Abstract

In this paper we develop an algebraic approach to data integration by combining techniques from functional programming, category theory, and database theory. In our formalism, database schemas and instances are algebraic (multi-sorted equational) theories of a certain form. Schemas denote categories, and instances denote their initial (term) algebras. The instances on a schema $S$ form a category, $S$–Inst, and a morphism of schemas $F : S \to T$ induces three adjoint data migration functors: $\Sigma_F : S$–Inst $\to T$–Inst, defined by substitution along $F$, which has a right adjoint $\Delta_F : T$–Inst $\to S$–Inst, which in turn has a right adjoint $\Pi_F : S$–Inst $\to T$–Inst. We present a query language based on for/where/return syntax where each query denotes a sequence of data migration functors; a pushout-based design pattern for performing data integration using our formalism; and describe the implementation of the formalism in a tool we call FQL.

1 Introduction

In this paper we develop an algebraic approach to data integration by combining techniques from functional programming, category theory, and database theory. By data integration we mean combining separate but related database schemas and instances to form a coherent, unified schema and instance, and we consider query and data migration to be special cases of data integration. By algebraic we mean that our schemas are algebraic (purely equational) theories and our instances denote algebras (models) of our schemas. We use category theory to define the semantics of our approach: schemas and instances form categories, and data integration operations are characterized with categorical constructions such as adjoint functors and pushouts. We use techniques from functional programming to implement our approach by constructing syntactic objects and reasoning about them, both on paper and using automated techniques. We use database theory as a baseline with which to compare our approach.

The mathematics of the semantics of our approach are worked out in detail in Schultz et al. (2017) using sophisticated category theory. In this paper, our goal is to implement a syntax for this semantics, and crucially, to do so in a computable way. How to do this is not obvious, because the mathematical objects defined in Schultz et al. (2017) are almost always infinite and not computable. This paper is a comprehensive description of the implementation of the FQL tool (http://categoricaldata.net/fql.html), an
open-source data integration tool capable of solving problems similar to those solved by relational data integration tools such as Clio (Haas et al., 2005) and Rondo (Melnik et al., 2003), as well as query languages such as SQL and LINQ (Grust, 2004).

Because our approach draws on functional programming, category theory, and database theory, the more knowledge a reader has about each of these fields the more the reader will get out of the paper. These three theories are used in a deep, rather than wide, way: we use mostly basic categorical concepts such as category, functor, natural transformation, and adjunction; we use mostly basic functional programming concepts such as equational logic and algebraic data types; and we use mostly basic database theory concepts such as relational conjunctive queries and labelled nulls. For this reason, we believe that a reader well-versed in only one of these areas can still get something out of this paper, and will be rewarded with a deeper insight into the other areas, at least from a data integration perspective. We include short primers on category theory (section 2) and equational logic (section 3), and connections to database theory are made as remarks in the text.

1.1 Background

Our data model extends a particular category-theoretic data model which we call the functorial data model (FDM) (Spivak, 2012). Originating in the late 1990s (Fleming et al., 2003), the FDM defines a schema to be a finitely-presented category (Barr & Wells, 1995): a directed, labelled multi-graph and a set of path equality constraints; and defines an instance on a schema $S$ to be a set-valued functor: a set of tables representing functions. An example FDM schema and instance about employees and departments is shown in figure 1.

![Diagram](image-url)

**Fig. 1. A Schema and Instance in the Functorial Data Model**
In the FDM, the database instances on a schema $S$ constitute a category, denoted $S$–Inst, and a functor (a.k.a schema mapping (Fagin et al., 2005b)) $F : S \rightarrow T$ between schemas $S$ and $T$ induces three adjoint data migration functors: $\Delta_F : T$–Inst $\rightarrow$ $S$–Inst, defined as $\Delta_F(I) := I \circ F$, and the left and right adjoints to $\Delta_F$, respectively: $\Sigma_F : S$–Inst $\rightarrow$ $T$–Inst and $\Pi_F : S$–Inst $\rightarrow$ $T$–Inst. The $\Sigma, \Delta, \Pi$ data migration functors provide a category-theoretic alternative to the traditional relational operations for both querying data (relational algebra) and migrating / integrating data (“chasing” embedded dependencies (Fagin et al., 2005b)). Their relative advantages and disadvantages over the relational operations are still being studied, but see Spivak (2012) for a preliminary discussion. At a high-level, $\Delta$ can be thought of as a projection, $\Pi$ as join, and $\Sigma$ as union.

The FDM’s basic idea of schemas-as-categories and three adjoint data migration functors $\Delta, \Sigma, \Pi$ recurs in our data model, but we base our formalism entirely on algebraic (equational) logic and therefore diverge from the original FDM. We define database schemas and instances to be equational theories of a certain kind. A schema mapping $F : C \rightarrow D$ is defined as a morphism (provability-respecting translation) of equational theories $C$ and $D$, and we define the $\Sigma_F$ data migration as substitution along $F$. The conditions we impose on our equational theories guarantee that $\Sigma_F$ has a right adjoint, $\Delta_F$, which in turn has a right adjoint, $\Pi_F$. Our query language is a generalization of for/where/return queries (Abiteboul et al., 1995) and corresponds semantically to combinations of $\Sigma, \Delta, \Pi$ operations. To integrate data we must go beyond $\Delta, \Sigma, \Pi$, and we use pushouts (Barr & Wells, 1995) of schemas and instances as the basis for a data integration design pattern suitable for building data warehouses.

1.2 Outline

The rest of this paper is divided into five sections. In section 2 we review category theory, and in section 3 we review algebraic (multi-sorted equational) logic. In section 4 we describe how we use algebraic theories to define schemas, instances, and the other artifacts necessary to perform data migration and query. In section 5 we describe how our formalism is implemented in the FQL tool. In section 6 we describe how we use pushouts of schemas and instances to perform data integration and include an extended example.

1.3 Related Work

In this section we describe how our work relates to the functional data model (section 1.3.1), functional programming (section 1.3.2), relational data integration (section 1.3.3), and other work which treats schemas as categories (section 1.3.4). Mathematical related work is discussed in Schultz et al. (2017).

1.3.1 vs The Functional Data Model

Our data model is simultaneously an extension of, and a restriction of, the functional data model (Shipman, 1981). Both formalisms use functions, rather than relations, over entities and data types as their principle data structure. The two primary ways the functional data model extends ours is that it allows products of entities (e.g., a function Person ×
Department $\rightarrow$ String), and it allows some non-equational data integrity constraints (e.g., a constraint $f(x) = f(y) \rightarrow x = y$). These two features can be added to our data model as described in Spivak (2014), although doing so weakens our data model’s properties (e.g., the existence of $\Pi$ cannot in general be guaranteed in the presence of non-equational constraints). Therefore, our data model can be thought of as extending the fragment of the functional data model where schemas are categories. Restricting to schemas-as-categories allows us to extend the functional data model with additional operations (e.g., $\Pi$) and to provide strong static reasoning principles (e.g., eliminating the need for run time data integrity checking; see section 4.3.1). The functional data model also includes updates, a topic we are still studying. The schemas of our data model and the functional data model both induce graphs, so both data models are graph data models in the sense of Angles and Gutierrez (2008). See Spivak (2012) for a discussion of how our formalism relates to RDF.

1.3.2 vs Functional Programming

Our formalism and functional programming languages both extend equational logic (Mitchell, 1996). As discussed in section 4.1, the schemas, type sides, and user-defined functions of our formalism are “algebraic data types” and our instances denote implementations (algebras) of these types. (This use of “algebraic data type” must be understood in the algebraic specification sense, as a data type specified by a set of equations (Mitchell, 1996), rather than in the Haskell/ML sense of a data type specified by products and sums.) As such, the implementation of our formalism draws heavily on techniques from functional programming: for example, when a set of data integrity constraints forms a confluent and terminating rewrite system, our implementation uses reduction to normal forms to decide equality under the constraints (section 5.1). Our formalism’s use of types, rather than terms, to represent sets (e.g., entity sets such as Person) is common in expressive type theories such as Coq (Bertot & Castran, 2010) and contrasts with the sets-as-terms approach used in simply-typed comprehension calculi (Grust, 2004); for a comparison of the two approaches (which we have dubbed QINL and LINQ, respectively), see Schultz et al. (2015). On certain type sides, our categories of schemas and instances are cartesian closed (Barr & Wells, 1995), meaning that schemas and instances can be defined by expressions in the simply-typed $\lambda$-calculus, and that our categories of schemas and instances are models of the simply-typed $\lambda$-calculus, but we have not yet found an application of this fact.

1.3.3 vs Relational Data Integration

Our formalism is an alternative to the traditional relational formalisms for both querying data (relational algebra) and migrating / integrating data (“chasing” embedded dependencies (EDs) (Fagin et al., 2005b)). In Spivak & Wisnesky (2015), we proved that the $\Delta, \Sigma, \Pi$ operations can express any (select, project, cartesian product, union) relational algebra query and gave conditions for the converse to hold. Our formalism uses purely equational data integrity constraints, which can be captured using first-order EDs, but not all first-order EDs can be captured using only equations. However, our formalism can be extended in a simple way, described in Spivak (2014), to capture all first-order EDs; when this is done, we find that a parallel chase step of an ED on an instance can be expressed using the
A proof that our formalism, extended as in Spivak (2014) and with a fixed point operation, is at least as expressive as chasing with first-order EDs is forthcoming. At present, we do not understand the relationship between our formalism and second-order EDs (Fagin et al., 2005a), although our “uber-flower” queries (section 4.3) can be written as second-order EDs of a particular form (section 4.3.6).

Our formalism defines databases “intensionally”, as sets of equations, and so in relational parlance our databases are “deductive databases” (Abiteboul et al., 1995). As such, some care must be taken when mediating between relational definitions and categorical definitions. For example, our instances can be inconsistent, in the sense that an instance might prove that 1 = 2 for two distinct constant symbols 1 and 2, and only consistent instances can be meaningfully translated into relational instances. In addition, our schemas do not define a set of constants (a “domain”) that all the instances on that schema share, as is customary in relational data integration (Fagin et al., 2005b). For these and other reasons (mentioned throughout this paper), our work is closer in spirit to traditional logic (Enderton, 2001) than database theory (Doan et al., 2012).

The pushout data integration pattern (section 6) is a “global as view” (Doan et al., 2012) pattern, because the integrated schema is a function (the pushout) of the source schemas. But rather than relating the integrated schema to the source schemas by EDs or queries, we use functors. A pushed-out instance satisfies a universal property similar to that of a universal solution to a set of EDs (Fagin et al., 2005b). Pushouts are investigated for relational data integration purposes in Alagic & Bernstein (2001). In that paper, the authors describe a design pattern for data integration which applies to a large class of formalisms: the so-called institutions (Goguen & Burstall, 1984). Our formalism is an institution, but their work differs from ours in key respects. First, they are primarily concerned with the Δ data migration functor (they call our Δ functor “Db” in their paper), because Δ exists in all institutions. They recognize that pushouts (what they call “schema joins”) are a canonical way of obtaining integrated schemas, and that not all institutions have pushouts of schemas (ours does). Their main theorem is that in any institution the Δ functor can be used to migrate the data on a pushout schema back to the source schemas. Our design pattern uses the Σ functor to go the other way: pushing data from source schemas to the integrated schema. See Goguen (2004) for more information about data integration in institutions. In the more general setting of algebraic specification, pushouts have received considerable attention as a means to integrate specifications (Blum et al., 1987).

1.3.4 vs The Functorial Data Model

Our work is related to a family of data models which treat database schemas as categories or variations thereof (Johnson et al., 2002) (Fleming et al., 2003) (Spivak, 2012) (Spivak, 2014) (Spivak & Wisnesky, 2015) (Schultz et al., 2017). We refer to these data models as “functorial data models”. The original functorial data model (Fleming et al., 2003) treated schemas as finitely presented categories and instances as set-valued functors, but is difficult to use for data integration purposes because most constructions on set-valued functors are only defined up to unique isomorphism, and in the context of data integration, we must distinguish two kinds of values in a database: atomic values such as strings and integers which must be preserved by morphisms (e.g., Bill), and meaningless identifiers which need not be preserved (e.g., auto-generated IDs). (In contexts outside of data integration, such
as query, there may not be a need to distinguish two types of values.) For example, the situation in figure 2, which holds in the original functorial data model, is untenable for data integration purposes.

<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>Bob</td>
<td>20</td>
<td>$250</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>30</td>
<td>$300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Alice</td>
<td>20</td>
<td>$100</td>
</tr>
<tr>
<td>5</td>
<td>Bob</td>
<td>20</td>
<td>$250</td>
</tr>
<tr>
<td>6</td>
<td>Sue</td>
<td>30</td>
<td>$300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amy</td>
<td>20</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>Bill</td>
<td>20</td>
<td>$250</td>
</tr>
<tr>
<td>3</td>
<td>Susan</td>
<td>30</td>
<td>$300</td>
</tr>
</tbody>
</table>

Fig. 2. The Attribute Problem

Several approaches to this “attribute problem” have been proposed, including Johnson et al. (2002) and Spivak & Wisnesky (2015). This paper extends the latter by paper by defining database schemas to exist in an ambient computational context called a “type-side”. Data values that inhabit types (e.g., Bill : String) are preserved by database morphisms, but other values, such as meaningless identifiers, are not. As a result, our formalism does not suffer from the attribute problem, and solving the attribute problem was a principle motivation for our work.

2 Review of Category Theory

In this section we review standard material on category theory (Barr & Wells, 1995). Readers familiar with category theory can safely skim or skip this section.

2.1 Categories with Products

A category C consists of a class of objects Ob(C) and a class of morphisms Hom(C) between objects. Each morphism m has a source object S and a target object T, which we write as m : S → T. Every object X has an identity morphism idX : X → X. When X is clear from the context we will write idX as simply id. Two morphisms f : B → C and g : A → B may be composed, written f ∘ g : A → C or g ∘ f : A → C. Composition is associative and id is its unit:

\[ f ∘ id = f \quad id ∘ f = f \quad f ∘ (g ∘ h) = (f ∘ g) ∘ h \]

A morphism f : X → Y is an isomorphism when there exists a g : Y → X such that

\[ f ∘ g = id \quad g ∘ f = id \]

Two objects are isomorphic when there exists an an isomorphism between them. Example categories include:

- **Set**, the category of sets. The objects of Set are sets, and a morphism f : X → Y is a (total) function from set X to set Y. Given morphisms f : Y → Z and g : X → Y, the
morphism \( f \circ g : X \to Z \) is defined as function composition: 
\[
(f \circ g)(x) := f(g(x)).
\]
The isomorphisms of \( \text{Set} \) are bijective functions. For each object \( X \), \( id_X \) is the identity function on \( X \).

- Any directed graph generates a category, called the free category on the graph: its objects are the vertices of the graph, and the morphisms are the paths in the graph. For each vertex \( X \), \( id_X \) is the 0-length path \( X \to X \). Composition of morphisms is concatenation of paths, and there are no non-identity isomorphisms.

A category \( C \) is said to have products when for every pair of objects \( X, Y \) in \( C \), there exists an object \( X \times Y \) in \( C \), morphisms \( \pi_1 : X \times Y \to X \) and \( \pi_2 : X \times Y \to Y \) in \( C \), and for every pair of morphisms \( f : A \to X, g : A \to Y \) in \( C \), there exists a morphism \( \langle f, g \rangle : A \to X \times Y \) in \( C \), such that
\[
\pi_1 \circ \langle f, g \rangle = f \quad \pi_2 \circ \langle f, g \rangle = g
\]
and such that for every morphism \( h : A \to X \times Y \),
\[
\langle \pi_1 \circ h, \pi_2 \circ h \rangle = h
\]
For example, set-theoretic cartesian product is a product in the category of sets, \( \text{Set} \).

