Relational Lenses: A Language for Updatable Views

Aaron Bohannon Benjamin C. Pierce
Jeffrey A. Vaughan

Technical Report MS-CIS-05-27
Department of Computer and Information Science
University of Pennsylvania

December 26, 2005

Abstract

We propose a novel approach to the classical *view update problem*. The view update problem 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.

1 Introduction

Our interest in the view update problem arose in the context of our work on a "universal data synchronizer" called Harmony [7, 4, 5] 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. In particular, a major component of our work on Harmony has been on developing the foundations of *bi-directional programming languages* [5], 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 the system 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.

As it stands, Harmony deals only with tree-structured data. However, as we have begun applying it to a broader range of applications, we have encountered many situations where we would like to use it 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 the present work has been to design a new bi-directional language, based on the abstract framework of lenses but 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 described as follows. Consider the following 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 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 express 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*. (The suffix `_dl` indicates that the update policy for this lens is to delete rows from its left-hand argument, as we shall see shortly.) The second line drops the *Date* attribute from the table

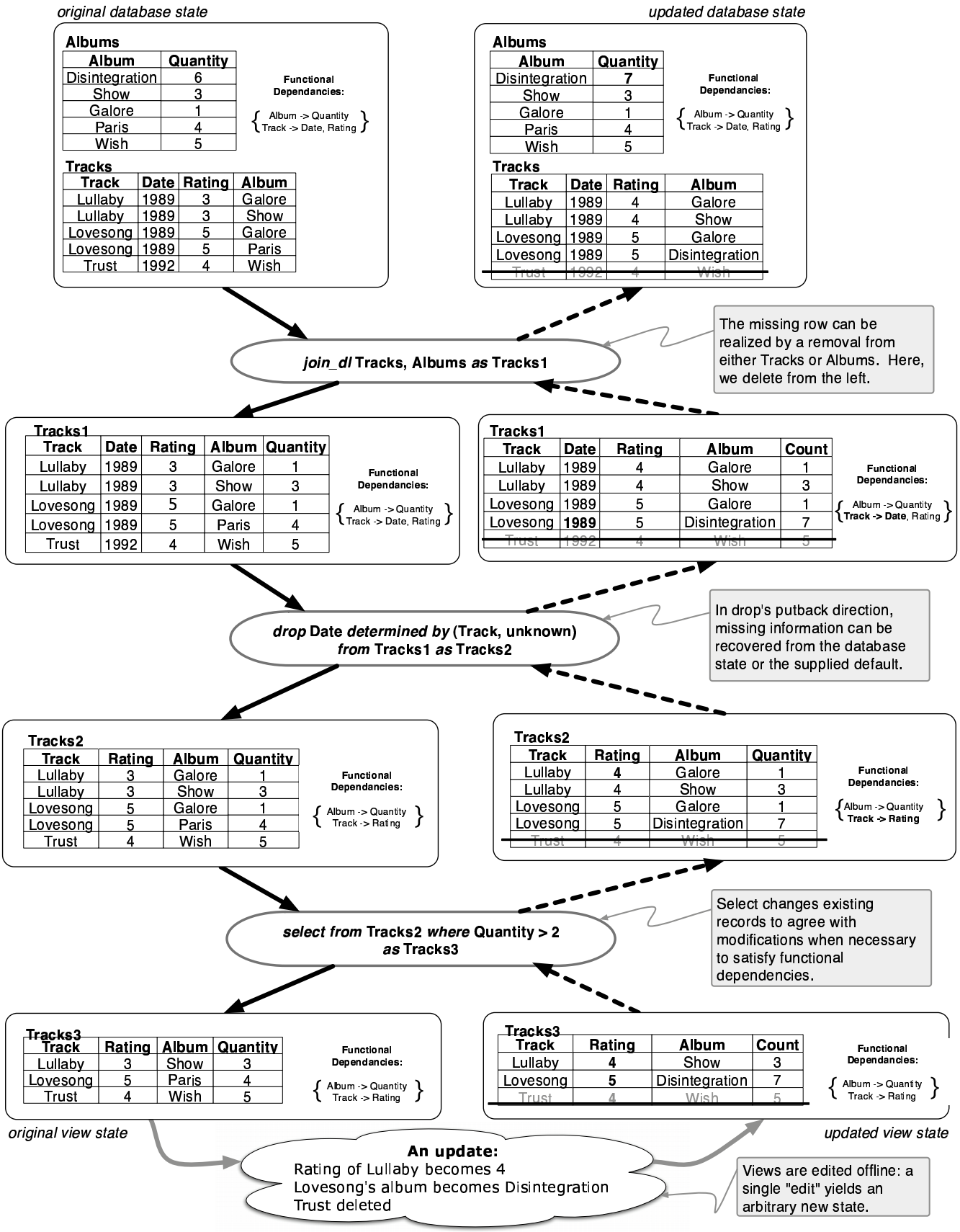


Figure 1: A Composite Lens

Tracks1, yielding a new state with table *Tracks2*. (The `drop` operation is a variant of project; again, the annotations (*Track*, *Unknown*) determine the update policy for this step). Finally, the third line selects the rows in *Tracks2* satisfying the predicate *Quantity* > 2, yielding a final state with a table *Tracks3*.

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 `drop` lens, for example, 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. 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` [12]—have been studied previously. But the others turn out to be surprisingly interesting. For example, the way `select`'s behavior interacts with functional dependencies turns out to be quite subtle.

The rest of the paper is organized as follows. Section 2 reviews some common 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.

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 attribute. A *record* is a partial function from attributes to values. Records are ranged over by m, n, l. We write records as $\{A = a_1, B = b_1\}$—or sometimes 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. We write $m[X]$ for the record with domain $X \cap dom(m)$ that agrees with m where it is definedIf $A \in dom(m)$, then we define $m[B/A] = m[U - A][B \mapsto m(A)]$.

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, written $M : U$, if $m : U$ for all $m \in M$. Note that if such a U exists, it is unique, except in the case that M is the empty set. The usual set-theoretic operations are defined on relations when both arguments have the same domain. We lift the operations of field update, attribute renaming, and domain restriction to sets of records (the last being equivalent to the usual notion of relational projection):

$$\begin{aligned} M[B/A] &= \{m[B/A] \mid m \in M\} \\ M[A \mapsto a] &= \{m[A \mapsto a] \mid m \in M\} \\ M[X] &= \{m[X] \mid m \in M\} \end{aligned}$$

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

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

We pause to record a couple of facts about joins that will come in handy later.

2.1 Fact: If $M_1 \subseteq N_1$ and $M_2 \subseteq N_2$, then $M_1 \bowtie M_2 \subseteq N_1 \bowtie N_2$.

2.2 Fact: If $L \subseteq M \bowtie N$ and $X \subseteq dom(M)$, then $L[X] \subseteq M[X]$.

P, 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. Negation of predicates is set 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 ($P \cap M$) suffices for expressing relational selection. Furthermore, we assume that all notation and definitions on relations is equally applicable to predicates.

We will be interested in cases where predicates are insensitive to certain fields. We write "P *ignores* X" to mean that 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 $m, n \in P$, 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 write $X \to Y : U$, to mean than $X \subseteq U$ 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 $X \to Y \in F$.

We will often find it useful to check whether a relation satisfies a set of functional dependencies by considering all of the records in the relation pairwise.

2.3 Fact: $M \models F$ iff $\{m_1, m_2\} \models F$ for all $m_1, m_2 \in M$.

Suppose F and F' are sets of functional dependencies over U. We say that F *implies* F', written $F \models_U F'$, if, for all M over U, $M \models F$ implies $M \models F'$. When U is clear from the context, we simply write $F \models F'$. We write $F \equiv_U F'$ (or just $F \equiv F'$) 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 is a predicate over U, and F is a set of functional dependencies over U. If $sort(R) = (U, P, F)$, then we define $dom(R) = U$, $pred(R) = P$, and $fd(R) = F$. We say M *satisfies* (U, P, F) when then $M : U$, $M \subseteq P$, and $M \models F$.

We use I, J to range over *database instances* (or just *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 [6]. 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 $v\nearrow \in \Sigma \to \Delta$ (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. In order to support a compositional approach, we take the perspective that a view state is an entire database (rather than just a single relation, as in many treatments of views). 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. If the *get* function preserves all information in the underlying data, then a unique inverse function exists and the Σ argument to the *putback* is not necessary for choosing among potential inverses; in this case, the lens is called *oblivious*. The requirement that the components of a lens be total has some interesting pragmatic ramifications, which we discuss below.

Most approaches to the view-update problem are formulated in terms of *update translators*, which map update *functions* on the abstract domain ($\Delta \to \Delta$) to update *functions* on the concrete domain ($\Sigma \to \Sigma$). The lens model replaces update translators with *putback* functions. But this difference is actually superficial. The type of the *putback* function, $(\Delta \times \Sigma \to \Sigma)$, is isomorphic to $\Delta \to (\Sigma \to \Sigma)$—that is, a *putback* function that maps *states* in the abstract domain (the results of updates) to *functions* on the concrete domain. It can be shown [15] that the class of lenses is a subset of the class of *consistent views* defined by Gottlob, Paolini, and Zicari [8]; indeed, modulo some technicalities the sets are isomorphic. This and other connections to previous treatments of view update are discussed further in the related work section.

As a specific example, let us define a lens $v_{\bowtie}$—a naive (and, as we shall see shortly, not quite satisfactory) first attempt at a bi-directional version of a natural join. The *get* and *putback* components of $v_{\bowtie}$ are defined as follows:

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

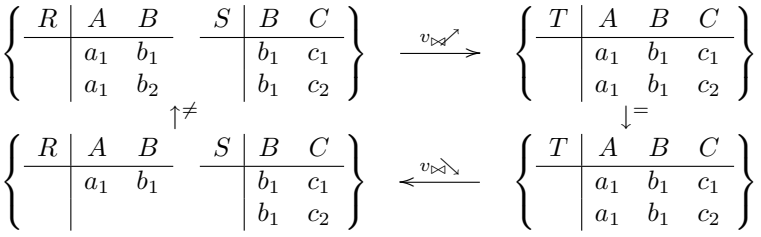
In the *get* direction, this lens forms the natural join of the tables R and S, calling the result T and removing R and S from the database. In the *putback* direction, the lens computes the projections of T on the fields of R and S to reconstruct those tables. Thus, $v_{\bowtie} \in \{R, S\} \leftrightarrow \{T\}$. (In fact, $v_{\bowtie} \in \Sigma \cup \{R, S\} \leftrightarrow \Sigma \cup \{T\}$ for any schema Σ with $R, S, T \notin \Sigma$.)

This lens is oblivious since the *putback* function does not use the argument I. However, one may rightly wonder whether the $v_{\bowtie}\searrow$ is an appropriate inverse for $v_{\bowtie}\nearrow$. We use two laws to ensure the *get* and *putback* functions of a lens are suitable inverses of one another.

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:

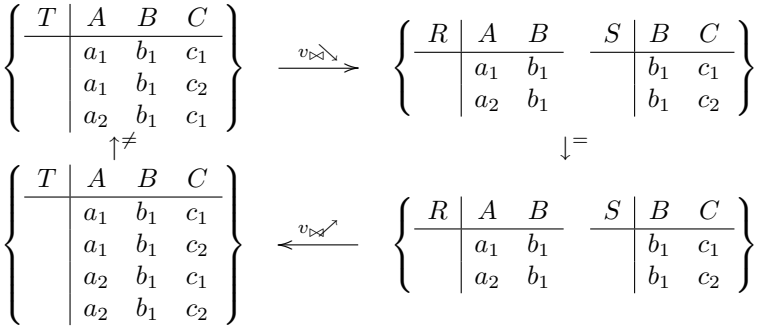
$$\begin{aligned} &v\searrow(v\nearrow(I), I) = I \quad \text{for all } I \in \Sigma && \text{(GetPut)} \\ &v\nearrow(v\searrow(J, I)) = J \quad \text{for all } (J, I) \in \Delta \times \Sigma && \text{(PutGet)} \end{aligned}$$

These properties provide a concrete basis for demonstrating the shortcomings of the $v_{\bowtie}$ lens. The following example demonstrates 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 lens is oblivious. 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.

In section 5 we introduce an assortment of primitive lenses which can be combined to create complex views and update policies. We must prove that these lenses are total and well-behaved at some type $\Sigma \Leftrightarrow \Delta$; we do so by establishing, for each lens v, the following four properties:

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

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. On the other hand, totality of a lens provides the benefit that the success or failure of an update program on a view can be determined offline—without examining the state of the concrete database (we return to this point in more detail in Section 6). 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 Record Revision

The technical keystone of our approach is an operation that "revises" a record from the original database state so that it agrees with a set of new records from an updated view, with respect to a given set of functional dependencies. This operation will be used to define several of the fundamental lens primitives in Section 5.

