An Equational Formalism and Design Pattern for Functorial Data Integration

Ryan Wisnesky
Categorical Informatics, Inc.
and
Patrick Schultz
Department of Mathematics
Massachusetts Institute of Technology

Abstract

In this paper we present the first truly practical formalism for querying and integrating data using the functorial data model. In our formalism, database schemas and instances are 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: $\Sigma_F : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$, defined by substitution along F, which has a right adjoint $\Delta_F : T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst}$, which in turn has a right adjoint $\Pi_F : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$. 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 the *functorial data model* (Spivak, 2012), which originated in the late 1990s (Fleming *et al.*, 2002), a database schema is a finitely presented category (Barr & Wells, 1995) (essentially, a directed multi-graph and path equality constraints) and a database instance on a schema S is a set-valued functor from S (essentially, a set of tables). The database instances on a schema S constitute a category, denoted S–**Inst**, and a functor (a.k.a schema mapping (Fagin *et al.*, 2005)) $F : S \to T$ between schemas S and T induces three adjoint data migration functors: $\Delta_F : T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst}$, defined as $\Delta_F(I) := I \circ F$, and the left and right adjoints to Δ_F, respectively: $\Sigma_F : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$ and $\Pi_F : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$.

The Σ, Δ, Π data migration functors provide a category-theoretic alternative to the traditional relational operations for both querying data (relational algebra) and migrating / integrating data (the chase (Fagin *et al.*, 2005)). The relative advantages and disadvantages of the functorial data model and the relational model are still being studied, but see (Spivak, 2012) for a discussion. In this paper we describe what we believe to be the first truly practical formalism for both query and data integration using (an extension of) the functorial data model, and use the formalism to describe a commonly occurring data integration design pattern by way of an example. We have implemented the formalism and the example in a prototype data integration system FQL, available at http://categoricaldata.net.

Hereafter we refer to the data model developed in this paper as “the functorial data model”, and refer to the data model of (Fleming *et al.*, 2002) as “the original functorial data model”.

The mathematics behind our formalism are worked out in detail in (Schultz *et al.*, 2016) using category theory. In this paper, our goal is to present our formalism and its implementation using only basic concepts from functional programming. In particular, we do not assume readers are familiar with category theory, although we do make “categorical remarks”, and proofs are deferred to (Schultz *et al.*, 2016). Our formalism is based entirely on multi-sorted equational logic: database schemas and instances are presented by equational theories of a certain kind. A schema mapping $F : C \to D$ is defined as a morphism of equational theories C and D, and the Σ_F data migration is defined as substitution along F. The conditions we impose on our equational theories guarantee that Σ_F has a right adjoint, Δ_F, which in turn has a right adjoint, Π_F. Getting the conditions right is tricky: even slight changes to our conditions result in at least one of Σ, Δ, Π no longer existing, or losing its adjointness property. We can recover the original functorial data model of (Fleming *et al.*, 2002), and the extension of the original model in (Spivak & Wisnesky, 2015), by imposing restrictions on our equational theories. Our query language, which is a generalization of for/where/return queries (Abiteboul *et al.*, 1995), corresponds semantically to combinations of Σ, Δ, Π operations and generalizes the idea from relational database theory that conjunctive queries can be represented as “frozen” instances (Abiteboul *et al.*, 1995). Our design pattern for data integration is based on a categorical construction called a “pushout” (Barr & Wells, 1995).

1.1 Related Work

Our motivation for extending the original functorial data model (Fleming *et al.*, 2002) is that the original model is difficult to use for data integration: the original model represents database instances as set-valued functors, but 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 that must be preserved by morphisms (e.g., Bill), and meaningless identifiers that need not be preserved (e.g., an auto-generated ID). (In contexts outside of data integration, such as relational query, there may not be a need to distinguish two types of values.) For example, the following situation, which holds in the original model, is untenable for data integration:

ID	Name	Age	Salary
1	Alice	20	$100
2	Bob	20	$250
3	Sue	30	$300

$\cong$ (good)

ID	Name	Age	Salary
4	Alice	20	$100
5	Bob	20	$250
6	Sue	30	$300

$\cong$ (bad)

ID	Name	Age	Salary
1	Amy	20	$100
2	Bill	20	$250
3	Susan	30	$300

Fig. 1. The Attribute Problem

Several approaches to this problem have been proposed, including (Johnson *et al.*, 2002) and (Spivak & Wisnesky, 2015). The latter paper also gives (partial) translations between the Δ, Σ, Π data migration functors and the $\sigma, \times, \cup, \pi$ fragment of relational algebra.

Pushouts, the basis of our data integration design pattern, are investigated for data integration purposes in (Alagic & Bernstein, 2001). In that paper, the authors use the theory of institutions (Goguen & Burstall, 1984) to describe a design pattern for data integration. The functorial data model is an institution, but their work differs from ours in key respects. First, they are primarily concerned with the Δ, rather than Σ, 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 (the functorial data model does). Their main theorem is that in any institution (including ours), 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 – the left adjoint to Δ, which exists in the functorial data model but not in all institutions – to go the other way: pushing data from source schemas to the integrated schema. See (Goguen, 2004) for more information about data integration in arbitrary institutions. In the more general setting of algebraic specification, pushouts have received considerable attention as a means to integrate specifications (Blum *et al.*, 1987).

1.2 Outline

This paper is divided into four sections. In Section 2 we review multi-sorted equational logic. In Section 3 we describe how we use equational theories to define schemas, instances, and the other artifacts necessary to perform data integration and query. In Section 4 we describe how our formalism is implemented in the FQL tool. In Section 5 we describe our design pattern for functorial data integration and include an extended example. We think of this paper as defining a reference implementation of (Schultz *et al.*, 2016) which a motivated reader could use to re-implement the FQL tool and its library of examples.

2 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 (Adámek *et al.*, 2011). Reader familiars with equational logic can safely skim or skip this section. To save space, we will use the phrase "theory", rather than "multi-sorted equational theory", in this section.

2.1 Syntax

A *signature* Sig consists of:

1. A set $Sorts$ whose elements are called *sorts*,
2. A set $Symbols$ of pairs $(f, s_1 \times \ldots \times s_k \to s)$ with $s_1, \ldots, s_k, s \in Sorts$ and no f occurring in two distinct pairs. We write $f : X$ instead of $(f, X) \in Symbols$. When

$k = 0$, we may call f a *constant symbol* and write $f : s$ instead of $f : \to s$. Otherwise, we may call f a *function symbol*.

We assume we have some infinite set $\mathscr{V} := \{v_1, v_2, \dots\}$, whose elements we call *variables* and which are assumed to be distinct from any sort or symbol we ever consider. A *context* Γ is defined as a finite set of variable-sort pairs, with no variable given more than one sort:

$$\Gamma := \{v_1 : s_1, \dots, v_k : s_k\}$$

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

1. $x \in Terms^s(Sig, \Gamma)$, if $x : s \in \Gamma$,
2. $f(t_1, \dots, t_k) \in Terms^s(Sig, \Gamma)$, if $f : s_1 \times \dots \times s_k \to s$ and $t_i \in Terms^{s_i}(Sig, \Gamma)$ for $i = 1, \dots, k$. 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 $Terms^s(Sig, \emptyset)$ as the set of *ground* terms of sort s. We will write $Terms(Sig, \Gamma)$ for the set of all terms in context Γ, i.e., $\bigcup_s Terms^s(Sig, \Gamma)$. 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:

$$v[v \mapsto t] = t \qquad v'[v \mapsto t] = v'\ (v \neq v') \qquad f(t_1, \dots, t_n)[v \mapsto t] = f(t_1[v \mapsto t], \dots, t_n[v \mapsto t])$$

We will only make use of substitutions that are sort-preserving; i.e., to consider $e[v \mapsto t]$, we require $e \in Terms(Sig, \Gamma)$ for some Γ such that $v : s \in \Gamma$ and $t \in Terms^s(Sig, \Gamma)$. We may abbreviate $[v_1 \mapsto t_1] \circ [v_2 \mapsto t_2]$ as $[v_2 \mapsto t_2, v_1 \mapsto t_1]$.

An *equation* over Sig is a formula $\forall \Gamma.\ t_1 = t_2 : s$ with $t_1, t_2 \in Terms^s(Sig, \Gamma)$; we will omit the $: s$ when it is not relevant. A *theory* is a pair of a signature and a set of equations over that signature. In this paper, we will make use of the following theory we call *Type*:

$$Sorts := \{\mathsf{Nat},\ \mathsf{Char},\ \mathsf{String}\}$$

$$Symbols := \{\mathsf{zero} : \mathsf{Nat},\ \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{A} : \mathsf{Char}, \mathsf{B} : \mathsf{Char}, \dots, \mathsf{Z} : \mathsf{Char},$$

$$\mathsf{nil} : \mathsf{String},\ \mathsf{cons} : \mathsf{Char} \times \mathsf{String} \to \mathsf{String},\ \mathsf{length} : \mathsf{String} \to \mathsf{Nat}, + : \mathsf{String} \times \mathsf{String} \to \mathsf{String}\}$$

$$Equations := \{\mathsf{length}(\mathsf{nil}) = \mathsf{zero},\ \ \forall c : \mathsf{Char}, s : \mathsf{String}.\ \mathsf{length}(\mathsf{cons}(c, s)) = \mathsf{succ}(\mathsf{length}(s))\}$$

We will abbreviate zero as 0 and $\mathsf{succ}^n(\mathsf{zero})$ as n. We will write e.g. Bill for

$$\mathsf{cons}(\mathsf{B}, \mathsf{cons}(\mathsf{I}, \mathsf{cons}(\mathsf{L}, \mathsf{cons}(\mathsf{L}, \mathsf{nil})))).$$

Fig. 2. The Multi-sorted Equational Theory *Type*

Associated with a theory Th is a binary relation between (not necessarily ground) terms, called *provable equality*. We write $Th \vdash \forall \Gamma.\ t = t' : s$ to indicate that the theory Th proves that terms $t, t' \in Terms^s(Sig, \Gamma)$ 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:

$$\frac{t \in Terms^s(Sig, \Gamma)}{Th \vdash \forall \Gamma.\, t = t : s} \qquad \frac{Th \vdash \forall \Gamma.\, t = t' : s}{Th \vdash \forall \Gamma.\, t' = t : s} \qquad \frac{Th \vdash \forall \Gamma.\, t = t' : s \qquad Th \vdash \forall \Gamma.\, t' = t'' : s}{Th \vdash \forall \Gamma.\, t = t'' : s}$$

$$\frac{Th \vdash \forall \Gamma.\, t = t' : s \qquad v \notin \Gamma}{Th \vdash \forall \Gamma, v : s'.\, t = t' : s} \qquad \frac{Th \vdash \forall \Gamma, v : s.\, t = t' : s' \qquad Th \vdash \forall \Gamma.\, e = e' : s}{Th \vdash \forall \Gamma.\, t[v \mapsto e] = t'[v \mapsto e'] : s'}$$

Fig. 3. Inference Rules for Multi-sorted Equational Logic

A *morphism of signatures* $F : Sig_1 \to Sig_2$ consists of:

- a function F from sorts in Sig_1 to sorts in Sig_2, and
- a function F from function symbols $f : s_1 \times \ldots \times s_n \to s$ in Sig_1 to terms in

$$Terms^{F(s)}(Sig_2, \{v_1 : F(s_1), \ldots, v_n : F(s_n)\})$$

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

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

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

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

In the theory $Type$ (Figure 2), any permutation of A, B, ... 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 necessary to formalize the functorial data model. 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) ostensibly 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 $s_1, \ldots, s_k \to s$ of $(\!| Th |\!)$ are equivalence classes of terms of sort s in context $\{v_1 : s_1, \ldots, v : s_k\}$, modulo the provable equality relation $Th \vdash$. Composition is defined by substitution. A morphism of theories $F : Th_1 \to Th_2$ denotes a functor $(\!| F |\!) : (\!| Th_1 |\!) \to (\!| Th_2 |\!)$. Although cartesian multi-categories are the most direct categorical semantics for theories, it many cases it is technically more convenient to work with product categories instead. Every cartesian multi-category is equivalent to a product category, so in our categorical remarks we may conflate product categories and cartesian multi-categories depending on which is more convenient. For details, see (Schultz *et al.*, 2016).

