ over signature $\text{Sig}$ consists of
- a set $A(s)$ for each sort $s$
- a function $A(f) : A(s_1) \times \ldots \times A(s_k) \rightarrow A(s)$ for each symbol $f : s_1 \times \ldots \times s_k \rightarrow s$

An environment $\eta$ for context $\Gamma$ takes each $v : s \in \Gamma$ to some $A(s)$.

We write $A\llbracket t\rrbracket \eta$ for the meaning of term $t$ in environment $\eta$.

$A$ is a model of a theory $Th$ ($Th \models A$) when $Th \vdash \Gamma. t = t' : s$ implies $A\llbracket t\rrbracket \eta = A\llbracket t'\rrbracket \eta$ for all terms $t, t'$ and environments $\eta$.

A morphism of $\text{Sig}$-algebras $h : A \rightarrow B$ is a family of functions $h(s) : A(s) \rightarrow B(s)$ such that

$$h(s)(A(f)(a_1, \ldots, a_n)) = B(f)(h(s_1)(a_1), \ldots, h(s_n)(a_n))$$

for every symbol $f : s_1 \times \ldots \times s_n \rightarrow s$ and $a_i \in A(s_i)$. 
Review of Algebraic Theories (Key Properties)

- Entailment is semi-decidable.

- Deduction is sound and complete.

- Every theory $Th$ admits a term model $M$:
  - $M(s)$ is the set of ground terms of sort $s$, modulo $Th \vdash$.
  - $M$ is initial: for every $M' \models Th$, there is a unique $M \to M'$.
  - Construction of $M$ is semi-computable.

- Algebraic theories are presentations of cartesian multi-categories.
A *typeside* \( Ty \) is an algebraic theory.
- Its sorts are called *types*.
- It represents an ambient computational context.
Example typeside

- **Types**
  
  Nat, Char, String

- **Symbols**


  A, B, C, ..., Z : Char

  nil: String, cons: Char × String → String

- **Equations**

  \[ \forall x. \,(zero, x) = x \]

  \[ \forall x, y. \,(succ(x), y) = succ(+(x, y)) \]
Algebraic Databases (Schemas)

- A *typeside* $T_y$ is an algebraic theory.
  - Its sorts are called *types*.
  - It represents an ambient computational context.

- A *schema* $S$ on $T_y$ extends $T_y$ with
  - new sorts (called *entities*).
  - new symbols $att : entity \rightarrow type$ (called *attributes*).
  - new symbols $fk : entity \rightarrow entity$ (called *foreign keys*).
  - new unary equations of the form $\forall v : entity. e = e'$.
Example Schema

- **Entities**
  - Emp, Dept

- **Foreign Keys**
  - manager: Emp → Emp, works: Emp → Dept, secretary: Dept → Emp

- **Attributes**
  - dname: Dept → String, ename: Emp → String

- **Equations**
  \[
  \forall e. \text{works(manager}(e)) = \text{works}(e)
  \]
  \[
  \forall d. \text{works(secretary}(d)) = d
  \]
  \[
  \forall e. \text{manager(manager}(e)) = \text{manager}(e)
  \]
Algebraic Databases (Instances)

- A *typeside* $Ty$ is an algebraic theory.
  - Its sorts are called *types*.
  - It represents an ambient computational context.

- A *schema* $S$ on $Ty$ extends $Ty$ with
  - new sorts (called *entities*).
  - new symbols $att : entity \rightarrow type$ (called *attributes*).
  - new symbols $fk : entity \rightarrow entity$ (called *foreign keys*).
  - new unary equations of the form $\forall v : entity. e = e'$.

