Implementing Mapping Composition
(Extended Technical Report)

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Todd J. Green*
U. of Pennsylvania
tjgreen@cis.upenn.edu

Sergey Melnik
Microsoft Research
melnik@microsoft.com

Alan Nash*
U.C. San Diego
anash@cs.ucsd.edu

Abstract
Mapping composition is a fundamental operation in metadata driven applications. Given a mapping over schemas \( \sigma_1, \sigma_2, \) and \( \sigma_3, \) the composition problem is to find an equivalent mapping over only \( \sigma_1 \) and \( \sigma_3. \) We describe a new composition algorithm that targets practical applications. It incorporates view unfolding. It eliminates as many \( \sigma_2 \) symbols as possible, even if not all can be eliminated. It covers constraints expressed using arbitrary monotone relational operators and, to a lesser extent, non-monotone operators. And it introduces the new technique of left composition. We describe our implementation, explain how to extend it to support user-defined operators, and present experimental results which validates its effectiveness.

1 Introduction
A mapping is a relationship between the instances of two schemas. Some common types of mappings are relational queries, relational view definitions, global-and-local-as-view (GLAV) assertions, XQuery queries, and XSL transformations. The manipulation of mappings is at the core of many important data management problems, such as data integration, database design, and schema evolution. Hence, general-purpose algorithms for manipulating mappings have broad application to data management.

Data management problems like those above often require that mappings be composed. The composition of a mapping \( m_{12} \) between schemas \( \sigma_1, \sigma_2 \) and a mapping \( m_{23} \) between schemas \( \sigma_2, \sigma_3 \) is a mapping between \( \sigma_1 \) and \( \sigma_3 \) that captures the same relationship between \( \sigma_1 \) and \( \sigma_3 \) as \( m_{12} \) and \( m_{23} \) taken together.

Given that mapping composition is useful for a variety of database problems, it is desirable to develop a general-purpose composition component that can be reused in many application settings, as was proposed in [1, 2]. This paper reports on the development of such a component, an implementation of a new algorithm for composing mappings between relational schemas. Compared to past approaches, the algorithm handles more expressive mappings, makes a best-effort when it cannot obtain a perfect answer, includes several new heuristics, and is designed to be extensible.

1.1 Applications of Mapping Composition
Composition arises in many practical settings. In data integration, a query needs to be composed with a view definition. If the view definition is expressed using global-as-view (GA\textsc{v}), then this is an example of composing two functional mappings: a view definition that maps a database to a view, and a query that maps a view to a query result. The standard approach is view unfolding, where references to the view in the query are replaced by the view definition [11]. View unfolding is simply function composition, where a function definition (i.e., the body of the view) is substituted for references to the function (i.e., the view schema) in the query.

In peer-to-peer data exchange, composition is used as an optimization to speed up queries and updates on peer databases connected through a network of mappings. For example, Piazza [12] uses composition to optimize reformulation of XML queries. In the ORCHESTRA collaborative data sharing system [6, 13], when updates are published, the system composes mappings to relate peers directly and avoid computing intermediate relations [5].

A third example is schema evolution, where a schema \( \sigma_1 \) evolves to become a schema \( \sigma_1'. \) The relationship between \( \sigma_1' \) and \( \sigma_1 \) can be described by a mapping. After \( \sigma_1 \) has evolved, any existing mappings involving \( \sigma_1, \) such as a mapping from \( \sigma_1 \) to schema \( \sigma_2, \) must now be upgraded to a mapping from \( \sigma_1' \) to \( \sigma_2. \) This can be done by composing the \( \sigma_1', \sigma_3 \) mapping with the \( \sigma_1', \sigma_2 \) mapping. Depending on the application, one or both of these mappings may be non-functional, in which case composing mappings is no
longer simply function composition.

A different schema evolution problem arises when an initial schema $\sigma_1$ is modified by two independent designers, producing schemas $\sigma_2$ and $\sigma_3$. To merge them into a single schema, we need a mapping between $\sigma_2$ and $\sigma_3$ that describes their overlapping content [3, 10]. This $\sigma_2$-$\sigma_3$ mapping can be obtained by composing the $\sigma_1$-$\sigma_2$ and $\sigma_1$-$\sigma_3$ mappings. Even if the latter two mappings are functions, one of them needs to be inverted before they can be composed. Since the inverse of a function may not be a function, this too entails the composition of non-functional mappings.

Finally, consider a database design process that evolves a schema $\sigma_1$ via a sequence of incremental modifications. This produces a sequence of mappings between successive versions of the schema, from $\sigma_1$ to $\sigma_2$, then to $\sigma_3$, and so forth, until the desired schema $\sigma_n$ is reached. At the end of this process, a mapping from $\sigma_1$ to the evolved schema $\sigma_n$ is needed, for example, as input to the schema evolution scenarios above. This mapping can be obtained by composing the mappings between the successive versions of the schema. The following example illustrates this last scenario.

**EXAMPLE 1** Consider a schema editor, in which the designer modifies a database schema, resulting in a sequence of schemas with mappings between them. She starts with the schema

\[
\text{Movies}(\text{mid}, \text{name}, \text{year}, \text{rating}, \text{genre}, \text{theater})
\]

where $\text{mid}$ means movie identifier. The designer decides that only 5-star movies and no ‘theater’ should be present in the database; she edits the table obtaining the following schema and mapping:

\[
\text{FiveStarMovies}(\text{mid}, \text{name}, \text{year}, \text{genre})
\]

\[\pi_{\text{mid}, \text{name}, \text{year}, \text{genre}}(\sigma_{\text{rating}=5}(\text{Movies})) \subseteq \text{FiveStarMovies}\]

To improve the data organization, the designer then splits the $\text{FiveStarMovies}$ table, resulting in a new schema and mapping

\[
\text{Names}(\text{mid}, \text{name}) \quad \text{Years}(\text{mid}, \text{year})
\]

\[\pi_{\text{mid}, \text{name}, \text{year}}(\text{FiveStarMovies}) \subseteq \text{Names} \bowtie \text{Years}\]

The system composes mappings (1) and (2) into a new mapping:

\[
\pi_{\text{mid}, \text{name}}(\sigma_{\text{rating}=5}(\text{Movies})) \subseteq \text{Names}
\]

\[
\pi_{\text{mid}, \text{year}}(\sigma_{\text{rating}=5}(\text{Movies})) \subseteq \text{Years}
\]

With this mapping, the designer can now migrate data from the old schema to the new schema, reformulate queries posed over one schema to equivalent queries over the other schema, etc.

### 1.2 Related Work

Mapping composition is a challenging problem. Madhavan and Halevy [7] showed that the composition of two given mappings expressed as GLAV formulas may not be expressible in a finite set of first-order constraints. Fagin, Kolaitis, Popa, and Tan [4] showed that the composition of certain kinds of first-order mappings may not be expressible in any first-order language, even by an infinite set of constraints. That is, that language is not closed under composition. Nash, Bernstein, and Melnik [9] showed that for certain classes of first-order languages, it is undecidable to determine whether there is a finite set of constraints in the same language that represents the composition of two given mappings. These results are sensitive to the particular class of mappings under consideration. But in all cases the mapping languages are first-order and are therefore of practical interest.

In [4] Fagin et al. introduced a second-order mapping language that is closed under composition, namely source-to-target tuple-generating dependencies. A second-order language is one that can quantify over function and relation symbols. A constraint over schemas $\sigma_1$ and $\sigma_2$ is source-to-target if it says that a formula over $\sigma_1$ implies a formula over $\sigma_2$. In [4], a second-order tuple-generating dependency is of the form $\exists f_1 \ldots \exists f_m ((\forall x_1 (\phi_1 (x_1) \rightarrow \psi_1 (x_1))) \ldots (\forall x_n (\phi_n (x_n) \rightarrow \psi_n (x_n))))$, where $f_1, \ldots, f_m$ are function symbols (essentially Skolem functions), each $\phi_i$ is a formula over $\sigma_1$, and $\psi_i$ is a formula over $\sigma_2$. Fagin et al. present a composition algorithm for this language, and show it can have practical value for some data management problems, such as data exchange. However, using it for that purpose requires a custom implementation, since the language of second-order tuple-generating dependencies is not supported by standard SQL-based database tools.

Yu and Popa [15] considered mapping composition for second-order source-to-target constraints over nested relational schemas in support of schema evolution. They presented a composition algorithm similar to the one in [4], with extensions to handle nesting, and with significant attention to minimizing the size of the result. They reported on a set of experiments using mappings on both synthetic and real life schemas, to demonstrate that their algorithm is fast and is effective at minimizing the size of the result.

Nash et al. [9] studied the composition of first-order constraints that are not necessarily source-to-target. That is, in the formalism of tuple-generating dependencies above, they allow each premise $\phi$ and conclusion $\psi$ to refer to relations in both the source and target schemas $\sigma_1$ and $\sigma_2$, and they do not allow existential quantifiers over function symbols. The composition of constraints in this language is not closed and determining whether a composition result exists is undecidable. Nevertheless, they gave an algorithm that produces a composition, if it halts (which it may not do).