2.2 Semantics

An *algebra* A over a signature Sig consists of:

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

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

$$A[\![v]\!]\eta = \eta(v) \qquad A[\![f(t_1, \ldots, t_n)]\!]\eta = A(f)(A[\![t_1]\!]\eta, \ldots, A[\![t_i]\!]\eta)$$

An algebra A over a signature Sig is a *model* of a theory Th on Sig when $Th \vdash \forall\Gamma.\ t = t' : s$ implies $A[\![t]\!]\eta = A[\![t']\!]\eta$ for all terms $t, t' \in Terms^s(Sig, \Gamma)$ and A-environments η for Γ. 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 2) 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 Sig we form its *term algebra* $[\![Sig]\!]$, a process called *saturation*, as follows. The carrier set $[\![Sig]\!](s)$ is defined as the set of ground terms of sort s. The function $[\![Sig]\!](f)$ for $f : s_1 \times \ldots s_k \to s$ is defined as the function $t_1, \ldots t_n \mapsto f(t_1, \ldots, t_n)$. From a theory Th on Sig we define its *term model* $[\![Th]\!]$ to be the quotient of $[\![Sig]\!]$ by the equivalence relation $Th \vdash$. In other words, the carrier set $[\![Th]\!](s)$ is defined as the set of equivalence classes of ground terms of sort s that are provably equal under Th. The function $[\![Th]\!](f)$ is $[\![Sig]\!](f)$ lifted to operate on equivalence classes of terms. To represent $[\![Th]\!]$ on a computer, or to write down $[\![Th]\!]$ succinctly, we must choose a *representative* for each equivalence class of terms; this detail can be safely ignored for now, but we will return to it in the implementation section 4. When we must choose representatives for $[\![Th]\!]$, we will write $nf_{Th}(e)$ to indicate the unique $e' \in [\![Th]\!]$ such that $Th \vdash e = e'$ (i.e., the *normal form* for e in Th). For example, the term model of the algebraic theory $Type$ (Figure 2) is:

A *morphism of algebras* (also called a *homomorphism* (Mitchell, 1996)) $h : A \to B$ on a signature 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 Sig are initial among all Sig-algebras: there is a unique morphism from the term algebra to any other 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.

$$[\![Type]\!](\mathsf{Nat}) = \{0,1,2,\ldots\}$$
$$[\![Type]\!](\mathsf{Char}) = \{\mathsf{A},\mathsf{B},\mathsf{C},\ldots\}$$
$$[\![Type]\!](\mathsf{String}) = \{\mathsf{nil},\mathsf{A},\mathsf{B},\ldots,\mathsf{AA},\mathsf{AB},\ldots\}$$
$$[\![Type]\!](\mathsf{succ}) = \{(0,1),(1,2),\ldots\}$$
$$\ldots$$

Fig. 4. The Term Model $[\![Type]\!]$ of Theory *Type* (Figure 2)

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

3 An Equational Formalism for Functorial Data Migration

In this section we describe how to use multi-sorted equational theories for functorial data migration. 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 *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, $father : Person \to Person$ and $age : Person \to Nat$; 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 mappings of sorts to sorts and function symbols to (open) terms that preserves entailment, that is, $h : C \to D$ exactly when $C \vdash p = q$ implies $D \vdash h(p) = h(q)$. Our query language is based on a generalization of for-where-return notation, and generalizes the idea from relational database theory that conjunctive queries can be represented as “frozen” instances (Abiteboul *et al.*, 1995).

3.1 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 LENGTH : VARCHAR $\to$ INTEGER; a type-side for SK combinator calculus (a Turing-complete formalism) would contain a sort o, constants $S, K : o$, a function symbol $\cdot : o \to o$, and equations $K \cdot x \cdot y = x$ and $S \cdot x \cdot y \cdot z = x \cdot z \cdot (y \cdot z)$. Note that simply-typed, first-order functional programs can be expressed using multi-sorted equational theories; in fact, using a functional program as a type-side is a "best-case scenario" for the automated reasoning required to implement our formalism; see Section 4 for details. We will also abbreviate "multi-sorted equational theory" as "theory" in the remainder of 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*. S must meet the following conditions:

- If $\forall \Gamma .\ t_1 = t_2$ is in S but not in Ty, then $\Gamma = \{v : s\}$ where s is an entity.
- If $f : s_1 \times \ldots \times s_n \to s$ 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* (or part) 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*, $S_1 \to S_2$ on type-side Ty is a morphism of theories $S_1 \to S_2$ that is the identity on Ty. An example schema Emp on type-side $Type$ (Figure 2) is in Figure 5 below. 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.

$$Sorts := \{\mathsf{Emp}, \mathsf{Dept}\}$$

$$Symbols := \{\mathsf{mgr} : \mathsf{Emp} \to \mathsf{Emp},\ \mathsf{wrk} : \mathsf{Emp} \to \mathsf{Dept},\ \mathsf{sec} : \mathsf{Dept} \to \mathsf{Emp}\},$$

$$\mathsf{dname} : \mathsf{Dept} \to \mathsf{String},\ \mathsf{ename} : \mathsf{Emp} \to \mathsf{String}$$

$$Equations := \{\forall v.\ v.\mathsf{mgr}.\mathsf{wrk} = v.\mathsf{wrk},\ \forall v.\ v = v.\mathsf{sec}.\mathsf{wrk}, \forall v.\ v.\mathsf{mgr}.\mathsf{mgr} = v.\mathsf{mgr}\}$$

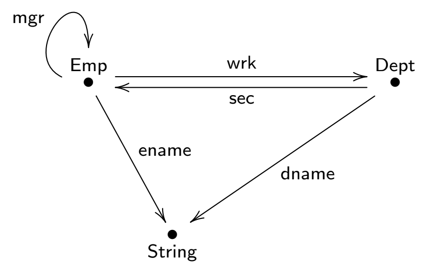


Fig. 5. Schema *Emp*, on Type-side *Type* (Figure 2)

In schema *Emp* (Figure 5), Emp and Dept are sorts (entities) of employees and departments, respectively; mgr takes an employee to their manager; sec takes a department to its secretary; and 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. The first restriction on schemas rules out products of entities (for example, using a symbol $\mathsf{CommonBoss} : \mathsf{Emp} \times \mathsf{Emp} \to \mathsf{Emp}$), and the second restriction on schemas rules out the use of types as domains (for example, using a symbol $\mathsf{EmpNumber} : \mathsf{Nat} \to \mathsf{Emp}$). Without these restrictions, the existence and adjointness of the Σ, Δ, Π functors that we will later define cannot be guaranteed.

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

- If s is a sort in I, then s is a sort in S.
- If $\forall\Gamma.\ t_1 = t_2$ is in I but not S, then $\Gamma = \emptyset$.
- 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 *generator* of sort s.

That is, an instance only adds 0-ary symbols and ground equations. Mirroring a similar practice in relational database theory, we use the phrase *skolem term* to refer 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 Ty, because I can declare new constant symbols whose sorts are types (so-called skolem variables), as well as additional equations; for example, $\mathsf{infinity} : \mathsf{Nat}$ and $\mathsf{succ}(\mathsf{infinity}) = \mathsf{infinity}$. A morphism of instances, or *transform*, $h : I_1 \to I_2$ is a morphism of theories $I_1 \to I_2$ that is the identity on S. (The requirement of identity on S rules out the "attribute problem" from figure 1).

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 $att : s \to t$ 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 $e.fk_1.\ldots.fk_n.att$; 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 $trans : Terms^t(I, \emptyset) \to talg(I)$, for every type (non-entity) sort t, as:

$$trans(e.fk_1.\ldots.fk_n.att) := nf_I(e.fk_1.\ldots.fk_n).att \text{ for observables}$$

$$trans(f(e_1,\ldots,e_n)) := f(trans(e_1),\ldots,trans(e_n)) \text{ otherwise}$$

[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.

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

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

Dept

ID	dname	secr
m	Math	b
s	s.dname	c

Emp

ID	ename	mgr	wrk
a	Al	a.mgr	m
b	b.ename	b.mgr	m
c	Carl	c.mgr	s
a.mgr	a.mgr.ename	a.mgr	m
b.mgr	b.mgr.ename	b.mgr	m
c.mgr	c.mgr.ename	c.mgr	s

The type-algebra extends *Type* with:

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

$$Eqs := \{\mathsf{a.ename} = \mathsf{Al},\ \ \mathsf{c.ename} = \mathsf{Carl},\ \ \mathsf{m.dname} = \mathsf{Math}\}$$

Fig. 6. Instance *Inst* on Schema *Emp* (Figure 5)

The images of the equations-at-type of I under *trans* are the equations for the instance $talg(I)$. Note that $talg(I)$ does not present $[\![I_T]\!]$ (the restriction of I to types), rather, $talg(I)$ presents $[\![I]\!]_T$ (i.e., 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, is shown in Figure 6.

In many cases, we would like for an instance I to be a *conservative extension* of its schema S, meaning that for all terms $t, t' \in Terms^s(S, \Gamma)$, $I \vdash \forall \Gamma.\ t = t' : s$ if and only if $S \vdash \forall \Gamma.\ t = t' : s$. (Similarly, we may also want schemas to conservatively extend their type-sides.) For example, $Emp \nvdash \mathsf{Al} = \mathsf{Carl} : \mathsf{String}$, but there is an *Emp*-instance I for which $I \vdash \mathsf{Al} = \mathsf{Carl} : \mathsf{String}$. 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 conservativity 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.*, 2005) that underlies much work on relational data integration (conservativity is checked during the chase process). In Section 4.3 we give a simple algorithm that soundly approximates conservativity. Note that the Δ and Π migration functors preserve the conservative extension property, but Σ does not; hence, one may want to be careful when using Σ. (More pedantically, Δ and Π preserve type-algebras, but Σ does not.)

