Relational Lenses: A Language for Updatable Views

Aaron Bohannon Benjamin C. Pierce Jeffrey A. Vaughan
University of Pennsylvania

ABSTRACT

We propose a novel approach to the classical *view update problem*. The view update problem arises from the fact that modifications to a database view may not correspond uniquely to modifications on the underlying database; we need a means of determining an "update policy" that guides how view updates are reflected in the database. Our approach is to define a *bi-directional* query language, in which every expression can be read both (from left to right) as a view definition and (from right to left) as an update policy. The primitives of this language are based on standard relational operators. Its type system, which includes record-level predicates and functional dependencies, plays a crucial role in guaranteeing that update policies are *well-behaved*, in a precise sense, and that they are *total*—i.e., able to handle arbitrary changes to the view.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*relational databases*

General Terms

Languages, Theory

Keywords

View update, lenses

1. INTRODUCTION

Our interest in the view update problem arose from of our work on a "universal data synchronizer" called Harmony [3, 4]. Harmony is a generic framework for reconciling disconnected updates to heterogeneous, replicated XML data. It can be used, for instance, to synchronize the bookmark files of several different web browsers, allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized by different users running different browser applications on disconnected machines. A central theme of the Harmony project has been bringing ideas from programming languages to bear on a set of problems more commonly regarded as belonging to databases or distributed systems.

Much of our work on Harmony has focused on developing the foundations of *bi-directional programming languages* [4], in which every program denotes a pair of functions—one for extracting a view of some complex data structure, and another for putting back an updated view into the original structure; we call these programs *lenses*. Lenses play a crucial role in the way Harmony deals with heterogeneous structures, mapping between diverse concrete application data formats and common abstract formats suitable for synchronization, and then translating the updates resulting from synchronization back to the original concrete data sources. The lens programming language of Harmony can be viewed as a solution to a specific instance of the general view update problem, where the data structures involved are trees.

As we have begun applying it to a broader range of applications, we have encountered many situations where we would like to use Harmony to synchronize information in traditional relational formats. Of course, relational data can be encoded as trees easily enough. But we have found that Harmony's tree-oriented programming language is not appropriate for the sorts of transformations commonly performed on relational data. In particular, its type system, which is based on regular tree automata, is good at capturing common XML schemas, but cannot encode familiar concepts from relational schemas, such as functional dependencies. This, in turn, means that the typing rules for familiar relational primitives such as joins are overly rigid, disallowing many useful cases.

Our aim in this paper is to design a new bi-directional language, based on the abstract framework of lenses but with a set of primitives and a type system specifically targeted at relational data. We plan to use this language in a new version of the Harmony system that will deal natively with synchronizing relational data, but the language also stands on its own as a novel approach to the classical view update problem in relational databases.

The view update problem can be illustrated as follows. Suppose we have a relation T, which is the result of joining relations R and S:

$$\left\{\begin{array}{c|cc} R & A & B \\ \hline & a & b \end{array}\right\} \quad \left\{\begin{array}{c|cc} S & B & C \\ \hline & b & c \end{array}\right\} \xrightarrow{\bowtie} \left\{\begin{array}{c|ccc} T & A & B & C \\ \hline & a & b & c \end{array}\right\}$$

If we update T—say, by deleting its single row—we may want to reflect this update in the original relations—i.e., to change R and/or S so that $R \bowtie S$ is the empty table. Here, the desired effect can be achieved by deleting the single

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

row in either or both of R and S; each of these options is a concrete example of an *update policy*. The *view update problem* is the problem of associating "reasonable" update policies with views.

Our approach to the view update problem is to design a new query language based on the relational algebra where every expression denotes *both* a view definition *and* a view update policy. Each primitive is annotated with enough parameters to guide the choice among a range of reasonable update policies, and the update policy for a compound expression is calculated by composing together the update policies of its constituents.

The example in Figure 1 illustrates the essential features of our approach. The three ovals together represent the following composite lens expression:

```
join_dl Tracks, Albums as Tracks1;
drop Date determined by (Track, unknown)
    from Tracks1 as Tracks2;
select from Tracks2 where Quantity > 2 as Tracks3
```

The first line joins the *Tracks* and *Albums* tables from the original database state, yielding a new state with a single table *Tracks1*. This new state is a database in its own right; this permits us to compose the other two lenses onto its right, with each stage in the composition each taking the "view" calculated by the previous stage as its starting "database." Likewise, the `join_dl` operation is itself a complete view definition, with its own individual update translation policy (the suffix `_dl` indicates that this policy will delete rows from its left-hand argument, as we shall see shortly). The `drop` in the second line projects away the *Date* attribute from the table *Tracks1*, yielding a new database with a single table *Tracks2*; again, the annotations (*Track*, *unknown*) will determine the update policy for this lens. Finally, the `select` lens in the third line chooses the rows in *Tracks2* satisfying the predicate *Quantity* > 2, yielding a final state with a table *Tracks3*. Each stage in the composition constructs a new table containing its result and removes the tables it uses as inputs; this ensures that each table can be used at most once in calculating the final view, which avoids the need to introduce complex schemas for tracking duplicated information in the view.

The top left box in the figure is the original state of the database. The solid arrows going down the left-hand side represent the (standard) step-by-step computation of the view state, yielding the original view state at the bottom left. We then perform an update on the view state, yielding the updated view state on the bottom right. To propagate the updates back to the original database, we apply the *putback* functions of each of the lenses in turn, represented by the dashed arrows moving up the right-hand side of the figure. The *putback* of the `select` lens uses the functional dependency *Track* $\rightarrow$ *Rating* to update the rating of the unseen "Lullaby" record. The *putback* of the `drop` lens uses the information in both original and updated states to restore the values in the *Date* column that were projected away by the *get*; also, the date "1989" is inferred for the new row containing "Lovesong," using the functional dependency. For the `join_dl` lens, records containing unseen information such as ("Paris", "4") are correctly restored to the table *Albums*, while deletions, such as the record for "Trust," and modifications, such as the quantity of "Disintegration," are correctly propagated. The final result is the updated database state on the top right.

The technical contributions of our work are twofold. First, our language incorporates a fairly rich notion of schemas for databases and views, including the functional dependencies shown in the example as well as record-level predicates. Each of our primitive lenses comes equipped with a typing rule specifying the domain (database schema) and range (view schema) on which its behavior is *total*—i.e., for which arbitrary schema-preserving updates to views are guaranteed to have reasonable translations. Second, our primitive lenses constitute a detailed analysis of the view update behavior of a number of fundamental relational operations in the presence of predicates and functional dependencies. Some—in particular, `join` [7]—have been studied previously. But the others turn out to be surprisingly interesting. For example, the way `select`'s behavior interacts with functional dependencies is quite subtle.