### 2.2 Functors

A functor \( F : C \to D \) between two categories \( C \) and \( D \) is a mapping of objects of \( C \) to objects of \( D \) and morphisms of \( C \) to morphisms of \( D \) that preserves identities and composition:
\[
F(f : X \to Y) : F(X) \to F(Y) \quad F(id_X) = id_{F(X)} \quad F(f \circ g) = F(f) \circ F(g)
\]
Example functors include:

- For any category \( C \), the identity functor \( Id : C \to C \) maps each object and morphism to itself.
- For any categories \( C \) and \( D \) and an object of \( D \), there exists a constant functor taking each object \( C \) in \( C \) to \( D \) and each morphism in \( C \) to \( id_D \).
- The power set functor \( \mathcal{P} : \text{Set} \to \text{Set} \) maps each set to its power set and each function \( f : X \to Y \) to the function which sends \( U \subseteq X \) to its image \( f(U) \subseteq Y \).
- For each set \( A \), there is a product functor \( - \times A : \text{Set} \to \text{Set} \) mapping each set \( X \) to the cartesian product \( X \times A \) and each function \( f : X \to Y \) to the function \( f \times id_A : X \times A \to Y \times A \).

### 2.3 Natural Transformations

A natural transformation \( \alpha : F \Rightarrow G \) between two functors \( F : C \to D \) and \( G : C \to D \) is a family of morphisms \( \alpha_X : F(X) \to G(X) \) in \( D \), one for each object \( X \) in \( C \), such that for every \( f : X \to Y \) in \( C \),
\[
\alpha_Y \circ F(f) = G(f) \circ \alpha_X
\]
This equation may conveniently expressed as a commutative diagram:

\[
\begin{array}{c}
F(X) \xrightarrow{F(f)} F(Y) \\
\downarrow \alpha_X \quad \downarrow \alpha_Y \\
G(X) \xrightarrow{G(f)} G(Y)
\end{array}
\]

A natural transformation \( \alpha \) is a natural isomorphism when for every object \( X \) in \( C \), the morphism \( \alpha_X \) is an isomorphism in \( D \). Example natural transformations include:

- The identity natural isomorphism \( id_F : F \Rightarrow F \) for a functor \( F : C \rightarrow D \) is defined as \( id_{F_X} : F(X) \rightarrow F(X) := id_{F(X)} \).
- Consider the power set functor \( \mathcal{P} : \text{Set} \rightarrow \text{Set} \). There is a natural transformation \( \text{sng} : \text{Id}_\text{Set} \Rightarrow \mathcal{P} \) that maps every set \( X \) to the singleton set \( \{X\} \) (i.e., \( \text{sng}_X : X \rightarrow \mathcal{P}(X) \)), and there is a natural transformation \( \text{union} : \mathcal{P} \circ \mathcal{P} \Rightarrow \mathcal{P} \) that maps a set of sets \( \{X_1, \ldots, X_n\} \) to its \( n \)-ary union \( X_1 \cup \ldots \cup X_n \) (i.e., \( \text{union}_X : \mathcal{P}(\mathcal{P}(X)) \rightarrow \mathcal{P}(X) \)).

### 2.4 Adjunctions

An adjunction between categories \( C \) and \( D \) consists of a functor \( F : D \rightarrow C \) called the left adjoint, a functor \( G : C \rightarrow D \) called the right adjoint, a natural transformation \( \varepsilon : F \circ G \Rightarrow \text{Id}_C \) called the counit, and a natural transformation \( \eta : \text{Id}_D \Rightarrow G \circ F \) called the unit, such that for every object \( X \) in \( C \) and \( Y \) in \( D \), the following equations hold:

\[
id_{F(Y)} = \varepsilon_{F(Y)} \circ F(\eta_Y) \quad id_{G(X)} = G(\varepsilon_X) \circ \eta_{G(X)}
\]

Consequently, the set of morphisms \( F(Y) \rightarrow X \) is bijective with the set of morphisms \( Y \rightarrow G(X) \). Example adjunctions include:

- Let \( A \) be a set and consider the product functor \( - \times A : \text{Set} \rightarrow \text{Set} \). The exponential functor \( -^A : \text{Set} \rightarrow \text{Set} \), which maps each set \( X \) to the set of functions from \( A \) to \( X \) (written \( X^A \)), is right adjoint to \( - \times A \). Intuitively, this is because the set of functions \( X \times Y \rightarrow Z \) is bijective with the set of functions \( X \rightarrow Z^Y \).
- Consider the category of groups and group homomorphisms, \( \text{Grp} \). The functor \( \text{free} : \text{Set} \rightarrow \text{Grp} \), which maps each set \( X \) to the free group generated by \( X \), and the functor \( \text{forget} : \text{Grp} \rightarrow \text{Set} \) which maps each group to its underlying set, are adjoint. Intuitively, maps from the free group \( \text{free}(X) \) to a group \( Y \) correspond precisely to maps from the set \( X \) to the set \( \text{forget}(Y) \): each homomorphism from \( \text{free}(X) \) to \( Y \) is fully determined by its action on generators.

### 3 Review of Multi-sorted Equational Logic

In this section we review standard material on multi-sorted equational logic, following the exposition in Mitchell (1996). Theories in multi-sorted equational logic are also called “algebraic theories”, as well as “Lawvere theories” and “product theories”. We will use these phrases interchangeably. For a category-theoretic study of such theories, see Adamek et al. (2011). Readers familiar with equational logic can safely skim or skip this section. We will write “theory” instead of “multi-sorted equational theory” in this section.
3 January 2017

3.1 Syntax

A signature \( \text{Sig} \) consists of:

1. A set \( \text{Sorts} \) whose elements are called \textit{sorts},
2. A set \( \text{Symbols} \) of pairs \((f, s_1 \times \ldots \times s_k \rightarrow s)\) with \( s_1, \ldots, s_k, s \in \text{Sorts} \) and no \( f \) occurring in two distinct pairs. We write \( f : X \) instead of \((f, X) \in \text{Symbols}\). When \( k = 0 \), we may call \( f \) a \textit{constant symbol} and write \( f : s \) instead of \( f : \rightarrow s \). Otherwise, we may call \( f \) a \textit{function symbol}.

We assume we have some countably infinite set \( \{v_1, v_2, \ldots\} \), whose elements we call \textit{variables} and which are assumed to be distinct from any sort or symbol we ever consider. A \textit{context} \( \Gamma \) is defined as a finite set of variable-sort pairs, with no variable given more than one sort:

\[
\Gamma := \{ v_1 : s_1, \ldots, v_k : s_k \}
\]

When the sorts \( s_1, \ldots, s_k \) can be inferred, we may write a context as \( \{v_1, \ldots, v_k\} \). We may write \( \{v_1 : s, \ldots, v_k : s\} \) as \( \{v_1, \ldots, v_k : s\} \). We may write \( \Gamma \cup \{v : s\} \) as \( \Gamma, v : s \). We inductively define the set \( \text{Terms}^s(\text{Sig}, \Gamma) \) of terms of sort \( s \) over signature \( \text{Sig} \) and context \( \Gamma \) as:

1. \( x \in \text{Terms}^s(\text{Sig}, \Gamma) \), if \( x : s \in \Gamma \),
2. \( f(t_1, \ldots, t_k) \in \text{Terms}^s(\text{Sig}, \Gamma) \), if \( f : s_1 \times \ldots \times s_k \rightarrow s \) and \( t_i \in \text{Terms}^{s_i}(\text{Sig}, \Gamma) \) for \( i = 1, \ldots, k \). When \( k = 0 \), we may write \( f \) for \( f() \). When \( k = 1 \), we may write \( t_1.f \) instead of \( f(t_1) \). When \( k = 2 \), we may write \( t_1\ f \ t_2 \) instead of \( f(t_1, t_2) \).

We refer to \( \text{Terms}^s(\text{Sig}, \emptyset) \) as the set of \textit{ground} terms of sort \( s \). We will write \( \text{Terms}(\text{Sig}, \Gamma) \) for the set of all terms in context \( \Gamma \), i.e., \( \bigcup_s \text{Terms}^s(\text{Sig}, \Gamma) \). Substitution of a term \( t \) for a variable \( v \) in a term \( e \) is written as \( e[v \mapsto t] \) and is recursively defined as usual:

\[
v[v \mapsto t] = t \quad v'[v \mapsto t] = v' \quad (v \neq v') \quad f(t_1, \ldots, t_n)[v \mapsto t] = f(t_1[v \mapsto t], \ldots, t_n[v \mapsto t])
\]

We will only make use of substitutions that are sort-preserving; i.e., to consider \( e[v \mapsto t] \), we require \( e \in \text{Terms}(\text{Sig}, \Gamma) \) for some \( \Gamma \) such that \( v : s \in \Gamma \) and \( t \in \text{Terms}^s(\text{Sig}, \Gamma) \). To indicate a simultaneous substitution for many variables we will write e.g., \( [v_2 \mapsto t_2, v_1 \mapsto t_1] \). To indicate the composition of two substitutions will write e.g., \( [v_1 \mapsto t_1] \circ [v_2 \mapsto t_2] \).

An \textit{equation} over \( \text{Sig} \) is a formula \( \forall \Gamma \cdot t_1 = t_2 : s \) with \( t_1, t_2 \in \text{Terms}^s(\text{Sig}, \Gamma) \); we will omit the : \( s \) when doing so will not lead to confusion. A \textit{theory} is a pair of a signature and a set of equations over that signature. In this paper, we will make use of a theory we call \textit{Type}, which is displayed in figure 3. Additional axioms, such as the associativity and commutativity of \( + \), can be added to \textit{Type}, but doing so does not impact the examples in this paper.

Associated with a theory \( Th \) is a binary relation between (not necessarily ground) terms, called \textit{provable equality}. We write \( Th \vdash \forall \Gamma \cdot t = t' : s \) to indicate that the theory \( Th \) proves that terms \( t, t' \in \text{Terms}^s(\text{Sig}, \Gamma) \) are equal according to the usual rules of multi-sorted equational logic. From these rules it follows that provable equality is the smallest equivalence relation on terms that is a congruence, closed under substitution, closed under adding variables to contexts, and contains the equations of \( Th \). In general, provable equality is semi-decidable (Bachmair et al., 1989). Formally, \( Th \vdash \) is defined by the inference rules in figure 4.
A morphism of signatures $F : \text{Sig}_1 \to \text{Sig}_2$ consists of:

- a function $F$ from sorts in $\text{Sig}_1$ to sorts in $\text{Sig}_2$, and
- a function $F$ from function symbols $f : s_1 \times \ldots \times s_n \to s$ in $\text{Sig}_1$ to terms in $\text{Terms}^{F(s)}(\text{Sig}_2, \{v_1 : F(s_1), \ldots, v_n : F(s_n)\})$

For example, let $\text{Sig}_1$ consist of two sorts, $a, b$, and one function symbol, $f : a \to b$, and let $\text{Sig}_2$ consist of one sort, $c$, and one function symbol, $g : c \to c$. There are countably infinitely many morphisms $F : \text{Sig}_1 \to \text{Sig}_2$, one of which is defined as $F(a) := c, F(b) := c$, and $F(f) := v : c, g(v))$. In the literature on algebraic specification, our definition of signature morphism is called a “derived signature morphism” (Mossakowski et al., 2014).

The function $F$ taking function symbols to terms can be extended to take terms to terms:

$$F(v) = v \quad F(f(t_1, \ldots, t_n)) = F(f)[v_1 \mapsto F(t_1), \ldots, v_n \mapsto F(t_n)]$$

When we are defining the action of a specific $F$ on a specific $f : s_1 \times \ldots \times s_n \to s$, to make clear the variables we are using, we may write $F(f) := \forall v_1, \ldots, v_n. \phi$, where $\phi$ may contain $v_1, \ldots, v_n$. A morphism of theories $F : Th_1 \to Th_2$ is a morphism of signatures that preserves provable equality of terms:

$$Th_1 \vdash \forall v_1 : s_1, \ldots, v_n : s_n. t_1 = t_2 : s \implies Th_2 \vdash \forall v_1 : F(s_1), \ldots, v_n : F(s_n). F(t_1) = F(t_2) : F(s)$$

In the theory $Type$ (figure 3), any permutation of $A, B, \ldots Z$ induces a morphism $Type \to Type$, for example. Although morphisms of signatures are commonly used in the categorical approach to logic (Adámek et al., 2011), such morphisms do not appear to be as commonly used in the traditional set-theoretic approach to logic. Checking that a morphism of signatures is a morphism of theories reduces to checking provable equality of terms and hence is semi-decidable.
Remark. Multi-sorted equational logic differs from single-sorted logic by allowing empty sorts (sorts that have no ground terms). Empty sorts are required by our formalism; without them, we could not express empty entities. For the theoretical development, this difference between multi-sorted and single-sorted logic can be safely ignored. But the fact that many algorithms are based on single-sorted logic means that care is required when implementing our formalism. For example, certain theorem proving methods based on Knuth-Bendix completion (Knuth & Bendix, 1970) require a ground term of every sort.

Categorical Remark. From a theory Th we form a cartesian multi-category (Barr & Wells, 1995) \(\{Th\}\) as follows. The objects of \(\{Th\}\) are the sorts of Th. The elements of the hom-set \(s_1, \ldots, s_k \rightarrow s\) of \(\{Th\}\) are equivalence classes of terms of sort \(s\) in context \(\{v_1 : s_1, \ldots, v_k : s_k\}\), modulo the provable equality relation \(Th \vdash\). Composition is defined by substitution. A morphism of theories \(F : Th_1 \rightarrow Th_2\) denotes a functor \(\{F\} : \{Th_1\} \rightarrow \{Th_2\}\). Although cartesian multi-categories are the most direct categorical semantics for theories, in many cases it is technically more convenient to work with product categories instead. Every cartesian multi-category generates a product category, and we often conflate the multi-category just described with the product category it generates, as is usually done in the categorical algebraic theories literature. For details, see Schultz et al. (2017).

3.2 Semantics

An algebra \(A\) over a signature \(\text{Sig}\) consists of:

- a set of carriers \(A(s)\) for each sort \(s\), and
- a function \(A(f) : A(s_1) \times \ldots \times A(s_k) \rightarrow A(s)\) for each symbol \(f : s_1 \times \ldots \times s_k \rightarrow s\).

Let \(\Gamma := \{v_1 : s_1, \ldots, v_n : s_n\}\) be a context. An \(A\)-environment \(\eta\) for \(\Gamma\) associates each \(v_i\) with an element of \(A(s_i)\). The meaning of a term in \(\text{Terms}(\text{Sig}, \Gamma)\) relative to \(A\)-environment \(\eta\) for \(\Gamma\) is recursively defined as:

\[
A[v] \eta = \eta(v) \quad A[f(t_1, \ldots, t_n)] \eta = A(f)(A[t_1] \eta, \ldots, A[t_n] \eta)
\]

An algebra \(A\) over a signature \(\text{Sig}\) is a model of a theory Th on \(\text{Sig}\) when \(Th \vdash \forall \Gamma. t = t' : s\) implies \(A[v] \eta = A[t'] \eta\) for all terms \(t, t' \in \text{Terms}(\text{Sig}, \Gamma)\) and \(A\)-environments \(\eta\) for \(\Gamma\). Deduction in multi-sorted equational logic is sound and complete: two terms \(t, t'\) are provably equal in a theory Th if and only if \(t\) and \(t'\) denote the same element in every model of Th. One model of the theory Type (figure 3) has carriers consisting of the natural numbers, the 26 character English alphabet, and all strings over the English alphabet. Another model of Type uses natural numbers modulo four as the carrier for Nat.