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 *et al.*, 1995) that in database theory a schema is a triple (dom, $null$, R), where dom is a set (called the *domain*), $null$ is a set disjoint from dom, (called the *labelled nulls*), and R is a set of relation names and arities; an instance I is a set of relations over $dom \cup null$, indexed by R, of appropriate arity; and a homomorphism $h : I_1 \to I_2$ is a function $h : dom \cup null \to dom \cup null$ that is constant on 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 $[\![I]\!]$ as a relational instance by considering every skolem term in $[\![I]\!]$ to be a labelled null and over non-skolem term to be a domain value, then a transform of instances in our formalism induces a homomorphism of the encoded relational instances. dom is playing the role of a free (equation-less), discrete (function-less) type-side.

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 $(\!|\hat{S}|\!)$ is called the *collage* of S. A schema mapping $F : S \to T$ is then a functor between collages $(\!|\hat{S}|\!) \to (\!|\hat{T}|\!)$ that is the identity on Ty. More pedantically, a schema S is a profunctor $(\!|S|\!) : (\!|S_E|\!)^{op} \times (\!|Ty|\!) \to Set$ which preserves products in $(\!|Ty|\!)$. The observables from an entity $e \in S_E$ to type $ty \in Ty$ are given by $(\!|S|\!)(e, ty)$. A schema mapping $F : S \to T$ denotes a functor $(\!|F_E|\!) : (\!|S_E|\!) \to (\!|T_E|\!)$ and a natural transformation $(\!|S|\!) \Rightarrow (\!|T|\!) \circ ((\!|F_E|\!)^{op} \times id)$:

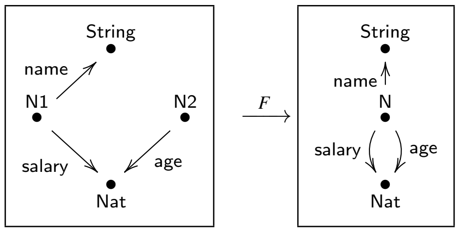
$$\begin{tikzcd}
(\!|S_E|\!)^{op} \times (\!|Ty|\!) \arrow[r, "(\!|S|\!)"] \arrow[d, "(\!|F_E|\!)^{op} \times id"'] & Set \\
(\!|T_E|\!)^{op} \times (\!|Ty|\!) \arrow[ur, "(\!|T|\!)"'] \arrow[u, phantom, shift right=6, "\Downarrow"] &
\end{tikzcd}$$

3.2 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.*, 2016):

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

Although Σ_F and $[\![\Delta_F]\!]$ 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 $\Sigma_F(\Sigma_G(I)) = \Sigma_{F \circ G}(I)$ and $[\![\Delta_F]\!]([\![\Delta_G]\!](I)) = [\![\Delta_{F \circ G}]\!](I)$ (for the other functors, these are not equalities, but unique isomorphisms).



N1

ID	Name	Salary
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2

ID	Age
1	20
2	20
3	30

$\xleftarrow{[\![\Delta_F]\!]}$

N

ID	Name	Salary	Age
1	Alice	$100	20
2	Bob	$250	20
3	Sue	$300	30

N1

ID	Name	Salary
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2

ID	Age
1	20
2	20
3	30

$\xrightarrow{[\![\Sigma_F]\!]}$

N

ID	Name	Salary	Age
a	Alice	$100	1.age
b	Bob	$250	2.age
c	Sue	$300	3.age
d	4.name	4.salary	20
e	5.name	5.salary	20
f	6.name	6.salary	30

N1

ID	Name	Salary
1	Alice	$100
2	Bob	$250
3	Sue	$300

N2

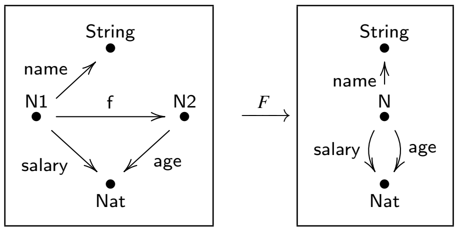
ID	Age
1	20
2	20
3	30

$\xrightarrow{[\![\Pi_F]\!]}$

N

ID	Name	Salary	Age
a	Alice	$100	20
b	Bob	$250	20
c	Sue	$300	20
d	Alice	$100	20
e	Bob	$250	20
f	Sue	$300	20
h	Alice	$100	30
i	Bob	$250	30
j	Sue	$300	30

Fig. 7. Example Functorial Data Migrations



N1

ID	Name	Salary	f
1	Alice	$100	1
2	Bob	$250	2
3	Sue	$300	3

N2

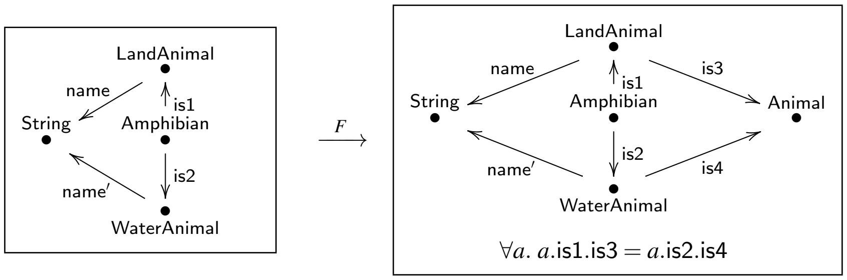
ID	Age
1	20
2	20
3	30

$\xleftarrow{[\![\Delta_F]\!]}$ $\xrightarrow{[\![\Pi_F]\!],[\![\Sigma_F]\!]}$

N

ID	Name	Salary	Age
1	Alice	$100	20
2	Bob	$250	20
3	Sue	$300	30

Fig. 8. Example Functorial Data Migrations, with Foreign Keys



LandAnimal

ID	Name
1	frog
2	cow

WaterAnimal

ID	Name
1	frog
3	fish

Amphibian

ID	Is1	Is2
1	1	1

$\xrightarrow{[\![\Sigma_F]\!]}$

LandAnimal

ID	Name	Is3
1	frog	1
2	cow	2

WaterAnimal

ID	Name	Is4
1	frog	1
3	fish	3

Animal

ID
1
2
3

Amphibian

ID	Is1	Is2
1	1	1

Fig. 9. An Example Σ Data Migration, with Path Equalities

3.3 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 $[\![\Sigma_F]\!] \circ [\![\Pi_G]\!] \circ [\![\Delta_H]\!]$. 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 extremely 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", Π is expensive to compute. Hence, in (Schultz *et al.*, 2016) we developed a new syntax, which we call *uber-flower* syntax, because it is a generalization of flwr (for-let-where-return) syntax (a.k.a select-from-where syntax, comprehension syntax). 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).

A *conjunctive* uber-flower $Q : S \to T$, where S and T are schemas on the same type-side, induces a data migration $eval(Q) : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst} \cong \Delta_G \circ \Pi_F$ an adjoint data migration $coeval(Q) : T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst} \cong \Delta_F \circ \Sigma_G$ for some $X,\ F : S \to X,\ G : T \to X$. In fact, all data migrations of the form $\Delta \circ \Pi$ can be expressed as the eval of a conjunctive uber-flower, and all migrations of the form $\Delta \circ \Sigma$ can be expressed as the coeval of a conjunctive uber-flower. In Sections 3.3.4 and 3.3.5 we describe the correspondence between uber-flowers and data migration functors in detail.

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 $Terms(S, fr)$, called the *where* clause *wh*.

Associated with a tableau over S is a canonical S-instance, called the "frozen" instance (Abiteboul *et al.*, 1995). In our formalism, a tableau trivially becomes an instance by a validity-preserving process called *Herbrandization* (the dual of the satisfiability-preserving *Skolemization* process) that "freezes" variables into fresh constant symbols. For example, we can consider the tableau $(\{v_1 : \mathsf{Emp}, v_2 : \mathsf{Dept}\}, v_1.\mathsf{wrk} = v_2, v_1.\mathsf{ename} = \mathsf{Peter})$ to be an *Emp*-instance with generators $\mathsf{v}_1, \mathsf{v}_2$. In this paper, we may silently pass between a tableau and its frozen instance.

A *conjunctive uber-flower* $S \to T$ consists of, for each entity $t \in T$:

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

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 require that the for clauses in an uber-flower only bind variables to entities, not to types (e.g., $v : \mathsf{Person}$ is allowed, but $v : \mathsf{Nat}$ 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 will 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 computed using relational algebra (Spivak & Wisnesky, 2015). Semantically, this restriction means that evaluations of conjunctive uber-flowers correspond to migrations of the form $\Delta_G \circ \Pi_F$, where F is surjective on attributes, a condition described in Section 3.3.4.