The rest of the paper is organized as follows. Section 2 reviews some familiar definitions and notational conventions. Section 3 introduces the abstract framework of lenses. Section 4 develops some fundamental operations involving relations and functional dependencies; these are used in Section 5 to define bi-directional versions of several fundamental relational operators. Sections 6 and 7 discuss related and future work. Proofs are omitted for the sake of brevity; they can be found in a companion technical report, available from `http://www.cis.upenn.edu/~bcpierce/harmony`.

2. BACKGROUND

We begin with a set of *attributes*, ranged over by A, B, C, and a homogeneous set of *values*, ranged over by a, b, c. (We do not assign specific domains or types to the attributes and do not have a distinguished null value.) We let U, V and X, Y, Z range over sets of attributes. *Records*, ranged over by m, n, l, are partial functions from attributes to values. We write records as $\{A = a_1, B = b_1\}$, or just (a_1, b_1) when the attributes are clear from context. We write $dom(m)$ for the domain of the record m and write $m : U$ to mean that $dom(m) = U$. If $A \in dom(m)$, then $m(A)$ is the value associated with A in m. We write $m[A \mapsto a]$ for the record with domain $dom(m) \cup \{A\}$ that maps A to a and agrees with m elsewhere, and $m[X]$ for the record with domain $X \cap dom(m)$ that agrees with m where it is defined.

We use M, N, L to range over *relations*—sets of records with the same domain. We say that M has domain U, or M is a relation *over* U, and write $M : U$, if $m : U$ for all $m \in M$. The usual set-theoretic operations are also defined on relations when both arguments have the same domain. We lift domain restriction to sets of records to define relational projection:

$$M[X] \quad = \quad \{m[X] \mid m \in M\}$$

Given $M : U$ and $N : V$, their natural join is defined by

$$M \bowtie N = \{l \mid l : UV \text{ with } l[U] \in M \text{ and } l[V] \in N\}.$$

P and Q range over *predicates*. In examples, we use familiar logical syntax for predicates; formally, however, we treat predicates simply as sets (generally infinite ones) of records having the same domain. We write $\top_U$ for the set of all records over the domain U—i.e., the always-true predicate over U. To lighten notation, we often just write $\top$ when U can be inferred from the context (and similarly for other U-subscripted notations below). Negation of predicates is set

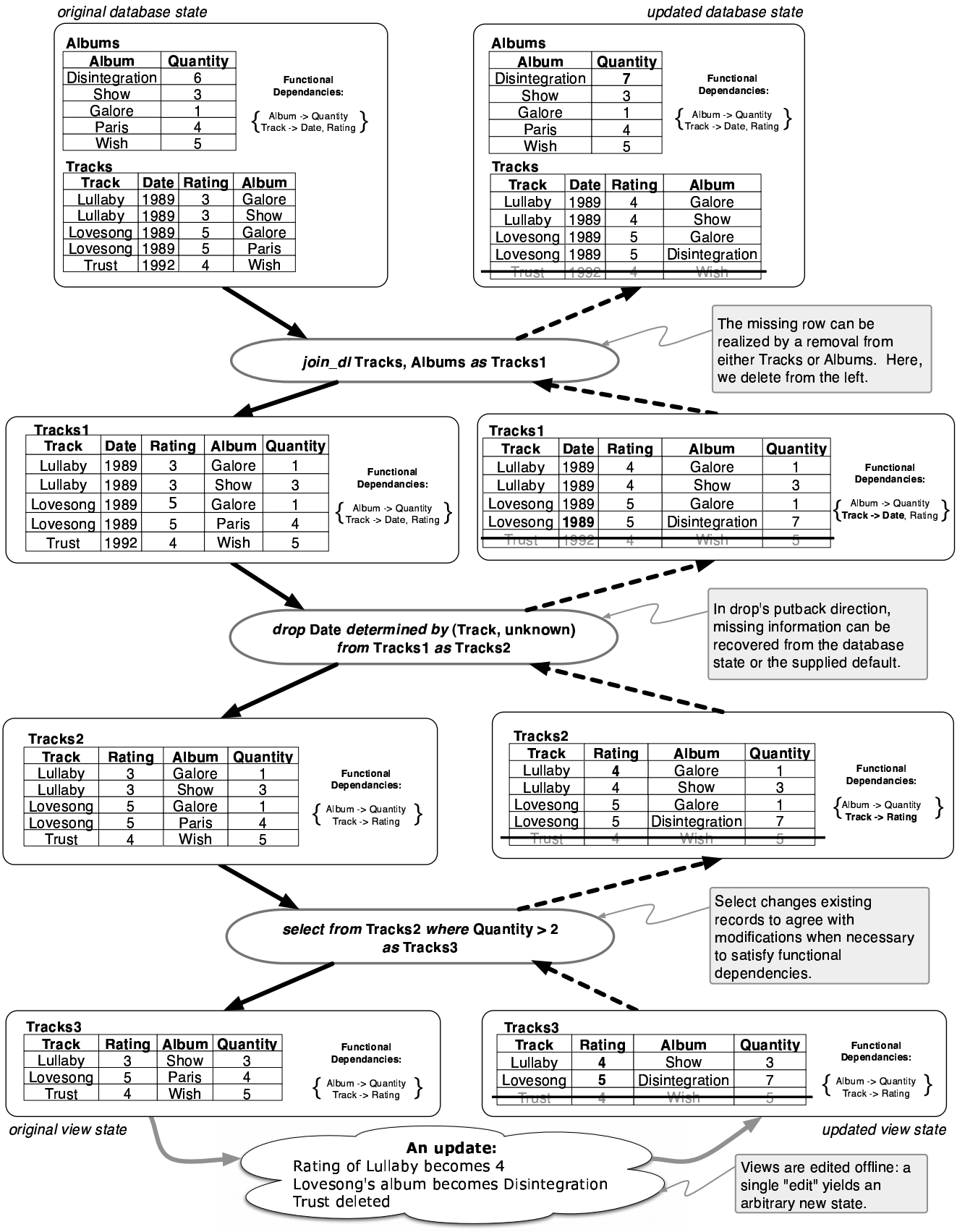


Figure 1: A Composite Lens

complement: we write $\neg_U M$ (or just $\neg M$) for $\top_U \setminus M$. Since predicates and relations are the same sorts of objects, mathematical intersection ($P \cap M$) expresses relational selection. Furthermore, all notation and definitions on relations are equally applicable to predicates.

We will be interested in cases where predicates are insensitive to the data associated with certain attributes. We write "P *ignores* X" when the truth of $m \in P$ can be determined without considering any of the values that m assigns to attributes in X—i.e., for all m and n, if $m[dom(m) - X] = n[dom(n) - X]$ then $m \in P \iff n \in P$.