4.1 Functions on Functional Dependencies

Before we come to record revision itself, we need a few more definitions involving 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-hand side in F.

$$\mathit{left}(F) \;=\; \bigcup_{X \to Y \in F} X \qquad \mathit{right}(F) \;=\; \bigcup_{X \to Y \in F} Y$$

We also define $\mathit{names}(F)$ as $\mathit{left}(F) \cup \mathit{right}(F)$

The functions $\mathit{left}(F)$ and $\mathit{right}(F)$ will be useful in some later definitions; however, they tell us very little about which attributes play an essential role in witnessing a statement $M \not\models F$ for some $M : U$. To

this end, we define a function that returns the set of all attributes that appear on the left-hand side of a functional dependency "in an essential way":

$$\mathit{inputs}(F) = \{A \in U \mid \exists M, a.\ M \models F \text{ and } M[A \mapsto a] \not\models F\}$$

A field A is in $\mathit{inputs}(F)$ if we can pick a relation satisfying F and make it inconsistent by filling column A with a constant. For example, $A \in \mathit{inputs}(A \to B)$ because, while the relation $\{\{A = a_1, B = b_1\}, \{A = a_2, B = b_2\}\}$ models $A \to B$, the relation $\{\{A = a_0, B = b_1\}, \{A = a_0, B = b_2\}\}$ does not. Similarly, we 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 non-trivially determined by some other fields in the relation. It is easy to check that, if $F \equiv F'$, then $\mathit{inputs}(F) = \mathit{inputs}(F')$ and $\mathit{outputs}(F) = \mathit{outputs}(F')$.

4.2 Tree Form

We will sometimes need to 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 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 consider 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. We know that, either $X_1 = X_2$, or else $X_1 \cap X_2 = \emptyset$ and $Y_1 \cap Y_2 = \emptyset$. We also know that, either $Y_1 = Y_2$ and $X_1 = X_2$, or else $Y_1 \cap Y_2 = \emptyset$. Additionally, we can show that, if $F \uplus \{X \to Y\}$ is in tree form, then $Y \cap \mathit{right}(F) = \emptyset$.

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

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

It is possible for two distinct sets of functional dependencies, both in tree form, to be equivalent—consider, for example, $\{A \to BC\}$ and $\{A \to B, A \to C\}$. However, every F in tree form has a unique representation F' where $|X| = 1$ for all $X \in \mathit{leaves}(F')$. We call this the *canonical* tree form of F.

Some examples of functional dependencies that are *not* in tree form are $\{A \to B, B \to A\}$, $\{A \to C, B \to C\}$, and $\{A \to B, BC \to D\}$.

When working with functional dependencies, it is sometimes useful to be able to manipulate them syntactically. To this end, we define the relation $F \vdash_U X \to Y$, pronounced "F *syntactically satisfies* $X \to Y$," with the following inference rules:

$$\frac{X \to Y : U \qquad X \to Y \in F}{F \vdash_U X \to Y} \tag{FD-Expl}$$

$$\frac{Y \subseteq X \subseteq U}{F \vdash_U X \to Y} \tag{FD-Refl}$$

$$\frac{F \vdash_U X \to Y}{F \vdash_U XZ \to YZ} \tag{FD-Aug}$$

$$\frac{F \vdash_U X \to Y \qquad F \vdash_U Y \to Z}{F \vdash_U X \to Z} \tag{FD-Trans}$$

The relation defined by these rules coincides with the usual (more semantic) definition of when a functional dependency is satisfied by a relation:

4.2.1 Proposition: $F \vdash_U X \to Y$ iff $F \models_U X \to Y$.

Proof: This is a standard result in the theory of functional dependencies. □

4.2.2 Lemma: If $F \vdash X_1 \to Y_1$ and $A \in Y_1$, then either $A \in X_1$ or else there exists $X_2 \to Y_2 \in F$ such that $A \in Y_2$.

Proof: By induction on the derivation of $F \vdash X_1 \to Y_1$:

- Case FD-Expl: We may choose $X_2 = X_1$ and $Y_2 = Y_1$ since we have $X_1 \to Y_1 \in F$.
- Case FD-Refl: We have $A \in X_1$ since $Y_1 \subseteq X_1$.
- Case FD-Aug: We have $X_1 = XZ$ and $Y_1 = YZ$. If $A \in Z$, then $A \in X_1$. If $A \notin Z$, then we have $Z \in Y$ and we apply the induction hypothesis.
- Case FD-Trans: We have some Z such that $F \vdash X_1 \to Z$ and $F \vdash Z \to Y_1$. By the induction hypothesis for the second premise, we have either $A \in Z$ or else there exists some $X_2 \to Y_2 \in F$ such that $A \in Y_2$. The latter case immediately gives us our desired conclusion, so let's consider the former case where $A \in Z$. Then we may use the induction hypothesis for the first premise, which gives us, either $A \in X_1$ or else there exists some $X_2 \to Y_2 \in F$ such that $A \in Y_2$, which is exactly what we need. □

4.2.3 Lemma: $\mathit{outputs}(F) \subseteq \mathit{right}(F)$

Proof: Let $A \in \mathit{outputs}(F)$. Then there exists X such that $F \models X \to A$ and $A \notin X$. By Lemma 4.2.5, since $A \notin X$, there must exist $X_2 \to Y_2 \in F$ such that $A \in Y_2$. Hence, $A \in \mathit{right}(F)$. □

4.2.4 Lemma: If F is a set of functional dependencies in tree form, then $\mathit{outputs}(F) = \mathit{right}(F)$.

Proof: Let U be the domain of F. By Lemma 4.2.6, $\mathit{outputs}(F) \subseteq \mathit{right}(F)$ for any set of functional dependencies F, we need only show that $\mathit{right}(F) \subseteq \mathit{outputs}(F)$. Pick $A \in \mathit{right}(F)$. Then then there must be $X, Y \subseteq U$ such that $X \to AY \in F$. We may assume that $A \notin Y$, and, as F is in tree form, we know $A \notin X$. Then we may construct a derivation showing $F \vdash_U X \to A$ as follows:

$$\dfrac{\dfrac{X \to AY \in F}{F \vdash_U X \to AY}\ \text{FD-Expl} \quad \dfrac{A \subseteq AY \subseteq U}{F \vdash_U AY \to A}\ \text{FD-Refl}}{F \vdash_U X \to A}\ \text{FD-Trans}$$

So, by Lemma 4.2.4, we may conclude that $F \models X \to A$, which shows that $A \in \mathit{outputs}(F)$. □

4.3 Single-Dependency Record Revision

The most basic operation using functional dependencies is one that updates some fields of a record so that it conforms to the other records in a relation.

We write $m \leftarrow\!\!+ n$ for the left-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 $\mathit{dom}(m) - \mathit{dom}(n)$. Alternatively, $m \leftarrow\!\!+ \{A_1 = a_1; \ldots; A_k = a_k\} = m[A_1 \mapsto a_1] \ldots [A_k \mapsto a_k]$.

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 $(X \to Y, N, m, m')$ and then showing that we can treat this relation as a function by observing that we 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 two inference rules define the relation:

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

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

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.

4.3.1 Lemma: Suppose $m \xrightarrow[N]{X \to Y} m'$, where $U = dom(m)$. Then $N : U$ and $X \to Y : U$. Furthermore, $N \models X \to Y$.

Proof: By inspection of the inference rules. □

4.3.2 Lemma: Suppose $m : U$, $N : U$, $X \to Y : U$, where $N \models X \to Y$. Then there exists m' such that $m \xrightarrow[N]{X \to Y} m'$.

Proof: Assume, first, that $m[X] \in N[X]$. Then there exists $n \in N$ such that $m[X] = n[X]$. So we may apply C-MATCH. On the other hand, if $m[X] \notin N[X]$, we may apply C-NOMATCH. □

4.3.3 Lemma: If $m \xrightarrow[N]{X \to Y} m'$ and $m \xrightarrow[N]{X \to Y} m''$, then $m' = m''$.

Proof: As noted, at most one of the rules C-MATCH and C-NOMATCH may apply. If the rule C-NOMATCH applied, then $m' = m$ and $m'' = m$; so $m' = m''$. If the rule C-MATCH applied, then it could be possibly be instantiated with different values n' and n''. However, since $N \models X \to Y$ and $m[X] = n[X]$, we may conclude that $n'[Y] = n''[Y]$. Hence, $m' = m''$. □

The unique result of revising a record $m : U$ with the functional dependency $X \to Y$ and the relation N has the same domain as m.

4.3.4 Lemma: If $m \xrightarrow[N]{X \to Y} m'$, then $dom(m) = dom(m')$.

Proof: The case of C-NOMATCH is immediate since $m = m'$. In the case of C-MATCH, we must show that $dom(m[n[Y] \mapsto)] = dom(m)$. The follows from the fact that $Y \subseteq dom(m)$, which is true since $X \to Y : dom(m)$ by Lemma 4.3.1. □

Values that were not originally in a record field must come from the relation that is involved in the revision.

4.3.5 Lemma: If $m \xrightarrow[N]{X \to Y} m'$, then for all $A \in dom(m)$, either $m'(A) = m(A)$ or $m'[A] \subseteq N[A]$.

Proof: In the case of C-MATCH where $A \in Y$, then $m'(A) = n(A)$, so $m[A] \in N[A]$. If $A \notin Y$, then $m'(A) = m(A)$. In the case of C-NOMATCH, $m'(A) = m(A)$ since $m' = m$. □

The revision operation leaves some fields of the record unchanged and may potentially update others. The fields that are guaranteed to remain unchanged are exactly $X \cup (U - Y)$.

4.3.6 Lemma: If $m \xrightarrow[N]{X \to Y} m'$, then $m[X \cap Y] = m'[X \cap Y]$.

Proof: If $m[X] \notin N[X]$, then the result is immediate because the rule C-NoMatch requires that $m = m'$. Otherwise, by inversion of the C-Match rules, we have an $n \in N$ such that $n[X] = m[X]$. Since $n : U$ and $Y \subseteq U$, it is easy to check that $(m \leftarrow\!\!+ n[Y])[Y] = n[Y]$. Hence:

$$\begin{aligned} m'[X \cap Y] &= (m \leftarrow\!\!+ n[Y])[X \cap Y] \\ &= (m \leftarrow\!\!+ n[Y])[Y][X] \\ &= n[Y][X] \\ &= n[X][Y] \\ &= m[X][Y] \\ &= m[X \cap Y] \end{aligned}$$

□

4.3.7 Lemma: If $m \xrightarrow[N]{X \to Y} m'$, then $m[U - Y] = m'[U - Y]$.

Proof: If $m[X] \notin N[X]$, then the result is immediate because the rule C-NoMatch requires that $m = m'$. Otherwise, by inversion of the C-Match rules, we have an $n \in N$ such that $n[X] = m[X]$. Then we have:

$$\begin{aligned} m'[U - Y] &= (m \leftarrow\!\!+ n[Y])[U - Y] \\ &= m[U - Y] \leftarrow\!\!+ n[Y][U - Y] \\ &= m[U - Y] \leftarrow\!\!+ n[\emptyset] \\ &= m[U - Y] \leftarrow\!\!+ \{\} \\ &= m[U - Y] \end{aligned}$$

□

4.3.8 Lemma: If $m \xrightarrow[N]{X \to Y} m'$, then $m[X] = m'[X]$.

Proof: By Lemmas 4.3.6 and 4.3.7. □

We also observe that if two records m and l agree on the fields in X, the records that result from revising m and l with the functional dependency $X \to Y$ and the relation N will agree on the fields in Y, assuming m and l also agree, on the fields in X, with some record in N.

4.3.9 Lemma: Suppose that $m_1 \xrightarrow[N]{X \to Y} m_1'$ and $m_2 \xrightarrow[N]{X \to Y} m_2'$. If there exists an $n \in N$ such that $m_1[X] = m_2[X] = n[X]$, then $m_1'[Y] = m_2'[Y]$.

Proof: As above, given that $n : U$ and $Y \subset U$, we can easily check that $m \leftarrow\!\!+ n[Y][Y] = n[Y]$, for any m. Therefore, we have

$$m_1'[Y] = m_1' \leftarrow\!\!+ n[Y][Y] = n[Y]$$

and

$$m_2'[Y] = m_2' \leftarrow\!\!+ n[Y][Y] = n[Y].$$

Hence, $m_1'[Y] = m_2'[Y]$. □

4.3.10 Lemma: Suppose $m_1 \xrightarrow[N]{X \to Y} m_1'$ and $m_2 \xrightarrow[N]{X \to Y} m_2'$. If $m_1[XY] = m_2[XY]$, then $m_1'[XY] = m_2'[XY]$.

Proof: Straightforward. □

As we might expect, revising a record twice with the same functional dependency and relation does nothing more than revising it just once.

