Query Rewriting for Semistructured Data

Yannis Papakonstantinou*
University of California, San Diego
yannis@cs.ucsd.edu

Vasilis Vassalos†
Stanford University
vassalos@cs.stanford.edu

Abstract

We address the problem of query rewriting for TSL, a language for querying semistructured data. We develop and present an algorithm that, given a semistructured query q and a set of semistructured views $\mathcal{V}$, finds *rewriting* queries, i.e., queries that access the views and produce the same result as q. Our algorithm is based on appropriately generalizing *containment mappings*, the *chase*, and *unification* – techniques that were developed for structured, relational data. We also develop an algorithm for equivalence checking of TSL queries.

We show that the algorithm is sound and complete for TSL, i.e., it always finds every TSL rewriting query of q, and we discuss its complexity. We extend the rewriting algorithm to use available structural constraints (such as DTDs) to find more opportunities for query rewriting.

We currently incorporate the algorithm in the TSIMMIS system.

1 Introduction

Recently, many semistructured data models, query and view definition languages have been proposed [20, 36, 8, 3, 38, 41, 1, 16] and are used for querying and management of Web data [16, 3, 38], biological databases [51], integration of heterogeneous data [20, 33], etc. The Harvest Information Discovery and Access system [5] and the Lotus Notes messaging and groupware system [34] are two important precursors to the "schema-less" approach to data modeling.

Semistructured models are necessary because of the flexible nature of non-database information systems. In particular, semistructured models are useful in the context of Web-based sources; Web data very often have irregular, partial or only implicit structure. The semistructured model XML [6] is emerging as the new standard for the modeling and exchange of Web data.

As it has been the case in the relational world, rewriting of semistructured queries using views is a fundamental query processing and optimization tool for semistructured queries. In this section we first

*Research partially sponsored by NSF, under Award Number IRI-9712239, the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339, and equipment donations by Intel Corporation.

†Research partially supported by NSF grant IRI–96–31952, Air Force contract F33615–93–1–1339 and the L. Voudouri Foundation.

present an abstract version of the rewriting problem and consequently we describe its applications, including a rewriter we build for the TSIMMIS system [20].

The Rewriting Problem At a sufficient level of abstraction the rewriting problem faced by the applications listed below is as follows: Given a query q accessing a semistructured database[1] D and a set of views $\mathcal{V} = \{V_1, \ldots, V_n\}$ over D, find *rewriting* queries, where a rewriting query of q given $\mathcal{V}$ is a query that accesses at least one view of $\mathcal{V}$ and returns the same result as q.[2] If the rewriting query uses views only (i.e., it does not access directly the database D) then it is called a *total rewriting* query.

Applications of the Rewriting Algorithm Semistructured models have been used by repositories that store semistructured data [36, 8] and by mediators that integrate heterogeneous information [56, 26, 42, 16, 37].

The importance of rewriting algorithms in mediators and repositories of relational systems, as described below, is a witness to the many applications they'll have in the semistructured world.

1. Relational query rewriting algorithms are used for answering queries using materialized views [28, 27, 49] and the query cache [25].

2. Views have been used in mediator systems to describe the source contents [29]. Furthermore, the different and limited query capabilities of the sources are often described by "views" where the constants are parameterized. For example, the parameterized view

   ```
   SELECT * FROM R WHERE R.A=$X
   ```

   where `R` resides at source S, declares that S can answer queries that pick all attributes of `R` and have `R.A` be bound to a constant. Then a query over the source data has to be rewritten to use correctly the contents and capabilities of the sources, i.e., to correctly use the available views [47, 30, 55, 24, 43, 14]. Indeed, in that case the query has to access *only* views and hence we need a total rewriting query.

The above points highlight the importance of rewriting algorithms in relational databases and mediators. We believe that rewriting algorithms will be equally important for semistructured databases and mediators.

Use of the Rewriting Algorithm in the TSIMMIS System: Capability-Based Rewriting and Cached Queries A TSIMMIS mediator integrates semistructured data from multiple heterogeneous information sources into a virtual view V_m — not to be confused with the views used by the rewriting algorithm. The general integration architecture is shown in Figure 1.

For example, a bibliographic mediator may combine the data of multiple bibliographic sources into a single "union" view. At run time, given a user query, the mediator decomposes it into multiple queries which refer to the source data. However, these bibliographic sources are accessible through interfaces that have

[1] The database may be distributed over multiple sites.

[2] We formalize the concept of "same result" and the definition of a rewriting query in Section 3.

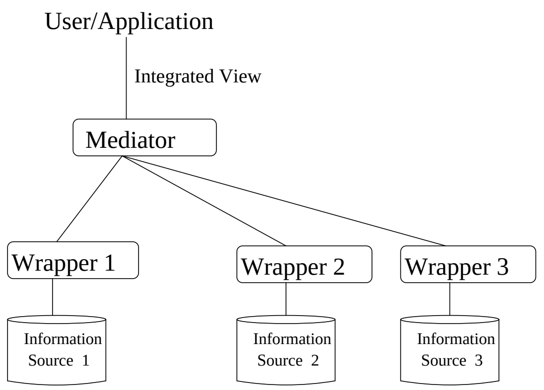

Figure 1: The TSIMMIS integration architecture

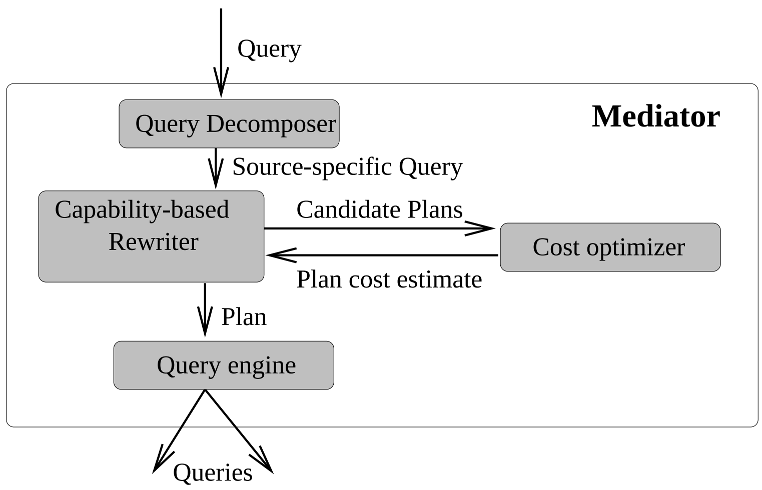

Figure 2: Mediator architecture

varying query capabilities; the queries emitted by the mediator must conform to these capabilities. Let us further illustrate this issue using an example.

The user query requests all "SIGMOD 97" publications. Then the mediator will decompose the user query into multiple "SIGMOD 97" queries where each one of them is source-specific, i.e., it refers to one source only (see Figure 2). To do the decomposition correctly and efficiently the mediator must figure out, using the capabilities of the underlying sources, how to extract the necessary information from the sources. This decision is made in the Capability-Based Rewriter (CBR) module. In our running example, if one source only supports queries on "year", the CBR will decide that a query that retrieves the "97" publications will be sent to this source. The rest, i.e., filtering for "SIGMOD", will be done at the mediator.

After such decisions are made, and the mediator formulates a query plan that respects the query capabilities of the sources, each query is sent to a wrapper, where it is translated into the native query language of the corresponding source. Then the individual query results, namely the "SIGMOD 97" publications each source contains, are collected, the information about each of them is appropriately consolidated into one entity by the mediator and the combined result is presented to the user.

The TSIMMIS system uses parametrized views to describe query capabilities. The mediator employs a simplified version of our rewriting algorithm to accomplish its task [33]. We are in the process of implementing the fully general query rewriting algorithm presented in this paper in the TSIMMIS system. Note that the existence of parameters in the views does not seriously affect the complexity of the problem [47, 55]. For presentation clarity we work in this paper with plain semistructured views - as opposed to parametrized ones.