From a signature \(\text{Sig}\) we form its term algebra \(\llbracket \text{Sig} \rrbracket\), a process called saturation, as follows. The carrier set \(\llbracket \text{Sig} \rrbracket(s)\) is defined as the set of ground terms of sort \(s\). The function \(\llbracket \text{Sig} \rrbracket(f)\) for \(f : s_1 \times \ldots \times s_k \rightarrow s\) is defined as the function \(t_1, \ldots, t_n \mapsto f(t_1, \ldots, t_n)\). From a theory Th on \(\text{Sig}\) we define its term model \(\llbracket Th \rrbracket\) to be the quotient of \(\llbracket \text{Sig} \rrbracket\) by the equivalence relation \(Th \vdash\). In other words, the carrier set \(\llbracket Th \rrbracket(s)\) is defined as the set of equivalence classes of ground terms of sort \(s\) that are provably equal under Th. The function \(\llbracket Th \rrbracket(f)\) is \(\llbracket \text{Sig} \rrbracket(f)\) lifted to operate on equivalence classes of terms. To represent \(\llbracket Th \rrbracket\) on a computer, or to write down \(\llbracket Th \rrbracket\) succinctly, we must choose a representative for each equivalence class of terms; this detail can be safely ignored for now, but we will return
A morphism of algebras \( h : A \to B \) on a signature \( \text{Sig} \) is a family of functions \( h(s) : A(s) \to B(s) \) indexed by sorts \( s \) 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 \( f : s_1 \times \ldots \times s_n \to s \) and \( a_i \in A(s_i) \). We may abbreviate \( h(s)(a) \) as \( h(a) \) when \( s \) can be inferred. The term algebras for a signature \( \text{Sig} \) are initial among all \( \text{Sig} \)-algebras: there is a unique morphism from the term algebra to any other \( \text{Sig} \)-algebra. Similarly, the term models are initial among all models. It is because of initiality that in many applications of equational logic to functional programming, such as algebraic datatypes (Mitchell, 1996), the intended meaning of a theory is its term model.

Categorical Remark. Models of a theory \( Th \) correspond to functors \( Th \to \text{Set} \), and the term model construction yields an initial such model. That is, an algebraic theory \( Th \) denotes a cartesian multi-category, \( \langle Th \rangle \), and the term model construction yields a functor \( \langle Th \rangle \to \text{Set} \). At the risk of confusion, we will write also write \( \langle Th \rangle \) for the functor \( \langle Th \rangle \to \text{Set} \); hence, we have \( \langle Th \rangle : \langle Th \rangle \to \text{Set} \). Morphisms between models correspond to natural transformations. A morphism of theories \( F : Th_1 \to Th_2 \) induces a functor, \( \langle F \rangle : \langle Th_1 \rangle \to \langle Th_2 \rangle \) between the cartesian multi-categories \( \langle Th_1 \rangle \) and \( \langle Th_2 \rangle \), as well as a natural transformation between the set-valued functors \( \langle Th_1 \rangle \) and \( \langle Th_2 \rangle \).

4 An Equational Formalism for Functorial Data Migration

In this section we describe how to use multi-sorted equational theories to define schemas and instances, and how to migrate data from one schema to another. To summarize, we proceed as follows; each of these steps is described in detail in this section. First, we fix an arbitrary multi-sorted equational theory \( Ty \) to serve as an ambient type-side or “background theory” against which we will develop our formalism. We say that the sorts in \( Ty \) are types. For example, we may define \( Ty \) to contain a sort \( \text{Nat} \); function symbols \( 0, 1, +, \times \); and equations such as \( 0 + x = x \). A schema \( S \) is an equational theory that extends \( Ty \) with new sorts (which we call entities), for example Person; unary function symbols between entities (which we call foreign keys) and from entities to types (which we call attributes),
for example, father : Person → Person and age : Person → Nat; and additional equations. An instance I on S is an equational theory that extends S with 0-ary constant symbols (which we call generators), such as Bill and Bob; as well as additional equations, such as father(Bill) = Bob. The intended meaning of I is its term model (i.e., the canonical model built from I-equivalence classes of terms). Morphisms of schemas and instances are defined as theory morphisms; i.e., mappings of sorts to sorts and function symbols to (open) terms that preserve entailment: \( h : C \rightarrow D \) exactly when \( C \vdash p = q \) implies \( D \vdash h(p) = h(q) \). Our “uber-flower” query language is based on a generalization of for-let-where-return (flwr) notation, and generalizes the idea from relational database theory that conjunctive queries can be represented as “frozen” instances (Abiteboul et al., 1995).

### 4.1 Type-sides, Schemas, Instances, Mappings, and Transforms

Our formalism begins by fixing a specific multi-sorted equational theory \( Ty \) which will be called the type-side of the formalism. The sorts in \( Ty \) are called types. The type-side is meant to represent the computational context within which our formalism will be deployed. For example, a type-side for SQL would contain sorts such as VARCHAR and INTEGER and functions such as \( \text{LENGTH} : \text{VARCHAR} \rightarrow \text{INTEGER} \), as well as any user-defined scalar functions we wish to consider (e.g., squaring a number); a type-side for SK combinator calculus would contain a sort \( o \), constants \( S, K : o \), a function symbol \( \cdot : o \rightarrow o \), and equations \( K \cdot x \cdot y = x \) and \( S \cdot x \cdot y \cdot z = x \cdot z \cdot (y \cdot z) \). From a database point of view, choosing a particular multi-sorted equational theory \( Ty \) can be thought of as choosing the set of built-in and user-defined types and functions that can appear in schemas and queries.

Simply-typed, first-order functional programs can be expressed using multi-sorted equational theories, and using a functional program as a type-side is a best-case scenario for the automated reasoning required to implement our formalism; see section 5 for details. We will abbreviate “multi-sorted equational theory” as “theory” in this section.

A schema \( S \) on type-side \( Ty \) is a theory extending \( Ty \). If \( s \) is a sort in \( S \) but not in \( Ty \), we say that \( s \) is an entity; otherwise, that \( s \) is a type. (Note that although we say “entity”, the synonyms “entity type” and “entity set” are also common in database literature.) \( S \) must meet the following conditions:

1. If \( \forall \Gamma. t_1 = t_2 \) is in \( S \) but not in \( Ty \), then \( \Gamma = \{ v : s \} \) where \( s \) is an entity.
2. If \( f : s_1 \times \ldots \times s_n \rightarrow s \) is in \( S \) but not \( Ty \), then \( n = 1 \) and \( s_1 \) is an entity. If \( s \) is an entity we say that \( f \) is a foreign key; otherwise, we say that \( f \) is an attribute.

In other words, every equation in a schema will have one of two forms: \( \forall v : t. v.p = v.p' : t' \), where \( t \) and \( t' \) are entities and \( p \) and \( p' \) are sequences of foreign keys; or some combination of type-side functions applied to attributes, for example \( \forall v : t. v.p_1.att_1 + v.p_2.att_2 = v.att : t' \) where \( t \) is an entity and \( t' \) is a type. Due to these restrictions, \( S \) admits three sub-theories: the type-side of \( S \), namely, \( Ty \); the entity-side of \( S \), namely, the restriction of \( S \) to entities (written \( S_E \)); and the attribute-side of \( S \), namely, the restriction of \( S \) to attributes (written \( S_A \)). We can also consider the entities and attributes together \( (S_{EA}) \), and the attributes and type-side together \( (S_{AT}) \). A morphism of schemas, or schema mapping, \( S_1 \rightarrow S_2 \) on type-side \( Ty \) is a morphism of theories \( S_1 \rightarrow S_2 \) that is the identity on \( Ty \). An example schema \( \text{Emp} \) on type-side \( Type \) (figure 3) is shown in figure 6. We may draw the entity and attribute...
part of the schema in a graphical notation, with every sort represented as a dot, and the foreign keys and attributes represented as edges.

\[
\text{Sorts} := \{\text{Emp}, \text{Dept}\}
\]

\[
\text{Symbols} := \{\text{mgr} : \text{Emp} \to \text{Emp}, \text{wrk} : \text{Emp} \to \text{Dept}, \text{secr} : \text{Dept} \to \text{Emp}, \text{dname} : \text{Dept} \to \text{String}, \text{ename} : \text{Emp} \to \text{String}\}
\]

\[
\text{Equations} := \{\forall v. v.\text{mgr} . \text{wrk} = v.\text{wrk}, \forall v. v = v.\text{secr} . \text{wrk}, \forall v. v.\text{mgr} . \text{mgr} = v.\text{mgr}\}
\]

In schema \(\text{Emp}\) (figure 6), Emp and Dept are sorts (entities) of employees and departments, respectively; mgr takes an employee to their manager; secr takes a department to its secretary; and \(\text{wrk}\) takes an employee to the department they work in. The equations are data integrity constraints saying that managers work in the same department as their employees, that secretaries work for the department they are the secretary for, and that the management hierarchy is two levels deep (this constraint ensures that \(\mathbb{E}_{\text{Emp}}\) is finite, a condition useful for our examples but not required by our implementation; see section 5).

An instance \(I\) on schema \(S\) is a theory extending \(S\), meeting the following conditions:

\begin{itemize}
  \item If \(s\) is a sort in \(I\), then \(s\) is a sort in \(S\).
  \item If \(\forall \Gamma. t_1 = t_2\) is in \(I\) but not \(S\), then \(\Gamma = \emptyset\).
  \item If \(f : s_1 \times \ldots \times s_n \to s\) is in \(I\) but not \(S\), then \(n = 0\). We say \(f\) is a \textit{generator} of sort \(s\).
\end{itemize}

That is, an instance only adds 0-ary symbols and ground equations to its schema. Mirroring a similar practice in database theory, we use the phrase \textit{skolem term} to refer to a term in an instance that is not provably equal to a term entirely from the type-side, but whose sort is a type. Although skolem terms are very natural in database theory, skolem terms wreak havoc in the theory of algebraic datatypes, where their existence typically implies an error in a datatype specification that causes computation to get “stuck” (Mitchell, 1996).

Similarly to how schemas admit sub-theories for entity, attribute, and type-sides, an instance \(I\) contains sub-theories for entities (\(I_E\)), attributes (\(I_A\)), and types (\(I_T\)). Note that \(I_T\) may not be the ambient type-side \(\mathcal{T}\), because \(I\) can declare new constant symbols whose
sorts are types (so-called skolem variables), as well as additional equations; for example, infinity : Nat and succ(infinity) = infinity. A morphism of instances, or transform, h : I₁ \rightarrow I₂ is a morphism of theories I₁ \rightarrow I₂ that is the identity on S, and the requirement of identity on S rules out the “attribute problem” from figure 2.

The intended meaning of an instance I is its term model, \([I]\). In practice, the term model \([I]\) will often have an infinite type-side, but \([I_EA]\) will be finite. Therefore, our implementation computes \([I_EA]\), as well as an instance \(talg(I)\) called the type-algebra\(^1\) for I. The type-algebra \(talg(I)\) is an instance on the empty schema over \(I_T\). For every attribute \(att : s \rightarrow t\) in S, and every term \(e \in [I_E](s)\), the type-algebra contains a generator \(e.att : t\).

We call these generators observables because they correspond to type-valued observations one can make about an entity. Observables have the form \(e.fk_1,...,fk_n.att\); i.e., have a 0-ary constant symbol as a head, followed by a possibly empty list of foreign keys, followed by an attribute. We define the function \(trans : Terms(I,\emptyset) \rightarrow talg(I)\), for every type (non-entity) sort \(t\), as:

\[
trans(e.fk_1,...,fk_n.att) := nf_{f}(e.fk_1,...,fk_n).att \text{ for observables}
\]

\[
trans(f(e_1,...,e_n)) := f(trans(e_1),...,trans(e_n)) \text{ otherwise}
\]

By \(nf_{f}(x)\) we mean the normal form for \(x\) in \([I]\); see section 3.2. The equations for \(talg(I)\) are the images under \(trans\) of the (necessarily ground) equations of I but not S and all substitution instances of the equations at types in S but not \(T_y\). Note that \(talg(I)\) does not present \([I_T]\) (the restriction of I to types), rather, \(talg(I)\) presents \([I_T]\) (the skolem terms of \([I]\) and their relationships).

We visually present term models using a set of tables, with one table per entity, with an ID column corresponding to the carrier set. Sometimes, we will present the type-algebra as well. An instance on the \(Emp\) schema, and its denotation, are shown in figure 7.

In many cases we would like for an instance I to be a conservative extension of its schema S, meaning that for all terms \(t,t' \in Terms(S,\Gamma)\), \(I \vdash \forall \Gamma : t = t' : s\) if and only if \(S \vdash \forall \Gamma : t = t' : s\). (Similarly, we may also want schemas to conservatively extend their typesides.) For example, \(Emp \not\models Al = Carl : String\), but there is an \(Emp\)-instance I for which \(I \vdash Al = Carl : String\). In the context of “deductive databases” (Abiteboul et al., 1995) (databases that are represented intensionally, as theories, rather than extensionally, as tables) such as our formalism, non-conservativity is usually regarded as non-desireable (Ghilardi et al., 2006), although nothing in our formalism requires conservativity. Checking for conservatism is decidable for the description logic underlying OWL (Ghilardi et al., 2006), but not decidable for multi-sorted equational logic (and hence our formalism), and not decidable for the formalism of embedded dependencies (Fagin et al., 2005b) that underlies much work on relational data integration (the chase fails when conservativity is violated). In section 5.3 we give a simple algorithm that soundly approximates conservativity. Note that the \(\Delta\) and \(\Pi\) migration functors preserve the conservative extension property,

\(^1\) Technically, it is \([talg(I)]\) that is a \(Ty\)-algebra, and \(talg(I)\) presents this algebra. But we will almost never be interested in \([talg(I)]\), so to save space we will refer to the equational theory \(talg(I)\) as \(I\)'s type-algebra.
The type-algebra extends \textit{Type} with:

\textbf{Symbols} := \{m.dname, s.dname, a.ename, b.ename, c.ename, \\
a.mgr.ename, b.mgr.ename, c.mgr.ename : String\}

\textbf{Eqs} := \{a.ename = Al, c.ename = Carl, \ m.dname = Math, \\
a.wrk = m, \ b.wrk = m, \ s.secr = c, \ m.secr = b\}

<table>
<thead>
<tr>
<th>Dept</th>
<th>ID</th>
<th>ename</th>
<th>mgr</th>
<th>wrk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>Al</td>
<td>a.mgr</td>
<td>m</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b.ename</td>
<td>b.mgr</td>
<td>m</td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>Math</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s</td>
<td>s.dname</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a.mgr</td>
<td>a.mgr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b.mgr</td>
<td>b.mgr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.mgr</td>
<td>c.mgr</td>
</tr>
</tbody>
</table>

The type-algebra extends \textit{Type} with:

\textbf{Symbols} := \{a, b, c : Emp, m, s : Dept\}

\textbf{Eqs} := \{a.ename = Al, c.ename = Carl, \ m.dname = Math, \\
a.wrk = m, \ b.wrk = m, \ s.secr = c, \ m.secr = b\}

\begin{tabular}{|c|c|c|c|}
\hline
ID & ename & mgr & wrk \\
\hline
a & Al & a.mgr & m \\
b & b.ename & b.mgr & m \\
m & Math & b & \\
s & s.dname & c & \\
a.mgr & a.mgr.ename & a.mgr & m \\
b.mgr & b.mgr.ename & b.mgr & m \\
c.mgr & c.mgr.ename & c.mgr & s \\
\hline
\end{tabular}

Fig. 7. Instance Inst on Schema Emp (figure 6)

but \(\Sigma\) does not; hence, one may want to be careful when using \(\Sigma\). (More pedantically, \(\Delta\) and \(\Pi\) preserve type-algebras, but \(\Sigma\) does not.)