4.3.11 Lemma: If $m \xrightarrow[N]{X \to Y} m'$ and $m' \xrightarrow[N]{X \to Y} m''$, then $m' = m''$.

Proof: If $m[X] \notin N[X]$, then the result follows from Lemma 4.3.3 and the fact that $m = m'$. So we will assume that C-MATCH was used to derive $m \xrightarrow[N]{X \to Y} m'$, and by inversion, we have an $n \in N$ such that $n[X] = m[X]$. We may then use Lemma 4.3.8 to show $m[X] = m'[X] = n[X]$. Therefore, by Lemma 4.3.9, $m'[Y] = m''[Y]$. Lemma 4.3.4 ensures us that $m{:}U$, so we may use Lemma 4.3.7 to establish that $m'[U - Y] = m''[U - Y]$. Then we put these together to obtain $m'[U] = m''[U]$. By Lemma 4.3.4, $m'' : U$, so we may conclude that $m' = m''$. □

4.3.12 Lemma: Let $m \in M$, $M \models X \to Y$, and $N \subseteq M$. If $m \xrightarrow[N]{X \to Y} m'$, then $m = m'$.

Proof: If $m[X] \notin N[X]$, then the result follows immediately since FC-NOMATCH was used. On the other hand, if we have $n \in N$ such that $n[X] = m[X]$, then we must have $n[Y] = m[Y]$ since $\{m, n\} \models X \to Y$. Thus, $m' = m \leftarrow\!\!+\, n[Y] = m$. □

4.3.13 Lemma: If $M, m \models X \to Y$ then $m \xrightarrow[M]{F} m$.

Proof: Trivial. □

4.4 General Record Revision

Now, 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 clear 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). It would be unreasonable to revise it to (a_3, b_3) because we want the net change to the record to be minimal, but we don't know which part of the record to hold constant. Fortunately, if the functional dependencies are tree-like, 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:

$$\frac{m : U \qquad L : U}{m \xRightarrow[L]{\emptyset} m} \tag{FC-EMPTY}$$

$$\frac{L \models F, X \to Y \qquad X \to Y \notin F \qquad F \text{ in tree form} \qquad X \in \mathit{roots}(F, X \to Y) \qquad m \xrightarrow[L]{X \to Y} m' \qquad m' \xRightarrow[L]{F} n}{m \xRightarrow[L]{F, X \to Y} n} \tag{FC-STEP}$$

Under the empty set of functional dependencies, a record revises to itself by FC-EMPTY. For a non-empty set of functional dependencies, we first apply a single-dependency record revision using one of the functional dependencies from the roots of the tree. Then we proceed recursively according to FC-STEP.

4.4.1 Lemma: Suppose $m \xRightarrow[L]{F} n$. Then $L : U$ and $F : U$, where $U = \mathit{dom}(m)$. Furthermore, F is in tree form and $L \models F$.

Proof: Straightforward induction, using Lemma 4.3.1 in the FC-STEP case. □

We intend the notation $m \overset{F}{\underset{L}{\Longrightarrow}} m'$ to mean 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.

4.4.2 Lemma: If $m \overset{F}{\underset{L}{\Longrightarrow}} n$, then $dom(m) = dom(n)$.

Proof: Simple induction using Lemma 4.3.4. □

4.4.3 Lemma: If $m \overset{F}{\underset{L}{\Longrightarrow}} m'$, then for all $A \in dom(m)$, either $m'(A) = m(A)$ or $m'[A] \in L[A]$.

Proof: Simple induction using Lemma 4.3.5. □

4.4.4 Lemma: Suppose $m : U$, $L : U$ and $F : U$, where F is in tree form and $L \models F$. Then there exists n such that $m \overset{F}{\underset{L}{\Longrightarrow}} n$.

Proof: Proof by induction on the size of F:

- $|F| = 0$: Trivial since we may apply FC-STEP. In this case $n = m$.
- $|F| > 0$: As F is in tree form and not empty, we can pick $X \to Y \in F$ such that $X \in roots(F)$. Let $F' = F \setminus \{X \to Y\}$ and let m' be the unique record such that $m \xrightarrow[L]{X \to Y} m'$. (We know such a record exists by Lemmas 4.3.2 and 4.3.3.) Since $L \models F'$ and F' is in tree form, we may appeal to the induction hypothesis to show that there exists n such that $m' \overset{F'}{\underset{L}{\Longrightarrow}} n$. Hence, we may instantiate the rule FC-STEP to show that $m \overset{F'}{\underset{L}{\Longrightarrow}} n$. □

The record revision operation does not change more fields than necessary:

4.4.5 Lemma: If $m \overset{F}{\underset{L}{\Longrightarrow}} n$ and $Z \cap outputs(F) = \emptyset$, then $m[Z] = n[Z]$.

Proof: Simple induction on the derivation of $m \overset{F}{\underset{L}{\Longrightarrow}} n$ making use of Lemma 4.3.7. □

4.4.6 Lemma: If $m \overset{F}{\underset{L}{\Longrightarrow}} n$ and $X \to Y \in F$, then $n \xrightarrow[L]{X \to Y} n$.

Proof: We proceed by induction on the derivation of $m \overset{F}{\underset{L}{\Longrightarrow}} n$:

- Case FC-EMPTY: Vacuously true since F is empty.
- Case FC-STEP:

  $F = F', X' \to Y'$ where $X' \to Y' \notin F$
  F is in tree form
  $X \in roots(F)$
  $m \xrightarrow[L]{X' \to Y'} m'$
  $m' \overset{F'}{\underset{L}{\Longrightarrow}} n$

  We will consider two cases:

– $X \to Y \in F'$: Then the result is immediate from the induction hypothesis.
– $X \to Y \notin F'$: Then $X \to Y = X' \to Y'$. So $Y \cap \mathit{right}(F) = \emptyset$. Also, since $X \in \mathit{roots}(F)$, we know $X \cap \mathit{right}(F) = \emptyset$. Taken together, we have $m'[XY] = n[XY]$ by Lemma 4.4.5. By Lemma 4.3.2, we have a record m'' such that $m' \xrightarrow[L]{X \to Y} m''$ and a record n' be the unique record such that $n \xrightarrow[L]{X \to Y} n'$. By Lemma 4.3.10, we know $m''[XY] = n'[XY]$.

$$\begin{array}{rcll} n'[XY] & = & m''[XY] & \\ & = & m'[XY] & \text{(by Lemma 4.3.11)} \\ & = & n[XY] & \end{array}$$

Combining this with Lemma 4.3.7, we have $n'[U] = n[U]$, which is sufficient by Lemma 4.3.4. □

4.4.7 Lemma: Let $m \in M$, $M \models F$, and $L \subseteq M$. If $m \xRightarrow[L]{F} n$, then $m = n$.

Proof: We proceed by induction on the derivation of $m \xRightarrow[L]{F} n$.

- Case FC-EMPTY: Immediate.
- Case FC-STEP: Then, for some m' and G, we have $m \xrightarrow[L]{X \to Y} m'$ and $m' \xRightarrow[L]{G} n$, where $F = G, X \to Y$. By Lemma 4.3.12, $m = m'$, and, by the induction hypothesis $m' = n$. Hence $m = n$. □

Next, the revised record m' agrees with the relation L on the functional dependencies F:

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

Proof: Let $l \in L$ and $X \to Y \in F$. If $l[X] \neq m'[X]$, then $\{l, m'\} \models X \to Y$ is trivially true. So let's assume that $l[X] = m'[X]$. By Lemma 4.4.6, we know $m' \xrightarrow[L]{X \to Y} m'$. Since $m'[X] \in L[X]$, the rule FC-MATCH must have been used. Because of Lemma 4.3.3, we know that FC-MATCH may be instantiated with any record from L that agrees with m' on X. Instantiating it with l in particular, we then must have $m' = m' \leftarrow\!\!+ l[Y]$. Hence,

$$m'[Y] = (m' \leftarrow\!\!+ l[Y])[Y] = l[Y].$$

Thus, $\{l, m'\} \models X \to Y$. □

4.4.9 Lemma: If $m \xRightarrow[N]{F} m'$, then there exist functional dependencies $X_1 \to Y_1, \ldots, X_n \to Y_n$ and records $l_0, \ldots, l_n$ such that all of the following hold:

- $F = \{X_1 \to Y_1, \ldots, X_n \to Y_n\}$
- $Y_i \cap \mathit{names}(\{X_1 \to Y_1, \ldots, X_{i-1} \to Y_{i-1}\}) = \emptyset$
- $l_0 = m$ and $l_n = m'$
- $l_{i-1} \xrightarrow[N]{X_i \to Y_i} l_i$ for $i = 1, \ldots, n$
- $l_0[Y_i] = l_{i-1}[Y_i]$ and $l_i[Y_i] = l_n[Y_i]$

Proof: Simple induction on the derivation of $m \xRightarrow[N]{F} m'$. □

The crux of the previous lemma is the last two points, which can be explained as a statement that the fields of the left-hand side Y_i of each functional dependency get modified exactly once.

By inspecting the result of a revision operation with a single functional dependency and comparing it with the relation used in the operation, we may be able to learn some information about the original record. The following lemma describes this.

4.4.10 Lemma: If $l \xrightarrow[M]{X \to Y} l'$ and $l'[Y] \notin M[Y]$, then $l'[Y] = l[Y]$.

Proof: Proof is by inspection of the inference rules. We rule out the case where the derivation ends in C-MATCH by attempting to invert the inference rule and finding $l'[Y] \in M[Y]$. Therefore l and l' are related by C-NOMATCH, which trivially implies the desired result. □

4.4.11 Lemma: If $m \overset{F}{\underset{M}{\Longrightarrow}} n$, $X \to Y \in F$, F is tree-form and $n[X] \notin M[X]$ then $m[X] = n[X]$.

Proof: If $\nexists Z.Z \to X \in F$, then Lemma 4.4.5 shows $m[X] = n[X]$. However, if such a Z exists (because F is in tree form there is at most one), then by Lemma 4.4.9 there exist l and l' such that $m[X] = l[X]$ and $n[X] = l'[X]$ where $l \xrightarrow[M]{Z \to X} l'$. As $l'[X] \notin M[X]$, Lemma 4.4.10 gives $l[X] = l'[X]$. Therefore $m[X] = n[X]$. □

Also, any pair of records revised with the same relation and functional dependencies are guaranteed not to conflict with each other under the functional dependencies:

4.4.12 Lemma: Suppose $m_1 \overset{F}{\underset{L}{\Longrightarrow}} m_1'$ and $m_2 \overset{F}{\underset{L}{\Longrightarrow}} m_2'$, where F is in tree form. If $\{m_1, m_2\} \models F$, then $\{m_1', m_2'\} \models F$.

Proof: Let $X \to Y \in F$. If $m_1'[X] \neq m_2'[X]$, then we trivially have $\{m_1', m_2'\} \models X \to Y$. So let's assume $m_1'[X] = m_2'[X]$.

First, we consider the case where there exists $l \in L$ such that $l[X] = m_1'[X] = m_2'[X]$. Then Lemma 4.4.8 would give $\{l, m_1'\} \models F$, so from $l[X] = m_1'[X]$, we infer $m_1'[Y] = l[Y]$. Symmetrically, $m_2'[Y] = l[Y]$. Thus, we have $m_1'[Y] = m_2'[Y]$. Hence, $\{m_1', m_2'\} \models X \to Y$.

Now, we consider the case where there does not exist $l \in L$ such that $l[X] = m_1'[X] = m_2'[X]$. By Lemma 4.4.11, which gives $m_1'[X] = m_1[X]$ and $m_2'[X] = m_2[X]$, we conclude that $m_1[X] = m_2[X]$. From this and $\{m_1, m_2\} \models X \to Y$ we know $m_1[Y] = m_2[Y]$. By Lemma 4.4.9 we can pick records l_1 and l_1' such that $l_1[Y] = m_1[Y]$, $l_1'[Y] = m_1'[Y]$, and $l_1 \xrightarrow[L]{X \to Y} l_1'$. Because F is in tree form, we know that either $X \subseteq \mathit{right}(F)$ or else $X \cap \mathit{right}(F) = \emptyset$. If $X \cap \mathit{right}(F) = \emptyset$, then $l_1[X] = m_1'[X]$ (or, equivalently, $l_1[X] = m_1[X]$) by Lemmas 4.2.7 and 4.4.5. On the other hand, if $X \subseteq \mathit{right}(F)$, then we know there exists some functional dependency $Z \to X \in F$, and we may use Lemma 4.4.9 to show that $l_1[X] = m_1'[X]$ (or, equivalently, $l_1[X] = m_1[X]$). Hence, this gives $l_1[X] \notin L[X]$. From the definition of single record revision, $l_1'[Y] = l_1[Y]$. Therefore $m_1'[Y] = m_1[Y]$. Symmetrically, $m_2'[Y] = m_2[Y]$, so $\{m_1', m_2'\} \models X \to Y$. □

4.4.13 Lemma: If $M, m \models F$ and $m \overset{F}{\underset{M}{\Longrightarrow}} m'$, then $m = m'$.

Proof: Simple induction on the derivation of $m \overset{F}{\underset{M}{\Longrightarrow}} m'$ using Lemma 4.3.13. □

4.5 Relation Revision

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

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

Relation revision preserves the domain of the relation.