Like Nash et al. [9], we explore the mapping composition problem for constraints that are not restricted to be-

---

*Note: The natural text is an adaptation of the converted text to make it more coherent and readable.*
ing source-to-target. Our algorithm strictly extends that of Nash et al. [9], which in turn strictly extends that of Fagin et al. [4] for source-to-target embedded dependencies. If the input is a set of source-to-target embedded dependencies, our algorithm behaves similarly to that in [4], except that as in [9] we also attempt to express the result as embedded dependencies through a de-Skolemization step. It is known from results in [4] that such a step can not always succeed. Furthermore, we also apply a “left-compose” step which allows the algorithm to handle mappings on which the algorithm in [9] fails.

1.3 Contributions

Given the inherent difficulty of the problem and limitations of past approaches, we recognized that compromises and special features would be needed to produce a mapping composition algorithm of practical value. The first issue was which language to choose.

Algebra-based rather than logic-based. We wanted our composition algorithm to be directly usable by existing database tools. We therefore chose a relational algebraic language: Each mapping is expressed as a set of constraints, each of which is either a containment or equality of two relational algebraic expressions. This language extends the algebraic dependencies of [14]. Each constraint is of the form \( E_1 = E_2 \) or \( E_1 \subseteq E_2 \) where \( E_1 \) and \( E_2 \) are arbitrary relational expressions containing not only select, project, and join but possibly many other operators. Calculus-based languages have been used in all past work on mapping composition we know of. We chose relational algebra because it is the language implemented in all relational database systems and most tools. It is therefore familiar to the developers of such systems, who are the intended users of our component. It also makes it easy to extend our language simply by allowing the addition of new operators. Notice that the relational operators we handle are sufficient to express embedded dependencies. Therefore, the class of mappings which our algorithm accepts includes embedded dependencies and, by allowing additional operators such as set difference, goes beyond them.

Eliminates one symbol at a time. Our algorithm for composing these types of algebraic mappings gives a partial solution when it is unable to find a complete one. The heart of our algorithm is a procedure to eliminate relation symbols from the intermediate signature \( \sigma_2 \). Such elimination can be done one symbol at a time. Our algorithm makes a best effort to eliminate as many relation symbols from \( \sigma_2 \) as possible, even if it cannot eliminate all of them. By contrast, if the algorithm in [9] is unable to produce a mapping over \( \sigma_1 \) and \( \sigma_3 \) with no \( \sigma_2 \)-symbols, it simply runs forever or gives up. In some cases it may be better to eliminate some symbols from \( \sigma_2 \) successfully, rather than insist on either eliminating all of them or failing. Thus, the resulting mapping may be over \( \sigma_1, \sigma_2', \) and \( \sigma_3 \), where \( \sigma_2' \) is a subset of \( \sigma_2 \) instead of over just \( \sigma_1 \) and \( \sigma_3 \).

To see the value of this best-effort approach, consider a composition that produces a mapping \( m \) that contains a \( \sigma_2 \)-symbol \( S \). If \( m \) is later composed with another mapping, it is possible that the latter composition can eliminate \( S \). (We will see examples of this later, in our experiments.) Also, the inability to eliminate \( S \) may be inherent in the given mappings. For example, \( S \) may be involved in a recursive computation that cannot be expressed purely in terms of \( \sigma_1 \) and \( \sigma_3 \) such those of Theorem 1 in [9]:

\[
R \subseteq S, \quad S = tc(S), \quad S \subseteq T
\]

where \( \sigma_1 = \{ R \}, \sigma_2 = \{ S \}, \sigma_3 = \{ T \} \) with \( R, S, T \) binary and where the middle constraint says that \( S \) is transitively closed. In this case, \( S \) cannot be eliminated, but is definable as a recursive view on \( R \) and can be added to \( \sigma_1 \). To use the mapping, those non-eliminated \( \sigma_2 \)-symbols may need to be populated as intermediate relations that will be discarded at the end. In this example this involves low computational cost. In many applications it is better to have such an approximation to a desired composition mapping than no mapping at all. Moreover, in many cases the extra cost associated with maintaining the extra \( \sigma_2 \) symbols is low.

Tolerance for unknown or partially known operators. Instead of rejecting an algebraic expression because it contains unknown operators which we do not know how to handle, our algorithm simply delays handling such operators as long as possible. Sometimes, it needs no knowledge at all of the operators involved. This is the case, for example, when a subexpression that contains an unknown operator can be replaced by another expression. At other times, we need only partial knowledge about an operator. Even if we do not have the partial knowledge we need, our algorithm does not fail globally, but simply fails to eliminate one or more symbols that perhaps it could have eliminated if it had additional knowledge about the behavior of the operator.

Use of monotonicity. One type of partial knowledge that we exploit is monotonicity of operators. An operator is monotone in one of its relation symbol arguments if, when tuples are added to that relation, no tuples disappear from its output. For example, select, project, join, union, and semijoin are all monotone. Set difference (e.g., \( R - S \)) and left outerjoin are monotone in their first argument \( R \) but not in their second \( S \). Our key observation is that when an operator is monotone in an argument, that argument can sometimes be replaced by an expression from another constraint. For example, if we have \( E_1 \subseteq R \) and \( E_2 \subseteq S \), then in some cases it is valid to replace \( R \) by \( E_1 \) in \( R - S \), but not to replace \( S \) by \( E_2 \). Although existing algorithms only work with select project, join and union, this observation enables our algorithm to handle outerjoin, select difference, and anti-semijoin. Moreover, our algorithm can handle nulls and bag semantics in many cases.

Normalization and denormalization. We call left-normalization the process of bringing the constraints to a form where a relation symbol \( S \) that we are trying to eliminate appears in a single constraint alone on the left. The result is of the form \( S \subseteq E \) where \( E \) is an expression.
We define right-normalization similarly. Normalization may introduce “pseudo-operators” such as Skolem functions which then need to be eliminated by a denormalization step. Currently we do not do much left-normalization. Our right-normalization is more sophisticated and, in particular, can handle projections by Skolemization. The corresponding denormalization is very complex. An important observation here is that normalization and denormalization are general steps which may possibly be extended on an operator-by-operator basis.

Left compose. One way to eliminate a relation symbol $S$ is to replace $S$’s occurrences in some constraints by the expression on the other side of a constraint that is normalized for $S$. There are two versions of this replacement, right compose and left compose. In right compose, we use a constraint $E \subseteq S$ that is right-normalized for $S$ and substitute $E$ for $S$ on the left side of a constraint that is monotonic in $S$, such as transforming $R \times S \subseteq T$ into $R \times E \subseteq T$, thereby eliminating $S$ from the constraint. Right composition is an extension of the algorithms in [4, 9]. We also introduce left compose, which handles some additional cases where right compose fails. Suppose we have the constraints $E_2 \subseteq M(S)$ and $S \subseteq E_1$, where $M(S)$ is an expression that is monotonic in $S$ but which we either do not know how to right-normalize or which would fail to right-denormalize. Then left compose immediately yields $E_2 \subseteq M(E_1)$.

Extensibility and modularity. Our algorithm is extensible by allowing additional information to be added separately for each operator in the form of information about monotonicity and rules for normalization and denormalization. Many of the steps are rule-based and implemented in such a way that it is easy to add rules or new operators. Therefore, our algorithm can be easily adapted to handle additional operators without specialized knowledge about its overall design. Instead, all that is needed is to add new rules.

Experimental study. We implemented our algorithm and ran experiments to study its behavior. We used composition problems drawn from the recent literature [4, 8, 9], and a set of large synthetic composition tasks, in which the mappings were generated by composing sequences of elementary schema modifications. We used these mappings to test the scalability and effectiveness of our algorithm in a systematic fashion. Across a range of composition tasks, it eliminated 50-100% of the symbols and usually ran in under a second. We see this study as a step toward developing a benchmark for composition algorithms.

The rest of the paper is organized as follows. Section 2 presents notation needed to describe the algorithm. Section 3 describes the algorithm itself, starting with a high-level description and then drilling into the details of each step, one-by-one. Section 4 presents our experimental results. Section 5 is the conclusion.

2 Preliminaries

We adopt the unnamed perspective which references the attributes of a relation by index rather than by name. A relational expression is an expression formed using base relations and the basic operators union $\cup$, intersection $\cap$, cross product $\times$, set difference $-$, projection $\pi$ and selection $\sigma$ as follows. The name $S$ of a relation is a relational expression. If $E_1$ and $E_2$ are relational expressions, then so are

$$E_1 \cup E_2 \quad E_1 \cap E_2 \quad E_1 \times E_2$$

$$E_1 - E_2 \quad \sigma_c(E_1) \quad \pi_1(E_1)$$

where $c$ is an arbitrary boolean formula on attributes (identified by index) and constants and $I$ is a list of indexes. The meaning of a relational expression is given by the standard set semantics. To simplify the presentation in this paper, we focus on these basic six relational operators and view the join operator $\bowtie$ as a derived operator formed from $\times$, $\pi$, and $\sigma$. We also allow for user-defined operators to appear in expressions. The basic operators should therefore be considered as those which have “built-in” support, but they are not the only operators supported.

The basic operators differ in their behavior with respect to arity. Assume expression $E_1$ has arity $r$ and expression $E_2$ has arity $s$. Then the arity of $E_1 \cup E_2$, $E_1 \cap E_2$, and $E_1 - E_2$ is $r = s$; the arity of $E_1 \times E_2$ is $r + s$; the arity of $\sigma_c(E_1)$ is $r$; and the arity of $\pi_1(E_1)$ is $|I|$.