Functional dependencies play a crucial role in our development. A functional dependency is a pair of attribute sets, written $X \to Y$. We say that $X \to Y$ is a functional dependency *over* the domain U, written $X \to Y : U$, if $X \subseteq U$ and $Y \subseteq U$. If $X \to Y$ is a functional dependency over U and M is a relation over U, we say that M satisfies $X \to Y$, written $M \models X \to Y$, if $m_1[X] = m_2[X]$ implies $m_1[Y] = m_2[Y]$ for all $m_1, m_2 \in M$.

Generally, we work with sets of functional dependencies. We say that F is a set of functional dependencies over U (written $F : U$) if $X \to Y : U$ for all $X \to Y \in F$. If F is a set of functional dependencies over U and M is a relation over U, then $M \models F$ means that $M \models X \to Y$ for all $X \to Y \in F$.

If F and F' are sets of functional dependencies over U, we say that F *implies* F', written $F \models_U F'$ when, for all M over U, $M \models F$ implies $M \models F'$. We write $F \equiv_U F'$ to mean that $F \models_U F'$ and $F' \models_U F$.

We use R, S to range over *relation names*; the function *sort* assigns a tuple (U, P, F) to each name, where U is a domain of attributes, P a predicate over U, and F a set of functional dependencies over U. If $sort(R) = (U, P, F)$, then $dom(R) = U$, $pred(R) = P$, and $fd(R) = F$. We say M *satisfies* (U, P, F) when $M : U$, $M \subseteq P$, and $M \models F$.

I and J range over *database instances* (or *databases*). A database I is a finite map from relation names to relations such that, if $I(R) = M$, then M satisfies $sort(R)$. A *database schema* (ranged over by Σ, Δ) is a set of relation names. A database I conforms to a schema Σ, written $I \models \Sigma$, if $dom(I) = \Sigma$.

3. LENSES

The starting point for this work is the class of bi-directional transformations known as *lenses*, which have previously been applied in the domain of semistructured data [4]. Lenses are bi-directional mappings between a *concrete domain*, thought of as a set of database states, and an *abstract domain*, thought of as a set of view states. (The abstract domain is "abstract" in the sense that, in general, abstract states contain less information than concrete ones—i.e., a view is usually smaller than the original database.) In the relational setting, both of these domains are database schemas.

3.1 Definition [Lenses]: Given schemas Σ and Δ, a *lens* v from Σ to Δ (written $v \in \Sigma \leftrightarrow \Delta$) is a pair of total functions $v\nearrow \in \Sigma \to \Delta$ ("$v\nearrow$" is pronounced "v get") and $v\searrow \in \Delta \times \Sigma \to \Sigma$ (pronounced "v putback").

The *get* component of a lens corresponds exactly to a view definition. Intuitively, the *get* and *putback* functions are intended to be be "inverses," in a sense that we will shortly make precise by imposing additional restrictions on their behavior. Since the view may discard information from the concrete domain, there is generally more than one way of inverting the *get* function. Hence, the *putback* function may be seen as a class of inverse functions from Δ to Σ that is indexed by an element of the concrete domain Σ.

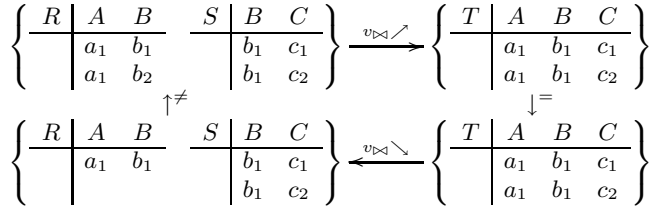
3.2 Definition [Well-behaved lenses]: Given schemas Σ and Δ along with a lens $v \in \Sigma \leftrightarrow \Delta$, we say that v is a *well-behaved* lens from Σ to Δ (written $v \in \Sigma \Leftrightarrow \Delta$) if it satisfies the laws GetPut and PutGet:

$$v\searrow (v\nearrow (I), I) = I \quad \text{for all } I \in \Sigma \qquad \text{(GetPut)}$$
$$v\nearrow (v\searrow (J, I)) = J \quad \text{for all } (J, I) \in \Delta \times \Sigma \qquad \text{(PutGet)}$$

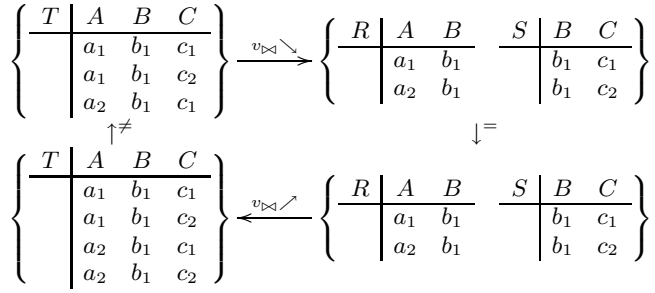
To highlight the importance of these properties, let us define a lens $v_{\bowtie}$—a naive first attempt at a bi-directional version of a natural join—that is *not* well-behaved. (Besides illustrating the definition and exercising some of the notation we have introduced, this lens will form the kernel of a better attempt at defining selection in Section 5.1.) We define the lens in a context where $dom(R) = AB$, $dom(S) = BC$, $dom(T) = ABC$, and the sorts of R, S, and T are otherwise unconstrained. In the *get* direction, this lens forms the natural join of the tables named R and S, calling the result T and removing R and S from the database. In the *putback* direction, the lens projects T on AB and BC to reconstruct R and S. Thus, $v_{\bowtie} \in \{R, S\} \leftrightarrow \{T\}$. (In fact, $v_{\bowtie} \in \Sigma \cup \{R, S\} \leftrightarrow \Sigma \cup \{T\}$ for any schema Σ with $R, S, T \notin \Sigma$.) More formally:

$$\begin{aligned} v_{\bowtie}\nearrow (I) &= I \backslash_{R,S} [T \mapsto I(R) \bowtie I(S)] \\ v_{\bowtie}\searrow (J, I) &= J \backslash_T [R \mapsto J(T)[AB]][S \mapsto J(T)[BC]] \end{aligned}$$

Although this definition has a pleasing simplicity and symmetry, one may rightly wonder whether the $v_{\bowtie}\searrow$ is an appropriate inverse for $v_{\bowtie}\nearrow$. The following example demonstrates that it is not—in particular, that $v_{\bowtie}$ does not satisfy GetPut:



Intuitively, the failure here is related to the fact that the view does not maintain all information present in the underlying data, but the *putback* function does not use the original database. A different problem is illustrated by the fact that the lens also fails to satisfy PutGet:



In this case, the initial state of table T could not have been the result of applying $v_{\bowtie}\nearrow$ to *any* database with the schema $\{R, S\}$, which directly implies that PutGet will fail.