Use of the Rewriting Algorithm in semistructured repositories: Our algorithm can be used to answer queries using materialized views and cached queries of repositories for semistructured data, such as Lore [36]. For example, if a cached query result contains all "SIGMOD" publications, our rewriting algorithm can create a rewriting query where "SIGMOD 97" publications are obtained by filtering the cached query for "1997" publications. Notice that the rewriting algorithm only needs the query and the cached query statements - it does not need to examine the source data. The cached queries play in this case the role of views.[3]

Materialized views and cached queries were the main original motivation for relational query rewriting [57, 11] and we believe they are as important for semistructured databases. Indeed our algorithm is applicable to repositories of Web data stored using the XML [6] data model, which is very similar to our data model. The query language — TSL, for Tree Specification Language — that we are working with is very similar to recent proposals for an XML query language [13].

Web site management and structured Web search: Some recent work [16] has applied concepts from information integration to the task of building complex Web sites that serve information derived from multiple data sources. In this scenario, a Web site is a declaratively-defined *site graph* over the semistructured *data graph* of the contents of the information sources. If we only have access to the information through the Web site(s), queries asked over the data graph need to be rewritten as queries over the Web site structure and contents. The Web site definitions are just view definitions over the data graph; the necessary query rewriting can thus be handled by our algorithm.

Data Model and Query and View Definition Language In TSIMMIS we model semistructured data using the OEM data model [42, 46], which is a labeled graph model where the nodes are uniquely identified by object-id's. The data models used by other semistructured information systems and languages, such as Strudel [16], Lore [36], and UnQL [8], can be easily reduced to OEM. To express queries and views we have implemented and use TSL, a logic-based language that allows creation of semantic object-id's using Skolem functions [40, 35]. That feature gives TSL strong restructuring capabilities: a query result can be

[3] Of course, given the autonomy of the bibliographic sources and the mediator, the rewriting query may deliver a stale result to the user. Nevertheless this result may still be very useful to the user. For example, assuming that the sources do not delete information, the cached result contains correct answers to the user's query, and it can be quickly delivered. Then, while the user inspects the cached query, the system can collect the current result from the sources [50]. Furthermore, if an update-propagation system is in place it can account for the "deltas" between the cache and the sources [23, 58, 2]. In this paper we will not deal any further with these consistency issues. Instead we focus on the rewriting algorithm.

an arbitrary *answer tree*. For the majority of Web-based sources and for most of the practical applications described above, the restructuring capabilities of TSL are sufficiently powerful.

The resemblance of TSL to Datalog [52] facilitates the presentation of similarities and differences between the rewriting of relational conjunctive queries and semistructured TSL queries. Highlighting these similarities is one of the goals of this paper.

We will describe TSL in more detail in Section 2. Besides its restructuring capabilities, note that TSL allows querying and copying arbitrarily deeply-nested, schema-less data, and supports *label variables* that enable querying the "structure" of the data.

Algorithm The proposed algorithm solves the rewriting problem by outputting a finite set $\mathcal{Q}$ of rewriting queries, i.e., queries equivalent to q that have at least one condition referring to one of the views. Furthermore, for every rewriting query q_r that does not appear in $\mathcal{Q}$ there is a $q_r' \in \mathcal{Q}$ such that every view that is used by q_r' is also used by q_r. Under any reasonable cost model, q_r' will be as or more efficient (if it uses strictly fewer views) than q_r and hence we do not include q_r in $\mathcal{Q}$. From now on, we will say that the algorithm returns *all* rewriting queries, though we actually mean that it returns a set of rewriting queries $\mathcal{Q}$ with the above properties.

The algorithm operates in two steps: First it finds candidate rewriting queries. Then it retains the candidate rewriting queries which are equivalent to the original query. During the first phase, the algorithm uses a generalization of containment mappings [10] from the views to the original query q in order to narrow the space of candidate queries. Furthermore, the algorithm uses the *chase* technique [53] to deal with the key dependencies that hold because of object identity.

Testing the equivalence of the candidate rewriting query with the original query is also accomplished in two steps. First, the algorithm "composes" the rewriting query and the views to obtain an "expanded" query q_r' that is equivalent to the rewriting query and does not refer to the views any more. Then q_r' is tested for equivalence with the original query. To do the testing, we develop conditions for the equivalence of TSL queries that extend the conditions for equivalence of unions of conjunctive queries [10, 48] to queries with path expressions.

We extend the rewriting algorithm to make use of structural constraints on the source data. In particular, we consider constraints that can easily be expressed by standards such as the XML DTDs or the newly proposed XML-Data. The existence of such constraints allows us find rewritings in cases where, in the absence of constraints, the algorithm would fail.

Lessons and Results Our rewriting algorithm is based on extending containment mappings, the chase, and unification from the relational into the semistructured world. In doing so we benefit from a vast body of knowledge on relational query rewriting. Furthermore, we obtain insight on how to interface with the optimizer of the TSIMMIS system (see Figure 2).

The reader may wonder whether, given a reduction of semistructured data to relations, such as the one presented in [39], the conjunctive TSL rewriting problem can be fully reduced to the well-understood relational conjunctive query rewriting problem. The answer is negative because TSL queries can not be

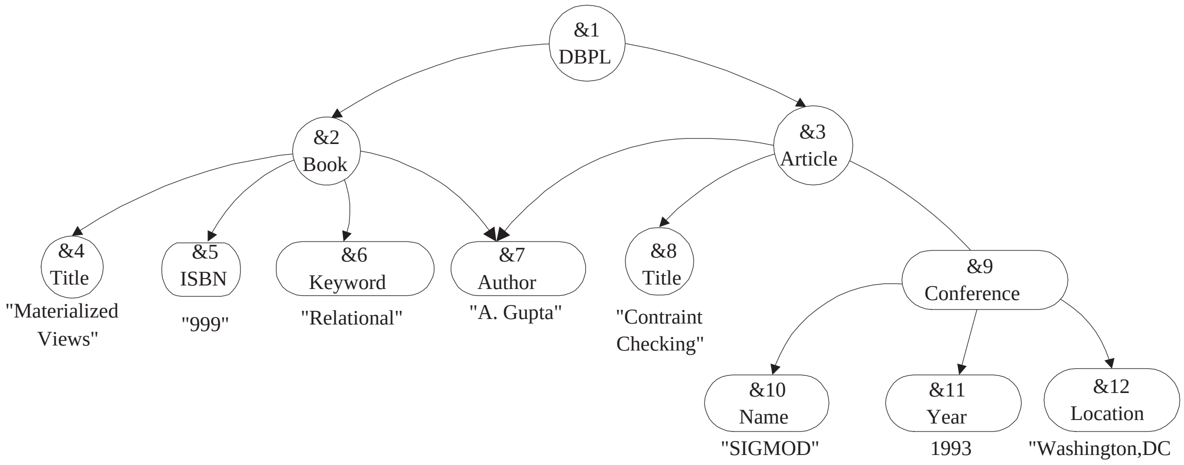

Figure 3: Example OEM objects

reduced to conjunctive relational queries. TSL queries and views are actually reducible to Datalog with function symbols and with a limited form of recursion. The function symbols are introduced to allow TSL to construct new objects through the use of semantic object-ids.[4] The recursion is necessary to copy arbitrarily deeply nested objects from the source to the query result [39]; this copying is an important feature for the integration of semistructured sources. Another factor that makes rewriting of semistructured queries harder than that of relational queries is of course the lack of schema that necessitates the use of label variables in the queries and the views. Finally, the existence of structural constraints gives more opportunities for query rewriting. We identified two situations where we can infer dependencies from structural constraints, and we extended the rewriting algorithm to use these dependencies.