An *uber-flower* is a conjunctive uber-flower which contains multiple for-where-return-keys clauses per target entity. These uber-flowers induce a single data migration of the form $\Sigma_F \circ \Delta_G \circ \Pi_H$[2], where F is a *discrete op-fibration* (Barr & Wells, 1995), and can be thought of as our formalism's equivalent of "unions of conjunctive queries". For simplicity, we will stick to exposition involving conjunctive uber-flowers, and in the rest of this paper, we may use the word "query" to mean "conjunctive uber-flower". We now present in Figure 10 an example query, from our *Emp* schema to our *Emp* schema (Figure 5), that when evaluated makes each employee their own boss and appends their old boss's name to their name. Note that in this query, Dept is copied over unchanged; only Emp changes.

```
Dept = for d:Dept
       where
       return (dname = d.dname)
       keys (sec = [e -> d.sec])

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

$$\begin{tikzcd}
\mathsf{Emp} \arrow[loop left, "\mathsf{mgr}"] \arrow[rr, shift left, "\mathsf{wrk}"] \arrow[dr, "\mathsf{ename}"'] & & \mathsf{Dept} \arrow[ll, shift left, "\mathsf{sec}"] \arrow[dl, "\mathsf{dname}"] \\
& \mathsf{String} &
\end{tikzcd}$$

Verification conditions:

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

$$\forall v : \mathsf{Dept}.v = v.\mathsf{sec.wrk} : \mathsf{Dept} \rightsquigarrow \mathit{Emp} \cup \mathsf{d} : \mathsf{Dept} \vdash \mathsf{d} = \mathsf{d}[\mathsf{d} \mapsto \mathsf{e.wrk}][\mathsf{e} \mapsto \mathsf{d.sec}] : \mathsf{Dept}$$

$$\forall v.v.\mathsf{mgr.mgr} = v.\mathsf{mgr} : \mathsf{Emp} \rightsquigarrow \mathit{Emp} \cup \mathsf{e} : \mathsf{Emp} \vdash \mathsf{e}[\mathsf{e} \mapsto \mathsf{e}][\mathsf{e} \mapsto \mathsf{e}] = \mathsf{e}[\mathsf{e} \mapsto \mathsf{e}] : \mathsf{Emp}$$

Fig. 10. The Conjunctive Uber-flower $\mathit{Promote} : \mathit{Emp} \to \mathit{Emp}$

[2] The order of Π and Δ is opposite that of (Spivak & Wisnesky, 2015)

Remark. Because boolean algebra can be equationally axiomatized, our conjunctive 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), our conjunctive uber-flowers can express queries such as $Q(R) := \{x \in R \mid P(x) \vee \neg P(x) = \top\}$. Because instances may contain skolem terms of type Bool, i.e., terms which are not provably equal to $\top$ or $\bot$, in general $Q(R)$ need not equal R.

$$\top, \bot : \mathsf{Bool} \qquad \neg : \mathsf{Bool} \to \mathsf{Bool} \qquad \wedge, \vee : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$$

$$\begin{array}{cc}
x \vee y = y \vee x & x \wedge y = y \wedge x \\
x \vee (y \vee z) = (x \vee y) \vee z & x \wedge (y \wedge z) = (x \wedge y) \wedge z \\
x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) & x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \\
x \vee \neg x = \top & x \wedge \neg x = \bot \\
x \vee \bot = x & x \wedge \top = x
\end{array}$$

Fig. 11. An Equational Axiomatization of Boolean Algebra

Similarly, our conjunctive 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 4) may diverge.

Categorical Remark. Let $Q : S \to T$ be a conjunctive uber-flower over type-side Ty. Then Q denotes a *bimodule* (Schultz *et al.*, 2016), i.e., a functor $(\!|Q|\!) : (\!|\hat{S}|\!)^{op} \to (\!|S\text{–}\mathbf{Inst}|\!)$ such that $(\!|Q|\!)(t) = y(t)$ for every $t \in (\!|Ty|\!)$, where $y : (\!|\hat{S}|\!)^{op} \to (\!|S\text{–}\mathbf{Inst}|\!)$ is the yoneda embedding.

3.3.1 Verification Conditions for Query Well-formedness

Let $Q : S \to T$ be a conjunctive uber-flower. A *path* in T, $p : t_0 \to t_n := fk_1.\dots.fk_n$ is a sequence of foreign keys, $fk_n : t_{n-1} \to t_n$, and we write $[p]$ to indicate the substitution $[fk_1] \circ \ldots \circ [fk_n]$ taking each $v : s \in fr_{t_n}$ to some term in $Terms^s(S, fr_{t_0})$ (when $n = 0$ this is the identity substitution on fr_{t_0}). 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. One kind of equation is equality at 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 is equality at types of arbitrary terms where paths share head variables in a singleton entity context, for example:

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

Here we must check, for example:

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

In Figure 10, $\phi \rightsquigarrow \psi$ means that equation $\phi \in T$ generates verification condition ψ. Generating verification conditions for a (not necessarily conjunctive) uber-flower is a slight generalization of the process described above.

3.3.2 Morphisms of Queries

Let $Q_1, Q_2 : S \to T$ be conjunctive uber-flowers. For every foreign key $fk : t \to t'$, we have transforms $Q_1(f) : fr^1_{t'} \to fr^1_t$, $Q_2(f) : fr^2_{t'} \to fr^2_t$. A morphism $h : Q_1 \to Q_2$ is, for each entity $t \in T$, a transform of frozen instances $h(t) : fr^1_t \to fr^2_t$, such that for every foreign key $fk : t \to t'$, and every $v' \in fr^1_{t'}$,

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

and for every attribute $att : t \to t'$,

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

h induces a transform $eval(h) : eval(Q_2)(I) \to eval(Q_1)(I)$ for each S-instance I (in this way, our conjunctive uber-flowers are similar to relational conjunctive queries) and a transform $coeval(h) : coeval(Q_1)(J) \to coeval(Q_2)(J)$ for each T-instance J.

3.3.3 Composing Queries

Let $Q_1 : A \to B$ and $Q_2 : B \to C$ be conjunctive uber-flowers on the same type-side. We define the uber-flower $Q_2 \circ Q_1 : A \to C$ 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 $coeval(Q_1) : B\text{–}\mathbf{Inst} \to A\text{–}\mathbf{Inst}$ to the frozen B-instance for c in Q_2:

$$(Q_2 \circ Q_1)(c) := coeval(Q_1)(Q_2(c))$$

- Similarly, the transform associated with a foreign key $fk : c \to c' \in C$ is:

$$(Q_2 \circ Q_1)(fk) := coeval(Q_1)(Q_2(fk))$$

- To define the term associated with an attribute $att : c \to c' \in C$ 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, $h : y(c') \to Q_2(c) := v \mapsto Q_2(att)$. Finally, we define:

$$(Q_2 \circ Q_1)(att) := coeval(Q_1)(h)(v)$$

(Not necessarily conjunctive) uber-flowers also compose (in a way similar to how polynomial functors compose), but describing their composition is significantly harder, and we omit a description here.

3.3.4 *Converting Data Migrations to Queries*

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

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

We now describe Q^F and Q_F:

- The for clause for $Q^F : T \to S$ 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 $fk : s \to s' \in S$, $F(fk)$ is (α-equivalent to) a term in $Terms^{F(s')}(T, \{v_s : F(s)\})$ and we define $Q^F(fk) : Q^F(s') \to Q^F(s)$ to be the transform $v_{s'} \mapsto F(fk)$. For each attribute $att : s \to s' \in S$, s' is a type and $F(att)$ is (α-equivalent to) a term in $Terms^{s'}(T, \{v_s : F(s)\})$ and we define $Q^F(att)$ to be $F(att)$. For example, the Q^F that corresponds to Figure 8 is:

```
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 $Q_F : S \to T$ 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 $fk : t \to t' \in T$, we define the transform $Q_F(fk) : Q_F(t') \to Q_F(t)$ to be $\Delta_F(v_{t'} \mapsto v_t.fk)$. For each attribute $att : t \to t' \in T$, $v_t.att \in Terms^{t'}(T, \{v_t : t\})$ and $trans(v_t.att) \in talg(\{v_t : t\})$. Since Δ preserves type algebras, we have $trans(v_t.att) \in talg(\Delta_F(\{v_t : t\}))$, and hence we can define $Q_F(att)$ to be $trans(v_t.att)$. For example, the Q_F that corresponds to Figure 8 is (note that we write 'x' to indicate an $x \in talg(y(N))$):

```
y(N) = vN : N

N := for vN1 : N1, vN2 : N2
     where vN1.f = vN2, vN1.name='vN.name',
      vN1.salary = 'vN.salary', vN2.salary = 'vN.salary'
     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 $talg(\Delta_F(I))$, i.e., $talg(\Delta_F(I)) \cong Ty$. Formally, F is surjective on attributes when for every attribute $att : t \to t' \in T$, there exists an entity $s \in S$ such that $F(s) = t$ and there exists an $e \in Terms^{t'}(T, \{v : s\})$ such that $T \vdash \forall v : F(s).\ F(e) = v.att$.

3.3.5 *Converting Queries to Data Migrations*

A conjunctive uber-flower $Q : S \to T$, where S and T are schemas on the same type-side Ty, induces a data migration $eval(Q) : S$–**Inst** $\to T$–**Inst** $\cong \Delta_G \circ \Pi_F$ and adjoint data migration

$coeval(Q) : T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst} \cong \Delta_F \circ \Sigma_G$ for some X, $F : S \to X$, $G : T \to X$. In this section, we construct X, F, and G. First, we define a schema X such that $S \subseteq X$ and we define $F : S \hookrightarrow X$ to be the inclusion mapping. We start with:

$$En(X) := En(S) \sqcup En(T) \qquad Att(X) := Att(S) \qquad Fk(X) \subseteq Fk(S) \sqcup Fk(T)$$

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

$$(v,s,t) : t \to s \ \in Fk(X)$$

Let us write σ_x for the substitution $[v_k \mapsto x.(v_k, s_k, t),\ \forall v_k : s_k \in fr_t]$. For each equation $e = e' \in wh_t$ we add an equation to X:

$$\forall x : t.\ e\sigma_x = e'\sigma_x \ \ \in Eq(X)$$

and for each foreign key $fk : t \to t'$ and for each $v' : s' \in fr_{t'}$, we add an equation to X:

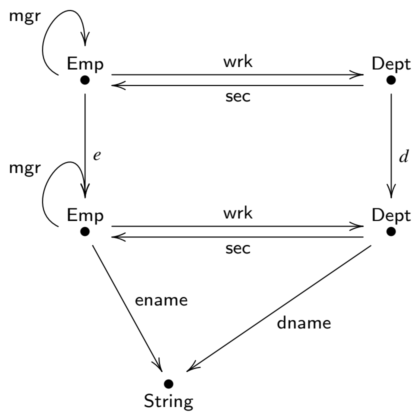
$$\forall x : t.\ x.fk.(v', s', t') = v'[fk]\sigma_x \ \ \in Eq(X)$$

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

$$G(att : t \to t') := \forall x : t.\ [att]\sigma_x$$

For example, the schema X for the query $Promote$ (Figure 10) is shown in Figure 12.

$X :=$



$$\begin{tikzcd}
\text{Emp} \arrow[loop left, "\text{mgr}"] \arrow[rr, shift left, "\text{wrk}"] \arrow[d, "e"] & & \text{Dept} \arrow[ll, shift left, "\text{sec}"] \arrow[d, "d"] \\
\text{Emp} \arrow[loop left, "\text{mgr}"] \arrow[rr, shift left, "\text{wrk}"] \arrow[dr, "\text{ename}"] & & \text{Dept} \arrow[ll, shift left, "\text{sec}"] \arrow[dl, "\text{dname}"'] \\
& \text{String} &
\end{tikzcd}$$

$$\forall x.\ x.\mathsf{wrk}.d = d[d \mapsto e.\mathsf{wrk}][e \mapsto x.e] \equiv x.e.\mathsf{wrk}$$

$$\forall x.\ x.\mathsf{sec}.e = e[e \mapsto d.\mathsf{sec}][d \mapsto x.d] \equiv x.d.\mathsf{sec}$$

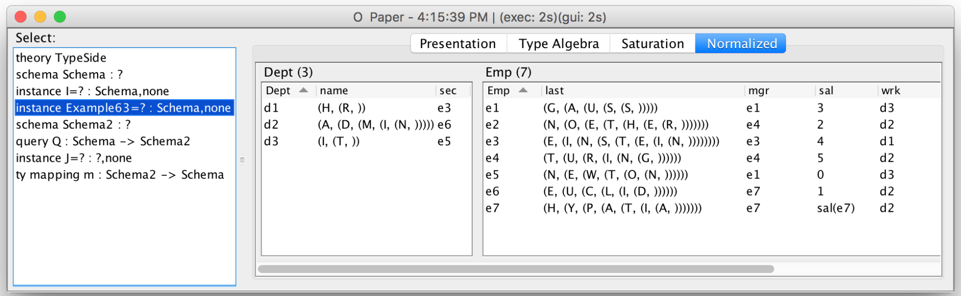
$$\forall x.\ x.\mathsf{mgr}.e = e[e \mapsto e][e \mapsto x.e] \equiv x.e$$

$$G(\mathsf{ename}) := \forall x.\ [\mathsf{ename}][e \mapsto x.e] \equiv x.e.\mathsf{ename} + x.e.\mathsf{mgr}.\mathsf{ename}$$

Fig. 12. Uber-flower *Promote* (Figure 10) as a Data Migration

4 Implementation

We have implemented the formalism defined in this paper in the FQL tool, which can be downloaded at `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).