4.5.1 Lemma: If $M : U$, then $M \leftarrow_F L : U$.

Proof: Immediate from Lemma 4.4.2. □

We now record several key properties of relation revision. First, it does not make up new values.

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

Proof: Consider some $m' \in M \leftarrow_F L$. Then there exists $m \in M$ such that $m \xRightarrow[L]{F} m'$. By Lemma 4.4.3, either $m'(A) = m(A)$, in which case $m'[A] \in M[A]$, or else $m'[A] \in L[A]$. □

Next, relation revision does not alter non-outputs.

4.5.3 Lemma: If $Z \cap \mathit{outputs}(F) = \emptyset$, then $(M \leftarrow_F L)[Z] \subseteq M[Z]$.

Proof: Let $n \in (M \leftarrow_F L)[Z]$. Then there is some $m \in M$ such that $m \xRightarrow[L]{F} m'$ and $m'[Z] = n$. By Lemma 4.4.5, $n = m[Z]$. So $n \in M[Z]$. □

Moreover, if P ignores the outputs of some set F of functional dependencies, then the property of a relation M satisfying P is preserved when M is revised with respect to F.

4.5.4 Lemma: If $M \subseteq P$ and P ignores $\mathit{outputs}(F)$, then $M \leftarrow_F L \subseteq P$.

Proof: Let $m' \in M \leftarrow_F L$. Then there is some $m \in M$ such that $m \xRightarrow[L]{F} m'$. By Lemma 4.4.5,

$$m[\mathit{dom}(m) - \mathit{outputs}(F)] = m'[\mathit{dom}(m) - \mathit{outputs}(F)].$$

Since $m \in P$, we have $m' \in P$ by the definition of "ignores." □

Most significantly, relation revision results in a relation that satisfies F if both of the relations involved in the operation do.

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

Proof: Let $m'_1, m'_2 \in M \leftarrow_F L$. By the definition of relation revision, there exist $m_1, m_2 \in M$ such that $m_1 \xRightarrow[L]{F} m'_1$ and $m_2 \xRightarrow[L]{F} m'_2$. By Fact 2.4, $\{m_1, m_2\} \models F$. So, by Lemma 4.4.12, $\{m'_1, m'_2\} \models F$. Since m'_1 and m'_2 were chosen arbitrarily, we conclude $M \leftarrow_F L \models F$ using Fact 2.4 in the other direction. □

4.5.6 Lemma: Let $M : U$ and $N : U$. If $M \models F$, $L \models F$, and $N \subseteq L$, then $(M \leftarrow_F L) \cup N \models F$.

Proof: Consider any pair of records $l_1, l_2 \in (M \leftarrow_F L) \cup N$. If $l_1, l_2 \in M \leftarrow_F L$, then $\{l_1, l_2\} \models F$ by Lemma 4.5.5 (and Fact 2.4). If $l_1, l_2 \in N$, then $l_1, l_2 \models F$ since $N \models F$ by assumption. Finally, let us assume, without loss of generality, that $l_1 = m' \in M \leftarrow_F L$ and $l_2 = n \in N$. Then $\{m', n\} \models F$ by Lemma 4.4.8. Since l_1 and l_2 were chosen arbitrarily, $(M \leftarrow_F N) \cup N \models F$ by Fact 2.4. □

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*. Suppose $M : U$ and $N : U$. Then we define

$$M \overset{\cup}{\leftarrow}_F N = (M \leftarrow_F N) \cup N.$$

The basic properties of relation revision also hold for relational merge:

4.5.7 Lemma: Let $M : U$ and $N : U$. If $M \models F$ and $N \models F$, then $M \overset{\cup}{\leftarrow}_F N \models F$.

Proof: Corollary of Lemma 4.5.6. □

4.5.8 Lemma: If $M \models F$ and $M' \subseteq M$ then $M' \models F$

Proof: For a contradiction assume $M' \not\models F$. Then by the definition of $\models$, $\exists m'_1, m'_2 \in M', X \to Y \in F. m'_1[X] = m'_2[X]$ and $m'_1[Y] \neq m'_2[Y]$. But $M' \subseteq M$ gives $m'_1, m'_2 \in M$, hence $M \not\models F$. Contradiction. □

The following technical result is necessary to define join in terms of merge.

4.5.9 Lemma: If $M \models F$, $N \models G$, F is in tree form, and $N \subseteq M$, then $M \overset{\cup}{\leftarrow}_F N = M$.

Proof: We must show two directions of set containment. First we demonstrate $M \overset{\cup}{\leftarrow}_F N \subseteq M$. Pick some $m' \in M \overset{\cup}{\leftarrow}_F N$. If $m' \in N$, then (using assumption $N \subseteq M$) we find $m' \in M$. Next, consider the case where $m' \notin N$. We know $m' \in M \leftarrow_R N$, and, from the definition of relation revision, we can pick some $m \in M$ such that $m \overset{F}{\underset{N}{\Longrightarrow}} m'$. As $N \cup \{m\} \subseteq M$ we know $N, m \models F$. From Lemma 4.4.13 we find $m' = m$ which gives $m' \in M$.

Now we must establish $M \subseteq M \overset{\cup}{\leftarrow}_F N$. Pick $m \in M$. By Lemma 4.4.4, there exists $m' \in M \overset{\cup}{\leftarrow}_F N$ such that $m \overset{F}{\underset{N}{\Longrightarrow}} m'$. As $N \subseteq M$ we have $N, m' \models F$ and, by Lemma 4.4.13, $m' = m$. Thus $m \in M \overset{\cup}{\leftarrow}_F N$. □

5 Relational Lens Primitives

We now proceed to describing our primitive lenses for updatable relational views.

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:

$$\begin{array}{rcl} v\nearrow(I) & = & I\backslash_R[S \mapsto P \cap I(R)] \\ v\searrow(J, I) & = & J\backslash_S[R \mapsto M_1 \setminus N_\#] \\ \text{where} & & \\ M_1 & = & (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F J(S) \\ N_\# & = & (P \cap M_1) \setminus J(S) \\ F & = & fd(R) \end{array}$$

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_1 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_1 ends up with records that, if put in the concrete database, would result in a violation of PutGet, but that may safely be removed from the concrete database. (Such a case will be illustrated later in this section.) We collect these records in $N_\#$ and remove them from the result.

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

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

This typing rule should be read as a *theorem* that describes a set of database schemas and view schemas over which the lens is well-behaved.

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

Here is a typical example of the use of the `select` lens. Let v stand for the expression

$$\texttt{select from } R \texttt{ where } C = c_2 \texttt{ as } S$$

and assume that our *sort* function has the following assignments for R and S:

$$\begin{array}{rcl} sort(R) & = & (ABC, \top, \{A \to B\}) \\ sort(S) & = & (ABC, C = c_2, \{A \to B\}) \end{array}$$

We may instantiate the rule T-Select to check that $v \in \{R\} \Leftrightarrow \{S\}$. Let us apply the lens in the *get* direction to a database I containing a single table R:

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

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

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

In the table R, the record (a_1, b_1, c_1) has been replaced with the record (a_1, b_2, c_1) in order to satisfy 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 sort function: Assume that there is an ordering on elements, such that $b_i \leq c_i$ exactly when $i \leq j$, and that we have the following assignments:

$$\begin{array}{rcl} sort(R) & = & (U, B \leq C, A \to B) \\ sort(S) & = & (U, B \leq C \wedge C = c_2, A \to B) \end{array}$$

Then we would have $I \models \{R\}$, $J \models \{S\}$, and $J' \models \{S\}$, but $I' \not\models \{R\}$, which violates our fundamental principle 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, we consider an example where a rather unusual behavior can occur. Given the sort function

$$\begin{aligned} sort(R) &= (U, \top, A \to B) \\ sort(S) &= (U, B = b_2, A \to B) \end{aligned}$$

consider the lens

$$\texttt{select from } R \texttt{ where } B = b_2 \texttt{ as } S$$

applied to the database I:

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \end{array} \right\}^I \xrightarrow{\not\nearrow(I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & & & \end{array} \right\}^J$$

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:

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_2 & c_1 \\ & a_1 & b_2 & c_2 \end{array} \right\}^{I'} \xleftarrow{\not\searrow(J',I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{J'}$$

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 row present in the concrete database.

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{I'} \xleftarrow{\not\searrow(J',I)} \left\{ \begin{array}{c|ccc} S & A & B & C \\ \hline & a_1 & b_2 & c_2 \end{array} \right\}^{J'}$$

In general, it is safe, but somewhat counter-intuitive, to delete records from the concrete database that result in conflicting values determined by functional dependencies. Although somewhat counter-intuitive, this behavior is consistent with the GetPut and PutGet laws.

Let us now check that the lens we have defined is indeed well behaved.

5.1.1 Theorem: Suppose

$sort(R) = (U, Q, F)$	$sort(S) = (U, P \cap Q, F)$
F is in tree form	Q ignores $outputs(F)$
$\Delta = \Sigma \uplus \{R\}$	$\Delta' = \Sigma \uplus \{S\}$
$v = \texttt{select from } R \texttt{ where } P \texttt{ as } S$	

Then $v \in \Delta \Leftrightarrow \Delta'$.

Proof: We must show the following statements:

$$\begin{array}{ll} v\nearrow \in \Delta \to \Delta' & \\ v\searrow \in \Delta' \times \Delta \to \Delta & \\ v\searrow (v\nearrow (I), I) = I & \text{for all } I \in \Delta \\ v\nearrow (v\searrow (J, I)) = J & \text{for all } (J, I) \in \Delta' \times \Delta \end{array}$$

We first show that $v\nearrow \in \Delta \to \Delta'$. Suppose $I \models \Delta$. From the assumptions, we know $I\backslash_R \models \Sigma$ and $I(R)$ satisfies (U, Q, F). From the definition of $v\nearrow$, it is easy to check that $S \in dom(v\nearrow (I))$ and $(v\nearrow (I))\backslash_S \models \Sigma$. It remains to show that $(v\nearrow (I))(S)$ satisfies $(U, P \cap Q, F)$, where $(v\nearrow (I))(S) = P \cap I(R)$. This involves checking three facts:

1. $P \cap I(R) : U$. This follows from $P : U$ and $I(R) : U$.

2. $P \cap I(R) \subseteq P \cap Q$. This follows from the assumption that $I(R) \subseteq Q$.

3. $P \cap I(R) \models F$. Immediate.

Hence, we conclude that $v\nearrow(I) \models \Delta'$, as required.

Next, we show that $v\searrow \in \Delta' \times \Delta \to \Delta$. Suppose $I \models \Delta$ and $J \models \Delta'$. From the assumptions we know $I\backslash_R \models \Sigma$ and $I(R)$ satisfies (U, Q, F), and similarly $J\backslash_S \models \Sigma$ and $J(S)$ satisfies $(U, P \cap Q, F)$. From the definition of $v\searrow$, it is easy to check that $R \in dom(v\searrow(J, I))$ and $(v\searrow(J, I))\backslash_R \models \Sigma$. It remains to show that $(v\searrow(J, I))(R)$ satisfies (U, Q, F), where

$$\begin{aligned}(v\searrow(J,I))(R) &= M_1 \setminus N_\# \\ M_1 &= (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F J(S) \\ N_\# &= (P \cap M_1) \setminus J(S).\end{aligned}$$

Again, this involves checking three facts:

1. $M_1 \setminus N_\# : U$. This follows from $P : U$, $I(R) : U$, $J(S) : U$, and Lemma 4.5.1.

2. $M_1 \setminus N_\# \subseteq Q$. Since $I(R) \subseteq Q$ and $J(S) \subseteq Q$, we have $M_1 \subseteq Q$ by Lemma 4.5.4, and the fact follows immediately.

3. $M_1 \setminus N_\# \models F$. Since $I(R) \models F$ and $J(S) \models F$, we have $M_1 \models F$ by Lemma 4.5.7, and the fact again follows immediately.

Next, we show that v satisfies the law GetPut—that is, $v\searrow(v\nearrow(I), I) = I$ for all $I \in \Delta$. It is easy to check that $dom(v\searrow(v\nearrow(I), I)) = dom(I)$ and that $(v\searrow(v\nearrow(I), I))\backslash_R = I\backslash_R$. It remains to show that $(v\searrow(v\nearrow(I), I))(R) = I(R)$. Expanding the definitions of $v\nearrow$ and $v\searrow$, we see that we must show $M_1 \setminus N_\# = I(R)$ where

$$\begin{aligned}M_1 &= (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F (P \cap I(R)) \\ N_\# &= (P \cap M_1) \setminus (P \cap I(R)).\end{aligned}$$

We first check that $M_1 \setminus N_\# \subseteq I(R)$. Of course, it suffices to show that $M_1 \subseteq I(R)$. Consider a record $n \in (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F (P \cap I(R))$. Then, from the definition of $\overset{\cup}{\leftarrow}_F$, either $n \in P \cap I(R)$ (in which case we immediately know that $n \in I(R)$) or there is some $m \in \neg P \cap I(R)$ such that $m \xRightarrow[P \cap I(R)]{F} n$. If such an m exists, then $m = n$ by Lemma 4.4.7, since $m \in I(R)$ and $I(R) \models F$, which also gives us $n \in I(R)$.