We define an additional operator which may be used in relational expressions called the Skolem function. A Skolem function has a name and a set of indexes. Let $f$ be a Skolem function on indexes $I$. Then $f_1(E_1)$ is an expression of arity $r + 1$. Intuitively, the meaning of the operator is to add an attribute to the output, whose values are some function $f$ of the attribute values identified by the indexes in $I$. We do not provide a formal semantics here. Skolem functions are used internally as a technical device in Section 3.5.

We consider constraints of two forms. A containment constraint is a constraint of the form $E_1 \subseteq E_2$, where $E_1$ and $E_2$ are relational expressions. An equality constraint is a constraint of the form $E_1 = E_2$, where $E_1$ and $E_2$ are relational expressions. We denote sets of constraints with capital Greek letters and individual constraints with lowercase Greek letters.

A signature is a function from a set of relation symbols to positive integers which give their arities. In this paper, we use the terms signature and schema synonymously. We denote signatures with the letter $\sigma$. (We denote relation symbols with uppercase Roman letters $R, S, T$, etc.) We sometimes abuse notation and use the same symbol $\sigma$ to mean simply the domain of the signature (a set of relations).

An instance of a database schema is a database that conforms to that schema. We use uppercase Roman letters $A, B, C$, etc to denote instances. If $A$ is an instance of a database schema containing the relation symbol $S$, we denote by $S^A$ the contents of the relation $S$ in $A$. 

4
Given a relational expression \( E \) and a relational symbol \( S \), we say that \( E \) is monotone in \( S \) if whenever instances \( A \) and \( B \) agree on all relations except \( S \) and \( S^A \subseteq S^B \), then \( E(A) \subseteq E(B) \). In other words, \( E \) is monotone in \( S \) if adding more tuples to \( S \) only adds more tuples to the query result. We say that \( E \) is anti-monotone in \( S \) if whenever \( A \) and \( B \) agree on all relations except \( S \) and \( S^A \subseteq S^B \), then \( E(A) \supseteq E(B) \).

The active domain of an instance is the set of values that appear in the instance. We allow the use of a special relational symbol \( D \) which denotes the active domain of an instance. \( D \) can be thought of as a shorthand for the relational expression \( \bigcup_{i=1}^{n} \bigcup_{j=1}^{a_i} \pi_j(S_i) \) where \( \sigma = \{S_1, \ldots, S_n\} \) is the signature of the database and \( a_i \) is the arity of relation \( S_i \). We also allow the use of another special relation in expressions, the empty relation \( \emptyset \).

Given a relational expression \( E \), we consider the containment constraints when \( E(A) \subseteq E(B) \), which is equivalent to \( E(A) = E(B) \).

EXAMPLE 2 The constraint that the first attribute of a binary relation \( S \) is the key for the relation, which can be expressed in a logic-based setting as the equality-generating dependency \( S(x, y), S(x, z) \rightarrow y = z \) may be expressed in our setting as a containment constraint by making use of the active domain relation

\[
\pi_{24}(\sigma_{1=3}(S^2)) \subseteq \sigma_{1=2}(D^2)
\]

where \( S^2 \) is short for \( S \times S \) and \( D^2 \) is short for \( D \times D \).

A mapping is a binary relation on instances of database schemas denoted by the letter \( m \). Given a class of constraints \( \mathcal{L} \), we associate to every expression of the form \( (\sigma_1, \sigma_2, \Sigma_{12}) \) the mapping

\[
\{ (A, B) : (A, B) \models \Sigma_{12} \}.
\]

That is, it defines which instances of two schemas correspond to each other. Here \( \Sigma_{12} \) is a finite subset of \( \mathcal{L} \) over the signature \( \sigma_1 \cup \sigma_2 \), \( \sigma_1 \) is the input (or source) signature, \( \sigma_2 \) is the output (or target) signature, \( A \) is a database with signature \( \sigma_1 \) and \( B \) is a database with signature \( \sigma_2 \). We assume that \( \sigma_1 \) and \( \sigma_2 \) are disjoint. \( (A, B) \) is the database with signature \( \sigma_1 \cup \sigma_2 \) obtained by taking all the relations in \( A \) and \( B \) together. Its active domain is the union of the active domains of \( A \) and \( B \). In this case, we say that \( m \) is given by \((\sigma_1, \sigma_2, \Sigma_{12})\).

Given two mappings \( m_{12} \) and \( m_{23} \), the composition \( m_{12} \circ m_{23} \) is the unique mapping

\[
\{ (A, C) : \exists B ( (A, B) \in m_{12} \text{ and } (B, C) \in m_{23}) \}.
\]

Assume two mappings \( m_{12} \) and \( m_{23} \) are given by \((\sigma_1, \sigma_2, \Sigma_{12})\) and \((\sigma_2, \sigma_3, \Sigma_{23})\). The mapping composition problem is to find \( \Sigma_{13} \) such that \( m_{12} \circ m_{23} \) is given by \((\sigma_1, \sigma_3, \Sigma_{13})\).

Given a finite set of constraints \( \Sigma \) over some schema \( \sigma \) and another finite set of constraints \( \Sigma' \) over some sub-schema \( \sigma' \) of \( \sigma \) we say that \( \Sigma \) is equivalent to \( \Sigma' \), denoted \( \Sigma \equiv \Sigma' \), if

1. **(Soundness)** Every database \( A \) over \( \sigma \) satisfying \( \Sigma \) when restricted to only those relations in \( \sigma' \) yields a database \( A' \) over \( \sigma' \) that satisfies \( \Sigma' \)

2. **(Completeness)** Every database \( A' \) over \( \sigma' \) satisfying the constraints \( \Sigma' \) can be extended to a database \( A \) over \( \sigma \) satisfying the constraints \( \Sigma \) by adding new relations in \( \sigma \setminus \sigma' \) (not limited to the domain of \( A' \)).

EXAMPLE 3 For example, the set of constraints

\[
\Sigma := \{ R \subseteq S, S \subseteq T \}
\]

is equivalent to the set of constraints \( \Sigma' := \{ R \subseteq T \} \).

Given this definition, we can restate the composition problem as follows. Given a set of constraints \( \Sigma_{12} \) over \( \sigma_1 \cup \sigma_2 \) and a set of constraints \( \Sigma_{23} \) over \( \sigma_2 \cup \sigma_3 \), find a set of constraints \( \Sigma_{13} \) over \( \sigma_1 \cup \sigma_3 \) such that \( \Sigma_{12} \cup \Sigma_{23} \equiv \Sigma_{13} \).

3 Algorithm

3.1 Overview

At the heart of the composition algorithm (which appears at the end of this subsection), we have the procedure \textsc{Eliminate} which takes as input a finite set of constraints \( \Sigma \) over some schema \( \sigma \) that includes the relation symbol \( S \) and which produces as output another finite set of constraints \( \Sigma' \) over \( \sigma \setminus \{S\} \) such that \( \Sigma \equiv \Sigma' \), or reports failure to do so. On success, we say that we have eliminated \( S \) from \( \Sigma \).

Given such a procedure \textsc{Eliminate} we have several choices on how to implement \textsc{Compose}, which takes as input three schemas \( \sigma_1, \sigma_2, \sigma_3 \) and two sets of constraints \( \Sigma_{12} \) and \( \Sigma_{23} \) over \( \sigma_1 \cup \sigma_2 \) and \( \sigma_2 \cup \sigma_3 \) respectively. The goal of \textsc{Compose} is to return a set of constraints \( \Sigma_{13} \) over \( \sigma_1 \cup \sigma_3 \). That is, its goal is to eliminate the relation symbols from \( \sigma_2 \). Since this may not be possible, we aim at eliminating from \( \Sigma := \Sigma_{12} \cup \Sigma_{23} \) a set \( S \) of relation symbols in \( \sigma_2 \) which is as large as possible or which is maximal under some other criterion than the number of relation symbols in it. There are many choices about how to do this, but we do not explore them in this paper. Instead we simply follow the user-specified ordering on the relation symbols in \( \sigma_2 \) and try to eliminate as many as possible in that order.

We therefore concentrate in the remainder of this section on \textsc{Eliminate}. It consists of the following three steps, which we describe in more detail in following sections:

1. View unfolding
2. Left compose
3. Right compose
Each of the steps 1, 2, and 3 attempts to remove $S$ from $\Sigma$. If any of them succeeds, ELIMINATE terminates successfully. Otherwise, ELIMINATE fails.

All three steps work in essentially the same way: given a constraint that contains $S$ alone on one side of a constraint and an expression $E$ on the other side, they substitute $E$ for $S$ in all other constraints.

**Example 4** Here are three one-line examples of how each of these three steps transforms a set of two constraints into an equivalent set with just one constraint:

1. $S = R \times T$, $\pi(U) - S \subseteq U \Rightarrow \pi(U) - (R \times T) \subseteq U$
2. $R \subseteq S \cap V$, $S \subseteq T \times U \Rightarrow R \subseteq (T \times U) \cap V$
3. $T \times U \subseteq S$, $S - \pi(U) \subseteq R \Rightarrow (T \times U) - \pi(U) \subseteq R$

In order to perform such a substitution we need an expression which contains $S$ alone on one side of a constraint. This holds in the example. However, this is typically not the case, and another key feature of our algorithm is that it performs normalization as necessary to get the constraints in such a form. In the case of left and right compose, we also need all other expressions which contain $S$ to be monotone in $S$.