Generally, the combination of well-behavedness and totality implies that *get* and *putback* functions must always be surjective. It is often very difficult to describe a pair of non-trivial domains over which surjectivity holds in both directions for a lens. Thus, totality imposes a stringent constraint on our lens design. The principle contribution of this research is showing how to use schemas with functional dependencies to describe useful domains for total, well-behaved lenses based upon traditional relational primitives.

4. REVISION

The technical keystone of our approach is an operation that "revises" a set of records from the original database state so that they agree with a set of new records from an updated view, with respect to a given set of functional dependencies; this operation is used in Section 5 to define several of the fundamental lens primitives. We begin its development in Sections 4.1 and 4.2 by building up a few preliminaries involving functional dependencies. The revision operator itself is presented in Section 4.3.

4.1 Functions on Functional Dependencies

We write $\mathit{left}(F)$ for the set of all attributes appearing on the left-hand side in a set of functional dependencies F. Similarly, $\mathit{right}(F)$ is the the set of attributes appearing on the right in F. We define $\mathit{names}(F)$ as $\mathit{left}(F) \cup \mathit{right}(F)$. We also define a function that returns the set of all attributes that appear on the right-hand side of a functional dependency "in an essential way":

$$\mathit{outputs}(F) = \{A \in U \mid \exists X \subseteq U.\ A \notin X \text{ and } F \models X \to A\}$$

That is, the outputs are the fields that are actually constrained by some other fields in the relation. It is easy to check that, if $F \equiv F'$, then $\mathit{outputs}(F) = \mathit{outputs}(F')$.

4.2 Tree Form

We will sometimes work with sets of functional dependencies with a special shape that we call *tree form*. We say F is in tree form if there exists a collection of pairwise disjoint sets of attributes $X_1, \ldots, X_n$ such that, if $X \to Y \in F$, then $X, Y \in \{X_1, \ldots, X_n\}$ and, moreover, the graph G over the nodes $\{1, \ldots, n\}$ with the edges $\{(i, j) \mid X_i \to X_j \in F\}$ is a directed acyclic graph with all nodes having in-degree at most one (i.e., a forest, in the sense of graph theory).

If F is in tree form and we have two functional dependencies $X_1 \to Y_1$ and $X_2 \to Y_2$ in F, we may draw some useful conclusions about their relationship. First, either $X_1 = X_2$ or else $X_1 \cap X_2 = \emptyset$ and $Y_1 \cap Y_2 = \emptyset$. Moreover, either $Y_1 = Y_2$ and $X_1 = X_2$, or else $Y_1 \cap Y_2 = \emptyset$. Finally, if $\{X \to Y\} \notin F$ and $F \cup \{X \to Y\}$ is in tree form, then $Y \cap \mathit{right}(F) = \emptyset$.

Consider the functional dependencies $\{A \to BC, C \to D\}$. This set is not literally in tree form by our definition, but it is semantically equivalent to a set of functional dependencies in tree form, namely $\{A \to B, A \to C, C \to D\}$.(Indeed, we conjecture that our theory can be extended to handle any set of functional dependencies that is equivalent to one in tree form.) By contrast, some sets of functional dependencies are *not* equivalent to any set in tree form. For example, $\{A \to B, B \to A\}$, $\{A \to C, B \to C\}$, and $\{A \to B, BC \to D\}$. Such sets of functional dependencies are problematic for the primitive lenses defined in the next section. Fortunately, they are also less commonly seen in practice than those in tree form, which include basic key constraints and the dependencies arising from typical joins on tables with key constraints.

If F is in tree form, we define $\mathit{leaves}(F)$ and $\mathit{roots}(F)$ as follows (note that these are sets of attribute *sets*):

$$\mathit{leaves}(F) = \{Y \mid \exists X.\ X \to Y \in F \text{ and } Y \cap \mathit{left}(F) = \emptyset\}$$
$$\mathit{roots}(F) = \{X \mid \exists Y.\ X \to Y \in F \text{ and } X \cap \mathit{right}(F) = \emptyset\}$$

4.3 Record and Relation Revision

Our goal is to define an operation $M \leftarrow_F L$ that will compute a new relation similar to M whose records do not conflict with those of L on the functional dependencies F. For instance, suppose $F = \{A \to B, B \to C\}$ and M and L are defined as follows:

$$M = \begin{array}{ccc} A & B & C \\ \hline a_1 & b_2 & c_1 \\ a_2 & b_1 & c_1 \\ a_3 & b_3 & c_3 \end{array} \qquad L = \begin{array}{ccc} A & B & C \\ \hline a_2 & b_2 & c_2 \\ a_4 & b_4 & c_4 \end{array}$$

Then we want the following behavior:

$$M \leftarrow_F L = \begin{array}{ccc} A & B & C \\ \hline a_1 & b_2 & c_2 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{array}$$

Relation revision is at the heart of several of our primitive lenses. In some cases, it appears in the form of a slightly higher-level operation that revises a relation and combines the result with the relation that was used during the revision. We call this operation *relational merge*. We define $M \overset{\cup}{\leftarrow}_F N = (M \leftarrow_F N) \cup N$, where $M : U$ and $N : U$. For example:

$$M \overset{\cup}{\leftarrow}_F L = \begin{array}{ccc} A & B & C \\ \hline a_1 & b_2 & c_2 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \\ a_4 & b_4 & c_4 \end{array}$$

It turns out to be somewhat tricky to define the revision operation precisely and prove that it behaves as desired. The remainder of Section 4 sketches this development; it can safely be skipped on a first reading.

The most basic operation using functional dependencies is one that updates some fields of a single record so that it conforms to the other records in a relation. We write $m \leftarrow\!\!+ n$ for the right-biased combination of records m and n—i.e., the record with domain $\mathit{dom}(m) \cup \mathit{dom}(n)$ that agrees with n on $\mathit{dom}(n)$ and agrees with m on $\mathit{dom}(m) - \mathit{dom}(n)$.

We first define a *single-dependency record revision* operation that takes a record m, a single functional dependency $X \to Y$, and a relation N satisfying $X \to Y$, and returns a revised record m' such that $\{m'\} \cup N$ satisfies $X \to Y$. Formally, this operation is defined by giving a mathematical relation over tuples $(X \to Y, N, m, m')$ (we typeset it as an arrow mapping m to m' with $X \to Y$ on top and N below) and then showing that we can treat this relation as a function, since there is always a unique such m' whenever $X \to Y$, N, and m share a domain U and $N \models X \to Y$. The following inference rules define the relation:

$$\frac{\begin{array}{c} m : U \qquad N : U \qquad X \to Y : U \\ N \models X \to Y \qquad n \in N \\ m[X] = n[X] \qquad m' = m \leftarrow\!\!+\, n[Y] \end{array}}{m \xrightarrow[N]{X \to Y} m'} \quad \text{(C-Match)}$$

$$\frac{\begin{array}{c} m : U \qquad N : U \qquad X \to Y : U \\ N \models X \to Y \qquad m[X] \notin N[X] \end{array}}{m \xrightarrow[N]{X \to Y} m} \quad \text{(C-NoMatch)}$$

By C-Match, if there exists $n \in N$ such that $m[X] = n[X]$, then m' is the result of overwriting the Y fields of m with those of n. Every such n will coincide on the values in the Y fields since $N \models X \to Y$. It should be clear that such an n exists exactly when $m[X] \in N[X]$. On the other hand, if $m[X] \notin N[X]$, then we may apply C-NoMatch to show that m is unchanged. The uniqueness of the result of record revision follows immediately.