Now we check that $I(R) \subseteq M_1 \setminus N_\#$. Consider a record $m \in I(R)$. First assume that $m \in P$. Then $m \in P \cap I(R)$, so $m \notin N_\#$. Furthermore, $m \in (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F (P \cap I(R))$ by the definition of $\overset{\cup}{\leftarrow}_F$. Thus, $m \in M_1 \setminus N_\#$. On the other hand, suppose $m \in \neg P$. We see that $m \notin N_\#$ because $m \notin P \cap M_1$. Now it remains to show that $m \in M_1$. We may use Lemmas 4.4.4 and 4.4.7 to show that $m \xRightarrow[P \cap I(R)]{F} m$ since $m \in \neg P \cap I(R)$ and $I(R) \models F$. Hence, $m \in (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F (P \cap I(R))$, and therefore $m \in M_1 \setminus N_\#$.

Finally, we show that v satisfies the law PutGet—that is, $v\nearrow(v\searrow(J, I)) = J$ for all $I \in \Delta$ and $J \in \Delta'$. It is easy to check that $dom(v\nearrow(v\searrow(J, I))) = dom(J)$ and that $(v\nearrow(v\searrow(J, I)))\backslash_S = J\backslash_S$. It remains to show that $(v\nearrow(v\searrow(J, I)))(S) = J(S)$. Expanding the definitions of $v\nearrow$ and $v\searrow$, we see that we must show $P \cap (M_1 \setminus N_\#) = J(S)$ where

$$\begin{aligned}M_1 &= (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F J(S) \\ N_\# &= (P \cap M_1) \setminus J(S).\end{aligned}$$

We first check that $P \cap (M_1 \setminus N_\#) \subseteq J(S)$. Suppose $m \in P \cap (M_1 \setminus N_\#)$—that is, $m \in P$, $m \in M_1$, and $m \notin N_\#$. But then it must be that $m \in J(S)$, by the definition of $N_\#$.

Now we check that $J(S) \subseteq P \cap (M_1 \setminus N_\#) \subseteq P$. We know by assumption that $J(S) \subseteq P \cap Q$. Also, $J(S) \subseteq M_1$ by the definition of the relational merge. Furthermore, we know that $J(S) \cap N_\# = \emptyset$ by the definition of $N_\#$. Hence, $J(S) \subseteq P \cap (M_1 \setminus N_\#)$, completing the proof. □

5.2 A Simple 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 begin in this section with a concrete lens illustrating one particular possible choice of update policy, and then in Section 5.3 show how this lens and several others can be obtained as instances of a more general scheme. In the *get* direction, the lens

$$\texttt{join\_dl}\ R,\ S\ \texttt{as}\ T$$

performs a natural join. In the *putback* direction `join_dl` may add records to both tables R and S, but may 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`:

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

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

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

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, note that the record (a_2, b_2, c_3) from 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:

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

where

$$\begin{aligned} (U,P,F) &= sort(R) \\ (V,Q,G) &= sort(S) \\ M_0 &= I(R) \overset{\cup}{\leftarrow}_F J(T)[U] \\ N &= I(S) \overset{\cup}{\leftarrow}_G J(T)[V] \\ L &= (M_0 \bowtie N)\ \setminus J(T) \\ M &= M_0 \setminus L[U] \end{aligned}$$

The *get* function just computes table T in the result from the join of R and S in the input; 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

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

satisfies neither GetPut nor PutGet. This definition fails because (i) records which 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, anomalous behavior may still occur. 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 this 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 not expressed in the initial view state.

Our actual definition avoids this problem, 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. Note that update anomalies and user deletions are indistinguishable in this calculation, and L handles both.

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

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

This reasonable result illustrates that `join_dl` is well-behaved.

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

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

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. This is necessary because relations such as

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

cannot be decomposed into relations over AB and BC. As desired, the typing rule prevents $J(T)$ from having this form when $U = AB$ and $V = BC$. Imposing the key constraint on the left table would work too; picking the right table was 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 merge to be well defined.

5.3 A Parameterized Join

A great variety of natural join lenses, including `join_dl` and many others, can be derived from a single generic lens, called `join_template`, which is parameterized by an operation $\overset{\cup?}{\leftarrow}$ and a boolean function[1]

[1] Φ is a *predicate* in the mathematical sense, but we use the term "boolean function" to avoid confusion with predicates over records.

Φ. The operation $\overset{\cup?}{\leftarrow}$ takes two relations over the same domain and is used in the *putback* direction to update the records in a concrete relation using an abstract relation and a set of functional dependencies. The accompanying boolean function Φ takes a set of field names, a predicate over those field names, and a set of functional dependencies over those field names as arguments; it is used in the typing rule to check that a particular use of `join_template` meets the constraints necessary to guarantee that $\overset{\cup?}{\leftarrow}$ behaves sensibly. For example, `join_dl` is an instantiation of `join_template` with

$$\begin{array}{rcl} \overset{\cup?}{\leftarrow} & = & \overset{\cup}{\leftarrow} \\ \Phi(U,P,F) & = & (F \text{ is tree-form}) \text{ and } (P \text{ ignores } \mathit{outputs}(F)). \end{array}$$

Formally we write an instance of the join template lens as $\texttt{join\_template}_{\overset{\cup?}{\leftarrow},\Phi}\ (R,P_d)\ (S,Q_d)\ \texttt{as}\ T$, explicitly showing the parameterization by $\overset{\cup?}{\leftarrow}$ and Φ, as well as on R and S, the relations to be joined, T, the result of the join, and P_d and Q_d, predicates that control the treatment of "ambiguous deletions," as described below. Most of the time, however, we will leave $\overset{\cup?}{\leftarrow}$ and Φ implicit and write just $\texttt{join\_template}\ (R,P_d)\ (S,Q_d)\ \texttt{as}\ T$.

Naturally, we cannot expect to obtain a well-behaved lens if we instantiate $\overset{\cup?}{\leftarrow}$ with a completely arbitrary update operator: we need to impose some constraints. Accordingly, we say that $\overset{\cup?}{\leftarrow}$ and Φ are *suitable at* U if, for all $M:U$, $N:U$, $F:U$ and $P:U$ such that $M \models F$, $N \models F$, and $\Phi(U,P,F)$, we have:

$$N \subseteq M \overset{\cup?}{\leftarrow}_F N \tag{1}$$

$$N \subseteq M \implies M \overset{\cup?}{\leftarrow}_F N = M \tag{2}$$

$$M \overset{\cup?}{\leftarrow}_F N : U \tag{3}$$

$$M \subseteq P \wedge N \subseteq P \implies M \overset{\cup?}{\leftarrow}_F N \subseteq P \tag{4}$$

$$M \overset{\cup?}{\leftarrow}_F N \models F \tag{5}$$

The first property ensures that, when we use the update operation to add some new records N to some existing ones M, all of the new ones actually make it into the result. The second limits the "aggressiveness" of the update operation: if there is no new information in N, then the update does nothing. The last three properties ensure that if M and N satisfy (V,Q,G) then $M \overset{\cup?}{\leftarrow}_F N$ does too.

In Section 5.4, we show that the record merge operator is suitable; we also define a variant called "squash" and prove it suitable as well. But for a simpler example, which also illustrates the role of Φ, consider ordinary set union. It is easy to see that $\cup$ satisfies the first four properties, but to make it satisfy 5 we must pick a restrictive Φ, such as

$$\Phi(U,P,F) = (F = \emptyset).$$

Given a suitable $\overset{\cup?}{\leftarrow}$ and Φ, the behavior of `join_template` is defined as

$$v = \texttt{join\_template}\ (R,P_d)\ (S,Q_d)\ \texttt{as}\ T$$

$$v\nearrow(I) = I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)]$$

$$v\searrow(J,I) = J\backslash_{T}[R \mapsto M][S \mapsto N]$$

where

$$\begin{aligned}
(U, P, F) &= sort(R) \\
(V, Q, G) &= sort(S) \\
M_0 &= I(R) \overset{\cup?}{\leftarrow}_F J(T)[U] \\
N_0 &= I(S) \overset{\cup?}{\leftarrow}_G J(T)[V] \\
L &= M_0 \bowtie N_0 \setminus J(T) \\
L_l &= L \bowtie (J(T)[U \cap V]) \\
L_a &= L \setminus L_l \\
M &= (M_0 \setminus (L_a \cap P_d)[U]) \setminus L_l[U] \\
N &= N_0 \setminus (L_a \cap Q_d)[V]
\end{aligned}$$

While the *get* direction is identical to `join_dl`'s, the *putback* has changed. As before, M_0 corresponds to a naive *putback* and L is a set of records corresponding to deletions and to anomalies in the naive update policy. A reasonable *putback* must ensure that an immediate *get* will not contain any $l \in L$. Where `join_dl` is defined such that deletions from the right table, M_0, handle L, `join_template` allows deletions on either side. Here N is defined by the naive right table N_0 less such deletions.

The two new arguments to the `join_template` lens—P_d and Q_d—specify a policy for ambiguous deletions. However not all deletions are ambiguous; sometimes well-behavedness forces a deletion to occur in left table alone. The sets L_l and L_a are a partition of L where L_l represents deletions which must be taken from the left table, M_0, and L_a represents ambiguous cases. This asymmetry stems from the different requirements (discussed in section 5.2) imposed on functional dependency sets F and G by the typing rule for the join template given below—while G is unconstrained, G will imply that $U \cap V$ is a key for S. The following example shows how these relations interact.

Consider the following database instances:

$$\left\{ \begin{array}{c|cc} R & A & B \\ \hline & a_1 & b_1 \\ & a_1' & b_1 \\ & a_2 & b_2 \end{array} \quad \begin{array}{c|c} S & B \\ \hline & b_1 \\ & b_2 \end{array} \right\}^I \qquad \left\{ \begin{array}{c|cc} T & A & B \\ \hline & a_1 & b_1 \end{array} \right\}^J$$

We can calculate:

$$L = \begin{array}{cc} A & B \\ \hline a_1' & b_1 \\ a_2 & b_2 \end{array} \qquad L_l = \begin{array}{cc} A & B \\ \hline a_1' & b_1 \end{array} \qquad L_a = \begin{array}{cc} A & B \\ \hline a_2 & b_2 \end{array}$$

Record $(a_1', b_1) \in L_l$ captures the intuition that $(a_1', b_1)[B] = (b_1)$ cannot be removed from the right hand table because this would prevent (a_1, b_1) from appearing after a *get*. In contrast $(a_2, b_2) \in L_a$ because we can delete from either R or S and maintain well-behavedness. The predicates P_d and Q_d determine the database instance I which results from a *putback*:

$$(a_2, b_2) \in P_d \wedge (a_2, b_2) \notin Q_d \quad \text{yields} \quad \left\{ \begin{array}{c|cc} R & A & B \\ \hline & a_1 & b_1 \end{array} \quad \begin{array}{c|c} S & B \\ \hline & b_1 \\ & b_2 \end{array} \right\}^I$$

$$(a_2, b_2) \notin P_d \wedge (a_2, b_2) \in Q_d \quad \text{yields} \quad \left\{ \begin{array}{c|cc} R & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \end{array} \quad \begin{array}{c|c} S & B \\ \hline & b_1 \end{array} \right\}^I$$

$$(a_2, b_2) \in P_d \wedge (a_2, b_2) \in Q_d \quad \text{yields} \quad \left\{ \begin{array}{c|cc} R & A & B \\ \hline & a_1 & b_1 \end{array} \quad \begin{array}{c|c} S & B \\ \hline & b_1 \end{array} \right\}^I$$

$$(a_2, b_2) \notin P_d \wedge (a_2, b_2) \notin Q_d \quad \text{yields} \quad \left\{ \begin{array}{c|cc} R & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \end{array} \quad \begin{array}{c|c} S & B \\ \hline & b_1 \\ & b_2 \end{array} \right\}^I$$

The astute reader will notice that PutGet will fail on the last example where $(a_2, b_2) \notin P_d \wedge (a_2, b_2) \notin Q_d$. This motivates one of the premises to the typing rule given below, namely, $P_d \cup Q_d = \top_{UV}$.

The typing rule for `join_template` defined below:

$$\frac{\begin{array}{ccccc} sort(R) = (U,P,F) & sort(S) = (V,Q,G) & sort(T) = (UV, P \bowtie Q, F \cup G) \\ G \models U \cap V \to V & P_d \cup Q_d = \top_{UV} & \Phi(U,P,F) & \Phi(V,Q,G) \end{array}}{\texttt{join\_template}\ (R,P_d)\ (S,Q_d)\ \texttt{as}\ T \in \Sigma \uplus \{R,S\} \Leftrightarrow \Sigma \uplus \{T\}}$$

This varies in two interesting ways from `join_dl`. We discussed one of these above. Additionally, the premises $\Phi(U,P,F)$ and $\Phi(V,Q,G)$ ensure the lens will only be applied to databases instances which can be properly handled by the update operator. As with select, we must show that these premises imply `join_template` (R,P_d) (S,Q_d) `as` $T \in \Sigma \uplus \{R,S\} \Leftrightarrow \Sigma \uplus \{T\}$.