\textbf{Remark.} There is a precise sense in which our definition of transform corresponds to the definition of database homomorphism in relational database theory. Recall (Abiteboul \textit{et al.}, 1995) that in database theory a schema is a triple \(\langle \text{dom}, \text{null}, R \rangle\), where \text{dom} is a set (called the \textit{domain}), \text{null} is a set disjoint from \text{dom} (called the \textit{labeled nulls}), and \(R\) is a set of relation names and arities; an instance \(I\) is a set of relations over \(\text{dom} \cup \text{null}\), indexed by \(R\), of appropriate arity; and a homomorphism \(h : I_1 \rightarrow I_2\) is a function \(h : \text{dom} \cup \text{null} \rightarrow \text{dom} \cup \text{null}\) that is constant on \text{dom} and such that \((c_1, \ldots, c_n) \in I_1(R)\) implies \((h(c_1), \ldots, h(c_n)) \in I_2(R)\) for every \(R\) of arity \(n\). If we interpret a term model \([\hat{I}]\) as a relational instance by considering every skolem term in \([\hat{I}]\) to be a labeled null and every non-skolem term to be a domain value, then a transform of instances in our formalism induces a homomorphism of the encoded relational instances. In this encoding, \text{dom} is playing the role of a free (equation-less), discrete (function-less) type-side.

\textbf{Categorical Remark.} By forgetting the entity/attribute distinction, we can consider a schema \(S\) as a single algebraic theory, \(\hat{S}\); the cartesian multi-category \(\langle \hat{S} \rangle\) is called the \textit{collage} of \(S\). A schema mapping \(F : S \rightarrow T\) is then a functor between collages \(\langle \hat{S} \rangle \rightarrow \langle \hat{T} \rangle\) that is the identity on \(Ty\). More pedantically, a schema \(S\) is a profunctor \(\langle S \rangle : \langle SL \rangle^{op} \times \langle Ty \rangle \rightarrow \text{Set}\) which preserves products in \(\langle Ty \rangle\). The observables from an entity \(e \in S_E\) to type \(ty \in Ty\) are given by \(\langle S \rangle(e, ty)\). A schema mapping \(F : S \rightarrow T\) denotes a functor \(\langle F_E \rangle : \langle S_L \rangle^{op} \times \langle T_L \rangle \rightarrow \text{Set}\). 
\[ \langle S \rangle \rightarrow \langle T \rangle \] and a natural transformation \( \langle S \rangle \Rightarrow \langle T \rangle \circ (\langle F \rangle \circ \text{id}) \):

\[
\begin{array}{c}
\langle S \rangle \circ \text{id} \\
\downarrow \\
\langle T \rangle \circ \text{id}
\end{array}
\]

\[
\downarrow
\]

\[ \text{Set} \]

\[ \langle T \rangle \]

4.2 Functorial Data Migration

We are now in a position to define the data migration functors. We first fix a type-side (multi-sorted equational theory), \( Ty \). The following are proved in Schultz et al. (2017):

- The schemas on \( Ty \) and their mappings form a category.
- The instances on a schema \( S \) and their transforms form a category, \( S \rightarrow \text{Inst} \).
- The models of \( S \) and their homomorphisms obtained by applying \( \text{[ ]} \) to \( S \rightarrow \text{Inst} \) form a category, \( \text{[S–Inst]} \), which is equivalent to, but not equal to, \( S \rightarrow \text{Inst} \).
- A schema mapping \( F : S \rightarrow T \) induces a unique functor \( \Sigma_F : S \rightarrow \text{Inst} \rightarrow T \rightarrow \text{Inst} \) defined by substitution, \( \Sigma_F(I) := F(I) \), with a right adjoint, \( \Delta_F : T \rightarrow \text{Inst} \rightarrow S \rightarrow \text{Inst} \), which itself has a right adjoint, \( \Pi_F : S \rightarrow \text{Inst} \rightarrow T \rightarrow \text{Inst} \).
- A schema mapping \( F : S \rightarrow T \) induces a unique functor \( \text{[ΔF]} : [T \rightarrow \text{Inst}] \rightarrow [S \rightarrow \text{Inst}] \)
  defined by composition, \( \text{[ΔF]}(I) := I \circ F \), with a left adjoint, \( \text{[ΣF]} : [S \rightarrow \text{Inst}] \rightarrow [T \rightarrow \text{Inst}] \), and a right adjoint \( \text{[ΠF]} : [S \rightarrow \text{Inst}] \rightarrow [T \rightarrow \text{Inst}] \).

Although \( \Sigma_F \) and \( \text{[ΔF]} \) are canonically defined, their adjoints are only defined up to unique isomorphism. The canonically defined migration functors enjoy properties that the other data migration functors do not, such as \( \Sigma_F(\text{[ΣG]}(I)) = \Sigma_F \circ \text{[ΣG]}(I) \) and \( \text{[ΔF]}(\text{[ΔG]}(I)) = \text{[ΔF]} \circ \text{[ΔG]}(I) \) (for the other functors, these are not equalities, but unique isomorphisms).

It is possible to give explicit formulæe to define the three data migration functors, \( \Delta, \Sigma, \Pi \) (Schultz et al., 2017). However, we have found that it is more convenient to work with two derived data migration functors, \( \Delta \circ \Pi \) and \( \Delta \circ \Sigma \), which we describe in the next section. Therefore, we now simply describe examples of \( \Delta, \Sigma, \Pi \) in figures 8, 9, and 10. Because these examples display instances as tables, rather than equational theories, we are actually illustrating \( [\Delta], [\Sigma], [\Pi] \).

Figures 8 and 9 shows a schema mapping \( F \) which takes two distinct source entities, \( N_1 \) and \( N_2 \), to the target entity \( N \). The \( [\Delta_F] \) functor projects in the opposite direction of \( F \); it projects columns from the single table for \( N \) to two separate tables for \( N_1 \) and \( N_2 \), similar to FROM \( N \) as \( N_1 \) and FROM \( N \) as \( N_2 \) in SQL. When there is a foreign key between \( N_1 \) and \( N_2 \), the \( [\Delta_F] \) functor populates it so that \( N \) can be recovered by joining \( N_1 \) and \( N_2 \). The \( [\Pi_F] \) functor takes the cartesian product of \( N_1 \) and \( N_2 \) when there is no foreign key between \( N_1 \) and \( N_2 \), and joins \( N_1 \) and \( N_2 \) along the foreign key when there is. The \( [\Sigma_F] \) functor disjointly unions \( N_1 \) and \( N_2 \); because \( N_1 \) and \( N_2 \) are not union compatible (have different columns), \( [\Sigma_F] \) creates null values. When there is a foreign key between \( N_1 \) and \( N_2 \), \( [\Sigma_F] \) merges the tuples that are related by the foreign key, resulting a join. As these two examples illustrate, \( \Delta \) can be thought of as “foreign-key aware” projection, \( \Pi \) can be thought of as a product followed by a filter (which can result in a join), and \( \Sigma \) can be thought of as a (not necessarily union compatible) disjoint union followed by a merge (which can also result in a join).
Figure 10 shows a traditional “data exchange setting” (Fagin et al., 2005b); data on a source schema about amphibians must be migrated onto a target schema about animals, where the target schema contains a data integrity constraint enforcing that each amphibian is only counted as a single animal. The schema mapping \( F \) is an inclusion, and \( \Box F \) has precisely the desired semantics.

### 4.3 Uber-flower Queries

It is possible to form a query language directly from schema mappings. This is the approach we took in Spivak & Wisnesky (2015), where a query is defined to be a triple of schema mappings \((F, G, H)\) denoting \( \Delta G \circ \Pi F \circ \Delta H \). Suitable conditions on \( F, G, H \) guarantee closure under composition, computability using relational algebra, and other properties desirable in a query language. In practice, however, we found this query language to be challenging to program. Having to specify entire schema mappings is onerous; it is difficult to know how to use the data migration functors to accomplish any particular task without a thorough understanding of category theory; and as a kind of “join all”, \( \Pi \) is expensive to compute. Hence, in Schultz et al. (2017) we developed a new syntax, which we call uber-flower syntax because it generalizes flwr (for-let-where-return) syntax (a.k.a. select-from-where syntax, a.k.a. comprehension syntax (Grust, 2004)). We have found uber-flower syntax to be more concise, easier to program, and easier to implement than the language based on triples of schema mappings in Spivak & Wisnesky (2015).

An uber-flower \( Q : S \rightarrow T \), where \( S \) and \( T \) are schemas on the same type-side, induces a data migration \( \text{eval}(Q) : S – \text{Inst} \rightarrow T – \text{Inst} \cong \Delta G \circ \Pi F \) and an adjoint data migration \( \text{coeval}(Q) : T – \text{Inst} \rightarrow S – \text{Inst} \cong \Delta F \circ \Sigma G \) for some \( X \), \( F : S \rightarrow X \), \( G : T \rightarrow X \). In fact, all data migrations of the form \( \Delta \circ \Pi \) can be expressed as the eval of an uber-flower, and all migrations of the form \( \Delta \circ \Sigma \) can be expressed as the coeval of an uber-flower. In sections 4.3.4 and 4.3.5 we describe the correspondence between uber-flowers and data migration functors in detail. In the remainder of this section we describe uber-flowers, but defer a description of how to (co-)evaluate them to sections 5.4 and 5.5.

A tableau (Abiteboul et al., 1995) over a schema \( S \) is a pair of:

- a context over \( S \), called the for clause, \( fr \) and
- a set of equations between terms in \( \text{Terms}(S, fr) \), called the where clause \( wh \).

Associated with a tableau over \( S \) is a canonical \( S \)-instance, the “frozen” instance (Abiteboul et al., 1995). In our formalism, a tableau trivially becomes an instance by the validity-preserving Herbrandization process (the dual of the satisfiability-preserving Skolemization process) which “freezes” variables into fresh constant symbols. For example, we can consider the tableau \( \{ v_1 : \text{Emp}, v_2 : \text{Dept}, v_1.\text{wrk} = v_2, v_1.\text{ename} = \text{Peter} \} \) to be an Emp-instance with generators \( v_1, v_2 \). In this paper, we may silently pass between a tableau and its frozen instance.
Fig. 8. Example Functorial Data Migrations
Fig. 9. Example Functorial Data Migrations, with Foreign Keys

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>f</th>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>$100</td>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>$250</td>
<td>2</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Sue</td>
<td>$300</td>
<td>3</td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

Fig. 10. An Example $\Sigma$ Data Migration, with Path Equalities

$\forall a. a.is1.is3 = a.is2.is4$
An uber-flower $S \to T$ consists of, for each entity $t \in T$:

- a tableau $(fr_t, wh_t)$ over $S$ and,
- for each attribute $att : t \to t' \in T$, a term $[att]$ in $Terms^\delta(S, fr_t)$, called the return clause for $att$, and
- for each foreign key $fk : t \to t' \in T$, a transform $[fk]$ from the tableau for $t'$ to the tableau for $t$ (note the reversed direction), called the keys clause for $fk$,
- such that an equality-preservation condition holds. We defer a description of this condition until section 4.3.1.

We prefer to use $fr_t, wh_t, [att], [fk]$ notation when discussing a single uber-flower. When we are discussing many queries $Q_1, \ldots, Q_n$, we will write $Q_k(t), Q_k(att), Q_k(fk)$ to indicate $(fr_t, wh_t), [att],$ and $[fk]$, respectively, for $Q_k$.

We usually require that the for clauses in an uber-flower only bind variables to entities, not to types (e.g., $v : \text{Person}$ is allowed, but $v : \text{Nat}$ is not). While not strictly necessary, there are two reasons for preferring this restriction. First, in practice, types will almost always be infinite, so the data migrations induced by a non-restricted uber-flower would often return infinite instances. Second, the restriction ensures that the induced data migrations are domain independent (Abiteboul et al., 1995), allowing some evaluations of uber-flowers to be computable by relational algebra (Spivak & Wisnesky, 2015). Semantically, this restriction means that evaluations of uber-flowers correspond to migrations of the form $\Delta_G \circ \Pi_F$, where $F$ is surjective on attributes, a condition described in section 4.3.4.

In figure 12 we present an uber-flower, from our $\text{Emp}$ schema to our $\text{Emp}$ schema (figure 6), which when evaluated makes each employee their own boss and appends their old boss’s name to their name. Note that Dept is copied over unchanged; only Emp changes.

**Remark.** Because boolean algebra can be equationally axiomatized, evaluation of uber-flowers can express queries that might not be considered conjunctive in certain relational settings. For example, when our type-side contains boolean algebra (figure 11), evaluation of uber-flowers can express queries such as $Q(R) := \{ x \in R \mid P(x) \lor \neg P'(x) = \top \}$. In addition, instances can contain skolem terms of type Bool, implying the existence of truth values besides $\top, \bot$.

\[
\begin{align*}
\top, \bot : \text{Bool} & \to \text{Bool} \\
\land, \lor : \text{Bool} \times \text{Bool} & \to \text{Bool} \\
x \lor y & = y \lor x \\
x \lor (y \lor z) & = (x \lor y) \lor z \\
x \land (y \land z) & = (x \land y) \land z \\
x \lor (y \land z) & = (x \lor y) \land (x \lor z) \\
x \land \neg x & = \top \\
x \lor \bot & = x \\
x \land \top & = x \\
x \lor \bot & = x \\
x \land \bot & = \bot \\
\end{align*}
\]

Fig. 11. An Equational Axiomatization of Boolean Algebra

Similarly, evaluation of uber-flowers can express non-computable queries whenever a type-side contains an equational theory that is Turing-complete. In such cases, the type-side will not have a decidable equality relation and the theorem proving methods employed by the implementation (section 5) may diverge.

**Categorical Remark.** An uber-flower is syntax for a structure which has several equivalent formulations. One is induced by a cospan of schemas (section 4.3.5). Another is a certain kind of profunctor between schemas: let $Q : S \to T$ be an uber-flower on type-side
Then $Q$ denotes a bimodule (Schultz et al., 2017), i.e., a functor $(\langle Q \rangle : \langle T \rangle^{op} \to \langle S\text{-}\text{Inst} \rangle)$ where $\langle Q \rangle(t) = y(t)$ for all $t \in \langle T \rangle$, where $y : \langle S \rangle^{op} \to \langle S\text{-}\text{Inst} \rangle$ is the Yoneda embedding.