It also follows from the definition that, if two records m_1 and m_2 agree on the fields in X, the records that result from revising m_1 and m_2 with the functional dependency $X \to Y$ and the relation N will agree on the fields in Y, assuming m_1 and m_2 also agree with some record in N on the fields in X. Moreover, as we might expect, revising a record twice with the same functional dependency and relation does nothing more than revising it just once.

The last step is to define a record revision operation that can use a *set* of functional dependencies to make a record conform to a relation. However, we need to be careful: there is no natural way to do this for some sets of functional dependencies. Consider the relation $N = \{(a_1, b_2), (a_2, b_1)\}$ over the attributes A, B, and assume that we want to revise the record (a_1, b_1) to conform to N using the functional dependencies $\{A \to B, B \to A\}$. Since we have no precedence among our functional dependencies, we do not know whether it would be better to revise it to (a_1, b_2) or (a_2, b_1). We clearly don't want to revise it to (a_3, b_3), because we want the net change to the record to be minimal. Fortunately, if the functional dependencies are in tree form, then the effect of a revision operation is clear: we should propagate updates down from the roots to the leaves.

Formally, the general revision operation is the least relation closed under the following inference rules:

$$\frac{m : U \qquad L : U}{m \xRightarrow[L]{\emptyset} m} \quad \text{(FC-Empty)}$$

$$\frac{\begin{array}{c} L \models F, X \to Y \qquad X \to Y \notin F \\ F, X \to Y \text{ in tree form} \qquad X \in \mathit{roots}(F, X \to Y) \\ m \xrightarrow[L]{X \to Y} m' \qquad m' \xRightarrow[L]{F} n \end{array}}{m \xRightarrow[L]{F, X \to Y} n} \quad \text{(FC-Step)}$$

Under the empty set of functional dependencies, a record revises to itself by FC-Empty. For a non-empty set of functional dependencies, we use FC-Step to first apply a single-dependency record revision using some root dependency and then proceed recursively with the remaining dependencies.

Intuitively, $m \xRightarrow[L]{F} m'$ means that m' is a version of m that has been minimally revised to conform to the relation L under the functional dependencies F. The following lemmas justify this intuition. The first says that the record revision operation does not change more fields than necessary. The second says that the revised record m' agrees with the relation L on the functional dependencies F. The third says that any pair of records revised with the same relation and functional dependencies are guaranteed not to conflict with each other according to the functional dependencies.

4.3.1 Lemma: If $m \xRightarrow[L]{F} n$ and $Z \cap \mathit{outputs}(F) = \emptyset$, then $m[Z] = n[Z]$.

4.3.2 Lemma: If $m \xRightarrow[L]{F} m'$, then $\{l, m'\} \models F$ for all $l \in L$.

4.3.3 Lemma: Suppose $m_1 \xRightarrow[L]{F} m_1'$ and $m_2 \xRightarrow[L]{F} m_2'$, where F is in tree form. If $\{m_1, m_2\} \models F$, then $\{m_1', m_2'\} \models F$.

The record revision operation can be lifted to sets of records in a natural way. We call this *relation revision*:

$$M \leftarrow_F L = \{m' \mid m \xRightarrow[L]{F} m' \text{ for some } m \in M\}.$$

A key property of relation revision is that it does not make up new values. Moreover (Lemma 4.3.5), it actually results in a relation satisfying F.

4.3.4 Lemma: Let $M : U$, $A \in U$, and $m' \in M \leftarrow_F L$. Then either $m'[A] \in M[A]$ or $m'[A] \in L[A]$.

4.3.5 Lemma: If $M \models F$, then $M \leftarrow_F L \models F$.

5. RELATIONAL LENS PRIMITIVES

We now describe some primitive lenses for updatable relational views. For brevity, we concentrate on three of the most interesting primitives; several others are described in the long version of the paper.

5.1 Selection

The *get* component of the `select` lens performs a relational selection on a table in the database; this part is simple. Equipping this *get* function with a *putback* function that behaves well in the presence of schemas with functional dependencies and predicates is a little trickier.

Letting v stand for the lens expression

$$\texttt{select from } R \texttt{ where } P \texttt{ as } S,$$

the behavior of the `select` lens is defined as follows:

$$\begin{array}{rcl} v\nearrow(I) & = & I \backslash_R [S \mapsto P \cap I(R)] \\ v\searrow(J, I) & = & J \backslash_S [R \mapsto M_0 \setminus N_\#] \\ \text{where} \quad M_0 & = & (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F J(S) \\ N_\# & = & (P \cap M_0) \setminus J(S) \\ F & = & \mathit{fd}(R) \end{array}$$

The *get* function extracts the relation R from I, selects with respect to the predicate P, and associates the resulting relation with the name S. The *putback* function forms an approximation M_0 of the updated table R in the concrete database by performing a relational merge of the records in the abstract database with those from the concrete database

that do not satisfy the predicate. However, we have to be careful: in some cases, M_0 contains records that, if put in the concrete database, would result in a violation of PutGet, but that may safely be removed. (Such a case will be illustrated later in this section.) We collect these records as $N_\#$ and remove them from the result (# stands for "conflict").

The following typing rule captures the domain over which the `select` lens is guaranteed to behave well:

$$\frac{\begin{array}{c} sort(R) = (U, Q, F) \\ sort(S) = (U, P \cap Q, F) \\ F \text{ is in tree form} \qquad Q \text{ ignores } outputs(F) \end{array}}{\begin{array}{c} \texttt{select from } R \texttt{ where } P \texttt{ as } S \in \\ \Sigma \uplus \{R\} \Leftrightarrow \Sigma \uplus \{S\} \end{array}} \quad \text{(T-Select)}$$

We use the notation $\Sigma_1 \uplus \Sigma_2$ for the disjoint union of Σ_1 and Σ_2 (which is defined only when $\Sigma_1 \cap \Sigma_2 = \emptyset$). Thus, the Σ in the conclusion of the typing rule may be instantiated with any database schema as long as $R, S \notin \Sigma$. Above the line, we declare the relationship that must exist between the sorts of the tables R and S, along with two other constraints. This typing rule can be read as a *theorem* that describes a set of database schemas and view schemas over which the above lens is well-behaved. Proving the theorem involves checking that the following facts are implied by the rule's premises:

Get Total: If $I \models \Sigma$ then $v\nearrow(I) \models \Delta$.