We now give a more detailed technical overview of the three steps we have just introduced. To simplify the discussion below, we take $\Sigma_0 := \Sigma$ to be the input to ELIMINATE and $\Sigma_n$ to be the result after step $s$ is complete. We use $E, E_1, E_2$ to stand for arbitrary relational expressions and $M(S)$ to stand for a relational expression monotonic in $S$.

1. **View Unfolding.** We look for a constraint $\xi$ of the form $S = E_1$ in $\Sigma_0$ where $E_1$ is an arbitrary expression which does not contain $S$. If there is no such constraint, we set $\Sigma_1 := \Sigma_0$ and go on to the next step. Otherwise, to obtain $\Sigma_1$ we remove $\xi$ and replace every occurrence of $S$ in every other constraint in $\Sigma_0$ with $E_1$. Then $\Sigma_1 \equiv \Sigma_0$. Soundness is obvious and to show completeness it is enough to set $S := E_1$.

2. **Left Compose.** If $S$ appears on both sides of some constraint in $\Sigma_1$, we exit. Otherwise, we convert every equality constraint $E_1 = E_2$ that contains $S$ into two containment constraints $E_1 \subseteq E_2$ and $E_2 \subseteq E_1$ to obtain $\Sigma_1'$. Next we check $\Sigma_1'$ for right-monotonicity in $S$. That is, we check whether every expression $E$ in which $S$ appears to the right of a containment constraint is monotone in $S$. If this check fails, we set $\Sigma_2 := \Sigma_1$ and go on to the next step.

Next we left-normalize every constraint in $\Sigma_1'$ for $S$ to obtain $\Sigma_2'$. That is, we replace all constraints in which $S$ appears on the left with a single equivalent constraint $\xi$ of the form $S \subseteq E_1$. That is, $S$ appears alone on the left in $\xi$. This is not always possible: if we fail, we set $\Sigma_2 := \Sigma_1$ and go on to the next step.

If $S$ does not appear on the left of any constraint, then we add to $\Sigma_1'$ the constraint $\xi : S \subseteq E_1$ and we set $E_1 := D^r$ where $r$ is the arity of $S$. Here $D$ is a special symbol which stands for the active domain. Clearly, any $S$ satisfies this constraint.

Now to obtain $\Sigma_3''$ from $\Sigma_3'$ we remove $\xi$ and for every constraint in $\Sigma_3'$ of the form $E_2 \subseteq M(S)$ where $M$ is monotonic in $S$, we put a constraint of the form $E_2 \subseteq M(E_1)$ in $\Sigma_3''$. We call this step basic left-composition. Finally, to the extent that our knowledge of the operators allows us, we attempt to eliminate $D^r$ (if introduced) from any constraints, to obtain $\Sigma_3$. For example $E_1 \cap D^r$ becomes $E_1$.

Then $\Sigma_2 \equiv \Sigma_1$. Soundness follows from monotonicity since $E_2 \subseteq M(S) \subseteq M(E_1)$ and to show completeness it is enough to set $S := E_1$.

3. **Right Compose.** Right compose is dual to left-compose. We check for left-monotonicity and we right-normalize as in the previous step to obtain $\Sigma_2''$ with a constraint $\xi$ of the form $E_1 \subseteq S$. If $S$ does not appear on the right of any constraint, then we add to $\Sigma_2''$ the constraint $\xi : E_1 \subseteq S$ and set $E_1 := \emptyset$. Clearly, any $S$ satisfies this constraint.

In order to handle projection during the normalization step we may introduce Skolem functions. For example $R \subseteq \pi_1(S)$ where $R$ is unary and $S$ is binary becomes $f(R) \subseteq S$. The expression $f(R)$ is binary and denotes the result of applying some unknown Skolem function $f$ to the expression $R$. The right-normalization step always succeeds for select, project, and join, but may fail for other operators. If we fail to normalize, we set $\Sigma_3 := \Sigma_2$ and we exit.

Now to obtain $\Sigma_3''$ from $\Sigma_3'$ we remove $\xi$ and for every constraint in $\Sigma_3'$ of the form $M(S) \subseteq E_2$ where $M$ is monotonic in $S$, we put a constraint of the form $M(E_1) \subseteq E_2$ in $\Sigma_3''$. We call this step basic right-composition. Finally, to the extent that our knowledge of the operators allows us, we attempt to eliminate $\emptyset$ (if introduced) from any constraints, to obtain $\Sigma_3$. For example $E_1 \cup \emptyset$ becomes $E_1$.

Then $\Sigma_2'' \equiv \Sigma_2$. Soundness follows from monotonicity since $M(E_1) \subseteq M(S) \subseteq E_2$ and to show completeness it is enough to set $S := E_1$.

Since during normalization we may have introduced Skolem functions, we now need a right-denormalization step to remove such Skolem functions. Following [9], we call this part deskolemization. Deskolemization is very complex and may fail. If it does, we set $\Sigma_3 := \Sigma_2$ and we exit. Otherwise, we set $\Sigma_3$ to be the result of deskolemization.
Procedure ELIMINATE  
**Input:** Signature \( \sigma \)  
Constraints \( \Sigma \)  
Relation Symbol \( S \)  
**Output:** Constraints \( \Sigma' \) over \( \sigma \) or \( \sigma - \{ S \} \)  
1. \( \Sigma' := \text{VIEW UNFOLD}(\Sigma, S) \), On success, return \( \Sigma' \).  
2. \( \Sigma' := \text{LEFTCOMPOSE}(\Sigma, S) \), On success, return \( \Sigma' \).  
3. \( \Sigma' := \text{RIGHTCOMPOSE}(\Sigma, S) \). On success, return \( \Sigma' \).  
4. Return \( \Sigma \) and indicate failure.

Procedure COMPOSE  
**Input:** Signatures \( \sigma_1, \sigma_2, \sigma_3 \)  
Constraints \( \Sigma_{12}, \Sigma_{23} \)  
Relation Symbol \( S \)  
**Output:** Signature \( \sigma \) satisfying \( \sigma_1 \cup \sigma_3 \subseteq \sigma \subseteq \sigma_1 \cup \sigma_2 \cup \sigma_3 \)  
1. Set \( \sigma := \sigma_1 \cup \sigma_2 \cup \sigma_3 \).  
2. Set \( \Sigma = \Sigma_{12} \cup \Sigma_{23} \).  
3. For every relation symbol \( S \in \sigma_2 \) do:  
4. \( \Sigma := \text{ELIMINATE}(\sigma, \Sigma, S) \)  
5. On success, set \( \sigma := \sigma - \{ S \} \).  
6. Return \( \sigma, \Sigma \).

**THEOREM 1** Algorithm COMPOSE is correct.

**PROOF 1** (Sketch) Correctness follows from points 1-3 above and the proof of correctness of the de-Skolemization algorithm in [9].

### 3.2 View Unfolding

The goal of the unfold views step is to eliminate \( S \) at an early stage by applying the technique of view unfolding. It takes as input a set of constraints \( \Sigma_0 \) and a symbol \( S \) to be eliminated. It produces as output an equivalent set of constraints \( \Sigma_1 \) with \( S \) eliminated (in the success case), or returns \( \Sigma_0 \) (in the failure case). The step proceeds as follows. We look for a constraint \( \xi \) of the form \( S = E_1 \) in \( \Sigma_0 \) where \( E_1 \) is an arbitrary expression which does not contain \( S \). If there is no such constraint, we set \( \Sigma_1 := \Sigma_0 \) and report failure. Otherwise, to obtain \( \Sigma_1 \) we remove \( \xi \) and replace every occurrence of \( S \) in every other constraint in \( \Sigma_0 \) with \( E_1 \). Note that \( S \) may occur in expressions which are not necessarily monotone in \( S \), or which contain user-defined operators about which little is known. In either case, because \( S \) is defined by an equality constraint, the result is still an equivalent set of constraints. This is in contrast to left compose and right compose, which rely for correctness on the monotonicity of expressions in \( S \) when performing substitution.

**EXAMPLE 5** Suppose the input constraints are given by  
\[ S = R_1 \times R_2, \quad \pi(R_3 - S) \subseteq T_1, \quad T_2 \subseteq T_3 - \sigma_c(S). \]  
Then unfold views deletes the first constraint and substitutes \( R_1 \times R_2 \) for \( S \) in the second two constraints, producing  
\[ \pi(R_3 - (R_1 \times R_2)) \subseteq T_1, \quad T_2 \subseteq T_3 - \sigma_c(R_1 \times R_2). \]

Note that in this example, neither left compose nor right compose would succeed in eliminating \( S \). Left compose would fail because the expression \( T_3 - \sigma_c(S) \) is not monotone in \( S \). Right compose would fail because the expression \( \pi(R_3 - S) \) is not monotone in \( S \). Therefore view unfolding does indeed give us some extra power compared to left compose and right compose alone.