5.3.1 Theorem: If $\overset{\cup?}{\leftarrow}$ and Φ are suitable at U and V and

$$\begin{array}{l} sort(R) = (U,P,F) \\ sort(S) = (V,Q,G) \\ sort(T) = (UV, P \bowtie Q, F \cup G) \\ G \models U \cap V \to V \\ P_d \cup Q_d = \top_{UV} \\ \Phi(U,P,F) \\ \Phi(V,Q,G) \\ \Delta = \Sigma \uplus \{R,S\} \\ \Delta' = \Sigma \uplus \{T\} \\ v = \texttt{join\_template}\ (R,P_d)\ (S,Q_d)\ \texttt{as}\ T, \end{array}$$

then

$$v \in \Delta \Leftrightarrow \Delta'$$

Proof: By the definition of lenses, there are four properties to be established. We consider them in turn.

Subclaim Get Total**:** If $I \models \Delta$ then $v\nearrow(I) \models \Delta'$.

The definition of v gives

$$v\nearrow(I) = I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)].$$

As $I \models \Delta$, we know $I\backslash_{R,S} \models \Sigma$. Hence to prove $v\nearrow(I) \models \Delta'$, we need to show $I(R) \bowtie I(S)$ satisfies $sort(T)$. From the form of $sort(T)$ we see that it suffices to show the following:

- $I(R) \bowtie I(S) : UV$. Trivial.
- $I(R) \bowtie I(J) \subseteq P \bowtie Q$. Follows from Fact 2.1.
- $I(R) \bowtie I(S) \models F \cup G$. Assume, for a contradiction, that we can pick $X \to Y \in F \cup G$ and $l_1, l_2 \in I(R) \bowtie I(S)$ such that $l_1[X] = l_2[X]$ but $l_1[Y] \neq l_2[Y]$. By the definition of union, either $X \to Y \in F$ or $X \to Y \in G$. Without loss of generality, assume the former. From the definition of $\bowtie$ we find $l_1[U], l_2[U] \in I(R)$. I's schema gives $I(R) \models F$, thus, by Lemma 4.5.8, $l_1[U], l_2[U] \models F$. Consequently (as $l_1[X] = l_2[X]$) we find $l_1[Y] = l_2[Y]$. This contradicts $l_1[Y] \neq l_2[Y]$; therefore $I(R) \bowtie I(S) \models F \cup G$.

This proves Get Total.

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

Unrolling `join_template`'s definition we obtain

$$v\searrow(J,I) = J\backslash_T[R \mapsto M][S \mapsto N]$$

where

$$M \subseteq M_0 = I(R) \overset{\cup?}{\leftarrow}_F J(T)[U]$$

$$N \subseteq N_0 = I(S) \overset{\cup?}{\leftarrow}_G J(T)[V].$$

By the definitions of Δ' and database instances, $J\backslash_T \models \Sigma$. Therefore, to show $v\searrow (J, I) \models \Delta$, we must demonstrate that M satisfies $sort(R)$ and N satisfies $sort(S)$. Thus, it suffices to show the following:

- $M : U$. Straightforward.
- $M \subseteq P$. The definition of I and J gives $I(R) \subseteq P$ and $J(T)[U] \subseteq P$. Property 4 of $\overset{\cup?}{\leftarrow}$ gives $I(R) \overset{\cup?}{\leftarrow}_F J(T)[U] \subseteq P$, from which the result follows since $M \subseteq M_0 \subseteq I(R) \overset{\cup?}{\leftarrow}_F J(T)[U]$.
- $M \models F$. By property 5, we find $M_0 \models F$. Lemma 4.5.8 gives $M \models F$.
- $N : V$, $N \subseteq Q$, and $N \models Q$. These statements can be proved symmetrically.

This proves Put Total.

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

Unfolding the definition of `join_template` gives

$$v\searrow (v\nearrow (I), I) = J\backslash_T[R \mapsto M][S \mapsto N]$$

where

$$\begin{aligned}
J &= I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)] \\
M_0 &= I(R) \overset{\cup?}{\leftarrow}_F J(T)[U] \\
N_0 &= I(S) \overset{\cup?}{\leftarrow}_G J(T)[V] \\
L &= M_0 \bowtie N_0 \setminus J(T) \\
L_l &= L \bowtie (J(T)[U \cap V]) \\
L_a &= L \setminus L_l \\
M &= (M_0 \setminus (L_a \cap P_d)[U]) \setminus L_l[U] \\
N &= N_0 \setminus (L_a \cap Q_d)[V].
\end{aligned}$$

By the definition of $\bowtie$ (Fact 2.2, we have $J(T)[U] \subseteq I(R)$. Applying property 2 of $\overset{\cup?}{\leftarrow}$ gives $M_0 = I(R)$. Similarly, $N_0 = I(S)$. Thus we can unroll the definition of L as follows:

$$L = M_0 \bowtie N_0 \setminus J(T) = I(R) \bowtie I(S) \setminus I(R) \bowtie I(S) = \emptyset$$

Therefore

$$\begin{aligned}
L_l &= L_a = \emptyset \\
M &= M_0 = I(R) \\
N &= N_0 = I(S).
\end{aligned}$$

We now calculate as follows:

$$\begin{aligned}
v\searrow (v\nearrow (I), I) &= J\backslash_T[R \mapsto M][S \mapsto N] \\
&= (I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)])\backslash_T[R \mapsto M][S \mapsto N] \\
&= I\backslash_{R,S,T}[R \mapsto M][S \mapsto N] \\
&= I\backslash_{R,S}[R \mapsto M][S \mapsto N] \qquad (\text{as } I \models \Sigma \uplus \{R, S\} \text{ and } T \notin \Sigma \uplus \{R, S\}) \\
&= I\backslash_{R,S}[R \mapsto I(R)][S \mapsto I(S)] \\
&= I
\end{aligned}$$

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

With the definition of `join_template`, we calculate

$$\begin{aligned} v\nearrow(v\searrow(J,I)) &= v\nearrow(J\backslash_T[R \mapsto M][S \mapsto N]) \\ &= J\backslash_{R,S,T}[T \mapsto M \bowtie N] \\ &= J\backslash_T[T \mapsto M \bowtie N] \qquad (\text{as } J\backslash_T \models \Sigma \text{ and } R, S \notin \Sigma) \end{aligned}$$

where

$$\begin{aligned} M_0 &= I(R) \overset{\cup?}{\leftarrow}_F J(T)[U] \\ N_0 &= I(S) \overset{\cup?}{\leftarrow}_G J(T)[V] \\ L &= M_0 \bowtie N_0 \setminus J(T) \\ L_l &= L \bowtie (J(T)[U \cap V]) \\ L_a &= L \setminus L_l \\ M &= (M_0 \setminus (L_a \cap P_d)[U]) \setminus L_l[U] \\ N &= N_0 \setminus (L_a \cap Q_d)[V]. \end{aligned}$$

To conclude, we must show $M \bowtie N = J(T)$. We do this by showing that each is contained in the other.

- $J(T) \subseteq M \bowtie N$. Pick $l \in J(T)$. We want to show $l[U] \in M$ and $l[V] \in N$.

  Applying property 1, we see $J(T)[U] \subseteq M_0$ and $J(T)[V] \subseteq N_0$; this gives $l[U] \in M_0$ and $l[V] \in N_0$. Therefore it suffices to show that $l[U] \notin L[U]$ (from which it immediately follows that $l[U] \in M$ by the definition of M) and $l[V] \notin L_a[V]$ (from which it immediately follows that $l[V] \in N$ by the definition of N).

  To show $l[U] \notin L[U]$, let us assume, for a contradiction, that there is some $l' \in L$ such that $l'[U] = l[U]$. Clearly, $l'[U \cap V] = l[U \cap V]$. From $l' \in L$ we know that $l' \in M_0 \bowtie N_0$ and, consequently, $l'[V] \in N_0$. Likewise $l[V] \in N_0$, because $l \in M_0 \bowtie N_0$.

  Now, by assumption, we know $I(S) \models G$ and $J(T)[V] \models G$. Therefore, property 5 of $\overset{\cup?}{\leftarrow}$ gives $N_0 \models G$. From this, Lemma 4.5.8 gives $l[V], l'[V] \models G$. The hypothesis also tells us $G \models (U \cap V) \to V$. Therefore $l[V], l'[V] \models (U \cap V) \to V$. By assumption, we have $l'[U \cap V] = l[U \cap V]$, which, with this functional dependency, implies $l'[V] = l[V]$. The foregoing—coupled with the assumption that $l'[U] = l[U]$—gives $l' = l$ and so $l' \in J(T)$. But L and $J(T)$ are disjoint, so $l' \in J(T)$ contradicts the assumption $l' \in L$ and proves $l[U] \notin L[U]$.

  To show $l[V] \notin L_a[V]$, let us assume the opposite (for a contradiction), and pick some $l' \in L_a$ such that $l'[V] = l[V]$. From $l \in J(T)$ we know $l[U \cup V] \in J(T)[UV]$. As $l'[U \cap V] = l[U \cap V]$, we find $l'[U \cap V] \in J(T)[U \cap V]$. We also know $l' \in L$ because $L_a \subseteq L$; thus, $l' \in L \bowtie J(T)[U \cap V]$—that is, $l' \in L_l$. But this is a contradiction: we assumed $l' \in L_a$, and L_a and L_l are disjoint.

- $M \bowtie N \subseteq J(T)$. Pick $l \in M \bowtie N$ and assume, for a contradiction, that $l \notin J(T)$. Then $l[U] \in M$ and $l[V] \in N$, which imply $l[U] \in M_0$ and $l[V] \in N_0$. From these facts, we can see that $l \in M_0 \bowtie N_0 \backslash J(T)$—i.e., $l \in L$.

  We continue by examining the form of L. The definitions of L_l and L_a give $L = L_l \cup L_a$. We assumed $P_d \cup Q_d = \top_{UV}$, so we can rewrite L as follows:

$$\begin{aligned} L &= L_l \cup L_a \\ &= L_l \cup (L_a \cap (P_d \cup Q_d)) \\ &= (L_l \cup (L_a \cap P_d)) \cup (L_a \cap Q_d) \end{aligned}$$

  Thus, we can have $l \in L$ in two (non-exclusive) ways:

– $l \in (L_l \cup (L_a \cap P_d))$. This is a contradiction because $l[U] \in M$ and M is disjoint from $L_l \cup (L_a \cap P_d)[U]$.
– $l \in (L_a \cap Q_d)$. This is a contradiction because $l[V] \in N$ and N is disjoint from $L_a \cap Q_d[V]$.

This proves PutGet. □

5.4 Derived Joins

Using the join template, we can now define a variety of concrete join lenses with different update policies and different restrictions on the situations in which they can be applied. We begin with some using relation merge as the update operation, then introduce a simpler update operation called *squash* that yields a more draconian update policy but that can be used in a wider variety of situations (because its accompanying boolean function Φ is always true).

The first set of derived natural joins differ in their treatment of ambiguous deletions. Consider a simple database with two relations R and S which are joined to create relation T:

$$R \mapsto \begin{array}{cc} A & B \\ \hline a & b \end{array} \qquad S \mapsto \begin{array}{cc} B & C \\ \hline b & c \end{array} \qquad T \mapsto \begin{array}{ccc} A & B & C \\ \hline a & b & c \end{array}$$

If the view state is modified to $T = \emptyset$, then there are three reasonable ways to update the database (plus some unreasonable ones, which add irrelevant records to either R or S). We can delete from R—i.e., take $R = \emptyset$—or delete from S, or both.[2] These choices can be implemented by passing different deletion predicates to the join template.

Let $\Phi_m(U, P, F) = F$ is tree form and P ignores $outputs(F)$. The three merge-based variants of natural join are defined as follows:

$$\begin{aligned}
\texttt{join}\ (R)\ (S)\ \texttt{as}\ T &= \texttt{join_template}_{\overset{\cup}{\leftarrow},\Phi_m}\ (R, \top_{dom(R)})\ (S, \top_{dom(S)})\ \texttt{as}\ T \\
\texttt{join_dl}\ (R)\ (S)\ \texttt{as}\ T &= \texttt{join_template}_{\overset{\cup}{\leftarrow},\Phi_m}\ (R, \top_{dom(R)})\ (S, \emptyset)\ \texttt{as}\ T \\
\texttt{join_dr}\ (R)\ (S)\ \texttt{as}\ T &= \texttt{join_template}_{\overset{\cup}{\leftarrow},\Phi_m}\ (R, \emptyset)\ (S, \top_{dom(S)})\ \texttt{as}\ T
\end{aligned}$$

The differences between these alternative forms of join is clearly conveyed by example. To this end, we define a pair of database states J with relation T and I with relations R and S.