Put Total: If $I \models \Sigma$ and $J \models \Delta$ then $v\searrow(J, I) \models \Sigma$.

GetPut: If $I \models \Sigma$ then $v\searrow(v\nearrow(I), I) = I$.

PutGet: If $I \models \Sigma$ and $J \models \Delta$ then $v\nearrow(v\searrow(J, I)) = J$.

The restriction on the schema predicate Q is needed because the relational merge results in record revisions that may change any fields in $outputs(F)$. The requirement that F be in tree form is needed because our relational merge operation is only defined for such functional dependencies.

Here is a typical example of the use of `select`. Let v stand for the expression `select from` R `where` $C = c_2$ `as` S, and assume that $sort(R)$ is $(ABC, \top, \{A \to B\})$ and $sort(S)$ is $(ABC, C = c_2, \{A \to B\})$. We apply the lens in the *get* direction to a database I containing a single table R:

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_1 & b_1 & c_2 \\ & a_2 & b_2 & c_2 \end{array} \right\}^I \xrightarrow{v\nearrow(I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_1 & c_2 \\ & a_2 & b_2 & c_2 \end{array} \right\}^J$$

Now assume that an update to the table S results in the new database J'. Applying the lens in the *putback* direction results in an updated concrete database I':

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_2 & c_1 \\ & a_1 & b_2 & c_2 \\ & a_2 & b_2 & c_2 \end{array} \right\}^{I'} \xleftarrow{v\searrow(J',I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_2 & c_2 \\ & a_2 & b_2 & c_2 \end{array} \right\}^{J'}$$

In the table R, the record (a_1, b_1, c_1) has been replaced with the record (a_1, b_2, c_1) to preserve the functional dependency $A \to B$, even though this record was not visible in the abstract view.

One of the conditions imposed upon v by T-Select is that Q ignores $outputs(F)$. We can justify this by considering the following modification to the sorts: Assume that there is an ordering on elements, such that $b_i \leq c_j$ exactly when $i \leq j$, and that $sort(R)$ is $(U, B \leq C, A \to B)$ and $sort(S)$ is $(U, B \leq C \wedge C = c_2, A \to B)$. Then, in the example above, we would have $I \models \{R\}$, $J \models \{S\}$, and $J' \models \{S\}$, but $I' \not\models \{R\}$, which violates our rule that lenses must be total on their specified domains. The problem arises because the field B is updated in the process of a merge operation, but the record-level predicate associated with R puts a restriction on the values in that field.

Finally, an example illustrating the role of $N_\#$. Suppose we apply the lens `select from` R `where` $B = b_2$ `as` S with $sort(R)$ is $(U, \top, A \to B)$ and $sort(S)$ is $(U, B = b_2, A \to B)$ to the database I:

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \end{array} \right\}^I \xrightarrow{v\nearrow(I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline \end{array} \right\}^J$$

The abstract database table S is empty, but suppose the record (a_1, b_2, c_2) were added. One might expect the following behavior from the *putback* function:

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_2 & c_1 \\ & a_1 & b_2 & c_2 \end{array} \right\}^{I'} \xleftarrow{v\searrow(J',I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{J'}$$

However, this behavior would fail to satisfy the law PutGet because a subsequent *get* operation would retrieve both rows from the concrete database, while only one was present in the abstract database. The actual behavior of the `select` lens in the *putback* direction will delete the existing row in the concrete database.

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{I'} \xleftarrow{v\searrow(J',I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{J'}$$

In general, it is safe, if somewhat counter-intuitive, to delete records from the concrete database if they result in conflicting values determined by functional dependencies.

5.2 Join

Relational join is another operation with non-obvious *putback* semantics. There are actually many variants, all sharing the same *get* component but with different update policies; we consider just one update policy here, for brevity (the long version of the paper explores several others). In the *get* direction, the lens `join_dl` R, S `as` T performs a natural join. In the *putback* direction `join_dl` may add records to both tables R and S, but will only delete from table R (the name is intended to suggest "deleting from the left table").

The following example illustrates a typical use of `join_dl`:

$$\begin{array}{rcl} v & = & \texttt{join\_dl } R, S \texttt{ as } T \\ fd(R) & = & \{A \to B\} \end{array}$$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b_2 & c_2 \\ & a_2 & b_2 & c_3 \end{array} \right\}^I \quad \left\{ \begin{array}{c|c} S & C \\ \hline & c_1 \\ & c_2 \end{array} \right\}^I \xrightarrow{v\nearrow(I)} \left\{ \begin{array}{c|ccc} T & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b_2 & c_2 \end{array} \right\}^J$$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_2 & b_2' & c_2 \\ & a_2 & b_2' & c_3 \end{array} \right\}^{I'} \quad \left\{ \begin{array}{c|c} S & C \\ \hline & c_1 \\ & c_2 \end{array} \right\}^{I'} \xleftarrow{v\searrow(I,J)} \left\{ \begin{array}{c|ccc} T & A & B & C \\ \hline & a_2 & b_2' & c_2 \end{array} \right\}^{J'}$$

Note that the record (a_1, b_1, c_1) is deleted from R, rather than deleting (c_1) from S, in accordance with the "delete from the left table" policy. Moreover, the record (a_2, b_2, c_3) in R, which does not appear in the view, is updated to

(a_2, b_2', c_3) by the *putback*; this a consequence of the functional dependency $A \to B$.

The behavior of `join_dl` is defined as follows:

$$\begin{array}{rrcl} & v\nearrow(I) & = & I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)] \\ & v\searrow(J,I) & = & J\backslash_T[R \mapsto M][S \mapsto N] \\ \text{where} & (U,P,F) & = & sort(R) \\ & (V,Q,G) & = & sort(S) \\ & M_0 & = & I(R) \overset{\cup}{\leftarrow}_F J(T)[U] \\ & N & = & I(S) \overset{\cup}{\leftarrow}_G J(T)[V] \\ & L & = & (M_0 \bowtie N) \setminus J(T) \\ & M & = & M_0 \setminus L[U] \end{array}$$

The *get* function associates name T with the inner join of R and S; the complexity is in the *putback*, where each piece of the definition is necessary to guarantee well-behavedness. To see why, recall from section 3 that defining *putback* by

$$v_{broken1}\searrow(J,I) = J\backslash_T[R \mapsto J(T)[U]][S \mapsto J(T)[V]]$$

is unsatisfactory. This definition fails because (i) records that are not included in the view are dropped after the *putback*, and (ii) records may be added to create a view state which is not the result of any natural join.