### 3.3 Checking Monotonicity

The correctness of performing substitution to eliminate a symbol \( S \) in the left compose and right compose steps depends upon the left-hand side (lhs) or right-hand side (rhs) of all constraints being monotone in \( S \). We describe here a sound but incomplete procedure MONOTONE for checking this property. MONOTONE takes as input an expression \( E \) and a symbol \( S \). It returns ‘m’ if the expression is monotone in \( S \), ‘a’ if the expression is anti-monotone in \( S \), ‘i’ if the expression is independent of \( S \) (for example, because it does not contain \( S \)), and ‘u’ if it cannot say how the expression depends on \( S \). For example, given the expression \( S \times T \) and symbol \( S \) as input, MONOTONE returns ‘m’, while given the expression \( \sigma_{c_1}(S) - \sigma_{c_2}(S) \) and the symbol \( S \), MONOTONE returns ‘u’.

The procedure is defined recursively in terms of the six basic relational operators. In the base case, the expression is a single relational symbol, in which case MONOTONE returns ‘m’ if that symbol is \( S \), and ‘i’ otherwise. Otherwise, in the recursive case, MONOTONE first calls itself recursively on the operands of the top-level operator, then performs a simple table lookup based on the return values and the operator. For the unary expressions \( \sigma(E_1) \) and \( \pi(E_1) \), we have that MONOTONE(\( \sigma(E_1), S \)) = MONOTONE(\( \pi(E_1), S \)) = MONOTONE(\( E_1, S \)) (in other words, \( \sigma \) and \( \pi \) do not affect the monotonicity of the expression). Otherwise, for the binary expressions \( E_1 \cup E_2 \), \( E_1 \cap E_2 \), \( E_1 \times E_2 \), and \( E_1 \times E_2 \), there are sixteen cases to consider, corresponding to the possible values of MONOTONE(\( E_1, S \)) and MONOTONE(\( E_2, S \)).

Note that \( \times \), \( \cap \), and \( \cup \) all behave in the same way from the point of view of MONOTONE, that is, MONOTONE(\( E_1 \cup E_2, S \)) = MONOTONE(\( E_1 \cap E_2, S \)) = MONOTONE(\( E_1 \times E_2, S \)), for all \( E_1, E_2 \). Set difference \( - \), on the other hand, behaves differently than the others.

In order to support user-defined operators in MONOTONE, we just need to know the rules regarding the monotonicity of the operator in \( S \), given the monotonicity of its operands in \( S \). Once these rules have been added to the appropriate tables, MONOTONE supports the user-defined operator automatically.

### 3.4 Left Compose

Recall from Section 3.1 that left compose consists of four main steps, once equality constraints have been converted to containment constraints. The first is to check the constraints for right-monotonicity in \( S \), that is, to check whether every expression \( E \) in which \( S \) appears to the right of a containment constraint is monotone in \( S \). Section 3.3
nary relation. Define the instance \( A \) to be \( R^A := \{1, 2\} \), \( S^A := \{1, 2\} \), \( T^A := \{11, 22\} \). Then \( A \models \{R \subseteq \pi_1(T), S \subseteq \pi_2(T)\} \), but \( A \not\models \{R \times S \subseteq T\} \).

In addition to the basic relational operators, left normalize may be extended to handle user-defined operators, by specifying a user-defined rewriting rule for each such operator.

Left normalize proceeds as follows. Let \( \Sigma_1 \) be the set of input constraints, and let \( S \) be the symbol to be eliminated from \( \Sigma_1 \). Left normalize computes a set \( \Sigma'_1 \) of constraints as follows. Set \( \Gamma_1 := \Sigma_1 \). We loop as follows, beginning at \( i = 1 \). In the \( i \)th iteration, there are two cases:

1. If there is no constraint in \( \Gamma_i \) which contains \( S \) on the rhs in a complex expression, set \( \Sigma'_i \) to be \( \Gamma_i \) with all the constraints containing \( S \) on the lhs collapsed into a single constraint, which has an intersection of expressions on the right. For example, \( S \subseteq E_1, S \subseteq E_2 \) becomes \( S \subseteq E_1 \cap E_2 \). If \( S \) does not appear on the lhs of any expression, we add to \( \Sigma'_i \) the constraint \( S \subseteq D^r \) where \( r \) is the arity of \( S \). Finally, return success.

2. Otherwise, choose some constraint \( \xi := E_1 \subseteq E_2 \), where \( E_1 \) contains \( S \). If there is no rewriting rule for the top-level operator in \( E_1 \), set \( \Sigma'_i := \Sigma_1 \) and return failure. Otherwise, set \( \Gamma_{i+1} \) to be the set of constraints obtained from \( \Gamma_i \) by replacing \( \xi \) with its rewriting, and iterate.

**Example 7** Suppose the input constraints are given by

\[
R - S \subseteq T, \quad \pi(S) \subseteq U.
\]

where \( S \) is the symbol to be eliminated. Then left normalization succeeds and returns the constraints

\[
R \subseteq S \cup T, \quad S \subseteq U \times D^r.
\]

**Example 8** Suppose the input constraints are given by

\[
R \cap S \subseteq T, \quad \pi(S) \subseteq U.
\]

Then left normalization fails, because there is no rule matching the lhs of \( R \cap S \subseteq T \).

**Example 9** Suppose the input constraints are given by

\[
R \cap T \subseteq S, \quad U \subseteq \pi(S).
\]

Since there is no constraint containing \( S \) on the left, left normalize adds the trivial constraint \( S \subseteq D^r \), producing

\[
R \cap T \subseteq S, \quad U \subseteq \pi(S), \quad S \subseteq D^r.
\]

**3.4.2 Basic Left Compose**

Among the constraints produced by left normalize, there is a single constraint \( \xi := S \subseteq E_1 \) which has \( S \) on its lhs. In basic left compose, we remove \( \xi \) from the set of constraints, and we replace every other constraint of the form \( E_2 \subseteq M(S) \), where \( M(S) \) is monotonic in \( S \), with a constraint of the form \( E_2 \subseteq M(E_1) \). This is easier to understand with the help of a few examples.
EXAMPLE 10 Consider the constraints from Example 7 after left normalization:

\[ R \subseteq S \cup T, \quad S \subseteq U \times D^r. \]

The expression \( S \cup T \) is monotone in \( S \). Therefore, we are able to left compose to obtain

\[ R \subseteq (U \times D^r) \cup T. \]

Note although the input constraints from Example 7 could just as well be put in right normal form, right compose would fail, because the expression \( R - S \) is not monotone in \( S \). Thus left compose does indeed give us some additional power.

EXAMPLE 11 We continue with the constraints from Example 9:

\[ R \cap T \subseteq S, \quad U \subseteq \pi(S), \quad U \subseteq \pi(D^r). \]

We left compose and obtain

\[ R \cap T \subseteq D^r, \quad U \subseteq \pi(D^r). \]

Note that the active domain relation \( D \) occurs in these constraints. In the next section, we explain how to eliminate it.

3.4.3 Eliminate Domain Relation

We have seen that left compose may produce a set of constraints containing the symbol \( D \) which represents the active domain relation. The goal of this step is to eliminate \( D \) from the constraints, to the extent that our knowledge of the operators allows, which may result in entire constraints disappearing in the process as well. We use rewriting rules are derived from the following identities for the basic relational operators:

\[
\begin{align*}
E_1 \cup D^r &= D^r & E_1 \cap D^r &= E_1 \\
E_1 - D^r &= \emptyset & \pi_1(D^r) &= D^{1|1}
\end{align*}
\]

(We do not know of any identities applicable to cross product or selection.) In addition, the user may supply rewriting rules for user-defined operators, which we will make use of if present. The constraints are rewritten using these rules until no rule applies. At this point, \( D \) may appear alone on the rhs of some constraints. We simply delete these, since a constraint of this form is satisfied by any instance. Note that we do not always succeed in eliminating \( D \) from the constraints, but this is acceptable, since in particular a constraint containing \( D \) can still be checked.

EXAMPLE 12 We continue with the constraints from Example 11:

\[ R \cap T \subseteq D^r, \quad U \subseteq \pi(D^r). \]

First, the domain relation rewriting rules are applied, yielding

\[ R \cap T \subseteq D^r, \quad U \subseteq D^k, \]

where \( k \) is the arity of \( \pi(D^r) \). Then, since both of these constraints have the domain relation alone on the rhs, we are able to simply delete them.

3.5 Right Compose

Recall from Section 3.1 that right compose proceeds through five main steps. The first step is to check that every expression \( E \) that appears to the left of a containment constraint is monotone in \( S \). The procedure for checking this was described in Section 3.3. The other four steps are right normalize, basic right compose, right-denormalize, and eliminate empty relations. In this section, we describe these steps in more detail and provide some examples.