Contents The following section introduces the OEM data model and our query language for semistructured data. Section 3 states the rewriting problem, describes our algorithm and describes mappings and query composition in TSL. Section 4 presents an algorithm for equivalence testing of TSL queries. Section 5 proves the correctness of our rewriting algorithm for TSL queries and views and discusses the complexity of the rewriting problem. Finally, Section 6 discusses related work and Section 7 offers some concluding remarks and discusses future work.

2 The OEM Data Model and the TSL Query Language

In the OEM data model, the data are represented as a rooted graph with labeled nodes (also called *objects*) that have unique object-id's. Figure 3 illustrates some bibliographic data represented in OEM. *Atomic* objects have an atomic value (e.g., `SIGMOD`) while the value of the other objects (called *set objects*) is the

[4] Other semistructured languages, such as StruQL [17] and XML-QL [13], have recently adopted the same approach.

set of objects (not just object-ids) pointed to by the outgoing edges. Notice that this definition is inherently recursive, since the value of an object is part of the object: the value of a set object o is essentially the OEM subgraph rooted o.[5] The roots of the graph are illustrated as top level objects. They are the starting points for querying the sources. Note that we ignore objects that are not reachable from the roots of the graph.

The object-id's are typically atomic data. In the general case they are terms from the Herbrand universe composed from

- a set of atomic data, which includes but is not necessarily confined to, the atomic data appearing as labels and values and
- an arbitrary set of uninterpreted function symbols. For example, `f(&10, ashish)` is a possible object-id, and the function symbol `f` "defines" the term.

Object-id's may be symbols with no particular meaning. In other cases object-id's may have a semantic meaning. For example, if the object is a Web page then it is typically a good idea to have the URL be the object-id. Furthermore, meaningful term object-id's can facilitate the integration tasks.

Even though OEM can model data that can naturally be represented as an arbitrary graph, we expect that in many applications, especially those dealing with XML data, data will instead be naturally represented as a directed acyclic graph, or a tree.

A TSL query is a *rule* that defines the query result using minimal model semantics. A rule consists of a *head* followed by a `:-` and a *body*, in the style of Datalog [53]. Intuitively, the head describes the result objects in the *answer graph*, whereas the body describes one or more conditions that must be satisfied by the source objects. The head and the body conditions are based on *object patterns* of the form <*object-id label value*>. The *value* field can be either a term (variable, atomic constant, or function symbol followed by a term list) or a *set* value pattern which contains zero or more object patterns. Terms that appear in an object-id field in the *head* of a TSL query must be unique. This restriction forces TSL to produce fresh object-ids for the objects in the query result. It also forces TSL to produce *answer trees* instead of arbitrary graphs as query results. We discuss removing this restriction (and the resulting language) in Section 6.

Semantics and power of TSL: We illustrate the semantics with the following example.

(Q1) `<fem(P) female {<f(X) Y Z>}> :- <P person {<G gender female> <X Y Z>}>@db`

The semantics of the above query are

if there is a tuple of *bindings* p, g, x, y and z for the variables `P`, `G`, `X`, `Y`, and `Z` such that
the data source `db` contains a `person` top-level (root) object identified by p,
the p object has a `gender` subobject with value `female` and object id g, and
the p object has a y subobject with value z and object id x
%the object p may also have sub-objects other than the g and the x
then the query result has

[5] Excluding o itself.

a `female` object, with object-id `fem`(p),

the `fem`(p) object has a y subobject with value z and object id `f`(x).

%the object `fem`(p) *may have subobjects other than y*

%because the result of another rule may "fuse" more subobjects into the object `fem`(p)

Note that z could be a subgraph of the data in `db`. The answer to query (Q1) is an object with a new, unique object-id and the structure denoted by the query head. In general, a TSL query can construct answer objects that are tree restructurings of source data, hence we refer to the result of a TSL query as an *answer tree*. Because of the copying semantics of TSL, (e.g., z above could be a subgraph of the data), the query result can actually be a graph: a constructed tree with (perhaps cyclic) subgraphs potentially hanging off some branches. Note finally that a TSL query may refer to more than one data source, e.g., one condition may refer to `db1` and a second one to `db2`.

Formally, for an OEM database D, let P_D be the set of all subgraphs[6] of D, O be the set of all object-ids in D, and C be the set of all labels and atomic values. Let V_O be the set of all object-id variables[7] and V_C be the set of all other (label and value) variables, with $V_O \cap V_C = \emptyset$. Let $V = V_O \cup V_C$ be the set of all variables. The meaning of the query body is the set of assignments $\theta : V \rightarrow O \cup C \cup P_D$ that satisfy all conditions in the body. Each assignment maps object-id variables to O, label variables to C and value variables to $C \cup P_D$.

The meaning of the query head is as follows. We create and label the new nodes of the answer tree, and make the top-level object pattern of the query the root of the answer tree. In particular, for each object pattern <$f(X_1, \ldots, X_m)$ L V>. in the query head, and for each assignment θ above, create a new object with object-id $f(\theta(X_1), \ldots, \theta(X_m))$, label $\theta(L)$ and value $\theta(V)$. If instead of V, the object pattern above has $\{o_1 \ldots o_n\}$, the value of the created object is $\{\theta(o_1), \ldots \theta(o_n)\}$.

Notice that when two assignments produce the same term as the object-id of an object, the same object is "returned", and the values of the two objects are "fused".

TSL can be translated to Datalog with function symbols and limited recursion over a fixed schema. It can be shown to be less expressive than StruQL and thus less expressive than linear datalog [17]. TSL queries can be computed in polylogarithmic parallel time with polynomially many processors (i.e., TSL $\subset$ QNC).

In the rest of this paper, we only consider positive TSL queries without cyclic object patterns in the body conditions (i.e., without object patterns that look for cycles in the OEM database). To simplify the presentation, we focus on normal form queries, defined next. Every TSL query can be easily converted into normal form and hence the focus on normal form does not limit the power of the language.

Definition: Normal Form TSL Queries are the TSL queries in whose body all set-valued *value* fields contain *at most* one object pattern. Additionally, a normal form query with just one condition in its body is called a *single path* query. □

The query (Q1) can be easily transformed into the following normal form query:

[6] Remember that the value of a set object is essentially the OEM subgraph rooted at that object.

[7] Object-id variables are variables appearing in the object-id field of object patterns.

```
(Q2) <fem(P) female {<f(X) Y Z>}> :- <P person {<G gender female>}>@db
                                     AND <P person {<X Y Z>}>@db
```

Safe TSL queries A TSL query is safe if every variable appearing in the query head also appears in the query body. Thus, the same simple syntactic test that is used by [53] to define safety of conjunctive queries can be used to define safety in TSL. In the remainder of this paper we are only discussing safe TSL queries.

TSL views A TSL view is defined simply by a TSL query. Each view defines its own OEM database, with its own space of unique object-id's. That can easily be accomplished for example by qualifying the object-id's by the name of the view.

It is important to point out that TSL has features essential for querying and integrating semistructured data, namely the ability to query and copy arbitrarily nested schema-less data, the ability to restructure such data through the use of semantic object-id's, and the ability to query the "structure" of the data through the use of label variables.

3 TSL Query Rewriting

Given a TSL query Q referring to an OEM database D and conjunctive views $\mathcal{V} = V_1, \ldots, V_n$, also referring to D, the rewriting problem is to find a TSL query Q' such that (i) Q' refers to at least one of $V_1, \ldots, V_n$ and (ii) for all OEM databases D, the result of Q is equivalent to the result of Q'. (See definition of equivalence below.)

We call Q' the *rewriting* query. In general, there may be more than one rewriting queries. We start our discussion with a straightforward definition of equivalence of OEM databases.