```
theory TypeSide
schema Schema : ?
instance I=? : Schema,none
instance Example63=? : Schema,none
schema Schema2 : ?
query Q : Schema -> Schema2
instance J=? : ?,none
ty mapping m : Schema2 -> Schema
```

Dept (3)

Dept	name	sec
d1	(H, (R,))	e3
d2	(A, (D, (M, (I, (N,)))))	e6
d3	(I, (T,))	e5

Emp (7)

Emp	last	mgr	sal	wrk
e1	(G, (A, (U, (S, (S,)))))	e1	3	d3
e2	(N, (O, (E, (T, (H, (E, (R,)))))))	e4	2	d2
e3	(E, (I, (N, (S, (T, (E, (I, (N,))))))))	e3	4	d1
e4	(T, (U, (R, (I, (N, (G,))))))	e4	5	d2
e5	(N, (E, (W, (T, (O, (N,))))))	e1	0	d3
e6	(E, (U, (C, (L, (I, (D,))))))	e7	1	d2
e7	(H, (Y, (P, (A, (T, (I, (A,)))))))	e7	sal(e7)	d2

Fig. 13. The FQL Tool Displaying an Instance

4.1 Deciding Equality in Equational Theories

Many, but not all, constructions involving equational theories, including query evaluation, depend crucially 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 query 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”[3]. 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 re-write rules (oriented equations), R, such that p and q are equal under E if and only if p and q re-write 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, below. Knuth-Bendix completion yields the re-write system on the right in Figure 14. To see how these re-write rules are used to decide the word problem, consider the two terms $(a^{-1} * a) * (b * b^{-1})$ and $b * ((a * b)^{-1} * a)$. Both of these terms re-write to 1 under the above re-write rules; hence, we conclude that they are provably equal. In contrast, the two terms $1 * (a * b)$ and $b * (1 * a)$ re-write to $a * b$ and $b * a$, respectively, which are not syntactically the same; hence, we conclude that they are not provably equal (which is different than saying they are provably not equal).

[3] Technically, a decision procedure for *ground* terms only is sufficient, because a decision procedure for all terms can be obtained via Herbrandization; i.e., replacing universally quantified variables by fresh 0-ary constant symbols (Bachmair *et al.*, 1989).

Equations	Re-write rules	
$1 * x = x$	$1 * x \rightsquigarrow x$	$x * 1 \rightsquigarrow x$
$x^{-1} * x = 1$	$x^{-1} * x \rightsquigarrow 1$	$(x^{-1})^{-1} \rightsquigarrow x$
$(x * y) * z = x * (y * z)$	$(x * y) * z \rightsquigarrow x * (y * z)$	$x * x^{-1} \rightsquigarrow 1$
	$x^{-1} * (x * y) \rightsquigarrow y$	$x * (x^{-1} * y) \rightsquigarrow y$
	$1^{-1} \rightsquigarrow 1$	$(x * y)^{-1} \rightsquigarrow y^{-1} * x^{-1}$

Fig. 14. Knuth-Bendix Completion for Group Theory

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 re-write 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.

4.2 Saturating Theories into Term Models

Many, but not all, constructions involving equational theories, including query evaluation, depend crucially 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 (Carmody *et al.*, 1995) (Bush *et al.*, 2003), and arbitrary theories can be saturated by using a decision procedure for the theory's word problem. Let Th be an algebraic theory. We construct $[\![Th]\!]$ in stages: first, we find all not provably equal terms of size 0 in Th;[4] call this $[\![Th]\!]^0$. Then, we add to $[\![Th]\!]^0$ all not provably equal terms of size 1 that are not provably equal to a term in $[\![Th]\!]^0$; call this $[\![Th]\!]^1$. We iterate this procedure, obtaining a sequence $[\![Th]\!]^0, [\![Th]\!]^1, \ldots$. If $[\![Th]\!]$ is indeed finite, then there will exist some n such that $[\![Th]\!]^n = [\![Th]\!]^{n+1} = [\![Th]\!]$ and we can stop. Otherwise, our attempt to construct $[\![Th]\!]$ will run forever: it is not decidable whether a given theory Th generates a finite term model.

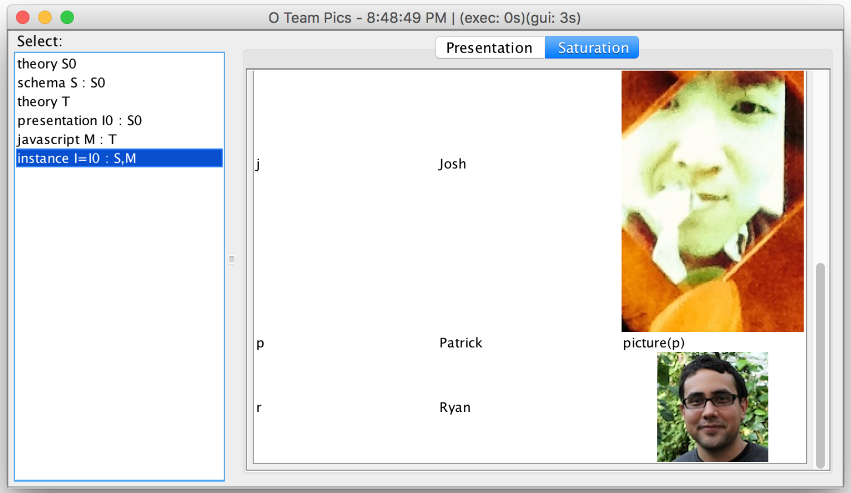
Note that the model $[\![Th]\!]$ computed using this procedure is technically not the canonical term model for the theory; rather, we have constructed a model that is isomorphic to the canonical term model by choosing representatives for equivalence classes of terms under the provable equality relation. Depending on how we enumerate terms, we can end up with different models. As a reminder, we write $nf_{Th}(e)$ to indicate the unique $e' \in [\![Th]\!]$ such that $Th \vdash e = e'$; i.e., e' is the normal form for e in $[\![Th]\!]$.

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

[4] By the *size* of a term, we mean the height of the associated syntax tree. For example $\mathsf{max}(\mathsf{x.sal}, \mathsf{x.mgr.sal})$ has size of three.

infinite, so we cannot saturate the equational theory of the instance directly (i.e., $[\![I]\!]$ is often infinite). For example, if the type-side of I is the free group on one generator a, then $[\![I]\!]$ will contain a, $\mathsf{a} * \mathsf{a}$, $\mathsf{a} * \mathsf{a} * \mathsf{a}$, and so on. Hence, as described in section 3, the FQL tool computes the term model for only the entity and attribute part of I (namely, $[\![I_{EA}]\!]$), along with an instance (equational theory) called the type-algebra of I (namely, $talg(I)$). The pair $([\![I_{EA}]\!], talg(I))$ is sufficient for all of FQL's purposes.

The FQL tool supports an experimental feature that 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 *$\mathscr{L}$-valued model* of $talg(I)$ is similar to a (set-valued) model of $talg(I)$, except that instead of providing a carrier set for each sort in $talg(I)$, an $\mathscr{L}$-valued model provides a type in $\mathscr{L}$, and instead of providing a function for each symbol in $talg(I)$, an $\mathscr{L}$-valued model provides an expression in $\mathscr{L}$. For example, if $\mathscr{L} =$ Java, then we can interpret String as $\mathsf{java.lang.String}$, Nat as $\mathsf{java.lang.Integer}$, $+ : \mathsf{String} \to \mathsf{String}$ as $\mathsf{java.lang.String.append}$, etc. (Note that our $\mathscr{L}$-models are on $talg(I)$, not Ty; so an $\mathscr{L}$-model must provide a meaning for the skolem terms in $talg(I)$, which can be tricky.) Given an $\mathscr{L}$-model M, we can take the image of $[\![I_{EA}]\!]$ under M by replacing each term $talg(t) \ni t \in [\![I_{EA}]\!]$ with the value of t in M. We write this as $M([\![I_{EA}]\!])$. The pair $(M([\![I_{EA}]\!]), M)$ can be used by the FQL tool in many situations where $([\![I_{EA}]\!], talg(A))$ is expected; for example, in displaying tables to users (see Figure 15), and when (co-)evaluating queries. We have used type-side computation to connect the FQL tool to a number of 3rd party libraries, including a machine-learning library, and formalizing computational type-sides is an important area for future work.



```
theory S0
schema S : S0
theory T
presentation I0 : S0
javascript M : T
instance I=I0 : S,M
```