3.5.1 Right Normalize

Right normalize is dual to left normalize. The goal of right normalize is to put the constraints in a form where \( S \) appears on the rhs of exactly one constraint, which has the form \( E_1 \subseteq S \). We say that the constraints are then in right normal form. We make use of the following identities for containment constraints in right normalization (note that here again the subscripts of the projection operator \( \pi \) are omitted for readability):

\[
\begin{align*}
\cup & : \quad E_1 \subseteq E_2 \cup E_3 \leftrightarrow E_1 \subseteq E_3 \subseteq E_2 \\
& \quad \leftrightarrow E_1 - E_2 \subseteq E_3 \\
\cap & : \quad E_1 \subseteq E_2 \cap E_3 \leftrightarrow E_1 \subseteq E_2, E_1 \subseteq E_3 \\
\times & : \quad E_1 \subseteq E_2 \times E_3 \leftrightarrow \pi(E_1) \subseteq E_2, \pi(E_1) \subseteq E_2 \\
- & : \quad E_1 \subseteq E_2 - E_3 \leftrightarrow E_1 \subseteq E_2, E_1 \cap E_3 \subseteq \emptyset \\
\pi & : \quad E_1 \subseteq \pi(E_2) \leftrightarrow f(E_1) \subseteq E_2 \\
\sigma & : \quad E_1 \subseteq \sigma_c(E_2) \leftrightarrow E_1 \subseteq E_2, E_1 \subseteq \sigma_c(D^r)
\end{align*}
\]

As in left normalize, to each identity in the list, we associate a rewriting rule which takes a constraint of the form given by the lhs of the identity and produces an equivalent constraint or set of constraints of the form given by the rhs of the identity. For example, from the identity for \( \sigma \) we obtain a rule which matches constraints of the form \( E_1 \subseteq \sigma_c(E_2) \) and produces the equivalent pair of constraints \( E_1 \subseteq E_2 \) and \( E_1 \subseteq \sigma_c(D^r) \). Note that, as with left normalize, there is at most one rule for each operator, so to find the rule which matches a particular expression, we need only look up the rule corresponding to the topmost operator in the expression. In contrast to the rules used by left normalize, there is a rule in this list for each of the six basic relational operators. Therefore right normalize always succeeds when applied to constraints which use only basic relational expressions.

Just as with left normalize, user-defined operators can be supported via user-specified rewriting rules. If there is
a user-defined operator which does not have a rewriting rule, then right normalize may fail in some cases.

Note that the rewriting rule for the projection operator \( \pi \) may introduce Skolem functions. The deskolemize step will later attempt to eliminate any Skolem functions introduced by this rule. If we have additional knowledge about key constraints for the base relations, we use this to minimize the list of attributes on which the Skolem depends. This increases our chances of success in deskolemize.

3.5.2 Basic Right Compose

After right normalize, there is a single constraint \( \xi := E_1 \subseteq S \) which has \( S \) on its rhs. In basic right compose, we remove \( \xi \) from the set of constraints, and we replace every other constraint of the form \( M(S) \subseteq E_2 \), where \( M(S) \) is monotonic in \( S \), with a constraint of the form \( M(E_1) \subseteq E_2 \). This is easier to understand with the help of a few examples.

**EXAMPLE 15** Recall the constraints produced by right normalize in Example 13:

\[
S \times T \subseteq U, \quad \pi(T) \subseteq S, \quad \pi(T) \subseteq \sigma_c(D^r), \quad \pi(T) \subseteq \pi(R).
\]

Given those constraints as input, basic right compose produces

\[
\pi(T) \times T \subseteq U, \quad \pi(T) \subseteq \sigma_c(D^r), \quad \pi(T) \subseteq \pi(R).
\]

Since the constraints contain no Skolem functions, in this case we are done.

**EXAMPLE 16** Recall the constraints produced by right normalize in Example 14:

\[
\pi(f(R)) \subseteq S, \quad \pi(f(R)) \subseteq T \cap U, \quad S \subseteq \sigma_c(T).
\]

Given those constraints as input, basic right compose produces

\[
\pi(f(R)) \subseteq T \cap U, \quad \pi(f(R)) \subseteq \sigma_c(T).
\]

Note that composition is not yet complete in this case. We will need to try to complete the process by deskolemizing the constraints to get rid of \( f \). This process is described in the next section.

3.5.3 Right-Denormalize

During right-normalization, we may introduce Skolem functions in order to handle projection. For example, we transform \( R \subseteq \pi_1(S) \) where \( R \) is unary and \( S \) is binary to \( f_1(R) \subseteq S \). The subscript 1 indicates that \( f \) depends on position 1 of \( R \). That is, \( f_1(R) \) is a binary expression where to every value in \( R \) another value is associated by \( f \). Thus, after basic right-composition, we may have constraints with Skolem functions in them. The semantics of such constraints is that they hold iff there exist some values for the Skolem functions which satisfy the constraints. The objective of the deskolemization step is to remove such Skolem functions. It is a complex 12-step procedure based on a similar procedure presented in [9].

Procedure \( \text{DESKOLEMIZE}(\Sigma) \):

1. Unnest
2. Check for cycles
3. Check for repeated function symbols
4. Align variables
5. Eliminate restricting atoms
6. Eliminate restricted constraints
7. Check for remaining restricted constraints
8. Check for dependencies
9. Combine dependencies
10. Remove redundant constraints
11. Replace functions with $\exists$-variables
12. Eliminate unnecessary $\exists$-variables

Here we only highlight some aspects specific to this implementation. First of all, as we already said, we use an algebra-based representation instead of a logic-based representation. A Skolem function for us is a relational operator which takes an $r$-ary expression and produces an expression of arity $r + 1$. Our goal at the end of step 3 is to produce expressions of the form

$$\pi \sigma f \ldots \sigma (R_1 \times R_2 \times \ldots \times R_k).$$

Here

- $\pi$ selects which positions will be in the final expression,
- the outer $\sigma$ selects some rows based on values in the Skolem functions,
- $f, g, \ldots$ is a sequence of Skolem functions,
- the inner $\sigma$ selects some rows independently of the values in the Skolem functions, and
- $(R_1 \times R_2 \times \ldots \times R_k)$ is a cross product of possibly repeated base relations.

The goal of step 4 is to make sure that across all constraints, all these expressions have the same arity for the part after the Skolem functions. This is achieved by possibly padding with the $D$ symbol. Furthermore, step 4 aligns the Skolems in such a way that across all constraints the same Skolem functions appear, in the same sequence. For example, if we have the two expressions $f_1(R)$ and $g_1(S)$ with $R, S$ unary, step 4 rewrites them as

$$\pi_{13}g_2f_1(R \times S) \text{ and } \pi_{24}g_2f_1(R \times S).$$

Here $R \times S$ is a binary expression with $R$ in position 1 and $S$ in position 2 and $g_2f_1(R \times S)$ is an expression of arity 4 with $R$ in position 1, $S$ in position 2, $f$ in position 3 depending only on position 1, and $g$ in position 4 depending only on position 2.

The goal of step 5 is to eliminate the outer selection $\sigma$ and of step 6 to eliminate constraints having such an outer selection. The remaining steps correspond closely to the logic-based approach.

Deskolemization is complex and may fail at several of the steps above. The following two examples illustrates some cases where deskolemization fails.

**Example 17** Consider the following constraints from [4] where $E, F, C, D$ are binary and $\sigma_2 = \{F, C\}$:

$$E \subseteq F, \quad \pi_1(E) \subseteq \pi_1(C), \quad \pi_2(E) \subseteq \pi_1(C) \quad \pi_{46} \sigma_{1=3,2=5}(F \times C \times C) \subseteq D$$

Right-composition succeeds at eliminating $F$ to get

$$\pi_1(E) \subseteq \pi_1(C), \quad \pi_2(E) \subseteq \pi_1(C) \quad \pi_{46} \sigma_{1=3,2=5}(E \times C \times C) \subseteq D$$

Right-normalization for $C$ yields

$$\pi_{13}f_{12}(E) \subseteq C, \quad \pi_{23}g_{12}(E) \subseteq C \quad \pi_{46} \sigma_{1=3,2=5}(E \times C \times C) \subseteq D$$

and basic right-composition yields 4 constraints including

$$\pi_{46} \sigma_{1=3,2=5}(E \times \pi_{13}f_{12}(E) \times \pi_{13}g_{12}(E)) \subseteq D$$

which causes deskolemize to fail at step 3. Therefore, right-compose fails to eliminate $C$. As shown in [4] eliminating $C$ is impossible by any means.

### 3.5.4 Eliminate Empty Relation

Right compose may introduce the empty relation symbol $\emptyset$ into the constraints during composition in the case where $S$ does not appear on the rhs of any constraint. In this step, we attempt to eliminate the symbol, to the extent that our knowledge of the operators allows. This is usually possible and often results in constraints disappearing entirely, as we shall see. For the basic relational operators, we make use of rewriting rules derived from the following identities for the basic relational operators:

$$E_1 \cup \emptyset = E_1, \quad E_1 \cap \emptyset = \emptyset, \quad E_1 - \emptyset = E_1$$

In addition we allow the user to supply rewriting rules for user-defined operators. The constraints are rewritten using these rules until no rule applies. At this point, some constraints may have the form $\emptyset \subseteq E_2$. These constraints are then simply deleted, since they are satisfied by any instance. Note that we do not always succeed in eliminating the empty relation symbol from the constraints, but this is acceptable, since in particular a constraint containing $\emptyset$ can still be checked.

### 4 Experiments

Prior work [4, 9] showed that characterizing the class of mappings that can be composed is very difficult, even when no outer joins, unions, or difference operators are present in the mapping constraints. In this section we make a first attempt to systematically explore the boundaries of mappings that can be composed. Ultimately, our goal is to develop a mapping composition benchmark that can be used to compare competing implementations of mapping composition. The experiments that we report on could form a part of such a benchmark.
exploits most of the techniques presented in earlier sections and the resulting mapping constraints. This example lists the symbols that the algorithm managed to eliminate and left outerjoin. The composition problem is stated as similar to the slotted algebraic notation used in previous sections. We built a parser that takes as input a textual specification of a composition problem and converts it into an internal algebraic representation, which is fed to our algorithm.