Equivalence of two OEM databases D_1 and D_2 Two OEM databases D_1 and D_2 are equivalent if they are *identical*, i.e., they have the same set of object-id's and for every object-id x the two objects identified by x in D_1 and D_2 (i) have the same label l (ii) both of them have an atomic value or both of them have a set value (iii) if they are atomic objects they have the same atomic value v and (iv) if they are set objects they have identical sets of subobjects.

Apparently the above definition carries to equivalence of query results and views. It is possible to define OEM database equivalence up to object-id renaming. We discuss this issue in Section 6.

3.1 Rewriting of Queries with Single Path Condition

We informally present an algorithm which decides whether a query Q having one single path condition can be rewritten using a single view V that has one or more path conditions. This algorithm, though a special case of the complete rewriting algorithm (see Section 3.5), illustrates the basic steps of our technique. The general algorithm is proven sound and complete for TSL in Section 5.

Step 1: Find Candidate Queries We first find mappings from the view to the condition and then we develop a candidate query for each mapping. Note that for the special case of queries with a single path condition there may be at most one mapping and consequently at most one candidate query.

Step 1A: Find Mappings Find, if it exists, the *mapping* from the body of V to the body of Q. Our mappings extend [10] to cope with object nesting and are formally defined in Section 3.2. If a mapping exists, then we can be sure that, if there is a variable binding that satisfies the body of Q, then there is also a binding that satisfies the body of V. Hence mappings are a necessary condition for the relevance of the view to the query condition. Furthermore, the mapping indicates which conditions of Q do not appear in V; these conditions will have to be checked by the rewriting query. Notice that there can be *at most* one mapping from the body of V to the one single path condition in the body of Q. However, in the general case (Section 3.5) we may have multiple mappings.

Example 3.1 Consider the view (V1), which restructures the `person` objects of `db` into objects that "group" their labels in `property` subobjects and their values in `value` subobjects.[8] Notice that (V1) "loses" information in the sense that it only shows the labels and values that appear in `db` but the label-value correspondence has disappeared. Queries such as (Q3), that ask whether the value `leland_stanford` appears in the database, can be answered using the view (V1) because they do not need information on the label-value correspondence. The example shows how our algorithm finds a rewriting query for (Q3).

```
(V1) <g(P') person {<prop(P',Y') property Y'> <h(X') value Z'>}> :-
                    <P' person {<X' Y' Z'>}>@db
(Q3) <f(P) stanford yes> :- <P person {<X Y leland_stanford>}>@db
```

The only mapping from the body of (V1) to the body of (Q3) is (M2). Intuitively, (M2) indicates that the condition `Z' = leland_stanford` must be enforced on the view in order to get objects relevant to the query.

```
(M2) [P' ↦ P, X' ↦ X, Y' ↦ Y, Z' ↦ leland_stanford]
```

□

Step 1B: Generate Candidate Queries Apply the mapping to V, resulting in an "instantiation" of V, namely V'. Then build the rewriting query Q' as follows: The head of Q' is identical to the head of Q. The body of Q' is the head of V'.

Example 3.1 continued The only candidate rewriting query (Q4) is created from the head of (Q3) and the result of applying (M2) to the head of (V1).

```
(Q4) <f(P) stanford yes> :- <g(P) person {<prop(P,Y) property Y>
                                          <h(X) value leland_stanford>}>
```

[8] The examples try to illustrate the technical issues concisely and clearly, and sometimes can appear artificial.

Step 2: Test Correctness of Candidate Query Check whether the composition of V and Q', denoted by $V \circ Q'$ is equivalent to Q. Step 2 is accomplished in two sub-steps:

Step 2A: Computation of Composition The composition $V \circ Q'$ of the rewriting query with the view is computed. We compute $V \circ Q'$ using a query-view composition algorithm based on extending resolution and unification for semistructured data. This algorithm in essence takes exponential time in the size of the query. The composition algorithm is illustrated using an example below, and is formally presented in Section 3.7.

Step 2B: Testing Equivalence of $V \circ Q'$ with Q The general idea of equivalence testing is to find (1) a mapping that maps $V \circ Q'$ into Q, i.e., (i) it maps the head of $V \circ Q'$ into the head of Q and every condition of $V \circ Q'$ is mapped into a condition of Q and (2) a mapping that maps Q into $V \circ Q'$. Note that the $V \circ Q'$ and Q have to be in normal form in order to test equivalence as described above.[9]

Example 3.1 continued We test whether (Q4) is a valid rewriting query by first transforming it into the normal form $(\mathrm{Q4})_{norm}$, then composing it with (V1), and finally comparing the resulting query $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ to (Q3). Indeed, $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ *is* equivalent to (Q3) because (i) the mapping (M3) maps $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ to (Q3) and (ii) the mapping (M4) maps (Q3) to $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$).

```
(Q4)norm <f(P) stanford yes> :- <g(P) person {<prop(P,Y) property Y>}>
                                AND <g(P) person {<h(X) value leland_stanford>}>
(V1)∘(Q4)norm <f(P) stanford yes> :- <P person {<X′ Y Z′>}>
                                     AND <P person {<X″ Y″ leland_stanford>}>
(M3) [P ↦ P, X′ ↦ X, Y ↦ Y, Z′ ↦ leland_stanford, X″ ↦ X, Y″ ↦ Y]
(M4) [P ↦ P, X ↦ X″, Y ↦ Y″]
```

Set Mappings The rewriting query may have to apply a "subobject membership" condition on a value variable. To handle this case, our mappings are extended to map a variable to a set pattern.

Example 3.2 Consider the query (Q5) and the view (V1). It is clear that Z′ must bind to set values that contain a <Z last stanford> subobject. The algorithm captures this intuition by developing the mapping (M5) from the body of (V1) to the body of (Q5). Notice that Z′ is mapped to {<Z last stanford>}>.

```
(Q5) <f(P) stanford yes> :- <P person {<X Y {<Z last stanford>}>}>@db
(M5) [P' ↦ P, X' ↦ X, Y' ↦ Y, Z' ↦ {<Z last stanford>} ]
(Q6) <f(P) stanford yes> :- <g(P) person {<prop(P,Y) property Y>
                                        <h(X) value {<Z last stanford>}>}>@V1
```

(Q6) is the candidate query created from the head of (Q5) and the result of applying (M5) to the head of (V1). □

[9] The general equivalence testing algorithm is actually more intricate, because of the existence of object-id's. For a full description of the equivalence testing algorithm for TSL see Section 4.

Mappings are necessary but not sufficient for the existence of a rewriting query as the following example illustrates. That is why a containment test is needed, as in Step 2B of the algorithm.

Example 3.3 Consider the query (Q7) and the view (V1).

```
(Q7) <f(P) stanford yes> :- <P person {<X name {<Z last stanford>}>}>@db
```

Intuitively, there is no rewriting query for (Q7) because the view "loses" the correspondence between labels and values. Hence, if the database contains a `name` attribute and a value v containing the `<last stanford>` subobject it is impossible for the rewriting query to discover whether there is a `name` object with value v or `name` and v appear in different objects of the database. Notice that despite the non-existence of a rewriting query there is the mapping (M6). Based on this mapping the algorithm derives the candidate rewriting query (Q8). However, the composition of the candidate rewriting query with the view results in the query (Q9) which is not equivalent to the original query (Q7). Notice that `name` is the label of the object `X'` while `<last stanford>` is a subobject of another object `X''`.

```
(M6) [P' ↦ P, X' ↦ X, Y' ↦ name, Z' ↦ {<Z last stanford>} ]
(Q8) <f(P) stanford yes> :-<g(P) person {<prop(P,Y) property name>
                                         <h(X) value {<Z last stanford>}>}>@V1
(Q9) <f(P) stanford yes> :- <P person {<X' name Z'>}>@db
                            AND <P person {<X'' Y'' {<Z last stanford>}>}>@db
```

□

In the following subsection, we formally define mappings and describe an algorithm for mapping composition. Subsection 3.3 extends the chase for set variables, which as will see is necessary to deal with the key dependency on object-id. Subsection 3.4 discusses how the algorithm can exploit structural constraints, such as DTDs, that are known about source data. Subsections 3.5 and 3.6 present a general algorithm for query rewriting.