```
j    Josh
p    Patrick    picture(p)
r    Ryan
```

Fig. 15. The FQL Tool Displaying an Instance with a Computational Type-side

4.3 Deciding that a Theory Conservatively Extends Another

As described in Section 3, we may want instances to conservatively extend their schemas, where a theory Th_2 conservatively extends Th_1 when $Th_1 \vdash \forall\Gamma.\, t = t' : s$ iff $Th_2 \vdash \forall\Gamma.\, t = t' : s$ for all $t, t' \in Terms^s(Th_1, \Gamma)$ for every Γ. 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 the system (Lüth *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 a 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:

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

is not conservative over $Type$ (Figure 2), 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:

$$\{+ : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}, \mathsf{infinity} : \mathsf{Nat},\ \mathsf{undef} : \mathsf{Nat}, \mathsf{infinity}{+}1 = \mathsf{undef}{+}1\}$$

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. $\mathsf{age.bill}$ when 45 and $\mathsf{age.bill}$ are provably equal.

4.4 Evaluating Queries

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

- We define the generators of entity t in $eval(Q)(I)$ to be those $[\![I_{EA}]\!]$ environments for fr_t that satisfy wh_t. Formally, we represent these environments as ground substitutions $fr_t \to Terms(S, \emptyset)$ and define, where $fr_t := \{\overrightarrow{v_i : s_i}\}$:

$$eval(Q)(I)(t) := \{[\overrightarrow{v_i \mapsto e_i}] \mid I \vdash eq[\overrightarrow{v_i \mapsto e_i}]\ ,\ \forall eq \in wh_t\ , \forall e_i \in [\![I_{EA}]\!](s_i)\}$$

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

$Terms^{t'}(S,\emptyset)$, and we add:

$$\sigma.att = trans([att]\sigma) \ \in eval(Q)(I)$$

The reason that $trans([att]\sigma) \in Terms^{t'}(eval(Q)(I),\emptyset)$ is because $trans([att]\sigma) \in Terms^{t'}(talg(I),\emptyset)$ and $talg(I) \subseteq eval(Q)(I)$. See 3.1 for the definition of $trans$.

- 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 $[fk] : fr_{t'} \to Terms(S, fr_t)$. For every substitution $\sigma : fr_t \to Terms(S,\emptyset) \in eval(Q)(t)$, we add the equation:

$$\sigma.fk = \sigma \circ [fk] \ \in eval(Q)(I)$$

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

Note that in order to build the instance $eval(Q)(I)$, we have effectively constructed the term model $[\![eval(Q)(I)_{EA}]\!]$, and then "de-saturated" the term model into an equational theory. Evaluation of uber-flowers is similar, except that there are multiple tableau associated with each target entity, so the results for each tableau associated with target entity t must be disjointly-unioned together to populate $eval(Q)(I)(t)$.

To make the above description concrete, we will now evaluate the conjunctive uber-flower $Promote : Emp \to Emp$ from figure 10 on the instance $Inst$ from figure 6, which in turn is on the schema Emp from figure 5 on the type-side $Type$ from figure 2. Our goal is to compute the instance (equational theory) $eval(Promote, Inst)$. We start by copying $talg(Inst)$ into $eval(Promote, Inst)$. Next, we process the tableau. We start with target entity $\mathsf{Dept} \in T$. The from and where clauses give us a set of substitutions $\{[d \mapsto \mathsf{m}], [d \mapsto \mathsf{s})\}$, which are the generators of $eval(Promote, Inst)$ at entity Dept. The return clause adds equations $[d \mapsto \mathsf{m}].\mathsf{dname} = \mathsf{m.dname}$ and $[d \mapsto \mathsf{s}].\mathsf{dname} = \mathsf{s.dname}$; 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 $\mathsf{sec} : \mathsf{Dept} \to \mathsf{Emp}$ adds equations $[d \mapsto \mathsf{m}].\mathsf{secr} = [e \mapsto \mathsf{b}]$ and $[d \mapsto \mathsf{s}].\mathsf{secr} = [e \mapsto \mathsf{c}]$; we have not added $[e \mapsto \mathsf{b}]$ and $[e \mapsto \mathsf{b}]$ 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 $\mathsf{Emp} \in T$. The from and where clause give us a set of substitutions $\{[e \mapsto \mathsf{a}], [e \mapsto \mathsf{b}], [e \mapsto \mathsf{c}], [e \mapsto \mathsf{a.mgr}], [e \mapsto \mathsf{b.mgr}], [e \mapsto \mathsf{c.mgr}]\}$, which are the generators of $eval(Promote, Inst)$ at entity Emp. The return clause adds equations such as $[e \mapsto \mathsf{a}].\mathsf{ename} = \mathsf{a.ename} + \mathsf{a.mgr.ename}$, where a.ename and a.mgr.ename are generators in $talg(Inst)$. The keys clause for $\mathsf{mgr} : \mathsf{Emp} \to \mathsf{Emp}$ adds equations such as $[e \mapsto \mathsf{a}].\mathsf{mgr} = [e \mapsto \mathsf{a}]$, and the keys clause for $\mathsf{wrk} : \mathsf{Emp} \to \mathsf{Dept}$ add equations such as $[e \mapsto \mathsf{a}].\mathsf{wrk} = [d \mapsto m]$ (we added $[d \mapsto \mathsf{m}]$ to $eval(Promote, Inst)$ when processing the target entity Emp). The entire instance, and its denotation, are displayed in Figure 16.

4.4.1 Evaluating Queries on Transforms

Let $Q : S \to T$ be a conjunctive uber-flower and let $h : I \to J$ be a transform of S-instances I and J. Our goal is to define the transform $eval(Q)(h) : eval(Q)(I) \to eval(Q)(J)$. For

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

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

Dept		
ID	dname	secr
$[d \mapsto \mathsf{m}]$	Math	$[e \mapsto \mathsf{b}]$
$[d \mapsto \mathsf{s}]$	s.dname	$[e \mapsto \mathsf{c}]$

Emp			
ID	ename	mgr	wrk
$[e \mapsto \mathsf{a}]$	Al + a.mgr.ename	$[e \mapsto \mathsf{a}]$	$[d \mapsto \mathsf{m}]$
$[e \mapsto \mathsf{b}]$	b.ename + b.mgr.ename	$[e \mapsto \mathsf{b}]$	$[d \mapsto \mathsf{m}]$
$[e \mapsto \mathsf{c}]$	Carl + c.mgr.ename	$[e \mapsto \mathsf{c}]$	$[d \mapsto \mathsf{s}]$
$[e \mapsto \mathsf{a.mgr}]$	a.mgr.ename + a.mgr.ename	$[e \mapsto \mathsf{a.mgr}]$	$[d \mapsto \mathsf{m}]$
$[e \mapsto \mathsf{b.mgr}]$	b.mgr.ename + b.mgr.ename	$[e \mapsto \mathsf{b.mgr}]$	$[d \mapsto \mathsf{m}]$
$[e \mapsto \mathsf{c.mgr}]$	c.mgr.ename + c.mgr.ename	$[e \mapsto \mathsf{c.mgr}]$	$[d \mapsto \mathsf{s}]$

Fig. 16. Evaluation of Uber-flower *Promote* (Figure 10) on *Inst* (Figure 6)

each target entity $t \in T$, consider the generators in $eval(Q)(I)(t)$: they will be the ground substitutions $fr_t \to Terms(S, \emptyset)$ satisfying wh_t. We will map each such substitution to a substitution (generator) in $eval(Q)(J)(t)$. Given such a substitution $\sigma := [v_0 : s_0 \mapsto e_0, \ldots, v_n : s_n \mapsto e_n]$, where $e_i \in Terms^{s_i}(S, \emptyset)$, we define:

$$eval(Q)(\sigma) := [v_0 : s_0 \mapsto nf_{J_E}(h(e_1)), \ldots, v_n : s_n \mapsto nf_{J_E}(h(e_n))$$

In general, $h(e_i)$ need not appear in $[\![J_E]\!]$, so we must use the normal for $h(e_i)$ in J_E. Evaluation of (non-conjunctive) uber-flowers on transforms is similar.

4.4.2 *Evaluating Morphisms of Queries*

If $Q_1, Q_2 : S \to T$ are conjunctive 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 $eval(h) : eval(Q_2)(I) \to eval(Q_1)(I)$ for every S-instance I; in this section, we show how to compute to compute $[h]$. Let $t \in T$ be an entity and $fr_t^1 := \{v_1, \ldots, v_n\}$ be for for clause for t in Q_1. The generators of $eval(Q_2)(I)$ are substitutions $\sigma : fr_t^2 \to [\![I_E]\!]$. We define:

$$eval(h)(\sigma) := [v_1 \mapsto nf_{I_E}(h(v_1)\sigma), \ldots v_n \mapsto nf_{I_E}(h(i_n)\sigma)]$$

4.5 Co-Evaluating Queries

Although it is possible to co-evaluate a conjunctive[5] uber-flower query by translation into a data migration of the form $\Delta \circ \Pi$, we have implemented co-evaluation directly. Let $Q : S \to T$ be a conjunctive uber-flower and let J be a T-instance. We are not aware of any algorithm in relational database theory that is similar to $coeval(Q)$; intuitively, $coeval(Q)(J)$ 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) $coeval(Q)(J)$. First, we copy the generators and equations of the type-algebra $talg(J)$ into $coeval(Q)(J)$. We define $coeval(Q)(I)$ to be the smallest theory such that, for every target entity $t \in T$, where $fr_t := \{v_1 : s_1, \ldots, v_n : s_n\}$,

- $\forall (v : s) \in fr_t$, and $\forall j \in [\![J]\!](t)$,
$$(v, j) : s \ \in \ coeval(Q)(J)$$
- $\forall e = e' \in wh_t$, and $\forall j \in [\![J]\!](t)$,
$$(e = e')[v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)] \ \in \ coeval(Q)(J)$$
- $\forall att : t \to t' \in T$, and $\forall j \in [\![J]\!](t)$,
$$trans([\![J]\!](att)(j)) = [att][v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)] \ \in coeval(Q)(J)$$
recall that $[att] \in Terms^{t'}(S, fr_t)$ is the return clause for attribute att and $trans : Terms^{t'}(J, \emptyset) \to talg(J)$ is defined in Section 3.1.
- $\forall fk : t \to t' \in T$, and $\forall j \in [\![J]\!](t)$, and $\forall (v' : s') \in fr_{t'}$,
$$(v', [\![J]\!](fk)(j)) = v'[fk][v_1 \mapsto (v_1, j), \ldots, v_n \mapsto (v_n, j)] \ \in coeval(Q)(J)$$
recall that the substitution $[fk] : fr_{t'} \to Terms(S, fr_t)$ is the keys clause for foreign-key fk.

The co-evaluation the conjunctive uber-flower $Promote : Emp \to Emp$ from figure 10 on the instance $Inst$ from figure 6 is in fact isomorphic to the evaluation of $promote$ (Figure 16); the reason is that evaluation and co-evaluation of $Promote$ are semantically both projections (Δ-only operations).

4.5.1 Co-Evaluating Queries on Transforms

Let $Q : S \to T$ be a conjunctive 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 $\forall (v : s) \in fr_t$, and for every $j \in [\![J_E]\!](t)$, we define:

$$coeval(Q)((v, j)) := (v, nf_{J_E}(h(j)))$$

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

[5] non-conjunctive uber-flowers cannot be co-evaluated

4.5.2 Co-Evaluating Morphisms of Queries

If $Q_1, Q_2 : S \to T$ are conjunctive 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 $[h]$. Let $t \in T$ be an entity The generators of $coeval(Q_1)(J)$ are pairs (v, j) with $v : s \in fr_t^{Q_2}$ and $j \in [\![J_E]\!](t)$. Define:

$$coeval(h)((v,j)) := (v',j).fk_1 \ldots .fk_n \quad \text{where } h(v) := v'.fk_1.\ldots.fk_n$$

4.6 The Unit and Co-Unit of the Co-Eval $\dashv$ Eval Adjunction

Let $Q : S \to T$. 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 ε_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:

$$\eta_J(j) := [v_1 \mapsto (v_1, nf_{J_E}(j)), \ldots, v_n \mapsto (v_n, nf_{J_E}(j))]$$

4.7 Pivot (Instance $\to$ 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 $F : \int I \to S$, and a $\int I$-instance J, such that $\Sigma_F(J) = I$, defined as follows. First, we copy $talg(I)$ into J. Then for every entity $s \in S$, and every $i \in [\![I_E]\!](s)$, we have:

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

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

$$(i, att) : i \to s' \in \int I \qquad F((i, att)) := att \qquad i.(i, att) = trans([\![I_{EA}]\!](att)(i)) \in J$$