Below we show a sample run of the algorithm on mappings that contain relational operators union, difference, and left outerjoin. The composition problem is stated as that of eliminating a given set of relation symbols. The output lists the symbols that the algorithm managed to eliminate and the resulting mapping constraints. This example exploits most of the techniques presented in earlier sections.

```
SCHEMA
R1(2), R2(2), R3(2), R4(2), R5(2), S1(2), S2(2), S3(2), S4(2), S5(2), S6(2), S7(2), S8(2), T1(2), T2(2), T3(2), T4(2), T5(2), T6(2)

CONSTRAINTS
S1 = P_{0,2} LEFTOUTERJOIN_{0,1:1,2} (R2 R3),
JOIN_{0,1:1,0} (R2 R2) <= S5,
JOIN_{0,1:0,0} (R2 R2) <= S5,
R5 <= S7,
P_{0,1} R5 <= P_{0,1} S8,
P_{0,1} R5 <= P_{0,1} S8,
R1 <= S6,
P_{0,2} JOIN_{0,1:1,2} (S6 S6) <= S6,
DIFFERENCE (S2 S1) <= T1,
CROSS (P_{0,0} S1 P_{0,1} T2) <= T1,
UNION (T1 S3) <= T2,
T1 <= UNION (T2 T4),
P_{0,4} JOIN_{0,1:1,2:2,3:3,4} (S5 S5 S5 S5) <= T6,
P_{3,5} S_{0=2,1=3} CROSS(S7 S8 S8) <= T5,
S6 <= T4
ELIMINATED S1, S2, S3, S4, S5, S7
YIELDING
P_{3,5} S_{0=2,1=3} X (R5 S8 S8) <= T5,
P_{0,4} P_{0,1} S_{0=3,1=2} X (R2 R2),
UNION(P_{0,1} S_{0=2,2=3} X (R2 R2),
P_{0,1} S_{0=3,1=2} X (R2 R2),
UNION(P_{0,1} S_{0=2,2=3} X (R2 R2),
P_{0,1} S_{0=3,1=2} X (R2 R2),
UNION(P_{0,1} S_{0=2,2=3} X (R2 R2),
P_{0,1} S_{0=3,1=2} X (R2 R2)) <= T6,
T1 <= T2,
X (P_{0} P_{0,2} LEFTOUTERJOIN_{0,1:1,2} (R2 R3)
P_{0,1} T2) <= T1,
P_{0} S5 <= P_{0} S8,
P_{1} R5 <= P_{1} S8,
R1 <= S6,
P_{0,2} J_{0,1:1,2} (S6 S6) <= S6,
S6 <= T4
```

We used two data sets in our experiments. The first one contains 22 composition problems drawn from the recent literature [4, 8, 9], which illustrate subtle composition issues. The attractiveness of this data set is that the expected output mappings were verified manually and documented in the literature, sometimes using formal proofs. So, this data set serves as a correctness bar that can be used for testing implementations of composition.

We found that the output constraints produced by our algorithm are often more verbose than the ones derived manually, i.e., normalization of output mappings is essential. An example of such normalization is detecting and removing implied constraints. Mapping normalization appears to be a problem of independent interest and is out of scope of this paper. Furthermore, we found that our technique of representing key constraints using the active domain symbol works well in many cases, but fails in others due to de-Skolemization. Leveraging key constraints in a more direct way, e.g., using specialized rules in the composition algorithm, may increase its coverage.

The second data set used in the experiments is synthetic. Using synthetic mappings appears to be the primary way to study the scalability and effectiveness of mapping composition; this approach was also followed in [15]. Our study
is based on the schema evolution scenarios outlined in the introduction. Specifically, we focus on schema editing and schema reconciliation tasks since mapping adaptation can be viewed as a special case of schema reconciliation where one of the input mappings is fixed.

In the study based on synthetic mappings, we determine the success rate of our algorithm in eliminating symbols for various composition tasks, measure its execution time, and investigate the contributions of the main steps of the algorithm on its overall performance. To generate synthetic data for our experiments, we developed a tool which we call a schema evolution simulator. The simulator is driven by a weighted set of schema evolution primitives, such as adding or dropping attributes and relations, schema normalization, or vertical partitioning, and produces a mapping between the original schema and the evolved schema.

In the schema editing scenario, we run the simulator to mimic the schema transformation operations done by a database designer. The mapping between the original schema and the current state of the schema is composed with the mapping produced by each subsequent schema evolution primitive. We record the success or failure of each composition operation for the applied primitives.

To study schema reconciliation, we use the simulator to produce a large number of evolved schemas and mappings for a given original schema. We then compose the generated mappings pairwise using our composition tool.

Our key observations from this study are summarized below:

- Our algorithm is able to eliminate 50-100% of the symbols in the examined composition tasks (Figures 4,7–10).
- The algorithm’s median running time is in the subsecond range for mappings containing hundreds of constraints (Figures 5–7,9,10).
- Composition becomes increasingly harder as more schema evolution primitives are applied to schemas (Figure 9).
- Certain kinds of schema evolution primitives are more likely to produce complications for composition than others (Figures 4,5,7).
- Key constraints do not substantially affect the symbol-eliminating power of the algorithm, yet significantly increase the running time (Figures 4,5,10).
- It is beneficial to keep the symbols that could not be eliminated in the mappings as second-order constraints as long as possible. Subsequent composition operations may eliminate up to a third of those symbols (Figure 10).
- Our algorithm appears to be order-invariant on the studied data sets, i.e., it eliminates the same fraction of symbols no matter in what order the symbols are tried.

In the remaining space we present some details of our study. The experiments were conducted on a 1.5 GHz Pentium M machine.

### 4.2 Schema Evolution Simulator

Figure 1 lists the schema evolution primitives implemented in our simulator. Each primitive takes zero or one relation as input, and produces zero or more new relations and constraints. The produced constraints link the output relations to the input relations, or represent key or inclusion constraints on the output relations. In the figure, we use the named perspective on the relational algebra to simplify the exposition. Attribute lists are shown in bold (e.g., $A$), keys are underlined, and lower case symbols denote constants. The shown list of primitives covers a large fraction of those used in schema evolution and data integration literature but we do not claim completeness.

The first four primitives AR, DR, AA, and DA add or delete relations and attributes. Primitive AR creates a new relation. The arity of the new relation is chosen at random between some minimal and maximal relation arity (2 and 10, in our study). If keys are enabled, the created relation may have a key whose size is chosen between some minimal and maximal value (1 and 3, in our study). Primitive AA adds a new attribute $C$ to the relation $R$, i.e., produces a new relation $S$ that contains $C$ and all existing attributes $A$ of $R$. The mapping constraint $R = \pi_A(S)$ states that $R$ can be reconstructed as a projection on $S$. Primitive DA is complementary to AA.

Primitives D, H, V, and N have forward and backward variants (an explicit list of all supported primitives appears in Figure 2). The forward variant, labeled with subscript ‘f’, contains only the constraints that define the output relations in terms of the input relations. The backward variant (‘b’) contains only the constraints that define the inputs in terms of the outputs. The forward and backward variants reflect distinct evolution scenarios, as we explain below in more detail.

Primitive D adds an attribute $C$ with a default value $c$. The forward variant $D_f$ outputs the mapping constraint $R \times \{c\} = S$, while $D_b$ outputs $R = \pi_A(\sigma_{C=c}(S))$. Thus, $D_f$ states that $S$ is determined by $R$ and that the newly added column $C$ contains $c$-values only. $D_b$ produces a weaker constraint that allows column $S.C$ to contain other values beyond $c$.

Primitive H performs a lossless horizontal partitioning of the input relation $R$. The forward variant $H_f$ outputs the mapping constraints $\sigma_{B=c_s}(R) = S$, $\sigma_{B=c_T}(R) = T$, which tell us how to obtain partitions $S$ and $T$ by selecting tuples from $R$. Some data from $R$ is allowed to be lost. The backward variant $H_b$ produces constraint $R = S \cup T$, which states that $R$ can be reconstructed from $S$ and $T$ but does not include the partitioning criterion. The constraints produced by $H_f$ and $H_b$ do not imply each other.

The vertical partitioning primitives $V$, $V_f$, $V_b$ are the only ones that require the input relation $R$ to have a key. The attribute set of $R$ is partitioned across output relations $S$ and $T$.

Primitive N captures the schema normalization rule from database textbooks. The input relation $R$, which may contain redundant data, is split into $S$ and $T$. $A$ becomes
a key in $S$, and a foreign key in $T$. This primitive assumes the existence of an implicit functional dependency $A \rightarrow B$ on $R$. Although such functional dependencies are not enforced by database systems and hence are missing in real-world schemas, database administrators use this implicit knowledge in schema design. Should this implicit dependency be violated for a particular instance of $R$, then applying the $N_f$ primitive incurs data loss, while $N$ produces an inconsistency.

The constants $c$, $c_T$, and $c_S$ used in the primitives $D$ and $H$ and their variants are drawn from a fixed-size pool of constants (in our study, of size 10).