- An *instance* $I$ on $S$ extends $S$ with
  - new symbols $gen : entity$ (called *generators*).
  - new symbols $sk : type$ (called *labelled nulls / skolem variables*).
  - new non-quantified equations.
Example Instance

- Generators
  
  \[ a, b, c : \text{Emp}, \quad m, s : \text{Dept} \]

- Equations
  
  \[
  \text{name}(a) = \text{Al}, \quad \text{name}(c) = \text{Carl}, \quad \text{name}(m) = \text{Math} \\
  \text{works}(a) = m, \quad \text{works}(b) = m, \quad \text{secretary}(s) = c, \quad \text{secretary}(m) = b
  \]

Abbreviated \( \text{cons}(A, \text{cons}(l, \text{nil})) \) as Al, etc.
### Dept

<table>
<thead>
<tr>
<th>ID</th>
<th>dname</th>
<th>secretary</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Math</td>
<td>b</td>
</tr>
<tr>
<td>s</td>
<td>dname(s)</td>
<td>c</td>
</tr>
</tbody>
</table>

### Emp

<table>
<thead>
<tr>
<th>ID</th>
<th>ename</th>
<th>manager</th>
<th>works</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Al</td>
<td>mgr(a)</td>
<td>m</td>
</tr>
<tr>
<td>b</td>
<td>ename(b)</td>
<td>mgr(b)</td>
<td>m</td>
</tr>
<tr>
<td>c</td>
<td>Carl</td>
<td>mgr(c)</td>
<td>s</td>
</tr>
<tr>
<td>mgr(a)</td>
<td>ename(mgr(a))</td>
<td>mgr(a)</td>
<td>m</td>
</tr>
<tr>
<td>mgr(b)</td>
<td>ename(mgr(b))</td>
<td>mgr(b)</td>
<td>m</td>
</tr>
<tr>
<td>mgr(c)</td>
<td>ename(mgr(c))</td>
<td>mgr(c)</td>
<td>s</td>
</tr>
</tbody>
</table>
Functorial Data Migration

- Let $S, T$ be two schemas on typeside $Ty$. A *morphism* $F: S \to T$ is a morphism of theories that is the identity on $Ty$.
  - The schemas on $Ty$ form a category.

- Let $I, J$ be two instances on schema $S$. A *morphism* $h: I \to J$ is a morphism of theories that is the identity on $S$.
  - The instances on $S$ form a category, $S$-inst.

- A morphism $F: S \to T$ induces adjoint data migration functors
  - $\Sigma_F: S$-inst $\to$ $T$-inst (like outer disjoint union then quotient)
  - defined as substitution
  - $\Delta_F: T$-inst $\to$ $S$-inst (like project)
    \[ \Sigma_F \dashv \Delta_F \]
  - $\Pi_F: S$-inst $\to$ $T$-inst (like join)
    \[ \Delta_F \dashv \Pi_F \]

Note: adjoints are only defined up to unique isomorphism.
Graphical (E/R Diagram) Notation for Schemas

\[
\text{Emp.manager.works = Emp.works} \quad \text{Dept.secretary.works = Dept} \\
\text{Emp.manager.manager = Emp.manager}
\]
In these examples we show instances as term models rather than theories. IDs are meaningless – instances are only defined up to isomorphism.
Π (Join)

N

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>Alice</td>
<td>$100</td>
<td>30</td>
</tr>
<tr>
<td>d</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>e</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>f</td>
<td>Bob</td>
<td>$250</td>
<td>30</td>
</tr>
<tr>
<td>g</td>
<td>Sue</td>
<td>$300</td>
<td>20</td>
</tr>
<tr>
<td>h</td>
<td>Sue</td>
<td>$300</td>
<td>20</td>
</tr>
<tr>
<td>i</td>
<td>Sue</td>
<td>$300</td>
<td>30</td>
</tr>
</tbody>
</table>
\( \sum \) (Outer Disjoint Union then Quotient)

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

\[
\begin{array}{c|c|c}
    \text{ID} & \text{Age} \\
    4 & 20 \\
    5 & 20 \\
    6 & 30 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
    \text{ID} & \text{Name} & \text{Salary} & \text{Age} \\
    a & Alice & $100 & \text{null}_1 \\
    b & Bob & $250 & \text{null}_2 \\
    c & Sue & $300 & \text{null}_3 \\
    d & \text{null}_4 & \text{null}_5 & 20 \\
    e & \text{null}_6 & \text{null}_7 & 20 \\
    f & \text{null}_8 & \text{null}_9 & 30 \\
\end{array}
\]

\((\text{null}_1 \text{ abbreviates Age}(a), \text{ etc.})\)
Foreign keys

### N1

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td>6</td>
</tr>
</tbody>
</table>