### 4.3.1 Verification Conditions for Uber-flower Well-formedness

Let $Q : S \to T$ be an uber-flower. To verify that $Q$ is well-formed, for every equation in $T$ we must verify an induced set of equations on $S$. There are two kinds of equations in $T$ that we must consider, both of which require the notion of path to describe. A path in $T$, namely $p : t_0 \to t_n := f_{k_1} \ldots f_{k_n}$, is a sequence of foreign keys, $f_{k_n} : t_{n-1} \to t_n$, and we write $[p]$ to indicate the substitution $[f_{k_1}] \circ \ldots \circ [f_{k_n}]$ taking each $v : s \in fr_t$ to some term in $\text{Terms}^s(S, fr_t)$ (when $n = 0$ this is the identity substitution on $fr_t$). One kind of equation to verify is an equality between entities of paths with a shared head variable in a singleton entity context:

$$\forall v : t. v.p = v.p' : t'$$

For each variable $u : s \in fr_t$, we must verify the following:

$$S \cup (fr_t, wh_t) \vdash u[p] = u[p'] : t'$$

The other kind of equation to verify is an equality between types of arbitrary terms where paths share head variables in a singleton entity context, for example:

$$\forall v : t. c + v.p'.a = v.p.a : t'$$

Here we must check, for example:

$$S \cup (fr_t, wh_t) \vdash c + [a][p'] = [a][p'] : t'$$

Figure 12 shows an uber-flower and its verification conditions. In that figure $\varphi \leadsto \psi$ means that equation $\varphi \in T$ generates verification condition $\psi$. Rather than simply give the verification conditions, the figure illustrates how the verification conditions are obtained. For example, the first verification condition,

$$\forall v. v.mgr.wrk = v.wrk : \text{Dept} \leadsto \text{Emp} \cup e : \text{Emp} \vdash d[d \mapsto e.wrk][e \mapsto e] = d[d \mapsto e.wrk] : \text{Dept}$$

means that the equation $eq := \forall v. v.mgr.wrk = v.wrk : \text{Dept}$ induces the (tautological) verification condition $\text{Emp} \cup e : \text{Emp} \vdash e.wrk = e.wrk : \text{Dept}$ by starting with a substitution instance $eq' := d.mgr.wrk = d.wrk : \text{Dept}$ of $eq$, and then applying the keys clauses (substitutions) of the uber-flower, $[wrk] = [d \mapsto e.wrk]$ and $[mgr] = [e \mapsto e]$, to $eq'$.

From a practical standpoint, well-formed queries are guaranteed to only materialize instances which obey their data integrity constraints, so runtime checking of data integrity constraints are not needed.
Dept = for d:Dept
  where
  return (dname = d.dname)
  keys (secr = [e -> d.secr])

Emp = for e:Emp
  where
  return (ename = e.ename + e.mgr.ename)
  keys (mgr = [e -> e], wrk = [d -> e.wrk])

Verification conditions:

\( \forall v. v.mgr.wrk = v.wrk : \text{Dept} \rightarrow \text{Emp} \cup e : \text{Emp} \vdash d[d \mapsto e.wrk][e \mapsto e] = d[d \mapsto e.wrk] : \text{Dept} \)

\( \forall v : \text{Dept} \vdash v.secr.wrk : \text{Dept} \rightarrow \text{Emp} \cup d : \text{Dept} \vdash d = d[d \mapsto e.wrk][e \mapsto d.secr] : \text{Dept} \)

\( \forall v.v.mgr.mgr = v.mgr : \text{Emp} \rightarrow \text{Emp} \cup e : \text{Emp} \vdash e[e \mapsto e][e \mapsto e] = e[e \mapsto e] : \text{Emp} \)

Fig. 12. The Uber-flower Promote : Emp \rightarrow Emp

4.3.2 Morphisms of Uber-flowers

Let \( Q_1, Q_2 : S \rightarrow T \) be uber-flowers. For every foreign key \( f_k : t \rightarrow t' \), we have transforms

\( Q_1(f) : f r^1 \rightarrow f r^1_1, Q_2(f) : f r^2 \rightarrow f r^2_1 \).

A morphism \( h : Q_1 \rightarrow Q_2 \) is, for each entity \( t \in T \),

a transform of frozen instances \( h(t) : f r^1 \rightarrow f r^2_1 \), such that for every foreign key \( f_k : t \rightarrow t' \in T \), and every \( v' \in f r^2_1 \),

\[
S \cup wh^2 \vdash [Q_2(f)] [h(t)] = v' [Q_1(f)] [h(t)]
\]

and for every attribute \( att : t \rightarrow t' \in T \),

\[
S \cup wh^2 \vdash Q_2(att) = Q_1(att)[h(t)]
\]

The morphism \( h \) induces a transform \( \text{eval}(h) : \text{eval}(Q_2)(I) \rightarrow \text{eval}(Q_1)(I) \) for each \( S \)-instance \( I \) (in this way, \( \text{evals} \) of uber-flowers are similar to relational conjunctive queries, for which \( Q \rightarrow Q' \) implies \( \forall J, Q'(J) \subseteq Q(J) \)) and a transform \( \text{coeval}(h) : \text{coeval}(Q_1)(J) \rightarrow \text{coeval}(Q_2)(J) \) for each \( T \)-instance \( J \), described in sections 5.4.2 and 5.5.2, respectively.
We now describe *uber-flowers*, $F\text{-Inst} \rightarrow A\text{-Inst}$ as follows. For every entity $c \in C$,

- We define (the frozen $A$-instance of) the for and where clause at $c$ of $Q_2 \circ Q_1$ to be the frozen $A$-instance that is obtained by applying $\text{coeval}(Q_1) : B\text{-Inst} \rightarrow A\text{-Inst}$ to the frozen $B$-instance for $c$ in $Q_2$:

$$ (Q_2 \circ Q_1)(c) := \text{coeval}(Q_1)(Q_2(c)) $$

- Similarly, the transform associated with a foreign key $f k : c \rightarrow c' \in C$ is:

$$ (Q_2 \circ Q_1)(f k) := \text{coeval}(Q_1)(Q_2(f k)) $$

- To define the term associated with an attribute $\text{att} : c \rightarrow c' \in C$ we first define the instance $y(c')$ (a so-called “representable” (Barr & Wells, 1995) instance) to be the $B$-instance with a single generator $v : c'$. We then define a transform / substitution from $y(c')$ to the frozen $B$-instance for $c$ in $Q_2$, namely, $h : y(c') \rightarrow Q_2(c) := v \mapsto Q_2(\text{att})$. Finally, we define:

$$ (Q_2 \circ Q_1)(\text{att}) := \text{coeval}(Q_1)(h) $$

### 4.3.4 Converting Data Migrations to Uber-flowers

Let $F : S \rightarrow T$ be a schema mapping on a type-side $T_y$. In this section we define two uber-flowers, $Q_F : S \rightarrow T$ and $Q^F : T \rightarrow S$ such that:

$$ \text{eval}(Q^F) \cong \Delta_F \quad \text{coeval}(Q^F) \cong \Sigma_F \quad \text{eval}(Q_F) \cong \Pi_F \quad \text{coeval}(Q_F) \cong \Delta_F $$

We now describe $Q^F$ and $Q_F$:

- The for clause for $Q^F : T \rightarrow S$ at entity $s \in S$ is defined to have a single variable, $v_s : F(s)$, and $Q^F$ has an empty where clause. For each foreign key $f k : s \rightarrow s' \in S$, $F(f k)$ is ($\alpha$-equivalent to) a term in $\text{Terms}^{F(s')}(T, \{v_s : F(s)\})$ and we define $Q^F(f k) : Q^F(s') \rightarrow Q^F(s)$ to be the transform $v_{f k} \mapsto F(f k)$. For each attribute $\text{att} : s \rightarrow s' \in S$, $s'$ is a type and $F(\text{att})$ is ($\alpha$-equivalent to) a term in $\text{Terms}^{s'}(T, \{v_s : F(s)\})$ and we define $Q^F(\text{att})$ to be $F(\text{att})$. For example, the $Q^F$ that corresponds to figure 9 is:

```plaintext
N1 := for vN1 : N,
    return name = vN1.name, salary = vN1.salary
keys f = [vN2 -> vN1.f]

N2 := for vN2 : N,
    return age = vN2.age
```

- The frozen instance (for/where clause) for $Q_F : S \rightarrow T$ at entity $t \in T$ is defined to be $\Delta_F(y(t))$, where $y(t)$ is the instance with a single generator $\{v_t : t\}$. For each foreign key $f k : t \rightarrow t' \in T$, we define the transform $Q_F(f k) : Q_F(t') \rightarrow Q_F(t)$ to be $\Delta_F(v_{f k} \mapsto v_t.f k)$. For each attribute $\text{att} : t \rightarrow t' \in T$, $v_t.\text{att} \in \text{Terms}^F(T, \{v_t : t\})$ and $\text{trans}(v_t.\text{att}) \in \text{alg}$$\{v_t : t\}$. Since $\Delta$ preserves type algebras, we have
trans(v.i.att) ∈ talg(ΔF(\{v.i : t\})), and hence we can define Q_F(\text{att}) to be trans(v.i.att).

For example, the Q_F that corresponds to figure 9 is (note that we write ‘x’ to indicate an x ∈ talg(y(N))):

\[ y(N) = vN : N \]
\[ N := \text{for } vN1 : N1, vN2 : N2 \]
\[ \text{where } vN1.f = vN2, vN1.name='vN.name', \]
\[ vN1.salary = 'vN.salary', vN2.salary = 'vN.salary' \]
\[ \text{return name = vN1.name, salary = vN2.salary, age = vN2.age} \]

For Q_F to be an uber-flower that obeys the restriction that variables in for clauses only bind entities and not types, F must be surjective on attributes. This semi-decidable condition implies that for every I, there will be no skolem terms in talg(ΔF(I)), i.e., talg(ΔF(I)) ⊆ Ty. Formally, F is surjective on attributes when for every attribute att : t → i' ∈ T, there exists an entity s ∈ S such that F(s) = t and there exists an e ∈ Terms'(T, \{v : s\}) such that T ⊢ ∀v : F(s). F(e) = v.att.

4.3.5 Converting Uber-flowers to Data Migrations

An uber-flower Q : S → T, where S and T are schemas on the same type-side Ty, induces a data migration eval(Q) : S-Inst → T-Inst ≃ ΔG ∘ ΠF and adjoint data migration coeval(Q) : T-Inst → S-Inst ≃ ΔF ∘ ΔG for some X, F : S → X, G : T → X. In this section, we construct X, F, and G. First, we define a schema X such that S ⊆ X and we define F : S → X to be the inclusion mapping. We start with:

\[ En(X) := En(S) ∪ En(T) \]
\[ \text{Att}(X) := \text{Att}(S) \]
\[ Fk(X) \subseteq Fk(S) \sqcup Fk(T) \]

Then, for each entity t ∈ T and each v : s in f_r_t (the frozen instance for t in Q), we add a foreign key to X:

\[ (v,s,t) : t → s ∈ Fk(X) \]

Let us write \( σ_x \) for the substitution \([v_k → x(v_k,s_k,t), \forall v_k : s_k ∈ f_r_t]\). For each equation e = e' ∈ wh, we add an equation to X:

\[ ∀x : t. eσ_x = e'σ_x ∈ Eq(X) \]

and for each foreign key f_k : t → t' and for each \( v' : s' ∈ f_r_{t'} \), we add an equation to X:

\[ ∀x : t. x.fk.(v',s,t') = v'[f_k]σ_x ∈ Eq(X) \] (1)

This completes the schema X. Finally, we define G : T → X to be the identity on entities and foreign keys, and on attributes we define:

\[ G(\text{att} : t → t') := ∀x : t. [\text{att}]σ_x \]

For example, the schema X for the uber-flower Promote (figure 12) is shown in figure 13. Rather than simply give the equations of the schema X, the figure illustrates how the equations conditions are obtained. For example, the first equation,

\[ ∀x. x.wrk.d = d[d → e.wrk][e → x.e] ≡ x.e.wrk \]

means that schema X contains the equation ∀x. x.wrk.d = x.e.wrk, which was obtained from foreign key wrk : Emp → Dept and for-bound variable d : Dept by equation 1.
4.3.6 ED syntax for Uber-flowers

Intriguingly, the intermediate schema and schema mappings that are created when translating uber-flowers into data migrations, as described in the previous section, suggest an alternative syntax for uber-flowers that resembles the syntax of second-order embedded dependencies (EDs) (Fagin et al., 2005a). The uber-flower **Promote** (figure 12) is shown as a data migration in figure 13, and we can express the intermediate schema and mapping in figure 13 using the following second-order ED:

\[ \exists e : \text{Emp}_{src} \rightarrow \text{Emp}_{dst}, \exists d : \text{Dept}_{src} \rightarrow \text{Dept}_{dst}. \]

\[ \forall x. d(\text{wrk}_{src}(x)) = \text{wrk}_{dst}(e(x)) \land \forall x. e(\text{secr}_{src}(x)) = \text{secr}_{dst}(d(x)) \land \forall x. e(\text{mgr}_{src}(x)) = e(x) \]

We do not understand how our formalism relates to second-order EDs, but the implementation of our formalism in the FQL tool allows users to input uber-flowers using the above second-order ED syntax.

\[ G(\text{name}) := \forall x. [\text{name}]e \rightarrow x.e] \equiv x.e.\text{name} + x.e.\text{mgr}.\text{name} \]

![Diagram](image)

Fig. 13. Uber-flower **Promote** (figure 12) as a Data Migration

5 Implementation: the FQL tool

We have implemented our formalism in the FQL tool, which can be downloaded at [http://categoricaldata.net/fql.html](http://categoricaldata.net/fql.html). In this section we discuss certain implementation issues that arise in negotiating between syntax and semantics, and provide algorithms for key parts of the implementation: deciding equality in equational theories, saturating theories into term models, checking conservativity of equational theories, (co-)evaluating queries, and (co-)pivoting (converting instances into schemas).
5.1 Deciding Equality in Equational Theories

Many constructions involving equational theories, including uber-flower (co-)evaluation, depend on having a decision procedure for provable equality in the theory. A decidable equational theory is said to have a decidable word problem. The word problem is obviously semi-decidable: to prove if two terms (words) \( p \) and \( q \) are equal under equations \( E \), we can systematically enumerate all of the (usually infinite) consequences of \( E \) until we find \( p = q \). However, if \( p \) and \( q \) are not equal, then this enumeration will never stop. In practice, not only is enumeration computationally infeasible, but for uber-flower (co-)evaluation, we require a true decision procedure: an algorithm which, when given \( p \) and \( q \) as input, will always terminate with “equal” or “not equal”. Hence, we must look to efficient, but incomplete, automated theorem proving techniques to decide word problems.

The FQL tool provides a built-in theorem prover based on Knuth-Bendix completion (Knuth & Bendix, 1970): from a set of equations \( E \), it attempts to construct a system of rewrite rules (oriented equations), \( R \), such that \( p \) and \( q \) are equal under \( E \) if and only if \( p \) and \( q \) rewrite to syntactically equal terms (so-called normal forms) under \( R \). We demonstrate this with an example. Consider the equational theory of groups, on the left, in figure 15. Knuth-Bendix completion yields the rewrite system on the right in figure 15. To see how these rewrite rules are used to decide the word problem, consider the two terms \( (a^{-1} \ast a) \ast (b \ast b^{-1}) \) and \( b \ast ((a \ast b)^{-1} \ast a) \). Both of these terms rewrite to 1 under the above rewrite rules; hence, we conclude that they are provably equal. In contrast, the two terms \( 1 \ast (a \ast b) \) and \( b \ast (1 \ast a) \) rewrite to \( a \ast b \) and \( b \ast a \), respectively, which are not syntactically the same; hence, we conclude that they are not provably equal.