where $trans$ is defined in Section 3.1, and for every foreign key $fk : s \to s' \in S$,

$$(i, fk) : i \to [\![I_E]\!](fk)(i) \in \int I \qquad F((i, fk)) := fk \qquad i.(i, fk) = [\![I_E]\!](fk)(i) \in J$$

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

$$nf_{I_E}(e) \;\; . \;\; (nf_{I_E}(e), fk_1) \;\; . \;\; (nf_{I_E}(e.fk_1), fk_2) \;\; . \;\; \ldots \;\; . \;\; (nf_{I_E}(e.fk_1., \ldots fk_n), att)$$

We add $g = [g]$ to J. See Figure 17 for an example on type-side $Type$ (Figure 2). Our pivot operation is related to the Grothendieck construction (Barr & Wells, 1995).

$$\begin{tikzcd}[row sep=small] S := & \text{String} & & & \int I := & \text{String} & \\ & \text{Person} \arrow[u, "\text{name}"] & & & & a \arrow[u, "\text{name}"] \arrow[d, "\text{livesIn}"'] & b \arrow[ul, "\text{name}"'] \arrow[dl, "\text{livesIn}"] \\ & \text{Home} \arrow[from=u, "\text{livesIn}"'] \arrow[d, "\text{size}"'] & & \arrow[ll, "F"'] & & h \arrow[d, "\text{size}"'] & \\ & \text{Nat} & & & & \text{Nat} & \end{tikzcd}$$

$$I := \xleftarrow{[\![\Sigma_F]\!]} J :=$$

$I :=$ Home

ID	size
h	4

Person

ID	livesIn	name
a	h	Alice
b	h	Bill

$J :=$ h

ID	size
h	4

a

ID	name	livesIn
a	Adam	h

b

ID	name	livesIn
b	Bill	h

Fig. 17. Example of a Pivot

4.8 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 $\oint S$ on Ty, a mapping $F : S \to \oint I$, and a $\oint I$-instance J, such that $\Delta_F(J) \cong I$, defined as follows. In fact, we obtain an inclusion $F : S \hookrightarrow \oint I$, so J is superfluous. First, we add a single entity $\star$ to S, and for every generator $g : s \in talg(I)$, an attribute g_A:

$$\star \in \oint I \qquad g_A : \star \to s \ \in \oint I$$

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

$$i_E : \star \to s \in \oint I \qquad F(s) := s$$

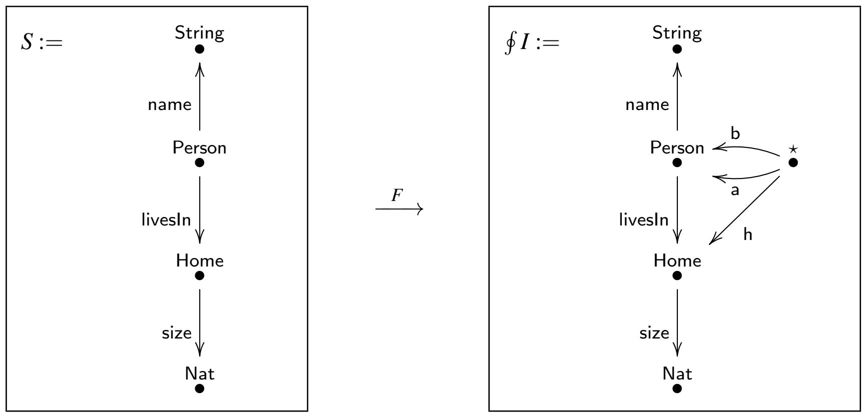
and additionally, for every attribute $att : s \to s'$, and every foreign key $fk : s \to s''$, we have:

$$\forall x : \star.\ x.i_E.att = x.trans(i.att)_A \qquad \forall x : \star.\ x.i_E.fk = x.nf_{I_E}(i.fk)_E$$

where $trans$ is defined in Section 3.1. See Figure 18 for an example on type-side $Type$ (Figure 2).

5 A Pushout Design Pattern for Functorial Data Integration

In this section we describe a design pattern for integrating two databases 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



$I :=$ Home

ID	size
h	4

Person

ID	livesIn	name
a	h	Alice
b	h	Bill

Fig. 18. Example of a Co-Pivot

instance is meant to capture the instance data that should be identified (e.g., that Pete and Peter are the same person).

Although the Σ, Δ, Π 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 will 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 (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 “O Integration” example.

5.1 Pushouts of Schemas and Instances

Let $\mathscr{C}$ be a category and $F_1 : S \to S_1$ and $F_2 : S \to S_2$ be morphisms in $\mathscr{C}$. 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 19.

Our formalism admits pushouts of schemas and instances. Let $S := (Sorts, Symbols, Eqs)$, $S_1 := (Sorts_1, Symbols_1, Eqs_1)$ and $S_2 := (Sorts_2, Symbols_2, Eqs_2)$ be schemas on some type-side, and $F_1 : S \to S_1$ and $F_2 : S \to S_2$ be schema mappings. The pushout schema T is

$$\begin{tikzcd}
S \arrow[r, "F_1"] \arrow[d, "F_2"'] & S_1 \arrow[d, "G_1"] \arrow[ddr, bend left, "G_1'"] & \\
S_2 \arrow[r, "G_2"'] \arrow[drr, bend right, "G_2'"'] & T \arrow[dr, dashed, "t"] & \\
 & & T'
\end{tikzcd}$$

Fig. 19. Pushouts

defined with sorts:

$$Sorts_T := (Sorts_1 \sqcup Sorts_2)/\sim$$

where $\sqcup$ means disjoint union, $\sim$ is the least equivalence relation such that $F_1(s) \sim F_2(s)$ for every sort $s \in S$, and $/$ means quotient. We define further that:

$$Symbols_T := Symbols_1 \ \sqcup \ Symbols_2$$

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

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

and the schema mappings G_1 and G_2 are essentially inclusions. Pushouts of instances are slightly easier. Let $S := (Gens, Eqs)$, $S_1 := (Gens_1, Eqs_1)$ and $S_2 := (Gens_2, Eqs_2)$ be instances on some schema, and $F_1 : S \to S_1$ and $F_2 : S \to S_2$ be transforms. The pushout instance T is:

$$Gens_T := Gens_1 \ \sqcup \ Gens_2$$

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

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). The canonical pushout schemas and instances can be simplified, and in fact FQL performs simplification, but the simplification process is necessarily non-canonical. In our extended medical example, we will use a simplified non-canonical pushout schema.

5.2 Overview of the Pattern

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

$$S_1 \overset{F_1}{\leftarrow} S \overset{F_2}{\rightarrow} S_2$$

we propose to use their pushout:

$$S_1 \overset{G_1}{\to} T \overset{G_2}{\leftarrow} S_2$$

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

$$\Sigma_{G_1}(I_1) \overset{\Sigma_{G_1}(h_1)}{\leftarrow} \left(\Sigma_{G_1 \circ F_1}(I) = \Sigma_{G_2 \circ F_2}(I)\right) \overset{\Sigma_{G_2}(h_2)}{\to} \Sigma_{G_2}(I_2)$$

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 process constructs weakly initial solutions to data integration problems (Fagin *et al.*, 2005).

5.3 An Example of the Pattern

As usual for our formalism, we being by fixing a type-side. We choose the *Type* type-side from Figure 2. 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 20. In that figure's graphical notation, an attribute $\bullet_A \to_{att} \bullet_{\mathsf{String}}$ is rendered as $\bullet_A - \circ_{att}$. Next, given input S_1-instance I_1, S_2-instance I_2, overlap S-instance I and morphisms $h_1 : \Sigma_{F_1}(I) \to I_1$ and $h_2 : \Sigma_{F_2}(I) \to I_2$, our goal is to construct a pushout T-instance J and morphisms j_1, j_2, as shown in Figure 21; note that in this figure, by $K \to_{(F,h)} L$ we mean that $h : \Sigma_F(K) \to L$, and that we use *variable* font for generators, sansserif font for sorts and symbols, and regular font for terms in the type-side.

Intuitively, this 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 giving the names of these methods). Another difference between schemas is that S_1 assigns each patient a gender, but S_2 does. Finally, entities with the same meaning in both schemas can have different names (Person vs Patient, for example).

We construct the overlap schema S 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 $P \;{}_f\!\leftarrow\; O \to_g T$ 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, so now we must think about the attributes. 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 will have the same patient named "Pete" in one database and "Peter" in the other database.

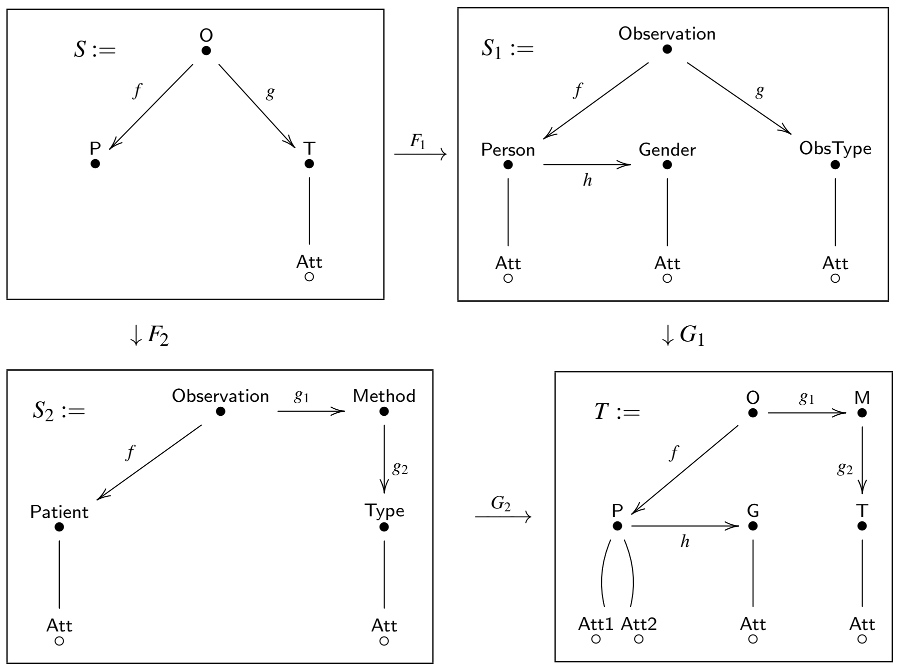


Fig. 20. Medical Records Schema Integration

Hence, we do not include an attribute for patient 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 4.3), 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 that 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 nomenclatures such as officer titles (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 a later 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 assumed 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.

In the output of our pattern 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 one patient, Jane, could not be assigned a gender.

$$\begin{tikzcd} I \arrow[r, "(F_1,h_1)"] \arrow[d, "(F_2,h_2)"'] & I_1 \arrow[d, "(G_2,j_2)"] \\ I_2 \arrow[r, "(G_1,j_1)"] & J \end{tikzcd}$$

$I :=$

O
ID

P
ID
p

T	
ID	Att
bp	BP
wt	Weight

$I_1 :=$

Gender	
ID	Att
f	F
m	M

ObsType	
ID	Att
bp	BP
wt	Weight
hr	HR

Observation		
ID	f	g
o_5	pe	bp
o_6	pa	hr
o_7	pe	wt

Person		
ID	Att	h
pa	Paul	m
pe	Peter	m

$I_2 :=$

Method	
ID	g2
m_1	bp
m_2	bp
m_3	wt
m_4	wt

Type	
ID	Att
bp	BP
wt	Weight

Observation		
ID	f	g
o_1	p	m_1
o_2	p	m_2
o_3	j	m_3
o_4	j	m_1

Patient	
ID	Att
j	Jane
p	Pete

$J :=$

M	
ID	g2
g1(o_5)	bp
g1(o_6)	wt
g1(o_7)	hr
m_1	bp
m_2	bp
m_3	wt
m_4	wt

O		
ID	f	g1
o_1	pe	m_1
o_2	pe	m_2
o_3	j	m_3
o_4	j	m_1
o_5	pe	g1(o_5)
o_6	pa	g1(o_6)
o_7	pe	g1(o_7)

G	
ID	Att
f	F
m	M
h(j)	Att(h(j))

T	
ID	Att
bp	BP
wt	Weight
hr	HR

P			
ID	Att1	Att2	h
j	Att1(j)	Jane	h(j)
pa	Paul	Att2(pa)	m
pe	Peter	Pete	m

Fig. 21. Medical Records Data Integration

5.4 Entity-resolution Using Queries

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

Technically, the overlap instance used in the pushout problem 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 resolve 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 $A(Person) := \{a_1, a_2\}$ and $B(Person) := \{b_1, b_2\}$. If the overlap instance O has $O(Person) := \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$, then this does not mean that the pushout will have 4 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. When I is empty, the pushout will be a co-product (“maximally large”), and when I is not a proper sub-instance, the pushout will be the pushout of a product, which will be small – perhaps terminal (“maximally small”).

Let $inc_1 : S_1 \to S_1 + S_2$ and $inc_2 : S_2 \to S_1 + S_2$ be inclusion schema mappings, and define the $S_1 + S_2$ instance $I' := \Sigma_{inc_1}(I_1) + \Sigma_{inc_2}(I_2)$. 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).

We construct overlap S-instance I by defining a query $Q : S_1 + S_2 \to S$ and evaluating it on the $S_1 + S_2$-instance I'. For each entity $s \in S$, we choose a set of pairs of attributes from $F_1(s)$ and $F_2(s)$ that we desire to be “close”. In the medical records example, for P we choose $(\mathsf{PatientAtt}, \mathsf{PersonAtt})$ and for T we choose $(\mathsf{ObsTypeAtt}, \mathsf{TypeAtt})$; we choose nothing for O. We next choose a way to compare these attributes; for example, we choose a string edit distance of less than two to indicate that the entities match. This comparison function must be added to our type side, e.g.,

$$\mathsf{strMatches} : \mathsf{String} \times \mathsf{String} \to \mathsf{Nat} \qquad \mathsf{true} : \mathsf{Nat} \qquad \mathsf{true} = 1$$

The function $\mathsf{strComp}$ can be defined using equations, although the FQL tool allows such functions to be defined using java code (see Section 4). With the String-comparator in hand, we can now define $Q : S_1 + S_2 \to S$ as in Figure 22. The overlap instance I is defined as $eval(Q)(I')$. To construct $h_n : \Sigma_{F_n}(I) \to I_n$ for $n = 1, 2$, we define projection queries $Q_n : S_1 + S_2 \to S$ and inclusion query morphisms $q_n : Q_n \to Q$ as in Figure 22. We start with the induced transforms for q_n, then apply Σ_{F_n}, then compose with the isomorphism $eval(Q_n)(I') \cong \Delta_{F_n}(I_n)$, and then compose the co-unit ε of the $\Sigma_{F_n} \dashv \Delta_{F_n}$ adjunction, to

$Q_1 : S_1 + S_2 \to S :=$