To address (i), we merge the concrete relations with projections of $J(T)$. Adding records from the concrete view fixes (i), and the definition of merge guarantees that functional dependencies are obeyed. However, we still do not account for deletions. Consider defining the putback function as

$$v_{broken2}\searrow(J,I) = J\backslash_T[R \mapsto M_0][S \mapsto N].$$

We can see that this definition is not correct by examining the database state:

$$\left\{ \begin{array}{c|cc} R & A & B \\ \hline & a & b \end{array} \quad \begin{array}{c|cc} S & B & C \\ \hline & b & c \end{array} \right\}^I$$

In the *get* direction, v yields a view state with one row in table T. Removing this row and invoking $v\searrow$ yields database state I again, in violation of PutGet. The lens definition fails to distinguish deleted records from those simply absent from the initial view state.

Our actual definition avoids this problem by removing any records that would, when using *broken2*, violate PutGet. To achieve this, we simulate a *putback-get* with $v_{broken2}$ as the *putback* function and calculate which records appear that should not. This yields L. We find the final right hand relation by removing $L[U]$ from M_0.

Working the previous example with the full, correct definition of *putback* gives:

$$\left\{ \begin{array}{c|cc} R & A & B \\ \hline \end{array} \quad \begin{array}{c|cc} S & B & C \\ \hline & b & c \end{array} \right\}^I$$

We still need to address problem (ii). This is accomplished by the typing rule:

$$\frac{\begin{array}{c} sort(R) = (U,P,F) \qquad sort(S) = (V,Q,G) \\ sort(T) = (UV, P \bowtie Q, F \cup G) \\ G \models U \cap V \to V \\ F \text{ is in tree form} \qquad G \text{ is in tree form} \\ P \text{ ignores } outputs(F) \qquad Q \text{ ignores } outputs(G) \end{array}}{\texttt{join\_dl } R, S \texttt{ as } T \in \Sigma \uplus \{R,S\} \Leftrightarrow \Sigma \uplus \{T\}} \quad \text{(T-Join)}$$

The most interesting premise is $G \models U \cap V \to V$, which asserts that `join_dl` can only join two tables if the shared fields are a key for the right table. The following example shows why this restriction is necessary. Consider using `join_dl` where $U = AB$ and $V = BC$ and $dom(T) = ABC$. Table

T	A	B	C
	a_1	b_1	c_1
	a_1	b_1	c_2
	a_2	b_1	c_1

provides a counterexample to *putback*. Fortunately, our typing rule prevents $J(T)$ from having this form. (Imposing the key constraint on the right table is an arbitrary choice.) In a well typed join, $sort(T)$ ensures that any $J(T)$ is decomposable into components satisfying schemas $sort(R)$ and $sort(S)$. Additionally, F and G must be in tree form for relational merge to be well defined.

Space constraints preclude further discussion of join. In the long version of the paper, we define a variety of natural join lenses as instantiations of a generic lens, parameterized by two procedures: one that chooses records to delete in the case of ambiguities, and another that ensures functional dependencies are respected. For example, we define `join_dl'`, which is like `join_dl` but does not require tree form functional dependencies and updates, by using a destructive "squash" instead of merging.

5.3 Projection

Rather than defining a lens corresponding to a general relational projection operation, we consider a more basic lens, which we call `drop`, that projects away just a single column in the *get* direction. This simplification is useful because some special care must be taken to make sure that values in the missing column can be safely reconstructed; the policy for how to do this and the associated type constraints describing when it is feasible are most easily expressed by considering only a single column at a time. A column A can be dropped when the associated functional dependencies show that at most one X non-trivially determines A and that no other fields are fully or partially determined by A. In cases where a general projection makes sense, it can be implemented by composing several `drop` operations in sequence.

Letting v stand for the expression

$$\texttt{drop } A \texttt{ determined by } (X,a) \texttt{ from } R \texttt{ as } S$$

the behavior of the `drop` lens is:

$$\begin{array}{rrcl} & v\nearrow(I) & = & I\backslash_R[S \mapsto I(R)[U - A]] \\ & v\searrow(J,I) & = & J\backslash_S[R \mapsto M \leftarrow_{X \to A} I(R)] \\ \text{where} & M & = & (I(R) \bowtie J(S)) \cup (N_+ \bowtie \{\{A = a\}\}) \\ & N_+ & = & J(S) \setminus I(R)[U - A] \\ & U & = & dom(R) \end{array}$$

The syntax of the `drop` expression includes a set of attributes X upon which the field A has a functional dependency (the typing rule will ensure this) and a value a, which will be used as a default value for the column A when the unseen data do not determine it. The *get* component simply projects away the single field A. The behavior of the *putback* component revolves around reconstructing the values in the missing column. Records that were unchanged in the view are guaranteed to receive their original values. Records that were

added in the view (captured by N_+) are first paired with the default value, but this may be overwritten by the relational revision operation using the functional dependency $X \to A$.

We assign types to the `drop` lens with this rule:

$$\frac{\begin{array}{c} sort(R) = (U, P, F) \\ A \in U \qquad F \equiv F' \cup \{X \to A\} \\ sort(S) = (U - A, P[U - A], F') \\ P = P[U - A] \bowtie P[A] \qquad \{A = a\} \in P[A] \end{array}}{\begin{array}{c} \texttt{drop}\ A\ \texttt{determined by}\ (X, a)\ \texttt{from}\ R\ \texttt{as}\ S \in \\ \Sigma \uplus \{R\} \Leftrightarrow \Sigma \uplus \{S\} \end{array}} \quad \text{(T-Drop)}$$

This rule imposes several restrictions on the dropped column A. We require $A \in U$ as a sanity check. Then we require that F has a representation in which a set of fields X determines A. This set of fields must be unique because $F' : U - A$, as required by the sort of S. (Note that, if A is not determined by any other fields, then we may always choose $X = A$.) Indeed, it is easy to see that no reasonable behavior exists if this condition is not satisfied. For example, assume that $fd(R) = \{A \to C, B \to C\}$ (note that the typing rule does not require $fd(R)$ or $fd(S)$ to be in tree form) and we create a new table S by projecting away the field C. A *get* operation on the database I followed by the insertion of a new row (a_1, b_2) would cause a problem for the *putback* operation:

$$\left\{\begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b_2 & c_2 \end{array}\right\}^I \xrightarrow{v\nearrow(I)} \left\{\begin{array}{c|cc} S & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \end{array}\right\}^J$$

$$\left\{\begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b_2 & c_2 \\ & a_1 & b_2 & ? \end{array}\right\}^I \xleftarrow{v\searrow(I,J')} \left\{\begin{array}{c|cc} S & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \\ & a_1 & b_2 \end{array}\right\}^{J'}$$

Since B and C independently determine A, any value that is put in place of ? will result in a table that disobeys F.