All evolution primitives discussed above produce equality constraints. Some schema integration and data exchange scenarios in the literature assume the so-called open-world semantics for mapping constraints: informally speaking, subset is used in place of equality. To accommodate these scenarios in our experiments, we added two further primitives, $\text{Sub}$ and $\text{Sup}$. Composition of mapping constraints produced by $\text{Sub}$ and $\text{Sup}$ with those of other primitives yields inclusion constraints that generalize global-local-as-view (GLAV) settings. For example, applying the $\text{DA}$ primitive produces the constraint $\pi_{A \leftarrow \{C\}}(R) = S$. Applying the $\text{Sub}$ primitive thereafter may produce the constraint $S \subseteq T$. Composing the two yields the constraint $\pi_{A \leftarrow \{C\}}(R) \subseteq T$.

### 4.2.1 Event Vectors

In each run, the schema evolution simulator applies a sequence of edits. Each edit consists of executing a particular schema evolution primitive followed by mapping composition.

An event vector specifies the proportions of primitives of a certain kind appearing in an edit sequence. In our study we use four event vectors that are depicted in Figure 2. For example, in the Default vector the proportion of $\text{AR}$ is 1, the proportion of $\text{AA}$ is 2 meaning that an attribute is added twice as often as a relation. That is, in an edit sequence of size 100 the $\text{AR}$ primitive is applied on average $1 \cdot 100 ÷ (0.2 + 2 + 1 \cdot 16) = 5.5$ times.

We are not aware of studies that investigated the frequency of schema evolution primitives used in real-world evolution scenarios. Thus, we assume that all primitives are applied with the same frequency, with the exception of adding attributes ($\text{AA}$ is twice as frequent) and dropping relations ($\text{DR}$ is five times less frequent).

The Default event vector exercises all schema evolution primitives. In contrast, the Forward vector focuses on those primitives where the input relations determine the output relations. Intuitively, the mapping produced by using the Forward vector defines the evolved schema as a view on the original schema. In the Backward vector, the original schema is defined as a view on the evolved schema. The Preserving vector contains only the ‘lossless’ schema evolution primitives; all information of the original schema is preserved in the evolved schema and vice-versa.

### Table 1: Event vectors

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Default</th>
<th>Fwd</th>
<th>Bwd</th>
<th>Preserving</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{AR}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\text{DR}$</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>$\text{AA}$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$\text{DA}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\text{D}_f, \text{H}_f, \text{V}_f, N_f, \text{Sub}$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\text{D}_b, \text{H}_b, \text{V}_b, N_b, \text{Sup}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\text{D}, \text{H}, \text{V}, \text{N}$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2: Event vectors

### Figure 3: Parameters of experimental setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimal initial relation arity</td>
<td>2</td>
</tr>
<tr>
<td>Maximal initial relation arity</td>
<td>10</td>
</tr>
<tr>
<td>Minimal initial key size</td>
<td>1</td>
</tr>
<tr>
<td>Maximal initial key size</td>
<td>3</td>
</tr>
<tr>
<td>Size of constant pool used in primitives</td>
<td>10</td>
</tr>
<tr>
<td>Exponential blowup threshold factor</td>
<td>100</td>
</tr>
<tr>
<td>Default schema size</td>
<td>30</td>
</tr>
<tr>
<td>Default size of edit sequence</td>
<td>100</td>
</tr>
<tr>
<td>Number of edit runs to average over</td>
<td>100</td>
</tr>
<tr>
<td>Number of composition runs to average over</td>
<td>500</td>
</tr>
</tbody>
</table>

4.3 Study on Synthetic Mappings

The output mapping produced by the composition algorithm may be exponentially larger than the input mappings. To control the exponential blowup, the algorithm aborts whenever the output-to-input size ratio exceeds a certain factor (100, in our study). The size of mappings is measured as the total number of operators across all constraints. In our experiments, the algorithm fails to eliminate only about 1% of symbols due to exponential blowup.

### 4.3.1 Schema Editing Scenarios

Figure 4 shows the success rate of our algorithm in composing the mappings of an edit sequence. Four configurations are examined (‘no keys’, ‘keys’, ‘no unfolding’, ‘no right compose’). The data for each configuration was obtained as follows: in each run, 100 edits are applied to a randomly generated schema of a default size 30. The mappings produced in each edit are composed. The proportions of primitives in the edit sequence correspond to the Default event vector. The numbers shown for each config-

![Figure 4: Eliminated symbols per primitive](image-url)

14
In the ‘no keys’ and ‘keys’ configurations all features of the algorithm are exploited; in the latter the relations may contain keys. In the ‘no unfolding’ configuration, the View Unfolding module of the algorithm is disabled. Similarly, for ‘no right compose’. Vertical partitioning is not applicable if no keys are present, therefore the respective bars remain empty in all configurations but ‘keys’.

The figure shows that the symbols introduced by some primitives are easier to eliminate than others. \(H_f, V_f\), and \(N_f\) are the ‘hardest’ primitives. Adding keys to schemas does not significantly affect the symbol-eliminating power of the algorithm. Turning off view unfolding or right compose weakens the algorithm substantially.

The execution times for this experiment are in Figure 5. Disabling view unfolding or adding keys increases the running time significantly. When a keyed relation is eliminated, its key constraints need to be propagated to all expressions that reference it. So, mappings produced in the ‘keys’ setting contain on average 218 constraints with 4,300 relational operators, as opposed to 95 constraints with 800 operators for ‘no keys’. The median execution time per run (for all 100 edits) are around 0.2 sec for ‘no keys’ and ‘no right compose’, 2.8 sec for ‘keys’, 2.1 sec for ‘no unfolding’ — a tenfold increase (not shown in the figure). The reason for reporting the median time instead of the average time is exemplified in Figure 6: most runs have close running times except for a few outliers that skew the average. This graph is characteristic across all experiments.

Figure 7 illustrates the impact of inclusion constraints (vs. equality constraints) on a few selected schema evolution primitives and the overall running time. Each edit sequence corresponding to value \(x\) on the \(x\)-axis is constructed by using the event vector obtained as a copy of the Default vector in which the proportion of Sub and Sup primitives is set to \(x\). On average, the composition tasks become more difficult (the total fraction of symbols eliminated in all edits decreases) since the effectiveness of view unfolding drops. However, the algorithm fails faster as it detects the symbols that cannot be isolated on either the left or right side of constraints, and so the overall running time decreases.

4.3.2 Schema Reconciliation Scenarios

Figure 8 depicts a schema reconciliation scenario. Each task consists of composing two mappings produced by the simulator for two sequences of 100 edits each. To obtain first-order input mappings, only those edit sequences produced by the simulator were considered in which all symbols were eliminated successfully.

The data points shown in the figure were obtained by averaging over 500 composition tasks. Increasing the size of the intermediate schema (which contains the symbols to be eliminated) simplifies the composition problem. This is an expected result since the simulator applies the edits to randomly chosen relations, so a larger intermediate schema makes it less likely that the constraints in the two input mappings interact, i.e., mention the same symbols. For this same reason, increasing the number of edits makes the composition problem harder; the fraction of eliminated symbols drops while the running time increases (see Fig-
Disabling view unfolding or right compose has a similar effect on the composition performance as in the schema editing scenarios, as shown in Figure 8. 10-20% fewer symbols get eliminated. In addition, the execution time for disabling view unfolding increases by about an order of magnitude (not shown in the figure). Disabling left compose (not shown) does not have a noticeable impact of the symbol-eliminating power of the algorithm for the settings used in the study, mostly because the simulator does not introduce any relational operators beyond $\sigma, \pi, \cup, \bowtie, \times$.

Mapping composition may necessarily fail to eliminate symbols from the intermediate schema. As illustrated in Figure 10, it is beneficial to keep those symbols around as ‘bystanders’ in second-order constraints even if ultimately we are interested in obtaining first-order mappings. We found that subsequent mapping compositions (or other operations on mappings) may facilitate symbol elimination at a later stage. To quantify this effect, we configured the simulator to retain second-order mappings, which may contain symbols that could not be eliminated upon edits.

Figure 10 illustrates a schema reconciliation problem for schema size 30 and input mappings generated for edit sequences of length 100. No keys are used in the Base configuration and the input mappings are first-order. In Base+SO, second-order mappings are allowed. The figure shows that close to 20% of second-order symbols in the input mappings can be eliminated in the final composition. Moreover, going second-order increases the number of symbols from the intermediate schema that can be eliminated successfully. A similar effect is observed when keys are allowed.

Finally we discuss the impact of varying the event vectors. The labels B, F, and P in Figure 10 correspond to the event vectors Bwd, Fwd, and Preserving of Figure 2. For example, in task BB the Bwd vector is used for generating both input mappings. Consequently, BB corresponds to composing mappings each of which is a view pointing inward, i.e., expressing the intermediate schema in terms of an evolved schema. Close to 100% of symbols from the intermediate schema get eliminated in this setting, and about a third of bystander symbols for second-order mappings. In BF and FB tasks, the mappings are views pointing in the same direction. If the views point to the opposite directions (FF), composition turns out to be harder. In contrast, close to 90% of symbols get eliminated for the information-preserving vector P; the bystander bar is not shown since the input mappings produced by the simulator do not contain any second-order constraints.

5 Conclusions

This paper presented a new algorithm for composition of relational algebraic mappings that need not be source-to-target. It has many new features: it makes a best effort to eliminate as many symbols as possible; it can handle unknown or partially known operators, including ones that are not monotonic in all of their arguments; it introduces the left-compose transformation; and it is highly extensible. We demonstrated its value by an experimental study of its effectiveness on a large set of synthetically-generated mappings.

References