3.2 Mappings

In this section, we formally define mappings and mapping composition.

Definition: A Mapping is a set of structures of the form *variable* $\mapsto$ *rhs* that transforms a TSL object pattern (or a conjunction of TSL object patterns or a TSL rule) to another. *Rhs* can be a term or a set of TSL object patterns $\{pattern_1, \ldots, pattern_n\}$. The right hand sides cannot contain variables that appear in the left hand sides of $\mapsto$. □

Definition: Valid Application of a Mapping on an OEM Object Pattern The result of applying a mapping on an OEM object pattern is a pattern where every variable which appears in the left hand side is replaced by the corresponding right hand side. The mapping is applicable to the object pattern if (i) the resulting pattern has valid OEM syntax, i.e., set patterns do not appear in object-id or label positions, (ii) is compatible with key dependencies imposed by the object-id's. □

The composition of two mappings θ_1 and θ_2, denoted by $\theta_1 \circ \theta_2$, is derived by applying θ_2 to the variables and set patterns on the right hand side of θ_1, deriving θ', and subsequently concatenating θ_2 and θ'.

Definition: Mapping Composition The composition $\theta_1 \circ \theta_2$ is a mapping θ consisting of (i) all $V \mapsto rhs$ structures of θ_2 and (ii) for every $V \mapsto rhs$ structure of θ_1, θ includes a structure of the form $V \mapsto rhs'$ and $rhs' = \theta_2(rhs)$. □

The complete algorithm for discovering mappings from a set of single path conditions to another set of single path conditions appears in Appendix B.

3.3 Extending the chase for set variables

Object identity introduces a functional dependency in OEM (key dependency from the object id to the label and value). Moreover, structural constraints introduce functional dependencies, as we will see in the next subsection. The rewriting algorithm uses the chase technique [53] to deal with these dependencies. The technique has to be extended for the case of variables that can bind to sets. In what follows, we motivate the need for and present our extension to the chase, presented for the case of key dependencies on object-id. The extension applies in general to any functional dependency with value variables in the right hand side.

Example 3.4 Consider (Q13) and (Q10) below.

```
(Q10) <f(P) stan_student V> :- <P person {<U university stanford>}>@db
                               AND <P person V>@db
```

(Q10) is equivalent to (Q13), since `V` is a set variable. However, our algorithm, as described so far, will erroneously not discover a rewriting query because there is no mapping from the condition of (Q13) to the second condition of (Q10). Using the key dependency on object-id, we can infer that `V` is a set variable and transform (Q10) to (Q13). Notice how the "set" variable is transformed into a set pattern. □

Recall that TSL queries are not allowed to contain cyclic object patterns. This is necessary for the described simple extension to the chase to terminate.

Object-id dependency chase extension Let o_1, o_2 be object patterns of a query q with the same term in the object-id field.

- If o_1 and o_2 have L_1, V_1 and L_2, V_2 in their label and value field respectively, then we replace all occurrences of L_2, V_2 in q with L_1, V_1 respectively.
- If o_1 has object patterns $\{o_i, \ldots, o_j\}$ in its value field and o_2 has V_2, then replace all occurrences of V_2 in q with {`<X Y Z >`}, where X, Y, Z are variables not appearing in q.
- If o_1 has $\{o_i, \ldots, o_j\}$ in its value field and o_2 has $\{c_k, \ldots, c_m\}$, replace the value fields of both o_1 and o_2 with $\{o_i, \ldots, o_j, c_k, \ldots, c_m\}$.
- If one of o_1, o_2 have a constant in one of the fields, and the other has a variable, replace all occurrences of that variable in q with the constant.

- If both o_1 and o_2 have constants in one of the fields, then, if the constants are different, halt with an error (this query cannot be chased to an equivalent query satisfying the object-id key dependency). If the constants are the same, do nothing for this field.
- If o_2 is identical to o_1, drop o_2 from q.

In order to "chase" functional dependencies that do not involve value variables, we can use the "regular" chase rule.

3.4 Structural constraints and query rewriting

Semistructured data are often accompanied by constraints that partially define the structure of objects. Such structural constraints can be expressed as a DTD, a DataGuide [22] or an XML-Data "schema". For instance, we could know that the data in source `db` in the previous examples conform to the following DTD:[10]

```
<!ELEMENT person (name, phone, address, affiliation*)>
<!ELEMENT name (last, first, middle?, alias?)>
<!ELEMENT alias (last, first)>
<!ELEMENT address CDATA>
<!ELEMENT phone CDATA>
<!ELEMENT last CDATA>
<!ELEMENT first CDATA>
<!ELEMENT middle CDATA>
```

This DTD describes in a flexible way the structure of the source data. For example, it specifies that objects labeled `person` have exactly one subobject each with labels `name, phone` and `address`, and zero or more `affiliation` subobjects. It also specifies that subobjects `phone` and `address` are atomic. Given such a DTD, we can infer information in the form of dependencies between labels or object-ids, that will allow the rewriting algorithm to discover rewritings in cases where it would have otherwise failed.

Example 3.5 Given the above DTD, we can infer automatically that in `db` the only subobject of a `person` object with a `last` subobject is a `name` object. Therefore, if we look at (Q9) in Example 3.3, `Y''` has to be `name`. Moreover, there exists a "labeled" functional dependency from object-id P with label `person` to object-id X with label `name`, since according to the DTD a `person` object has exactly one `name` subobject. This implies that `X''` has to be `X'` (by application of the *chase* rule). Therefore (Q9) can be rewritten as

```
(Q11) <f(P) stanford yes> :- <P person {<X' name Z'>}>@db
                             AND <P person {<X' name {<Z last stanford>}>}>@db
```

Finally, we chase the dependency on `P` using the chase extension described previously to derive (Q12). It should be obvious that (Q12) is equivalent to (Q7), and therefore a valid rewriting query.

[10] Since OEM does not support order, we ignore the order in the DTD description as well.

```
(Q12) <f(P) stanford yes> :- <P person {<X' name {<Z last stanford> <A B C>}>}>@db
```

□

As illustrated in the previous example, we identify two cases where information can easily be inferred from a structural description, such as a DTD, or an XML-Data "schema":

- (label inference) Given a "path expression" of labels `a.?.c`, if the structural constraint specifies that the only subobject of an `a` object with a `c` subobject is a `b` subobject, we can infer that ? = `b`.
- (functional dependency) If the structural constraint specifies that objects labeled `a` have only one subobject labeled `b`, we can infer the functional dependency between object-id variables $X_{\mathtt{a}} \rightarrow Y_{\mathtt{b}}$.

The rewriting algorithm takes advantage of this information by performing label inference and the chase on the query, the views and the candidate queries, again as illustrated in Example 3.5. It is straightforward to show that applying label inference and the chase always terminates in time polynomial to the length of the queries and the constraints description. Moreover, it is easy to show that label inference and the chase do not affect the soundness of the rewriting algorithm.

In the presence of structural constraints, there is clearly more opportunity for query simplification and query rewriting. This is the subject of future work.

3.5 General case of query rewriting

We now treat the general case of the query rewriting problem, with any number of views in $\mathcal{V}$ and any number of conditions in the body of the query Q. We apply the *chase* technique to take functional dependencies into account. For the sake of simplicity, the following example uses a view set with only one view. The method used generalizes trivially to view sets of any size; the algorithm described in subsection 3.6 covers the general case.