![Fig. 15. Knuth-Bendix Completion for Group Theory](image)

**Equations**

\[
\begin{align*}
1 \ast x &= x \\
x^{-1} \ast x &= 1 \\
(x \ast y) \ast z &= x \ast (y \ast z)
\end{align*}
\]

**Rewrite Rules**

\[
\begin{align*}
1 \ast x &\Rightarrow x \\
x^{-1} \ast x &\Rightarrow 1 \\
(x \ast y) \ast z &\Rightarrow x \ast (y \ast z) \\
x^{-1} + x &\Rightarrow y \\
1^{-1} &\Rightarrow 1 \\
x \ast (x^{-1} \ast y) &\Rightarrow y \\
(x \ast y)^{-1} &\Rightarrow y^{-1} \ast x^{-1}
\end{align*}
\]
The details of how the Knuth-Bendix algorithm works are beyond the scope of this paper. However, we make several remarks. First, Knuth and Bendix’s original algorithm (Knuth & Bendix, 1970) can fail even when a rewrite system to decide a word problem exists; for this reason, we use the more modern, “unfailing” variant of Knuth-Bendix completion (Bachmair et al., 1989). Second, first-order, simply-typed functional programs are equational theories that are already complete in the sense of Knuth-Bendix. Third, specialized Knuth-Bendix algorithms (Kapur & Narendran, 1985) exist for theories where all function symbols are unary, such as for the entity and attribute parts of our schemas.

5.2 Saturating Theories into Term Models

Many constructions involving equational theories, including uber-flower (co-)evaluation, depend on having a procedure, called saturation, for constructing finite term models from theories. This process is semi-computable: there are algorithms that will construct a finite term model if it exists, but diverge if no finite term model exists. The FQL tool has two different methods for saturating theories: theories where all function symbols are unary can be saturated using an algorithm for computing Left-Kan extensions (Bush et al., 2003), and arbitrary theories can be saturated by using a decision procedure for the theory’s word problem as follows. Let \( T \) be an equational theory, and define the size of a term in \( T \) to be the height of the term’s abstract syntax tree; for example, \( \text{max}(x.\text{sal}, x.\text{mgr}\text{.sal}) \) has size of three. We construct \( \llbracket T \rrbracket \) in stages: first, we find all not provably equal terms of size 0 in \( T \); call this \( \llbracket T \rrbracket^0 \). Then, we add to \( \llbracket T \rrbracket^0 \) all not provably equal terms of size 1 that are not provably equal to a term in \( \llbracket T \rrbracket^0 \); call this \( \llbracket T \rrbracket^1 \). We iterate this procedure, obtaining a sequence \( \llbracket T \rrbracket^0, \llbracket T \rrbracket^1, \ldots \). If \( \llbracket T \rrbracket \) is indeed finite, then there will exist some \( n \) such that \( \llbracket T \rrbracket^n = \llbracket T \rrbracket^{n+1} = \llbracket T \rrbracket \) and we can stop. Otherwise, our attempt to construct \( \llbracket T \rrbracket \) will run forever: it is not decidable whether a given theory \( T \) has a finite term model.

Note that the model \( \llbracket T \rrbracket \) computed using the above procedure is technically not the canonical term model for the theory; rather, we have constructed a model that is isomorphic to the canonical term model by choosing representatives for equivalence classes of terms under the provable equality relation. Depending on how we enumerate terms, we can end up with different models.

Saturation is used for constructing tables from instances to display to the user, and for (co-)evaluating queries on instances. In general, the type-side \( Ty \) of an instance \( I \) will be infinite, so we cannot saturate the equational theory of the instance directly (i.e., \( \llbracket I \rrbracket \) is often infinite). For example, if the type-side of \( I \) is the free group on one generator \( a \), then \( \llbracket I \rrbracket \) will contain \( a, a*a, a*a*a \), and so on. Hence, as described in section 4, the FQL tool computes the term model for only the entity and attribute part of \( I \) (namely, \( \llbracket I_{EA} \rrbracket \)), along with an instance (equational theory) called the type-algebra of \( I \) (namely, \( \text{talg}(I) \)). The pair (\( \llbracket I_{EA} \rrbracket, \text{talg}(I) \)) is sufficient for all of FQL’s purposes.

The FQL tool supports an experimental feature which we call “computational type-sides”. The mathematics behind this feature have not been fully worked out, but it provides a mechanism to connect FQL to other programming languages. An \( L \)-valued model of \( \text{talg}(I) \) is similar to a (set-valued) model of \( \text{talg}(I) \), except that instead of providing a carrier set for each sort in \( \text{talg}(I) \), an \( L \)-valued model provides a type in \( L \), and instead of providing a function for each symbol in \( \text{talg}(I) \), an \( L \)-valued model provides an ex-
pression in $\mathcal{L}$. For example, if $\mathcal{L} = \text{Java}$, then we can interpret $\text{String}$ as $\text{java.lang.String}$, $\text{Nat}$ as $\text{java.lang.Integer}$, $\text{Nat} + \text{String} : \text{String} \to \text{String}$ as $\text{java.lang.String}$. (Note that our $\mathcal{L}$-models are on $\text{alg}(I)$, not $\text{Ty}$; so an $\mathcal{L}$-model must provide a meaning for the skolem terms in $\text{alg}(I)$, which can be tricky.) Given an $\mathcal{L}$-model $M$, we can take the image of $\mathcal{J}I_{EA}$ under $M$ by replacing each term $\text{alg}(t) \ni t \in \mathcal{J}I_{EA}$ with the value of $t$ in $M$. We write this as $M(\mathcal{J}I_{EA})$. The pair $(M(\mathcal{J}I_{EA}), M)$ can be used by the FQL tool in many situations where $(\mathcal{J}I_{EA}, \text{alg}(A))$ is expected; for example, displaying instances (see figure 16), and (co-)evaluating uber-flowers. Formalizing computational type-sides is an important area for future work.

Fig. 16. The FQL Tool Displaying an Instance with a Computational Type-side

5.3 Deciding that a Theory Conservatively Extends Another

As described in section 4, we may want instances to conservatively extend their schemas, where a theory $Th_2$ conservatively extends $Th_1$ when $Th_1 \vdash \forall \Gamma. \ t = t' : s$ if $Th_2 \vdash \forall \Gamma. \ t = t' : s$ for all $t, t' \in \text{Terms}(Th_1, \Gamma)$ for every $\Gamma$. Conservativity in equational logic is not decidable, and the only system we are aware of that automates conservativity checks in a language at least as expressive as equational logic is CCC (L"{u}th et al., 2005). In this section, we give a simple algorithm that soundly but incompletely checks that a theory $Th_2$ conservatively extends a theory $Th_1$ by showing that $Th_2$ freely extends $Th_1$.

Let $Th_1$ be an equational theory, and let $Th_2$ extend $Th_1$ with new sorts, symbols, and equations. We will simplify the presentation of $Th_2$ by repeatedly looking for equations of the form $g = t$, where $g$ is a generator (0-ary symbol) of $Th_2$ but not of $Th_1$, and $t$ does not contain $g$; we then substitute $g \mapsto t$ in $Th_2$. If after no more substitutions are possible, all equations in $Th_2$ are either reflexive or provable in $Th_1$, then $Th_2$ is conservative (actually, free) over $Th_1$. For example, we can show that the theory:

\{infinity : Nat, undef : Nat, infinity = 0, undef = 1, infinity = undef\}

is not conservative over $Type$ (figure 3), because the simplification process yields the non-reflexive equation $0 = 1$, which is not provable in $Type$. However, the algorithm is far from
complete. The theory:

\[\{+, \times : \text{Nat} \times \text{Nat} \to \text{Nat}, \infty : \text{Nat}, \text{undef} : \text{Nat}, \infty + 1 = \text{undef} + 1\}\]

does not pass our check, even though it is a conservative extension of Type. Developing a better conservativity checker is an important area for future work, lest we inadvertently “damage our ontologies” (Ghilardi et al., 2006). The process of repeatedly substituting \(g \mapsto t\), where \(g\) is a generator in an instance’s type-algebra and \(t\) is a type-side term is also used by the FQL tool to simplify the display of tables by biasing the tables to display e.g., 45 instead of e.g. age.bill when 45 and age.bill are provably equal.

### 5.4 Evaluating Uber-flowers

Although it is possible to evaluate an uber-flower by translation into a data migration of the form \(\Lambda \circ \Pi\), we have found that in practice it is faster to evaluate such queries directly, using an algorithm which extends relational query evaluation. Let \(Q : S \to T\) be an uber-flower and let \(I\) be an \(S\)-instance. We now describe how to compute the instance (theory) \(\text{eval}(Q)(I)\). First, we copy the generators and equations of the type-algebra \(\text{talg}(I)\) into \(\text{eval}(Q)(I)\). Then, for every target entity \(t \in T\), we perform the following:

- We define the generators of entity \(t\) in \(\text{eval}(Q)(I)\) to be those \([I_{EA}]\) environments for \(fr_t\) which satisfy \(wh_t\). Formally, we represent these environments as ground substitutions \(fr_t \mapsto \text{Terms}(S, \emptyset)\) and define, where \(fr_t := \{v^i_j : \tilde{x}^i_j\}:\)

  \[\text{eval}(Q)(I)(t) := \{[v^i_j \mapsto \tilde{e}^i_j] \mid I \vdash \text{eq}[v^i_j \mapsto \tilde{e}^i_j], \forall e \in wh_t, \forall \sigma_i \in [I_{EA}](s_i)\}\]

- For each attribute \(att : t \to t' \in T'\), we have a term \([att] \in \text{Terms}'(S, fr_t)\) from the return clause for \(t\). For every substitution \(\sigma \in \text{eval}(Q)(t)\), we have \([att]\sigma \in \text{Terms}'(S, \emptyset)\), and we add:

  \[\sigma.\text{att} = \text{trans}([att]\sigma) \in \text{eval}(Q)(I)\]

  The reason that \(\text{trans}([att]\sigma) \in \text{Terms}'(\text{eval}(Q)(I), \emptyset)\) is because \(\text{trans}([att]\sigma) \in \text{Terms}'(\text{talg}(I), \emptyset)\) and \(\text{talg}(I) \subseteq \text{eval}(Q)(I)\). See 4.1 for the definition of \(\text{trans}\).

- For each foreign key \(fk : t \to t' \in T\), we have a transform from the frozen instance for \(t'\) to the frozen instance for \(t\) from the keys clause for \(t\), which can be thought of as a substitution \([fk] : fr_t \mapsto \text{Terms}(S, fr_t)\). For every substitution \(\sigma : fr_t \mapsto \text{Terms}(S, \emptyset) \in \text{eval}(Q)(t)\), we add the equation:

  \[\sigma.fk = \sigma \circ [fk] \in \text{eval}(Q)(I)\]

  We know that \(\sigma \circ [fk] \in \text{eval}(Q)(I)(t')\) because \([fk]\) is a transform, not an arbitrary substitution.

Note that in order to build the instance \(\text{eval}(Q)(I)\), we have effectively constructed the term model \([\text{eval}(Q)(I)]_{EA}\] and then “de-saturated” it into an equational theory.

To make the above description concrete, we will now evaluate the uber-flower \(\text{Promote} : \text{Emp} \to \text{Emp}\) from figure 12 on the instance \(\text{Inst}\) from figure 7, which in turn is on the schema \(\text{Emp}\) from figure 6 on the type-side \(\text{Type}\) from figure 3. Our goal is to compute the instance (equational theory) \(\text{eval}(\text{Promote}, \text{Inst})\). We start by copying \(\text{talg}(\text{Inst})\) into
eval(Promote, Inst). Next, we process the tableau. We start with target entity Dept \( \in T \). The from and where clauses give us a set of substitutions \( \{[d \mapsto m], [d \mapsto s]\} \), which are the generators of eval(Promote, Inst) at entity Dept. The return clause adds equations \( [d \mapsto m].dname = m.dname \) and \( [d \mapsto s].dname = s.dname \); note that s.dname is one of the generators from talg(Inst), and s will not be a term in eval(Promote, Inst). The keys clause for secr : Dept \( \rightarrow \) Emp adds equations \( [d \mapsto m].secr = [e \mapsto b] \) and \( [d \mapsto s].secr = [e \mapsto c] \); we have not added \( [e \mapsto b] \) and \( [e \mapsto b] \) to eval(Promote, Inst) yet but we will momentarily. Note that so far, we have simply copied the table Dept from Inst to eval(Promote, Inst), up to isomorphism. We next consider the target entity Emp \( \in T \). The from and where clause give us a set of substitutions \( \{[e \mapsto a], [e \mapsto b], [e \mapsto c], [e \mapsto a.mgr], [e \mapsto b.mgr], [e \mapsto c.mgr]\} \), which are the generators of eval(Promote, Inst) at entity Emp. The return clause adds equations such as \( [e \mapsto a].ename = a.ename + a.mgr.ename \), where a.ename and a.mgr.ename are generators in talg(Inst). The keys clause for mgr : Emp \( \rightarrow \) Dept adds equations such as \( [e \mapsto a].mgr = [e \mapsto a] \), and the keys clause for wrk : Emp \( \rightarrow \) Dept adds equations such as \( [e \mapsto a].wrk = [d \mapsto m] \) (we added \( [d \mapsto m] \) to eval(Promote, Inst) when processing the target entity Emp). The entire instance is displayed in figure 17.

\[
\text{Generators} := \{m.dname, s.dname, a.ename, b.ename, c.ename : String, a.mgr.ename, b.mgr.ename, c.mgr.ename : String,}
\[d \mapsto m], [d \mapsto s] : \text{Dept,}\]
\[e \mapsto a], [e \mapsto b], [e \mapsto c], [e \mapsto a.mgr], [e \mapsto b.mgr], [e \mapsto c.mgr] : \text{Emp}\}
\]

\[
\text{Eqs} := \{a.ename = Al, c.ename = Carl, m.dname = Math,}
\[d \mapsto m].secr = [e \mapsto b], [d \mapsto s].secr = [e \mapsto c], [d \mapsto m].dname = Math, \ldots\}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{ID} & \text{ename} & \text{mgr} & \text{wrk} \\
\hline
[e \mapsto a] & \text{Al + a.mgr.ename} & [e \mapsto a] & [d \mapsto m] \\
\hline
[e \mapsto b] & \text{b.ename + b.mgr.ename} & [e \mapsto b] & [d \mapsto m] \\
\hline
[e \mapsto c] & \text{Carl + c.mgr.ename} & [e \mapsto c] & [d \mapsto s] \\
\hline
[d \mapsto m] & \text{Math} & [e \mapsto b] & [d \mapsto m] \\
\hline
[d \mapsto s] & s.dname & [e \mapsto c] & [d \mapsto m] \\
\hline
\end{array}
\]

Fig. 17. Evaluation of Uber-flower Promote (figure 12) on Inst (figure 7)
5.4.1 Evaluating Uber-flowers on Transforms

Let \( Q : S \rightarrow T \) be an uber-flower and let \( h : I \rightarrow J \) a transform of \( S \)-instances \( I \) and \( J \). Our goal is to define the transform \( \text{eval}(Q)(h) : \text{eval}(Q)(I) \rightarrow \text{eval}(Q)(J) \). For each target entity \( t \in T \), consider the generators in \( \text{eval}(Q)(I)(t) \): they will be the ground substitutions \( f_{r_t} \rightarrow \text{Terms}(S, \emptyset) \) satisfying \( wh_t \). We will map each such substitution to a substitution (generator) in \( \text{eval}(Q)(J)(t) \). Given such a substitution \( \sigma := [v_0 : s_0 \mapsto e_0, \ldots, v_n : s_n \mapsto e_n] \), where \( e_i \in \text{Terms}^s(S, \emptyset) \), we define:

\[
\text{eval}(Q)(\sigma) := [v_0 : s_0 \mapsto \text{nf}_{E_t}(h(e_0)), \ldots, v_n : s_n \mapsto \text{nf}_{E_t}(h(e_n))]
\]

In general, \( h(e_i) \) need not appear in \([J_E]\), so we must use \( nf \) (section 3.2) to find the normal form of \( h(e_i) \) in \( J_E \).