### N2

<table>
<thead>
<tr>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

### N

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Alice</td>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>Bob</td>
<td>$250</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>Sue</td>
<td>$300</td>
<td>30</td>
</tr>
</tbody>
</table>
Expressivity of $\Delta, \Sigma, \Pi$

- Data migrations of the form

$$\Sigma_F \circ \Delta_G \circ \Pi_H$$

can express any SPCU relational algebra query under \textbf{bag} semantics. SPCU under \textbf{set} semantics can express the data migration when

- $F$ is a discrete op-fibration (ensures union compatibility).
- $H$ is a surjection on attributes (ensures domain independence).
- All theories denote finite categories (ensures computability).
- The typeside has no function symbols (ensures atomicity of data).
- We extend SPCU with a key generator (need fresh constants).

- $\Sigma_F$ has similar semantics to an operation called the \textit{chase} which is the basis of relational data integration.

- Migrations of the form $\Delta_F \circ \Pi_G$ and $\Sigma_G \circ \Delta_F$ can be specified using for/where/return syntax.
Pivot (Instance ↔ Schema)

Emp

<table>
<thead>
<tr>
<th>ID</th>
<th>mgr</th>
<th>works</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>103</td>
<td>q10</td>
<td>Al</td>
<td>Akin</td>
</tr>
<tr>
<td>102</td>
<td>102</td>
<td>x02</td>
<td>Bob</td>
<td>Bo</td>
</tr>
<tr>
<td>103</td>
<td>103</td>
<td>q10</td>
<td>Carl</td>
<td>Cork</td>
</tr>
</tbody>
</table>

Dept

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>q10</td>
<td>CS</td>
</tr>
<tr>
<td>x02</td>
<td>Math</td>
</tr>
</tbody>
</table>
QINL vs LINQ

- Functorial data migration is a QINL (i.e., co-LINQ) query mechanism.

- LINQ enriches programs with (schemas, queries and instances).
  - Collections are terms

    \[
    \text{Employee: Set Int} \quad \text{manager: Set (Int} \times \text{Int)}
    \]

    - e: Employee is not a judgment.
    - There is a term \( \in : \text{Int} \times \text{Set Int} \rightarrow \text{Bool} \).

- QINL enriches (schemas, queries and instances) with programs.
  - Collections are types

    \[
    \text{Employee: Type} \quad \text{manager: Employee} \rightarrow \text{Employee}
    \]

    - e: Employee is a judgment.
    - There is not a term \( \in : \text{Employee} \times \text{Type} \rightarrow \text{Bool} \).

- LINQ is more popular, but QINL is common in Coq, Agda, etc.
QINL is “one level up” from LINQ

- LINQ
  - Schemas are collection types over a base type theory
    \[
    \text{Set} (\text{Int} \times \text{String})
    \]
  - Instances are terms
    \[
    \{(1, \text{CS})\} \cup \{(2, \text{Math})\}
    \]
  - Data migrations are functions
    \[
    \pi_1 : \text{Set} (\text{Int} \times \text{String}) \rightarrow \text{Set} \text{Int}
    \]

- QINL
  - Schemas are type theories over a base type theory
    \[
    \text{Dept, name} : \text{Dept} \rightarrow \text{String}
    \]
  - Instances are term models
    \[
    d_1, d_2 : \text{Dept, name}(d_1) = \text{CS}, \text{name}(d_2) = \text{Math}
    \]
  - Data migrations are functors
    \[
    \Delta_{\text{Dept}} : (\text{Dept, name} : \text{Dept} \rightarrow \text{String})-\text{inst} \rightarrow (\text{Dept})-\text{inst}
    \]
The FQL IDE is an open-source graphical schema mapping and data integration tool available at
categoricaldata.net/fql.html

It is being commercialized by Categorical Informatics, a recent MIT spin out
catinf.com
Conclusion

- I presented an expressive formalism for specifying and manipulating databases using algebraic theories.

- Many concepts from algebraic specification appear in this work. Some not mentioned:
  - Conservative extensions / consistency
  - Institutions, limits, colimits, etc
  - Automated theorem proving

- Looking for feedback, users, and collaborators.

  catinf.com