Example 3.6 Consider the following view (V7). Notice that the semantic object-ids of `property` and `value` objects retain information about the object that originally had that property and value. Then consider query (Q13).

```
(V7) <view(P') person {<prop(X') property Y'> <val(X') value Z'>}> :-
                                          <P' person {<X' Y' Z'>}>
(Q13) <f(P) stan_student {<X Y Z>}> :- <P person {<U university stanford>}>@db
                                          AND <P person {<X Y Z>}>@db
```

Intuitively, (Q13) can be answered using only (V7) as follows: First use (V7) to find the `P`'s that have a "university" subobject with value "stanford." The mapping (M8) from the body of (V7) to the first condition of (Q13) implies that this is possible. Then for every `P` that qualifies, pick all its subobjects `<X Y Z>`. Mapping (M9) from the body of (V7) to the second condition of (Q13) implies that this is also possible. Then, the head of the rewriting query (Q14) is the head of (Q13) and the body of (Q14) is the conjunction of $\theta_8(head(V7))$ and $\theta_9(head(V7))$.

(M8) θ_8 = $[\texttt{P}' \mapsto \texttt{P},\ \texttt{X}' \mapsto \texttt{U},\ \texttt{Y}' \mapsto \texttt{university},\ \texttt{Z}' \mapsto \texttt{stanford}]$

(M9) θ_9 = $[\texttt{P}' \mapsto \texttt{P},\ \texttt{X}' \mapsto \texttt{X},\ \texttt{Y}' \mapsto \texttt{Y},\ \texttt{Z}' \mapsto \texttt{Z}]$

(Q14)
```
<f(P) stan_student {<X Y Z>}> :-
    <view(P) person {<prop(X) property Y> <val(X) value Z>}>@V7 AND
    <view(P) person {<prop(U) property university> <val(U) value stanford>}>@V7
```

Let us now check whether (Q14) is a valid rewriting query. That means transforming (Q14) into normal form and checking whether (Q15)=(V7)$\circ$(Q14)$_{norm}$ is equivalent to (Q13).

(Q15)
```
<f(P) stan_student {<X Y Z>}> :-
        <P person {<X Y Z'>}>@db AND <P person {<X Y' Z>}>@db AND
        <P person {<U university Z''>}>@db AND <P person {<U Y'' stanford>}>@db
```

Notice that unless we make use of the key dependency $Oid \rightarrow Label\,Value$ there is no mapping from the body of the query (Q13) to the body of (Q15). By *chasing* (Q15), we infer that $\texttt{Y} \equiv \texttt{Y}'$, $\texttt{Z} \equiv \texttt{Z}'$, $\texttt{Y}'' \equiv$ `university`, and $\texttt{Z}'' \equiv$ `stanford`. □

3.6 Rewriting Algorithm

The following algorithm generates a rewriting query if one exists. The query bodies are converted into *normal form* and label inference and the chase are applied before we apply the algorithm.

Input: A TSL query Q with k single path conditions in the body
and a set of TSL views $\mathcal{V} = \{V_1, \ldots, V_n\}$.

Output: A set of rewriting queries.

Step 1A: Find the mappings θ_{i_j} from the body of each $V_i \in \mathcal{V}$ to the body of Q
using the mapping discovery algorithm of Appendix B.

Step 1B: construct candidate rewriting queries Q'

- $head(Q')$ is $head(Q)$
- $body(Q')$ is any conjunction of l conditions, $1 \leq l \leq k$,

where each condition is either a view "instantiation" $\theta_{i_j}(head(V_i))$ or a condition of Q.
If the resulting query is unsafe, then continue with next candidate
perform label inference and chase Q'

Step 2: test whether each constructed Q' is correct. Specifically,
construct the composition of Q' with $V_1, \ldots, V_n$ $Q'(V_1, \ldots, V_n)$ (see Section 3.7)
perform label inference and chase $Q'(V_1, \ldots, V_n)$
if $Q'(V_1, \ldots, V_n)$ is equivalent to Q (see Section 4) include Q' in the output;
else continue with the next candidate.

Notice that the above algorithm constructs candidate queries (in Step 1B) basically at random. The efficiency of the algorithm can be substantially improved with the use of simple heuristics. A particularly effective heuristic is the following:

- keep track of which conditions of the query body each instantiated view $\theta_{i_j}(head(V_i))$ maps into. These are the conditions that are "covered" by $\theta_{i_j}(head(V_i))$.
- only construct candidate queries Q' such that the views and conditions in the body of Q' "cover" all the conditions in the body of Q.

A variation of the above heuristic is implemented in the capability-based rewriting module of the TSIMMIS system [33].

The next subsection formally defines the composition of TSL queries. Composition of TSL queries is performed in Step 2 of the rewriting algorithm presented in Section 3.6.

3.7 Composition of TSL queries

The *composition* of TSL queries Q and V is a query $Q_c = V \circ Q$, such that for any OEM database D, $Q_c(D) = Q(V(D))$. Query composition is accomplished by *resolving* each condition in the body of Q with the head of V *in all possible ways*, using *unification* (which generalizes [21, 53]). A unifier in TSL is defined as follows:

Definition: [40] Unifier from a single path condition e_1 to a general condition e_2. θ is a unifier from e_1 to e_2 if the pattern $\theta(e_1)$ *is included in* the pattern $\theta(e_2)$, as described by the following definition. □

Definition: Object pattern inclusion A single path pattern e_1 is included in a pattern e_2 *if and only if*

(a) e_1 has the same object-id and label fields as e_2
(b) *if* the value field of e_1 is of the form $\{e'_1\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ and
there is a pattern $e_2^j, j = 1, \ldots, m$ such that e'_1 is included in e_2^j.
else if the value field of e_1 is of the form $\{\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ (m may be 0)
else (values are atomic) e_1 and e_2 have the same value field.

□

Query composition is easily generalized to multiple queries $V_1, \ldots, V_n$. Let us look at the following detailed example:

Example 3.7 Let us consider the following query

```
(Q16) <f(P) ans {<g(D) m V>}> :- <P p {<A l V>}> AND <Q q {<D m V>}>
```

and two views

```
(V10) <h(A',B) p {<i(A') l V'> <j(B) Y 'abc'>}> :-
                  <X label1 {<A' label2 V'> <B Y W> <C label3 T>}>
(V11) <k(F) L E> :- <G l {<F L E>}>
```

There exist two unifiers for the first condition of (Q16) and the head of (V10):

$$\theta_1 = [\texttt{P} \mapsto \texttt{h(A}',\texttt{B)}, \texttt{A} \mapsto \texttt{A}', \texttt{V} \mapsto \texttt{V}']$$

$$\theta_2 = [\texttt{P} \mapsto \texttt{h(A}',\texttt{B)}, \texttt{A} \mapsto \texttt{B}, \texttt{Y} \mapsto \texttt{l}, \texttt{V} \mapsto \text{`}\texttt{abc}']$$

There exists one unifier for the first condition of (Q16) and the head of (V11):

$$\theta_3 = [\texttt{P} \mapsto \texttt{k(F)}, \texttt{L} \mapsto \texttt{p}, \texttt{E} \mapsto \{\texttt{<A l V>}\}]$$

That means the result of resolving the first condition of (Q16) with the views gives 3 TSL queries:

```
(Q17) <f(h(A',B)) ans {<g(D) m V'>}> :-
                         <X label1 {<A' label2 V'> <B Y W> <C label3 T>}>
                         AND <Q q {<D m V'>}>
(Q18) <f(h(A',B)) ans {<g(D) m 'abc'>}> :-
                         <X label1 {<A' label2 V'> <B l W> <C label3 T>}>
                         AND <Q q {<D m 'abc'>}>
(Q19) <k(F) ans {<g(D) m V>}> :-  <G l {<F p {<A l V>}>}> AND <Q q {<D m V>}>
```