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b_2 & c_2 \\ & a_2 & b_2 & c_3 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_1 \\ & c_2 \end{array} \right\}^I \qquad \left\{ \begin{array}{c|ccc} T & A & B & C \\ \hline & a_2 & b_2' & c_2 \end{array} \right\}^J$$

Relation R is constrained by the functional dependency $F = \{A \to B\}$, but relation S has no dependencies. As a consequence of the typing rules we see T must also satisfy $\{A \to B\}$. The results of the joins are as follows:

- $\texttt{join}\ (R)\ (S)\ \texttt{as}\ T\searrow\ (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_2 & b_2' & c_2 \\ & a_2 & b_2' & c_3 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_2 \end{array} \right\}^{I'}$$

- $\texttt{join\_dl}\ (R)\ (S)\ \texttt{as}\ T\searrow\ (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_2 & b_2' & c_2 \\ & a_2 & b_2' & c_3 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_1 \\ & c_2 \end{array} \right\}^{I'}$$

[2] There is also a more refined possibility: we could parameterize the join template on a function that chooses, on a case by case basis, whether a given deletion in the view should be reflected as a deletion from the left table, the right table, or both.

- `join_dr` (R) (S) `as` $T\searrow (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b'_2 & c_2 \\ & a_2 & b'_2 & c_3 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_2 \end{array} \right\}^{I'}$$

These update policies give reasonable results; unfortunately, they are only applicable when the functional dependencies are in tree form. In cases where we need to perform a join and this is not the case, we can use a different update policy, defined in terms of an update operator called *squash* (and written $\overset{\cup}{\nleftarrow}$).

Squash is analogous to relation merge. However, while relation merge modifies conflicting records to conform to functional dependencies, squash simply deletes such records. Formally, given M, N, and F all over U with $M \models F$ and $N \models F$, we define

$$M \overset{\cup}{\nleftarrow}_F N = \{m \mid m \in M \text{ and } N, m \models F\} \cup N.$$

The chief advantage of the squash operator is that, unlike Merge, it is suitable when paired with the trivial boolean function, $\Phi_s(U, P, F) = true$.

The following three derived joins are analogous to those above, but use squash to perform updates and Φ_s for static checking:

$$\begin{array}{rcl} \texttt{join}^{\texttt{s}}\ (R)\ (S)\ \texttt{as}\ T & = & \texttt{join\_template}_{\overset{\cup}{\nleftarrow},\Phi_s}\ (R, \top_{dom(R)})\ (S, \top_{dom(S)})\ \texttt{as}\ T \\ \texttt{join\_dl}^{\texttt{s}}\ (R)\ (S)\ \texttt{as}\ T & = & \texttt{join\_template}_{\overset{\cup}{\nleftarrow},\Phi_s}\ (R, \top_{dom(R)})\ (S, \emptyset)\ \texttt{as}\ T \\ \texttt{join\_dr}^{\texttt{s}}\ (R)\ (S)\ \texttt{as}\ T & = & \texttt{join\_template}_{\overset{\cup}{\nleftarrow},\Phi_s}\ (R, \emptyset)\ (S, \top_{dom(S)})\ \texttt{as}\ T \end{array}$$

Using the example databases I and J from above, the results of the squashing variants of join are as follows:

- `join`$^{\texttt{s}}$ (R) (ABC) `as` $SCT\searrow (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_2 & b'_2 & c_2 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_2 \end{array} \right\}^{I'}$$

- `join_dl`$^{\texttt{s}}$ (R) (ABC) `as` $SCT\searrow (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_2 & b'_2 & c_2 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_1 \\ & c_2 \end{array} \right\}^{I'}$$

- `join_dr`$^{\texttt{s}}$ (R) (ABC) `as` $SCT\searrow (J, I)$

$$\left\{ \begin{array}{c|ccc} R & A & B & C \\ \hline & a_1 & b_1 & c_1 \\ & a_2 & b'_2 & c_2 \end{array} \quad \begin{array}{c|c} S & C \\ \hline & c_2 \end{array} \right\}^{I'}$$

To conclude the discussion of derived joins, we must check that relation merge and squash are actually suitable.

5.4.1 Theorem: The operator $\overset{\cup}{\leftarrow}$ and the boolean function

$$\Phi(U, P, F) = F \text{ is tree form and } P \text{ ignores } outputs(F)$$

are suitable at U for any U.

Proof: We must show that all five suitability properties hold.

1. $N \subseteq M \overset{\cup}{\leftarrow}_F N$. Immediate from the definition.
2. $N \subseteq M \implies M \overset{\cup}{\leftarrow}_F N = M$. Immediate from Lemma 4.5.9.
3. $M \overset{\cup}{\leftarrow}_F N : U$. Immediate from Lemma 4.4.2.
4. $M \subseteq P \wedge N \subseteq P \implies M \overset{\cup}{\leftarrow}_F N \subseteq P$. By the definition of $\overset{\cup}{\leftarrow}$ and Lemma 4.5.3, we know that

$$(M \overset{\cup}{\leftarrow}_F N)[U - \mathit{outputs}(F)] \subseteq (M \cup N)[U - \mathit{outputs}(F)] \subseteq P[\mathit{outputs}(F)].$$

   Because P ignores $\mathit{outputs}(F)$, this implies $M \overset{\cup}{\leftarrow}_F N \subseteq P$.

5. $M \overset{\cup}{\leftarrow}_F N \models F$. Immediate from Lemma 4.5.7. □

5.4.2 Theorem: The operator $\overset{\cup}{\not\leftarrow}$ and the boolean function

$$\Phi(U, P, F) = \text{true}$$

are suitable at any U.

Proof: All conditions can easily be checked. □

Of course, this discussion has not exhausted the possibilities for join lenses—far from it. The notion of suitability gives an easily checked criterion for other update operations. We have already remarked that ordinary union is suitable (with a strong Φ); there are undoubtedly others to be discovered.

5.5 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 understood by considering only a single column at a time. In cases where a general projection makes sense, it can be implemented by composing several `drop` operations in sequence.

Letting v stand for the lens expression

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

the behavior of the `drop` lens is given by the following definitions for the *get* and *putback* components:

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

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 functional dependencies are not sufficient to infer 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 the expression associated with 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 the following typing rule:

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

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 taking $X = A$ satisfies the premise.) 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:

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

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

Since B and C independently determine A, any value that is put in place of ? will result in a table that does not satisfy 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.

5.5.1 Theorem: Suppose

$$\begin{array}{ll} sort(R) = (U, P, F) & F \equiv G \cup \{X \to A\} \\ U = Z \uplus \{A\} & sort(S) = (Z, P[Z], G) \\ P = P[Z] \bowtie P[A] & \{A = a\} \in P[A] \\ \Delta = \Sigma \uplus \{R\} & \Delta' = \Sigma \uplus \{S\} \\ v = \texttt{drop } A \texttt{ determined by } (X, a) \texttt{ from } R \texttt{ as } S & \end{array}$$

Then $v \in \Delta \Leftrightarrow \Delta'$.

Proof: We first show that $v\nearrow \in \Delta \to \Delta'$. Suppose $I \models \Delta$. Then $R \in dom(I)$, where $I\backslash_R \models \Sigma$ and $I(R)$ satisfies (U, P, F). From the definition of $v\nearrow$, it is easy to check that $S \in dom(v\nearrow(I))$ and $(v\nearrow(I))\backslash_S \models \Sigma$. It remains to show that $(v\nearrow(I))(S)$ satisfies $(Z, P[Z], G)$, where $(v\nearrow(I))(S) = I(R)[Z]$. This involves checking three facts:

1. $I(R)[Z] : Z$. This follows from $Z \subseteq U$.
2. $I(R)[Z] \subseteq P[Z]$. This follows directly from the definitions.

3. $I(R)[Z] \models G$. Assume, for a contradiction, $\{m_1, m_2\} \not\models G$, where $m_1, m_2 \in I(R)[Z]$. Then there exists a functional dependency $Y_1 \rightarrow Y_2 \in G$ such that $\{m_1, m_2\} \not\models Y_1 \rightarrow Y_2$. There must then exist records $n_1, n_2 \in I(R)$ such that $m_1 = n_1[Z]$ and $m_2 = n_2[Z]$. Furthermore $F \models Y_1 \rightarrow Y_2$. So we would have $I(R) \not\models F$, which is a contradiction. Since m_1 and m_2 were arbitrary, $I(R)[Z] \models G$ follows by Fact 2.4.

Hence, we conclude that $v\nearrow(I) \models \Delta'$, as required.

Next, we show that $v\searrow \in \Delta' \times \Delta \rightarrow \Delta$. Suppose $I \models \Delta$ and $J \models \Delta'$. Then $R \in dom(I)$, where $I\backslash_R \models \Sigma$ and $I(R)$ satisfies (U, P, F). Furthermore, $S \in dom(J)$, where $J\backslash_S \models \Sigma$ and $J(S)$ satisfies $(U, P[Z], G)$. From the definition of $v\searrow$, it is easy to check that $R \in dom(v\searrow(J, I))$ and $(v\searrow(J, I))\backslash_R \models \Sigma$. It remains to show that $(v\searrow(J, I))(R)$ satisfies (U, P, F), where

$$\begin{aligned}(v\searrow(J,I))(R) &= M \leftarrow_{X\rightarrow A} I(R)\\ M &= (I(R) \bowtie J(S)) \cup (N_+ \bowtie \{\{A = a\}\})\\ N_+ &= J(S) \setminus I(R)[Z].\end{aligned}$$

Note that this definition may also be written as

$$\begin{aligned}(v\searrow(J,I))(R) &= (I(R) \bowtie J(S)) \cup M_+\\ M_+ &= (N_+ \bowtie \{\{A = a\}\}) \leftarrow_{X\rightarrow A} I(R)\\ N_+ &= J(S) \setminus I(R)[Z]\end{aligned}$$

since the revision operation can only affect the records in the right-hand side of the union: $N_+ \bowtie \{\{A = a\}\}$. We use this alternate presentation in several places below.

Again, showing the totality of $v\searrow$ involves checking three facts:

1. $M \leftarrow_{X\rightarrow A} I(R) : U$. Checking with the original definition, we have $N_+ \bowtie \{A = a\} : U$ since $N_+ : Z$. Also, $I(R) \bowtie J(S) : U$. So $M \leftarrow_{X\rightarrow A} I(R) : U$ by Lemma 4.5.1.

2. $M \leftarrow_{X\rightarrow A} I(R) \subseteq P$. Again, we use the original definition. Since $P = P[Z] \bowtie P[A]$, by Fact 2.1 it suffices to show $(M \leftarrow_{X\rightarrow A} I(R))[Z] \subseteq P[Z]$ and $(M \leftarrow_{X\rightarrow A} I(R))[A] \subseteq P[A]$. We consider these two requirements in turn.

   (a) Since $J(S) \subseteq P[Z]$, we can see that $N_+ \subseteq P[Z]$ and $(I(R) \bowtie J(S))[Z] \subseteq P[Z]$. Hence, $M[Z] \subseteq P[Z]$. By Lemma 4.5.3, $M \leftarrow_{X\rightarrow A} I(R)[Z] \subseteq M[Z]$, so we have $M \leftarrow_{X\rightarrow A} I(R)[Z] \subseteq P[Z]$.

   (b) Since $A \in dom(I(R))$, by Fact 2.2 we have $(I(R) \bowtie J(S))[A] \subseteq I(R)[A]$. Also, since $I(R) \subseteq P$ and $P = P[Z] \bowtie P[A]$, by Fact 2.2, $I(R)[A] \subseteq P[A]$. Furthermore, since $A \notin dom(N_+)$, by Fact 2.2, $(N_+ \bowtie \{\{A = a\}\})[A] \subseteq \{\{A = a\}\}$, and $\{A = a\} \in P[A]$ by assumption. So

$$\begin{aligned}M[A] &= ((I(R) \bowtie J(S)) \cup (N_+ \bowtie \{\{A = a\}\}))[A]\\ &= (I(R) \bowtie J(S))[A] \cup (N_+ \bowtie \{\{A = a\}\})[A]\\ &\subseteq I(R)[A] \cup (N_+ \bowtie \{\{A = a\}\})[A]\\ &\subseteq I(R)[A] \cup \{\{A = a\}\}[A]\\ &\subseteq P[A].\end{aligned}$$

   Since $M[A] \subseteq P[A]$ and $I(R)[A] \subseteq P[A]$, Lemma 4.5.2 tells us that $(M \leftarrow_{X\rightarrow A} I(R))[A] \subseteq P[A]$, as required.