5.4.2 Evaluating Morphisms of Uber-flowers

If \( Q_1, Q_2 : S \rightarrow T \) are uber-flowers, a morphism \( h : Q_1 \rightarrow Q_2 \) is, for each entity \( t \in T \), a morphism from the frozen instance for \( t \) in \( Q_1 \) to the frozen instance for \( t \) in \( Q_2 \), and it induces a transform \( \text{eval}(h) : \text{eval}(Q_2)(I) \rightarrow \text{eval}(Q_1)(I) \) for every \( S \)-instance \( I \); we now show how to compute \( \text{eval}(h) \). Let \( t \in T \) be an entity and \( f_{r_t}^I := \{v_1, \ldots, v_n\} \) be the for clause for \( t \) in \( Q_1 \). The generators of \( \text{eval}(Q_2)(I) \) are substitutions \( \sigma : f_{r_t}^I \rightarrow [J_E] \), and

\[
\text{eval}(h)(\sigma) := [v_1 \mapsto \text{nf}_{E_t}(h(v_1)\sigma), \ldots, v_n \mapsto \text{nf}_{E_t}(h(v_n)\sigma)]
\]

In the above we must use \( nf \) (section 3.2) to find appropriate normal forms.

5.5 Co-Evaluating Uber-flowers

Although it is possible to co-evaluate an uber-flower by translation into a data migration of the form \( \Delta \circ \Sigma \), we have implemented co-evaluation directly. Let \( Q : S \rightarrow T \) be an uber-flower and let \( J \) be a \( T \)-instance. We are not aware of any algorithm in relational database theory that is similar to \( \text{coeval}(Q) \); intuitively, \( \text{coeval}(Q)(J) \) products the frozen instances of \( Q \) with the input instance \( J \) and equates the resulting pairs based on either the frozen part or the input part. We now describe how to compute the \( S \)-instance (theory) \( \text{coeval}(Q)(J) \). First, we copy the generators and equations of the type-algebra \( \text{talg}(J) \) into \( \text{coeval}(Q)(J) \).

We define \( \text{coeval}(Q)(I) \) to be the smallest theory such that, for every target entity \( t \in T \), where \( f_{r_t} := \{v_1 : s_1, \ldots, v_n : s_n\} \),

- \( \forall (v : s) \in f_{r_t}, \text{ and } \forall j \in [J](t), \quad (v, j) : s \in \text{coeval}(Q)(J) \)

- \( \forall e = e' \in \text{wh}_t, \text{ and } \forall j \in [J](t), \quad (e = e')[v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)] \in \text{coeval}(Q)(J) \)

- \( \forall \text{att} : t \rightarrow t' \in T, \text{ and } \forall j \in [J](t), \quad \text{trans}([J](\text{att}(j))) = [\text{att}][v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)] \in \text{coeval}(Q)(J) \)

• recall that \( [\text{att}] \in \text{Terms}^s(S, f_{r_t}) \) is the return clause for attribute \( \text{att} \) and \( \text{trans} : \text{Terms}^s(J, \emptyset) \rightarrow \text{talg}(J) \) is defined in section 4.1.
3 January 2017

\[ \forall f k : t \to t' \in T, \text{ and } \forall j \in [J](t), \text{ and } \forall (v : s') \in f r_{t'}, \\
(v', [J](f k)(j)) = v'[f k]\{v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)\} \in coeval(Q)(J) \]

recall that the substitution \([f k] : f r_{t'} \to Terms(S, f r_t)\) is the keys clause for \(f k\).

The co-evaluation of the uber-flower \(Promote : Emp \to Emp\) from figure 12 on the instance \(Inst\) from figure 7 is in fact isomorphic to the evaluation of \(promote\) (figure 17); the reason is that evaluation and co-evaluation of \(Promote\) are semantically both projections (\(\Delta\)-only operations).

5.5.1 Co-Evaluating Uber-flowers on Transforms

Let \(Q : S \to T\) be an uber-flower and let \(h : I \to J\) be a transform of \(T\)-instances \(I\) and \(J\). Our goal is to define the transform \(coeval(Q)(h) : coeval(Q)(I) \to coeval(Q)(J)\). For each entity \(t \in T\), and for every \((v : s) \in f r_t\), and for every \(j \in [J_E](t)\), we define:

\[ coeval(Q)((v, j)) = (v, nf_{E_k}(h(j))) \]

As was the case for evaluation of uber-flowers on transforms, we must use \(nf\) (section 3.2) to find appropriate normal forms.

5.5.2 Co-Evaluating Morphisms of Uber-flowers

If \(Q_1, Q_2 : S \to T\) are uber-flowers, a morphism \(h : Q_1 \to Q_2\) is, for each entity \(t \in T\), a morphism from the frozen instance for \(t\) in \(Q_1\) to the frozen instance for \(t\) in \(Q_2\), and it induces a transform \(coeval(h) : coeval(Q_1)(J) \to coeval(Q_2)(J)\) for every \(T\)-instance \(J\); in this section, we show how to compute \(coeval(h)\). Let \(t \in T\) be an entity. The generators of \(coeval(Q_1)(J)\) are pairs \((v, j)\) with \(v : s \in f r_{Q_1}^t\) and \(j \in [J_E](t)\). Define:

\[ coeval(h)((v, j)) = (v', j).f k_1 \ldots f k_n \text{ where } h(v) := v'.f k_1 \ldots f k_n \]

5.6 The Unit and Co-Unit of the Co-Eval ⇸ Eval Adjunction

Let \(Q : S \to T\) be an uber-flower. Then \(coeval(Q)\) is left adjoint to \(eval(Q)\), i.e., \(coeval(Q) \dashv eval(Q)\). This means that the set of morphisms \(coeval(Q)(I) \to J\) is isomorphic to the set of morphisms \(I \to eval(Q)(J)\) for every \(I, J\). The unit and co-unit of the adjunction, defined here, describe this isomorphism. Let \(I\) be a \(S\)-instance. The component at \(I\) of the co-unit transform \(\varepsilon_I : coeval(Q)(eval(Q)(I)) \to I\) is defined as:

\[ \varepsilon_I((v_k, [v_1 \mapsto e_1, \ldots, v_n \mapsto e_2])) := e_k, \forall k \in \{1, \ldots, n\} \]

Let \(J\) be a \(T\)-instance. The component at \(I\) of the unit transform \(\eta_J : J \to eval(Q)(coeval(Q), J)\) is defined as:

\[ \eta_I(j) := [v_1 \mapsto (v_1, nf_{E_k}(j)), \ldots, v_n \mapsto (v_n, nf_{E_k}(j))] \]
5.7 Pivot (Instance → Schema)

Let $S$ be a schema on type-side $Ty$ and let $I$ be an $S$-instance. The pivot of $I$ is defined to be a schema $\int S$ on $Ty$, a mapping $F : \int I \rightarrow S$, and a $\int I$-instance $J$, such that $\Sigma_F(J) = I$, defined as follows. First, we copy $\text{tal}(I)$ into $J$. Then for every entity $s \in S$, and every $i \in [\|I_E\|](s)$, we have:

$$i : s \in \int I \quad F(i) := s \quad i : J$$

and for every attribute $att : s \rightarrow s' \in S$,

$$(i, att) : i \rightarrow s' \in \int I \quad F((i, att)) := att \quad i.(i, att) = \text{trans}([\|I_E\|][att](i)) \in J$$

where $\text{trans}$ is defined in section 4.1, and for every foreign key $fk : s \rightarrow s' \in S$,

$$(i, fk) : i \rightarrow [\|I_E\|](fk)(i) \in \int I \quad F((i, fk)) := fk \quad i.(i, fk) = [\|I_E\|](fk)(i) \in J$$

In addition, for each generator $g := e.fk_1, \ldots, fk_n, att$ in $\text{tal}(I)$, we have a term $[g] \in \text{Terms}(J, \emptyset)$ defined as:

$$nf_I(e) \cdot (nf_I(e), fk_1) \cdot (nf_I(e, fk_1), fk_2) \cdot \ldots \cdot (nf_I(e, fk_1, \ldots, fk_n), att)$$

We add $g = [g]$ to $J$. See figure 18 for an example on type-side $Type$ (figure 3). Our pivot operation is related to the Grothendieck construction (Barr & Wells, 1995).

5.8 Co-Pivot (Instance → Schema)

Let $S$ be a schema on type-side $Ty$ and let $I$ be an $S$-instance. The co-pivot of $I$ is defined to be a schema $\int S$ on $Ty$, a mapping $F : \int \rightarrow \int I$, and a $\int I$-instance $J$, such that $\Delta_F(J) \cong I$, defined as follows. In fact, we obtain an inclusion $F : \int S \hookrightarrow \int I$, so $J$ is superfluous (can be recovered from $F$). First, we add a single entity $\star$ to $S$, and for every generator $g : s \in \text{tal}(I)$, an attribute $g_A$:

$$\star \in \int I \quad g_A : \star \rightarrow s \in \int I$$

Then for every entity $s \in S$, and $i \in [\|I_E\|](s)$ we have a foreign key:

$$i_E : \star \rightarrow s \in \int I \quad F(s) := s$$

and additionally, for every attribute $att : s \rightarrow s'$, and every foreign key $fk : s \rightarrow s''$, we have:

$$\forall x : \star. x.i_E.att = x.\text{trans}(i.att)_A \quad \forall x : \star. x.i_E.fk = x.nf_I(i, fk)_E$$

where $\text{trans}$ is defined in section 4.1. See figure 19 for an example on type-side $Type$ (figure 3).
Fig. 18. Example of a Pivot

Fig. 19. Example of a Co-Pivot
6 A Pushout Design Pattern for Algebraic Data Integration

In this section we describe a design pattern for integrating two instances on two different schemas, relative to an overlap schema and an overlap instance, using the formalism defined in this paper. The overlap schema is meant to capture the schema elements common to the two input schemas (e.g., that Patient and Person should be identified), and the overlap instance is meant to capture the instance data that should be identified (e.g., that “Pete” and “Peter” are the same person).

Although the $\Sigma, \Delta, \Pi$ data migration functors are sufficient to express queries and data migrations, as unary operations they are insufficient to express data integrations, which involve many schemas and instances and their relationships (Doan et al., 2012). So, we need to define additional operations on our formalism as we develop our pattern. In particular, we will define pushouts (Barr & Wells, 1995) of schemas and instances and use pushouts as the basis of our pattern. The idea of using pushouts to integrate data is not new and was for example discussed in Goguen (2004); our goal here is to express this pattern using our formalism. With pushouts defined, we describe our pattern at an abstract level, and then we describe a medical records example that uses the pattern. This example is built into the FQL tool as the “Pharma Colim” example.

6.1 Pushouts of Schemas and Instances

Let $\mathcal{C}$ be a category and $F_1 : S \to S_1$ and $F_2 : S \to S_2$ be morphisms in $\mathcal{C}$. A pushout of $F_1, F_2$ is any pair $G_1 : S_1 \to T$ and $G_2 : S_2 \to T$ such that $G_2 \circ F_2 = G_1 \circ F_1$, with the property that for any other pair $G'_1 : S_1 \to T'$ and $G'_2 : S_2 \to T'$ for which $G'_2 \circ F_2 = G'_1 \circ F_1$, there exists a unique $t : T \to T'$ such that $G'_1 = t \circ G_1$ and $G'_2 = t \circ G_2$, as shown in figure 20.

![Fig. 20. Pushouts](image)

Our formalism admits pushouts of schemas and instances. Let $S := (\mathit{Ens}, \mathit{Symbols}, \mathit{Eqs})$, $S_1 := (\mathit{Ens}_1, \mathit{Symbols}_1, \mathit{Eqs}_1)$ and $S_2 := (\mathit{Ens}_2, \mathit{Symbols}_2, \mathit{Eqs}_2)$ be schemas on some type-side, where $\mathit{En}$ indicates entities, $\mathit{Symbols}$ indicates foreign keys and attributes, and $\mathit{Eqs}$ indicates schema, but not type-side, equations. Let $F_1 : S \to S_1$ and $F_2 : S \to S_2$ be schema mappings. The pushout schema $T$ is defined with entities:

$\mathit{Ens}_T := (\mathit{Ens}_1 \sqcup \mathit{Ens}_2) / \sim$
where $\sqcup$ means disjoint union, $\sim$ is the least equivalence relation such that $F_1(e) \sim F_2(e)$ for every entity $e \in S$, and $/$ means set-theoretic quotient. We define further that:

$$Symbols_T := Symbols_1 \sqcup Symbols_2$$

$$Eqs_T := Eqs_1 \sqcup Eqs_2 \sqcup$$

$$\{v_1 : F_1(s_1), \ldots, v_n : F_1(s_n), F_1(e) = F_2(e) : F_1(s) \mid e : s_1 \times \ldots \times s_n \rightarrow s \in Symbols_S\}$$

and the schema mappings $G_1$ and $G_2$ inject each entity into its equivalence class under $\sim$ and inject each symbol appropriately.

Pushouts of instances are slightly easier to define than pushouts of schemas. Let $S := (Gens, Eqs)$, $S_1 := (Gens_1, Eqs_1)$ and $S_2 := (Gens_2, Eqs_2)$ be instances on some schema, where $Gens$ indicates generators and $Eqs$ indicates instance, but not schema, equations. Let $F_1 : S \rightarrow S_1$ and $F_2 : S \rightarrow S_2$ be transforms. The pushout instance $T$ is:

$$Gen_T := Gens_1 \sqcup Gens_2$$

$$Eqs_T := Eqs_1 \sqcup Eqs_2 \sqcup \{F_1(e) = F_2(e) : s \mid e : s \in Gens_S\}$$

and the transforms $G_1, G_2$ are inclusions.

The pushout schemas and instances defined in this section are canonical, but the price for canonicity is that their underlying equational theories tend to be highly redundant (i.e., have many symbols that are provably equal to each other). These canonical pushout schemas and instances can be simplified, and in fact FQL can perform simplification, but the simplification process is necessarily non-canonical. In our extended medical example (figure 21), we will use a simplified non-canonical pushout schema.

6.2 Overview of the Pattern

Given input schemas $S_1, S_2$, an overlap schema $S$, and mappings $F_1, F_2$ as such:

$$S_1 \xleftarrow{F_1} S \xrightarrow{F_2} S_2$$

we propose to use their pushout:

$$S_1 \xleftarrow{G_1} T \xrightarrow{G_2} S_2$$

as the integrated schema. Given input $S_1$-instance $I_1$, $S_2$-instance $I_2$, overlap $S$-instance $I$ and transforms $h_1 : \Sigma F_1(I) \rightarrow I_1$ and $h_2 : \Sigma F_2(I) \rightarrow I_2$, we propose the pushout of:

$$\Sigma G_1(I_1) \xleftarrow{\Sigma G_1(h_1)} \Sigma G_1 \circ F_1(I) = \Sigma G_2 \circ F_2(I) \xrightarrow{\Sigma G_2(h_2)} \Sigma G_2(I_2)$$

as the integrated $T$-instance.