For the second condition of each one of (Q17,Q18,Q19), there exists one unifier with (V11):

$$\theta_4 = [\texttt{Q} \mapsto \texttt{k(F)}, \texttt{L} \mapsto \texttt{q}, \texttt{E} \mapsto \{\texttt{<D m V'>}\}]$$

$$\theta_5 = [\texttt{Q} \mapsto \texttt{k(F)}, \texttt{L} \mapsto \texttt{q}, \texttt{E} \mapsto \{\texttt{<D m 'abc'>}\}]$$

$$\theta_6 = [\texttt{Q} \mapsto \texttt{k(F)}, \texttt{L} \mapsto \texttt{q}, \texttt{E} \mapsto \{\texttt{<D m V>}\}]$$

respectively. Therefore, Q_c consists of 3 TSL rules:

```
(Q20) <f(h(A',B)) ans {<g(D) m V'>}> :-
                         <X label1 {<A' label2 V'> <B Y W> <C label3 T>}>
                         AND <G l {<F q {<D m V'>}>}>
(Q21) <f(h(A',B)) ans {<g(D) m 'abc'>}> :-
                         <X label1 {<A' label2 V'> <B l W> <C label3 T>}>
                         AND <G l {<F q {<D m 'abc'>}>}>
(Q22) <k(F) ans {<g(D) m V>}> :-  <G l {<F p {<A l V>}>}> AND <G l {<F q {<D m V>}>}>
```

□

Notice that in TSL there are multiple *mgu*s or most general unifiers. The practical consequence is that the result of $V \circ Q$, where V and Q are single-rule conjunctive TSL queries, could be a *union* of conjunctive TSL queries. In other words, Q_c could consist of several rules. In particular, Q_c could consist of an exponential number of rules (of at most polynomial length.) This observation gives us the following theorem.

Theorem 3.8 (Composition Complexity) Query composition in TSL is in EXPTIME.
Notice that the order of resolving query conditions with view heads does not matter. It is obvious that query composition "implements" view dereferencing: it transforms a query that refers to the object patterns in a view head to a query that refers to the "source" objects that the view is defined over.

For the complete unification algorithm see Appendix A.

4 Equivalence of TSL queries

Two queries Q_1, Q_2 are equivalent *if and only if* for all OEM databases D, their results $Q_1(D)$ and $Q_2(D)$ are equivalent. In this section, we will develop a compile-time test of equivalence of TSL queries, based on an extension of containment mappings [10]. We assume that the chase has already been applied to the queries.

The problem of TSL equivalence is complicated because of the restructuring capabilities of TSL: query heads construct arbitrary answer graphs and different rules can contribute different parts of the same answer graph. Hence we need to make sure that all the components of the result graph are the same. The required decomposition is in the same spirit as normal form decomposition for query bodies (see Section 2), but it has to go one step further by decomposing a TSL query into finer-grain rules. In Appendix C we show that normal form decomposition does *not* allow us to determine equivalence of TSL queries.

We decompose a TSL query into *graph component* queries that correspond to the components of the result graph: edges, nodes and *root*, i.e., top-level objects.[11] Every TSL rule Q is decomposed into three types of finer grain rules:

- one `top` rule corresponding to the top level condition of the head of Q (this query corresponds to the *root* of the OEM graph constructed by the head of Q)
- as many `member` rules as there are object-subobject relationships in the head of Q (these queries correspond to the *edges* of the OEM graph constructed by the head of Q, and specify their start and end objects) and
- one `object` type rules as object conditions in the query head of Q (corresponding to the *objects* of the OEM graph constructed by the head of Q and describing their labels and values).

The decomposition is illustrated by the following example. The reduced rules are essentially TSL: set values are allowed in the object "predicates". Indeed it is the possibility of sets including objects of unbounded depth that prevents conjunctive TSL from being reducible to non-recursive Datalog. Note that `member` and `top` "predicates" depart from TSL syntax to emphasize the connection to Datalog [39].

Example 4.1 Consider the following query:

```
(Q23) <l(X) l {<f(Y) m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
```

Its decomposition in graph component queries is as follows:

```
top(l(X)) :- <X a {<Y b {<Z c V>}>}>
member(l(X),f(Y)) :- <X a {<Y b {<Z c V>}>}>
member(f(Y),n(Z)) :- <X a {<Y b {<Z c V>}>}>
<l(X) l {}> :- <X a {<Y b {<Z c V>}>}>
<f(Y) m {}> :- <X a {<Y b {<Z c V>}>}>
<n(Z) n V> :- <X a {<Y b {<Z c V>}>}>
```

[11] Recall that OEM graphs are rooted.

□

The condition for equivalence of the resulting graph component queries is easily derived:

Theorem 4.2 Two sets $S_1 = \{P_1, \ldots, P_n\}$ and $S_2 = \{T_1, \ldots, T_m\}$ of graph component queries are equivalent if and only if for each P_i there exists a *mapping* to it from some T_j and for each T_i there exists a mapping to it from some P_j.

The proof of Theorem 4.2 is a straightforward generalization of the containment theorem for unions of relational conjunctive queries. Moreover, the following theorem holds:

Theorem 4.3 (TSL query equivalence) Two TSL queries are equivalent if and only if their decompositions into graph component queries are equivalent.

From the above, it is straightforward to derive a simple equivalence test for TSL queries.

5 Soundness, Completeness, and Complexity

In the previous section we have shown that query rewriting can be done in two steps. In the first step, we find mappings from the body of the views to the body of the query and we use "instantiated" view heads to construct candidate rewriting queries. In the second step we check the correctness of the rewriting. The second step establishes the soundness of our rewriting algorithm. We will now prove the completeness of the algorithm, i.e., we will show that it always finds a rewriting query if one exists. For this proof, we assume that there are no structural constraints, and therefore no functional dependencies except the key dependencies on object-id. In the presence of arbitrary functional dependencies, such as the ones that can be inferred from structural constraints, it is easy to show that our rewriting algorithm is not complete (see [15] for a simple counterexample for the case of relational query rewriting).

To prove the completeness of the algorithm, we first observe that if there is no mapping from a view body to the query body, then the view is not "relevant" to the query.

Lemma 5.1 Let Q and V be TSL queries. There is a rewriting query Q' of Q using view V only if there is a mapping from the body of V to the body of Q.

Moreover, we can bound both the number of conditions and the variables appearing in the rewriting.

Lemma 5.2 Let Q be a TSL query and $\mathcal{V}$ be a set of TSL views. If there exists a rewriting of Q using $\mathcal{V}$, then there exists such a rewriting consisting of at most k view heads, where k is the number of *single path* conditions in the body of the query.[12]

Lemma 5.3 If there exists a rewriting of query Q using the set of views $\mathcal{V}$, then there exists a rewriting of Q using $\mathcal{V}$ that doesn't use variables that don't exist in Q.

[12] Notice that, since view heads do not have to be single path, the number of single paths in the rewriting *can* be greater than k.

The above lemmata demonstrate that the theory of relational query rewriting, presented in [28], can be generalized for TSL. Notice that Lemmata 5.2 and 5.3 hold in the presence of the key dependencies on object-id. Intuitively, our algorithm is complete because no additional functional dependencies can be inferred from the object-id key dependencies. By using disjoint sets of object-id and other variables, a condition such as `<X Y {<Y Z W>}>`, which implies the extra functional dependency from `X` to `Z` and `W`, is disallowed.

The following lemma justifies why completeness is not compromised by only constructing rewriting queries Q' that have a head identical to the head of the query Q. Notice, this is an issue that is particular to semistructured and nested models while it is trivial in the relational model (Q' must have a head identical, up to variable renaming, to the head of Q.)

Lemma 5.4 If there exists a valid rewriting query Q'' such that $head(Q'')$ is not the same as $head(Q)$, then there exists a valid rewriting query Q' such that $head(Q') = head(Q)$.