3. $(I(R) \bowtie J(S)) \cup M_+ \models F$. (Here we use the alternate definition.) We must show that $(I(R) \bowtie J(S)) \cup M_+ \models G$ and $(I(R) \bowtie J(S)) \cup M_+ \models X \rightarrow A$. For the first part, we begin by noting that, if we have a relation $M : U$ and a set of functional dependencies $H : Z$ where $Z \subseteq U$, then $M \models H$ iff $M[Z] \models H$. Since $G : Z$, it suffices to show that $((I(R) \bowtie J(S)) \cup M_+)[Z] \models G$. We have $(I(R) \bowtie J(S))[Z] \subseteq$

$J(S)$ (by Fact 2.2). Since $N_+ \subseteq J(S)$, we also have $M_+[Z] \subseteq J(S)$, using Lemma 4.5.3 (noting that Z is disjoint from $\mathit{outputs}(X \to A) = \{A\}$). Hence $((I(R) \bowtie J(S)) \cup M_+)[Z] \subseteq J(S)$. So $((I(R) \bowtie J(S)) \cup M_+)[Z] \models G$ (by Fact 2.3).

For the other part, we have $I(R) \models X \to A$ since $F \models X \to A$, and it is easy to see that $N_+ \bowtie \{\{A = a\}\} \models X \to A$. Then, since $(I(R) \bowtie J(S))[Z] \subseteq I(R)$, we may use Lemma 4.5.6 to show that $((I(R) \bowtie J(S)) \cup M_+)[Z] \models X \to A$, as required.

Next, we show that v satisfies the law GetPut—that is, $v\searrow(v\nearrow(I), I) = I$ for all $I \in \Delta$. It is easy to check that $\mathit{dom}(v\searrow(v\nearrow(I), I)) = \mathit{dom}(I)$, and that $(v\searrow(v\nearrow(I), I))\backslash_R = I\backslash_R$. It remains to show that $(v\searrow(v\nearrow(I), I))(R) = I(R)$. Expanding the definitions of $v\nearrow$ and $v\searrow$ (using the alternate definition), we see that we must show $(I(R) \bowtie I(R)[Z]) \cup M_+ = I(R)$ where

$$\begin{array}{rcl} M_+ & = & (N_+ \bowtie \{\{A = a\}\}) \leftarrow_{X \to A} I(R) \\ N_+ & = & I(R)[Z] \setminus I(R)[Z]. \end{array}$$

We have $M_+ = \emptyset$ since $N_+ = \emptyset$, and it is easy to see that $(I(R) \bowtie I(R)[Z]) \cup \emptyset = I(R)$.

Finally, we show that v satisfies the law PutGet—that is, $v\nearrow(v\searrow(J, I)) = J$ for all $I \in \Delta$ and $J \in \Delta'$. It is easy to check that $\mathit{dom}(v\nearrow(v\searrow(J, I))) = \mathit{dom}(J)$ and that $(v\nearrow(v\searrow(J, I)))\backslash_S = J\backslash_S$. It remains to show that $(v\nearrow(v\searrow(J, I)))(S) = J(S)$. Expanding the definitions of $v\nearrow$ and $v\searrow$ (using the alternate definition), we see that we must show $((I(R) \bowtie J(S)) \cup M_+)[Z] = J(S)$ where

$$\begin{array}{rcl} M_+ & = & (N_+ \bowtie \{\{A = a\}\}) \leftarrow_{X \to A} I(R) \\ N_+ & = & J(S) \setminus I(R)[Z]. \end{array}$$

This follows from a straightforward calculation:

$$\begin{array}{rcl} ((I(R) \bowtie J(S)) \cup M_+)[Z] & = & (I(R) \bowtie J(S))[Z] \cup M_+[Z] \\ & = & (I(R)[Z] \cap J(S)) \cup M_+[Z] \\ & = & (I(R)[Z] \cap J(S)) \cup N_+ \\ & = & (I(R)[Z] \cap J(S)) \cup (J(S) \setminus I(R)[Z]) \\ & = & J(S) \qquad \square \end{array}$$

5.6 Lens Composition

Lens composition is at the heart of our language. Given two 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:

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

The proof of well-behavedness of composition can be found in [5].

5.7 Other Primitive Lenses

The operational behavior of our lenses does not permit any table to be referred to more than once in a view definition. This is because our schema language is not powerful enough to express arbitrary constraints across different relations, which may be necessary if data from tables in the concrete database were to appear in multiple places in the view. Thus, even without restrictions on schemas, our language for defining views is substantially weaker that the general relational algebra. In order to express operations such as an outer join, new basic lenses must be developed. We have investigated versions of outer join that give a wide range of flexibility.

6 Related Work

The basic framework of lenses was introduced in a 2005 paper by Foster, Greenwald, Moore, Pierce, and Schmitt [5], to which we refer readers for an extensive survey of the related literature. Here, we give a high-level picture of the similarities and differences between our work and other approaches to view update—in particular Dayal and Bernstein's notion [3] of "correct update translation," Bancilhon and Spyratos's notion [1] of "update translation under a constant complement," Gottlob, Paolini, and Zicari's "dynamic views" [8], and the basic view update and "relational triggers" mechanisms offered by commercial database systems such as Oracle.

The view update problem concerns translating updates on a view into "reasonable" updates on the underlying database. It is helpful to structure the discussion by breaking this broad problem statement down into more specific questions. First, how is a "reasonable" translation of an update defined? Second, what should we do about the possibility that, for some update, there may be *no* reasonable way of translating its effect to the underlying database? And third, how do we deal with the possibility that there are *many* reasonable translations from which we must choose? We consider these questions in order.

One can imagine many possible ways of assigning a precise meaning to "reasonable update translation," but in fact there is a remarkable degree of agreement in the literature, with most approaches adopting one of two basic positions. The stricter of these is enunciated in Bancilhon and Spyratos's [1] notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there exists 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." The constant complement approach has influenced numerous later works in the area, including recent papers by Lechtenbörger [14] and Hegner [9].

The other, more permissive, definition of "reasonable" is elegantly formulated by Gottlob, Paolini, and Zicari, who call it "dynamic views" [8]. They present a general framework and identify two special cases, one being formally equivalent to Bancilhon and Spyratos's constant complement translators and the other—which they advocate on pragmatic grounds—being their own dynamic views.

Our notion of lenses adopts the same, more permissive, attitude towards reasonable behavior of update translation. Our definition of `select`, for example, is similar to an example proposed by Keller [13] as an illustration of a natural update policy that would be disallowed under the constant complement approach. Indeed, modulo some small technical refinements required to connect our inclusive notion of "totality" to the specific sets of update operations considered by others, the correspondence is exact [5, 15]: the set of all well-behaved lenses is isomorphic to the set of dynamic views in the sense of Gottlob, Paolini, and Zicari. (Moreover, the set of well-behaved lenses that also obey an additional law called PutPut in [5]

$$v\searrow(J', v\searrow(J, I)) = v\searrow(J', I) \qquad \text{(PutPut)}$$
$$\text{for all } J, J' \in \Delta \text{ and } I \in \Sigma$$

is isomorphic to the set of translators under constant complement in the sense of Bancilhon and Spyratos. Intuitively, this law says that each *putback* completely overwrites the effects of all previous updates—the effect of doing two *putback*s is the same as doing just the second.)

Dayal and Bernstein's [3] seminal theory of "correct update translation" also adopts the more permissive position on "reasonableness." Their notion of "exactly performing an update" corresponds, intuitively, to our PutGet law.

The pragmatic tradeoffs between these two perspectives on reasonable update translations are discussed by Hegner [10, 9], who introduces the term *closed view* for the stricter constant complement approach and *open view* for the looser approach adopted by dynamic views and in the present work. Hegner himself works in a closed-world framework, but notes that both choices may have pragmatic advantages in different situations, open-world being useful when the users are aware that they are "really" using a view as a convenient way to edit an underlying database, while closed-world is preferable when users should be isolated from the existence of the underlying database, even at the cost of offering them a more restricted set of possible updates.

Hegner [9] also formalizes an additional condition on reasonableness (which has also been noted by others—e.g., [3]): *monotonicity* of update translations, in the sense that an update that only adds records from the view should be translated just into additions to the database, and that an update that adds more records to the view should be translated to a larger update to the database (and similarly for deletions). As we have noted, some of our primitive lenses are *not* monotone in general. Many specific uses of our lenses will be monotone, however, and we conjecture that our type system could be refined so as to track monotonicity and let users know when it may be violated.

Commercial databases such as Oracle, SQL Server, and DB2 typically provide two quite different mechanisms for updating through views. First, some very simple views—defined using select, project, and a very restricted form of join (where the key attributes in one relation are a subset of those in the other)—are considered *inherently updatable*. For these, the notion of reasonableness is essentially the constant complement position. Alternatively, programmers can support updates to arbitrary views by adding *relational triggers* that are invoked whenever an update is attempted on the view and that can execute arbitrary code to update the underlying database. In this case, the notion of reasonableness is left entirely to the programmer.

The second question posed at the beginning of the section was how to deal with the possibility that there are no reasonable translations for some update. The simplest response is just to let the translation of an update fail, if it sees that its effect is going to be unreasonable; this is Dayal and Bernstein's approach, for example. Its advantage is that we can determine reasonableness on a case-by-case basis, allowing translations that usually give reasonable results but that might fail under rare conditions. The disadvantage is that we lose the ability to perform updates to the view offline—we need the concrete database in order to tell whether an update is going to be allowed.

Another possibility is to restrict the set of operations to just the ones that can be guaranteed to correspond to reasonable translations; this is the position taken by most papers in the area.

A different approach—the one we have taken in this work—is to restrict the view schema so that *arbitrary* (schema-respecting) updates are guaranteed to make sense.

The third question posed above was how to deal with the possibility that there may be multiple reasonable translations for a given update.

One attractive idea is to somehow restrict the set of reasonable translations so that this possibility does not arise—i.e., so that every translatable update has a unique translation. For example, under the constant complement approach, for a particular choice of complement, there will be at most one translation. Hegner's additional condition of monotonicity [9] ensures that (at least for updates consisting of only inserts or only deletes), the translation of an update is unique, independent of the choice of complement.

Another possibility is to place an ordering on possible translations of a given update and choose one that is minimal in this ordering. This idea plays a central role, for example, in Johnson, Rosebrugh, and Dampney's account of view update in the Sketch Data Model [11]. Buneman, Khanna, and Tan [2] have established a variety of intractability results for the problem of inferring minimal view updates in the relational setting for query languages that include both join and either project or union.

The key idea in the present work is to allow the programmer to describe the update policy at the same time as the view definition, by enriching the relational primitives with enough annotations to select among a variety of reasonable update policies.

7 Future Work

The use of lenses represents a novel approach to the view update problem for relational databases, for which many avenues of inquiry remain to be examined. On a theoretical level we are curious about extensions to the lens language and type system. We are also interested in potential applications both within and outside the context of the Harmony system. Here we introduce a few of the most promising areas for future research.

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.

A weaker version of PutPut can be formulated if there is an order on states in the abstract domain. Let $v \in \Sigma \Leftrightarrow \Delta$ and let $\prec \subseteq \Delta \times \Delta$ be a preorder on the elements of Δ. Then we define the law PutPutOrd:

$$v\searrow(J', v\searrow(J, I)) = v\searrow(J', I) \quad \text{(PutPutOrd)}$$
$$\text{for all } J, J' \in \Delta \text{ and } I \in \Sigma$$
$$\text{such that } v\nearrow(I) \prec J \prec J'$$

We conjecture that all of our lenses in this paper satisfy PutPutOrd for the subset and superset partial orders on the relations in the database.

Other theoretical work may include ideas farther afield from lenses. In particular, we are curious about composable techniques in the realm of pure constant complement or dynamic views. For example, it may be possible to use types to describe the set of view updates available to an end user who supplies a few definition.

We are interested in practical concerns surrounding potential implementations of the theory we have presented. We believe 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 better in some cases to allow lenses to fail during *putback*, accepting the loss of offline operation in return for greater flexibility of behavior.

We have experimented with several examples of modest complexity (larger than Figure 1, but not huge). In the future, we plan to try larger applications. This will be aided by an implementation of relational lenses now under development in the context of the Harmony system.

References

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

[2] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *PODS'02*, pages 150–158, 2002.

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

[4] 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.

[5] 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*, 2005. Extended version available as University of Pennsylvania technical report MS-CIS-03-08. Earlier version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[6] 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*, 2005.

[7] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. HARMONY: A synchronization framework for heterogeneous tree-structured data, 2005. `http://www.cis.upenn.edu/~bcpierce/harmony`.

[8] 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.

[9] 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.

[10] Stephen J. Hegner. Foundations of canonical update support for closed database views. In *ICDT '90: Proceedings of the third international conference on database theory on Database theory*, pages 422–436, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[11] Michael Johnson, Robert Rosebrugh, and C. N. G. Dampney. View updates in a semantic data modelling paradigm. In *ADC '01: Proceedings of the 12th Australasian conference on Database technologies*, pages 29–36, Washington, DC, USA, 2001. IEEE Computer Society.

[12] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS'85*, 1985.

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

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

[15] Benjamin C. Pierce and Alan Schmitt. Lenses and view update translation. Manuscript; available from `http://www.cis.upenn.edu/~bcpierce/harmony`, 2003.