```
O := for o1 in Observation1
     keys f=[p1->o1.f], g=[t1->o1.g]
P := for p1 in Person
T := for t1 in ObsType, return att=t1.att
```

$q_1 \downarrow$

$Q : S_1 + S_2 \to S :=$

```
O := for o1 in Observation1, o2 in Observation2
     where 1 = 2
     keys f=[p1->o1.f, p2 -> o2.f],
          g=[t1->o1.g, t2 -> o2.g1.g2]
P := for p1 in Person, p2 in Patient
     where true = strMatches(p1.PatientAtt, p2.PersonAtt)
T := for t1 in ObsType, t2 in Type
     where t1.ObsTypeAtt = t2.TypeAtt
     return att=t1.ObsTypeAtt
```

$q_2 \uparrow$

$Q_2 : S_1 + S_2 \to S :=$

```
O := for o2 in Observation2
     keys f=[p1->o2.f], g=[t1->o2.g1.g2]
P := for p2 in Patient
T := for t2 in Type, return att=t2.att
```

$$\begin{tikzcd} S_1 \arrow[r, hook, "inc_1"] & S_1 + S_2 & S_2 \arrow[l, hook', "inc_2"'] \end{tikzcd}$$

$$I' \in S_1 + S_2\text{–}\mathbf{Inst} := \Sigma_{inc_1}(I_1) + \Sigma_{inc_2}(I_2) \qquad I \in S\text{–}\mathbf{Inst} := eval(Q)(I')$$

$$\begin{tikzcd} I \arrow[r, "eval(q_n)(I')"] & eval(Q_n)(I') \end{tikzcd}$$

$$\begin{tikzcd} \Sigma_{F_n}(I) \arrow[r, "\Sigma_{F_n}(eval(q_n)(I'))"] & \Sigma_{F_n}(eval(Q_n)(I')) \arrow[r, "\cong"] & \Sigma_{F_n}(\Delta_{F_n}(I_n)) \arrow[r, "\varepsilon"] & I_n \end{tikzcd}$$

Fig. 22. Entity-Resolution on Medical Records using Uber-flowers

obtain h_n as in Figure 22. The result of running Figure 22 on the medical records data I_1, I_2 from Figure 21 will be the overlap instance I from Figure 21.

To compute the isomorphism $eval(Q_n)(I') \to \Delta_{F_n}(I_n)$, we note that the generators of $eval(Q_n)(I')$ will be singleton substitutions such as $[v_n \mapsto inj_n\ a_n]$ where a_n is a term in $\Sigma_{inc_n}(I_n)$ and inj_n means co-product injection. But inc_n is an inclusion, so a_n is a term in I_1. Because we compute Δ_{F_n} by translation into an uber-flower similar to Q_n, the generators of $\Delta_{F_n}(I')$ will have a similar form: $[v'_n \mapsto a_n]$ which defines the necessary isomorphism. In the case where all schemas are disjoint and variables are chosen appropriately, the isomorphism can be made an equality.

5.5 Further Patterns

Pushouts have a dual, called a pullback, obtained by reversing the arrows in the pushout diagram (see Figure 19). 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 colimits to data integration, as well as finding other useful design patterns and constructions for functorial data integration, is an important area for future work.

6 Conclusion

In this paper we have described what we believe to be the first truly practical formalism for the functorial data model, and work continues. In the short term, we aim to formalize our experimental "computational type-sides", and to develop a better conservativity checker. In the long term, we are looking to develop other design patterns for functorial 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 seems to be 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, 2003).

7 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

Abiteboul, Serge, Hull, Richard, & Vianu, Victor. (1995). *Foundations of databases*. Addison-Wesley.

Adámek, J., Rosický, J., & Vitale, E. M. (2011). *Algebraic theories*. Cambridge Tracts in Mathematics, no. 184. Cambridge University Press.

Alagic, Suad, & Bernstein, Philip A. (2001). A model theory for generic schema management. *DBPL*.

Bachmair, L., Dershowitz, N., & Plaisted, D.A. (1989). Completion without failure. *Resolution of equations in algebraic structures - rewriting techniques*, **2**, 1–30.

Barr, Michael, & Wells, Charles (eds). (1995). *Category theory for computing science, 2nd ed.* Prentice-Hall.

Blum, E. K., Ehrig, H., & Parisi-Presicce, F. (1987). Algebraic specification of modules and their basic interconnections. *J. comput. syst. sci.*, **34**(2-3), 293–339.

Bush, M.R., Leeming, M., & Walters, R.F.C. (2003). Computing left kan extensions. *Journal of symbolic computation*, **35**(2), 107 – 126.

Carmody, S., Leeming, M., & Walters, R.F.C. (1995). The todd-coxeter procedure and left kan extensions. *Journal of symbolic computation*, **19**(5), 459 – 488.

Doan, AnHai, Halevy, Alon, & Ives, Zachary. (2012). *Principles of data integration*. 1st edn. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Fagin, Ronald, Kolaitis, Phokion G., Miller, Renée J., & Popa, Lucian. (2005). Data exchange: semantics and query answering. *Theor. comput. sci.*, **336**(1), 89–124.

Fleming, Michael, Gunther, Ryan, & Rosebrugh, Robert. (2002). A database of categories. *Journal of symbolic computing*, **35**, 127–135.

Ghilardi, Silvio, Lutz, Carsten, & Wolter, Frank. (2006). Did i damage my ontology? a case for conservative extensions in description logics. *Pages 187–197 of: proceedings of KR2006*.

Goguen, J. A., & Burstall, R. M. (1984). *Logics of programs: Workshop, carnegie mellon university pittsburgh, pa, june 6–8*. Chap. Introducing institutions, pages 221–256.

Goguen, Joseph. (2004). *Information integration in institutions*.

Grust, Torsten. (2003). *Monad comprehensions. a versatile representation for queries. in the functional approach to data management, p.m.d. gray and l. kerschberg and p.j.h. king and a. poulovassilis (eds.)*. Springer Verlag.

Johnson, Michael, Rosebrugh, Robert, & Wood, R.J. (2002). Entity-relationship-attribute designs and sketches. *Theory and applications of categories*, **10**, 94–112.

Kapur, D., & Narendran, P. (1985). The knuth-bendix completion procedure and thue systems. *Siam journal on computing*, **14**(4), 1052–1072.

Knuth, Donald, & Bendix, Peter. (1970). Simple word problems in universal algebra. *Pages 263–297 of:* Leech, John (ed), *Computational Problems in Abstract Algebra*.

Lüth, Christoph, Roggenbach, Markus, & Schröder, Lutz. (2005). Ccc - the casl consistency checker. *Pages 94–105 of:* Fiadeiro, José (ed), *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004)*. Lecture Notes in Computer Science, vol. 3423.

Mitchell, John C. (1996). *Foundations of programming languages*. MIT Press.

Schultz, Patrick, Spivak, David I., Vasilakopoulou, Christina, & Wisnesky, Ryan. (2016). *Algebraic databases*. http://arxiv.org/abs/1602.03501.

Spivak, David I. (2012). Functorial data migration. *Inf. comput.*, **217**(Aug.), 31–51.

Spivak, David I, & Wisnesky, Ryan. (2015). Relational foundations for functorial data migration. *Dbpl*.

Contents

List of Figures