Because pushouts are initial among the solutions to our design pattern, our integrated instance is the “best possible” solution in the sense that if there is another solution to our pattern, then there will be a unique transform from our solution to the other solution. In functional programming terminology, this means our solution has “no junk” (extra data that should not appear) and “no noise” (missing data that should appear) (Mitchell, 1996). Initial solutions also appear in the theory of relational data integration, where the chase constructs weakly initial solutions to data integration problems (Fagin et al., 2005b).
6.3 An Example of the Pattern

As usual for our formalism, we begin by fixing a type-side. We choose the Type type-side from figure 3. Then, given two source schemas $S_1$, $S_2$, an overlap schema $S$, and mappings $F_1$, $F_2$ as input, our goal is to construct a pushout schema $T$ and mappings $G_1$, $G_2$, as shown in figure 21. In that figure’s graphical notation, an attribute $\bullet_A \rightarrow_{\text{att}} \bullet_{\text{String}}$ is rendered as $\bullet_A - o_{\text{att}}$. Next, given input $S_1$-instance $I_1$, $S_2$-instance $I_2$, overlap $S$-instance $I$ and morphisms $h_1 : \Sigma_{F_1}(I) \rightarrow I_1$ and $h_2 : \Sigma_{F_2}(I) \rightarrow I_2$, our goal is to construct a pushout $T$-instance $J$ and morphisms $j_1$, $j_2$, as shown in figure 22; note that in this figure, by $K \gamma_{(F,K)} L$ we mean that $h : \Sigma_K(K) \rightarrow L$, and that we use variable font for generators, sans serif font for sorts and symbols, and regular font for terms in the type-side.

Our example involves integrating two different patient records databases. In $S_1$, the “observations” done on a patient have types, such as heart rate and blood pressure. In $S_2$, the observations still have types, but via “methods” (e.g., patient self-report, by a nurse, by a doctor, etc; for brevity, we have omitted attributes for the names of these methods). Another difference between schemas is that $S_1$ assigns each patient a gender, but $S_2$ does not. Finally, entities with the same meaning in both schemas can have different names (Person vs Patient, for example).

We construct the overlap schema $S$ and mappings $F_1$, $F_2$ (figure 21) by thinking about the meaning of $S_1$ and $S_2$; alternatively, schema-matching techniques (Doan et al., 2012) can be used to construct overlap schemas. In this example, it is clear that $S_1$ and $S_2$ share a common span $P_f \leftarrow O \rightarrow g T$ relating patients, observations, and observation types; in $S_1$, this span appears verbatim but in $S_2$, the path $g$ corresponds to $g_2 \circ g_1$. This common span defines the action of $F_1$ and $F_2$ on entities and foreign keys in $S$, so now we must think about the attributes in $S$. For purposes of exposition we assume that the names of observation types (“BP”, “Weight”, etc.) are the same between the instances we are integrating. Hence, we include an attribute for observation type in the overlap schema $S$. On the other hand, we do not assume that patients have the same names across the instances we are integrating; for example, we have the same patient named “Pete” in one database and “Peter” in the other database. Hence, we do not include an attribute for patient name in $S$. If we did include an attribute for patient name, then the pushout schema would have a single attribute for patient name, and the integrated instance would include the equation “Pete” = “Peter” : String. We would violate the conservative extension property (see section 5.3), which is not a desirable situation (Ghilardi et al., 2006). So, our design pattern explicitly recommends that when two entities in $S_1$ and $S_2$ are identified in an overlap schema, we should only include those attributes which appear in both $S_1$ and $S_2$ for which the actual values of these attributes will correspond in the overlap instance. As another example of this phenomenon, to a first approximation, attributes for globally unique identifiers such as social security numbers can be added to overlap schemas, but attributes for non-standard vocabularies such as titles (e.g., CEO vs Chief Executive Officer) should not be added to overlap schemas.

With the overlap schema in hand, we now turn toward our input data. We are given two input instances, $I_1$ on $S_1$ and $I_2$ on $S_2$. Entity-resolution (ER) techniques (Doan et al., 2012) can be applied to construct an overlap instance $I$ automatically. Certain ER techniques can even be implemented as queries in the FQL tool, as we will describe in the next section. But for the purposes of this example we will construct the overlap instance by hand.
We first assume there are no common observations across the instances; for example, perhaps a cardiologist and nephrologist are merging their records. We also assume that the observation type vocabulary (e.g., “BP” and “Weight”) are standard across the input instances, so we put these observation types into our overlap instance. Finally, we see that there is one patient common to both input instances, and he is named Peter in $I_1$ and Pete in $I_2$, so we add one entry for Pete/Peter in our overlap instance. We have thus completed the input to our design pattern (figure 22).

In the output of our pattern (figure 22) we see that the observations from $I_1$ and $I_2$ were disjointly unioned together, as desired; that the observation types were (not disjointly) unioned together, as desired; and that Pete and Peter correspond to the same person in the integrated instance. In addition, we see that Jane could not be assigned a gender.

### 6.4 Entity-resolution Using Uber-flowers

Let schemas $S, S_1, S_2$, mappings $F_1 : S \rightarrow S_1, F_2 : S \rightarrow S_2$, and $S_1$-instance $I_1$ and $S_2$-instance $I_2$ be given. In practice, we anticipate that sophisticated entity-resolution (Doan et al., 2012) techniques will be used to construct the overlap $S$-instance $I$ and transforms $h_1 : \Sigma_{F_1}(I) \rightarrow I_1$ and $h_2 : \Sigma_{F_2}(I) \rightarrow I_2$. However, it is possible to perform a particularly simple kind of entity resolution directly by evaluating uber-flowers.

Technically, the overlap instance used in the pushout pattern should not be thought of as containing resolved (unified) entities; rather, it should be thought of as containing the record linkages between entities that will resolve (unify) (Doan et al., 2012). The
pushout resolves entities by forming equivalence classes of entities under the equivalence relation induced by the links. As the size of the overlap instance gets larger, the size of the pushout gets smaller, which is the opposite of what would happen if the overlap instance contained the resolved entities themselves, rather than the links between them. For example, let $A$ and $B$ be instances on some schema that contains an entity Person, and let $A(\text{Person}) := \{a_1, a_2\}$ and $B(\text{Person}) := \{b_1, b_2\}$. If the overlap instance $O$ has $O(\text{Person}) := \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$, then this does not mean that the pushout will have four people; rather, the pushout will have one person corresponding to $\{a_1, a_2, b_1, b_2\}$, because these four people are linked. Intuitively, the overlap instance $I$ constructed by the technique in this section is (isomorphic to) a sub-instance of $\Delta_{F_1}(I_1) \times \Delta_{F_2}(I_2)$, where $\times$ denotes a kind of product of instances which we will not define here.
Let \( \text{inc}_1 : S_1 \rightarrow S_1 + S_2 \) and \( \text{inc}_2 : S_2 \rightarrow S_1 + S_2 \) be inclusion schema mappings, and define the \( S_1 + S_2 \) instance \( I' := \text{inc}_1(I_1) + \text{inc}_2(I_2) \). This instance will contain \( I_1 \) and \( I_2 \) within it, and will contain nothing else. (Here \( X + Y \) means co-product, which is equivalent to the pushout of \( X \) and \( Y \) over the empty schema or instance).

\[
\begin{align*}
Q_1 : S_1 + S_2 \rightarrow S := & \quad O := \text{for } o_1 \text{ in Observation1} \\
& \quad \text{keys } f = [p_1 \rightarrow o_1.f], \ g = [t_1 \rightarrow o_1.g] \\
& \quad P := \text{for } p_1 \text{ in Person} \\
& \quad T := \text{for } t_1 \text{ in Obstype, return } att = t_1.\text{att}
\end{align*}
\]

\[
q_1 \downarrow
\]

\[
\begin{align*}
Q_1 : S_1 + S_2 \rightarrow S := & \quad O := \text{for } o_1 \text{ in Observation1, } o_2 \text{ in Observation2} \\
& \quad \text{where } 1 = 2 \\
& \quad \text{keys } f = [p_1 \rightarrow o_1.f, \ p_2 \rightarrow o_2.f], \\
& \quad g = [t_1 \rightarrow o_1.g, \ t_2 \rightarrow o_2.g_1.g_2] \\
& \quad P := \text{for } p_1 \text{ in Person, } p_2 \text{ in Patient} \\
& \quad \text{where } \text{true} = \text{strMatches}(p_1.\text{PatientAtt}, \ p_2.\text{PersonAtt}) \\
& \quad T := \text{for } t_1 \text{ in Obstype, } t_2 \text{ in Type} \\
& \quad \text{where } t_1.\text{ObsTypeAtt} = t_2.\text{TypeAtt} \\
& \quad \text{return } att = t_1.\text{ObsTypeAtt}
\end{align*}
\]

\[
q_2 \uparrow
\]

\[
\begin{align*}
Q_2 : S_1 + S_2 \rightarrow S := & \quad O := \text{for } o_2 \text{ in Observation2} \\
& \quad \text{keys } f = [p_1 \rightarrow o_2.f], \ g = [t_1 \rightarrow o_2.g_1.g_2] \\
& \quad P := \text{for } p_2 \text{ in Patient} \\
& \quad T := \text{for } t_2 \text{ in Type, return } att = t_2.\text{att}
\end{align*}
\]

The function \( \text{strComp} \) can be defined using equations, although the FQL tool allows such functions to be defined using java code (see section 5). With the String-comparator in hand, we can now define \( Q : S_1 + S_2 \rightarrow S \) as in figure 23. The overlap instance \( I \) is defined as \( \text{eval}(Q)(I') \). To construct \( h_n : \Sigma F_n(I) \rightarrow I_n \) for \( n = 1, 2 \), we define projection queries \( Q_n : \Sigma F_n(I) \rightarrow I_n \) for \( n = 1, 2 \).
S₁ + S₂ → S and inclusion query morphisms qₙ : Qₙ → Q as in figure 23 as follows. We start with the induced transforms for qₙ, then apply Σ_Fₙ, then compose with the isomorphism eval(Qₙ)(I') ≅ Δ_Fₙ(Iₙ), and then compose the co-unit ε of the Σ_Fₙ ⊣ Δ_Fₙ adjunction, to obtain hₙ as in figure 23.

The result of running figure 23 on the medical records data I₁, I₂ from figure 22 is the overlap instance I from figure 22. To compute the isomorphism eval(Qₙ)(I') → Δ_Fₙ(Iₙ), we note that the generators of eval(Qₙ)(I') will be singleton substitutions such as \([vₙ ↦ in jₙ aₙ]\) where aₙ is a term in Σ_inc(Iₙ) and in jₙ means co-product injection. But incₙ is an inclusion, so aₙ is a term in I₁. Because we compute Δ_Fₙ by translation into an uber-flower similar to Qₙ, the generators of Δ_Fₙ(I') will have a similar form: \([v'_n ↦ a_n]\) which defines the necessary isomorphism. When all schemas are disjoint and variables are chosen appropriately, the isomorphism can be made an equality.

6.5 Further Patterns

Pushouts have a dual, called a pullback, obtained by reversing the arrows in the pushout diagram (figure 20). Pushouts can also be generalized to co-limits, which can be thought of as n-ary pushouts (Barr & Wells, 1995). Exploring applications of pullbacks and co-limits to data integration, as well as finding other useful design patterns for algebraic data integration, are important areas for future work.

7 Conclusion

In this paper we have described an algebraic formalism for integrating data, and work continues. In the short term, we aim to formalize our experimental “computational typesides”, and to develop a better conservativity checker. In the long term, we are looking to develop other design patterns for data integration and to study their compositions, and we are developing an equational theorem prover tailored to our needs. In addition to these concrete goals, we believe there is much to be gained from the careful study of the differences between our formalism, with its category-theoretic semantics, and the formalism of embedded dependencies, with its relational semantics (Doan et al., 2012). For example, there is a semantic similarity between our Σ operation and the chase; as another example, so far we have found no relational counterpart to the concept of query “co-evaluation”; and finally, our uber-flower queries may suggest generalizations of comprehension syntax (Grust, 2004).

8 Acknowledgements

The authors would like to thank David Spivak and Peter Gates. Patrick Schultz was supported by AFOSR grant FA9550-14-1-0031, ONR grant N000141310260, and NASA grant NNH13ZEA001N. Ryan Wisnesky was supported by NIST SBIR grant 70NANB15H290.

References


## Contents

1 Introduction 1  
1.1 Background 2  
1.2 Outline 3  
1.3 Related Work 3  
1.3.1 vs The Functional Data Model 3  
1.3.2 vs Functional Programming 4  
1.3.3 vs Relational Data Integration 4  
1.3.4 vs The Functorial Data Model 5  

2 Review of Category Theory 6  
2.1 Categories with Products 6  
2.2 Functors 7  
2.3 Natural Transformations 7  
2.4 Adjunctions 8  

3 Review of Multi-sorted Equational Logic 8  
3.1 Syntax 9  
3.2 Semantics 11  

4 An Equational Formalism for Functorial Data Migration 12  
4.1 Type-sides, Schemas, Instances, Mappings, and Transforms 13  
4.2 Functorial Data Migration 17  
4.3 Uber-flower Queries 18  
4.3.1 Verification Conditions for Uber-flower Well-formedness 22  
4.3.2 Morphisms of Uber-flowers 23  
4.3.3 Composing Uber-flowers 24  
4.3.4 Converting Data Migrations to Uber-flowers 24  
4.3.5 Converting Uber-flowers to Data Migrations 25  
4.3.6 ED syntax for Uber-flowers 26  

5 Implementation: the FQL tool 26  
5.1 Deciding Equality in Equational Theories 27  
5.2 Saturating Theories into Term Models 28  
5.3 Deciding that a Theory Conservatively Extends Another 29  
5.4 Evaluating Uber-flowers 30  
5.4.1 Evaluating Uber-flowers on Transforms 32  
5.4.2 Evaluating Morphisms of Uber-flowers 32  
5.5 Co-Evaluating Uber-flowers 32  
5.5.1 Co-Evaluating Uber-flowers on Transforms 33  
5.5.2 Co-Evaluating Morphisms of Uber-flowers 33  
5.6 The Unit and Co-Unit of the Co-Eval ⊣ Eval Adjunction 33  
5.7 Pivot (Instance → Schema) 34  
5.8 Co-Pivot (Instance → Schema) 34  

6 A Pushout Design Pattern for Algebraic Data Integration 36  
6.1 Pushouts of Schemas and Instances 36  
6.2 Overview of the Pattern 37  
6.3 An Example of the Pattern 38  
6.4 Entity-resolution Using Uber-flowers 39  
6.5 Further Patterns 42  

7 Conclusion 42  
8 Acknowledgements 42  
References 42
List of Figures

1. A Schema and Instance in the Functorial Data Model
2. The Attribute Problem
3. The Multi-sorted Equational Theory Type
4. Inference Rules for Multi-sorted Equational Logic
5. The Term Model \([\mathcal{Type}]\) of Theory Type (figure 3)
6. Schema \(\text{Emp}\), on Type-side Type (figure 3)
7. Instance \(\text{Inst}\) on Schema \(\text{Emp}\) (figure 6)
8. Example Functorial Data Migrations
9. Example Functorial Data Migrations, with Foreign Keys
10. An Example \(\Sigma\) Data Migration, with Path Equalities
11. An Equational Axiomatization of Boolean Algebra
12. The Uber-flower \(\text{Promote} : \text{Emp} \rightarrow \text{Emp}\)
13. Uber-flower \(\text{Promote}\) (figure 12) as a Data Migration
14. The FQL Tool Displaying an Instance
15. Knuth-Bendix Completion for Group Theory
16. The FQL Tool Displaying an Instance with a Computational Type-side
17. Evaluation of Uber-flower \(\text{Promote}\) (figure 12) on \(\text{Inst}\) (figure 7)
18. Example of a Pivot
19. Example of a Co-Pivot
20. Pushouts
21. Medical Records Schema Integration
22. Medical Records Data Integration
23. Entity-Resolution on Medical Records using Uber-flowers