We must also constrain the predicate in the schema. The expression $P = P[U - A] \bowtie P[A]$ implies that the predicate may not impose any sort of dependency between the value in field A and the values other fields. This is necessary because it is not possible to statically guarantee that the value assigned to the attribute A in the *putback* direction would have any particular relationship with the rest of the record. Finally, we require that $\{A = a\} \in P[A]$, so that filling in the default value is safe with respect to the predicate.

5.4 Lens Composition

Given lenses v and w, their composition $(v; w)$ has *get* and *putback* components with the following behavior:

$$\begin{array}{rcl} (v; w)\nearrow(I) & = & v\nearrow(w\nearrow(I)) \\ (v; w)\searrow(J, I) & = & w\searrow(v\searrow(J, w\nearrow(I)), I) \end{array}$$

The *get* direction applies the *get* function of v, yielding a first abstract database, to which the *get* function of w is applied. In the other direction, the two *putback* functions are applied in turn: first, the *putback* function of w is used to put J into the concrete database that the *get* of w was applied to, i.e., $w\nearrow(I)$; the result is then put into I using the *putback* function of v.

The typing rule for composition reflects the fact that the abstract domain of the first lens must coincide with the concrete domain of the second lens:

$$\frac{v \in \Sigma \Leftrightarrow \Sigma' \qquad w \in \Sigma' \Leftrightarrow \Delta}{v; w \in \Sigma \Leftrightarrow \Delta} \quad \text{(T-Compose)}$$

6. RELATED WORK

The framework of lenses was introduced by Foster, Greenwald, Moore, Pierce, and Schmitt [4], to which we refer readers for an extensive survey of the related literature. We review here just a few important points of comparison.

In this work, we have adopted the position that views should be "free-standing" entities—i.e., that it should be possible to perform updates on a view without examining the state of the concrete database. This perspective squarely aligns us with traditional theories of the view-update problem [1, 5], but distinguishes us from the traditional practice in the area, including the seminal work of Dayal and Bernstein [2], as well as the methods that have actually been implemented in commercial database systems. The use of free-standing views has practical advantages (views can be edited when the database is inaccessible) and a theory that naturally allows for composability. However, it is conceivable that a more ad-hoc solution may be appropriate for some applications.

View-update approaches inherently require a trade-off between the number of updates that can be translated successfully and the strength of the properties that update translation can be guaranteed to satisfy. At one extreme is the position embodied in Bancilhon and Spyratos's [1] notion of the *complement* of a view, which includes (at least) all the information missing from the view. When a complement is fixed, there is at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement." This approach has influenced numerous later works in the area, including recent papers by Lechtenbörger [9] and Hegner [6]. The approach places tight constraints on the update translation, however, allowing only a relatively small number of updates on a view to be translated.

Our lenses fall at the opposite end of the spectrum, guaranteeing a translation for any update on any view, but with necessarily fewer guarantees about the behavior of the translation. Others have also investigated looser restrictions on update translations. Our definition of `select`, for example, is similar to an example proposed by Keller [8] as an illustration of a natural update policy that would be disallowed under the constant complement approach. The notion of "dynamic views" formulated by Gottlob, Paolini, and Zicari [5] is another general formalism with relatively loose restrictions on update translation. Dayal and Bernstein [2] also adopt the more permissive position. These more lenient approaches all retain some version of our PutGet law, but vary in which additional laws are enforced.

Our careful treatment of functional dependencies in view update can be viewed as one of the main technical contributions of this work. The theories of "dynamic views" [5] and update translation under a constant complement [1] leave all such details abstract. Concrete solutions, on the other hand, tend to skirt the issue. For instance, Dayal and Bernstein's [2] update translations make some operational use of functional dependencies, but their translations are not generally guaranteed to respect functional dependencies, nor do they give a useful specification of when each update operations will preserve the functional dependencies of the un-

derlying data. Any rigorous mechanism for building updatable views over relational data with functional dependencies must choose between implementing some form of relational revision (as we have done) or putting severe restrictions on the sorts of views that can be built or the range of updates that can be performed.

7. FUTURE WORK

The choice of schema language is a fundamental issue in designing relational lenses, significantly constraining the design space for lens primitives. It may be fruitful to consider extensions to the schema language we have proposed here. Possibilities include multivalued dependencies and foreign key constraints—or, more generally, inclusion dependencies. Multivalued dependencies would allow us to support join lenses with wider domains. Among other benefits, inclusion constraints would allow us to define lenses for database normalization.

We are interested in practical concerns surrounding potential implementations of the theory we have presented. Our language is designed so that type-checking should be decidable, given a reasonable choice of language for expressing predicates. Another potential concern is the efficiency of implementing the *putback* operations as defined in the paper; it would be interesting to study whether we can preserve lens semantics while only working with small "deltas" instead of whole database states. We also want to consider how our lenses would interact with traditional DBMS requirements like transactionality. Finally, while we have confined this work to total lenses, it may be useful to investigate partial lenses that satisfy stronger properties or lenses that can fail during *putback* based upon some features of the unseen concrete data for greater flexibility of behavior.

We are also working on a prototype implementation, which will allow us to experiment with a larger set of examples, assess the usefulness of our approach, and compare its strengths and weaknesses against available technologies and prior proposals in the area.

Acknowledgments

We are grateful to Nate Foster, Stéphane Lescuyer, Zack Ives, and the PODS reviewers for helpful comments on earlier drafts of the paper, and to Tim Griffin, the members of the Penn Database Group (especially Zack Ives, Val Tannen, Susan Davidson, and Nick Taylor), and the whole Penn PL Club for stimulating discussions. The Harmony project has been supported by the National Science Foundation under grants ITR-0113226, *Principles and Practice of Synchronization*, CPA-0429836 *Harmony: The Art of Reconciliation*, and IIS-0534592 *Linguistic Foundations for XML View Update*.

8. REFERENCES

[1] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[2] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, September 1982.

[3] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. In *Database Programming Languages (DBPL)*, August 2005.

[4] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, pages 233–246, 2005.

[5] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

[6] Stephane J. Hegner. An order-based theory of updates for closed database views. 40:63–125, 2004. Summary in /emphFoundations of Information and Knowledge Systems, Second International Symposium, 2002, pp. 230–249.

[7] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *ACM SIGACT–SIGMOD Symposium on Principles of Database Systems, Portland, Oregon*, 1985.

[8] Arthur M. Keller. Comment on Bancilhon and Spyratos' "Update semantics and relational views". *ACM Trans. Database Syst.*, 12(3):521–523, 1987.

[9] Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, San Diego, California*, pages 49–55. ACM, June 9–12 2003.