To see that Lemma 5.4 holds, notice that if there exists such a query Q'', then we can always apply our rewriting algorithm to it, to derive a query Q' equivalent to Q'' (and therefore to Q) whose head is identical to the head of Q.

Theorem 5.5 The rewriting algorithm proposed in subsection 3.6 is sound and complete.

Proof: *(Sketch)* The algorithm is obviously sound, because its last step is a correctness test. It is complete because of the above lemmata, because the query composition algorithm is correct [39], and finally because the rewriting algorithm exhaustively searches the space of rewritings defined by the above lemmata. □

5.1 Complexity of TSL rewriting

The algorithm described in Section 3.5 takes exponential time. First, *Step 1* can generate an exponential in the size of the view bodies number of mappings. Then *Step 2* can generate an exponential number of candidate rewritings. Finally the construction of $Q'(V_1, \ldots, V_n)$ using a generic query composition algorithm, i.e., an algorithm which can compose two arbitrary queries Q_1 and Q_2, takes exponential time.[13]

6 Related work

TSL is derived from the Mediator Specification Language (MSL), discussed in [41, 40]. MSL is a more general language that allows arbitrary restructurings of source data. Because of its additional restructuring power, MSL (as well as StruQL, which has the same restructuring capabilities) is not closed under query composition. This significantly reduces the applicability of the rewriting algorithm.

The problem of query rewriting for conjunctive relational views is discussed in [28, 45, 30, 54, 55, 15, 14] and for recursive queries (but not recursive views) in [14]. It is also related to the problems of query containment and query equivalence [10, 12].

To the best of our knowledge, there is no work on the problem of rewriting semistructured queries using views. The only relevant work we are aware of is [19] on the problem of query containment in StruQL (a

[13] See also Section 3.7.

semistructured language similar to TSL and MSL). This work does not deal with the restructuring capabilities of the StruQL language, whereas our work deals with the restructuring capabilities of TSL. On the other hand, [19] deals with queries and views containing "wildcards" and regular path expressions, whereas TSL does not support regular path expressions.

The relational rewriting work cannot offer a straightforward solution to the TSL rewriting problem because nonrecursive TSL queries reduce to (a restricted form of) *recursive*[14] Datalog programs, as described in detail in [39], hence making inapplicable the conjunctive query rewriting results. The special form of the restricted recursion in TSL leads to decidability and complexity results which are known not to hold for general recursive Datalog programs [14].

Since our data model supports object oriented features, our work is also related to the problem of object oriented query rewriting. Previous work on the problem of containment and equivalence of object oriented queries [9, 31] relies on the existence of a static class hierarchy.[15] Work on the containment of queries on complex objects has been presented most recently in [32]. However, [32] considers the problem of query containment and equivalence, while we are solving the query rewriting problem. Furthermore, the language used in [32] is typed – as opposed to TSL which is semistructured.

Finally, there has been some recent work in using structural information about a semistructured source (such as graph schemas [7] or DTDs) in query processing [18, 44]. Abiteboul and Vianu in [4] address the problem of containment for regular path conditions in the presence of path constraints.

Isomorphism In the OEM data model every node of the semistructured graph has an object identity — unlike [8] and [32]. Furthermore, we require that the original and the rewritten queries compute identical graphs (i.e., same OID's) as opposed to graphs equivalent under bisimulation [8] or bisimulation's close relative, isomorphism. Following the isomorphism approach two OEM databases D_1 and D_2 would be equivalent if for every object x_1 of D_1 we can find an object x_2 of D_2 such that x_1 and x_2 have the same label, same value if atomic, or equivalent (i.e. isomorphic) sets of subobjects if they have set values. The isomorphism approach is based on the fact that typically we do not care for the object-id symbol; we only care for the object subobject relationships the object-id's create. For example, the URL names are not important; it is the hypertext structure created by the links that is important.

From the point of view of the rewriting algorithm it is not important whether the rewriting query Q' produces results identical to the original query Q or it produces isomorphic results. The reason is that we conjecture that if there is no rewriting query Q' with a result identical to Q then there is no rewriting query Q'' returning a result isomorphic to Q.

[14] A restricted form of recursion is needed to deal with the lack of schema and the unbounded nesting of the semistructured objects.

[15] Notice the difference with the unbounded nesting depth of the semistructured objects.

7 Conclusions and Future Work

We presented an algorithm that given a semistructured query q expressed in conjunctive TSL and a set of semistructured views $\mathcal{V}$, finds *rewriting* queries, i.e., queries that access the views and are equivalent to q. Our algorithm is based on appropriately generalizing *containment mappings*, the *chase*, and *unification*. The first step uses containment mappings to produce candidate rewriting queries. The second step composes each candidate rewriting query with the views and checks whether the composition is equivalent to the original query. Though the algorithm is similar to the one for the rewriting of conjunctive queries there are many challenges stemming from the semistructured nature of the data and the queries. For example, the composition of the rewriting query and the views is harder (from a complexity point of view) because of the lack of schema and of the restructuring capabilities of TSL views. Moreover, we extended the algorithm to use structural constraints to discover rewritings in cases where, in the absence of constraints, there would be no rewritings.

We currently incorporate our algorithm into the TSIMMIS system for use as a capability based rewriter. We will soon adapt its interfaces to the TSIMMIS system so that it will be able to also serve as a rewriter of queries using cached views. Furthermore, we are working on extensions to the algorithm so that it can handle extensions to TSL, such as regular path expressions in the query body. Notice that in the presence of regular path expressions, the opportunities (and difficulties) presented by the existence of structural constraints such as DTDs are more significant.

We are also currently developing rewriting algorithms that, instead of generating equivalent rewriting queries, will generate *maximally contained* rewriting queries, in the spirit of [15, 14].

Acknowledgements

We are grateful to Jeff Ullman and Victor Vianu for their comments and suggestions on an earlier draft of this paper. We would also like to thank Dan Suciu and Ramana Yerneni for fruitful discussions and comments.

References

[1] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *SIGMOD Workshop on Management of Semistructured Data*, pages 83–90, Tuscon, Arizona, May 1997.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. VLDB Conf.*, 1998.

[3] S. Abiteboul and V. Vianu. Queries and computation on the Web. In *Proc. ICDT Conf.*, 1997.

[4] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. PODS Conf.*, 1997.

[5] M. Bowman et al. The harvest information discovery and access system. `http://harvest.transarc.com`.

[6] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation. Latest version available at `http://www.w3.org.TR/REC-xml`.

[7] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. ICDT Conf.*, 1997.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, 1996.

[9] E. Chan. Containment and minimization of positive conjunctive queries in OODB's. In *Proc. PODS Conf.*, 1992.

[10] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. ICDE*, 1995.

[12] S. Chaudhuri and M. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proc. PODS Conf.*, 1992.

[13] A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to W3C. Latest version available at `http://www.w3.org/TR/NOTE-xml-ql`.

[14] O. Duschka and M. Genesereth. Answering queries using recursive views. In *Proc. PODS Conf.*, 1997.

[15] O. Duschka and A. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[16] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proc. SIGMOD Conf.*, 1998.

[17] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. In *Workshop on Management of Semistructured Data, ACM SIGMOD Conf.*, 1997.

[18] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. ICDE Conf.*, 1998.

[19] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. PODS Conf.*, 1998. Available from `www.research.att.com/~suciu/` under Publications.

[20] H. Garcia-Molina et al. The TSIMMIS approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8:117–132, 1997.

[21] M. Genesereth and N. Nillson. *Logical Foundations of Artificial Intelligence*. Morgan Cauffman, 1988.

[22] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB Conf.*, 1997.

[23] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD Conference*, 1995.

[24] L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.

[25] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5:35–47, Jan. 1996.

[26] L. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - a language for interoperability in relational multi-database systems. In *Proc. VLDB Conf.*, pages 239–250, 1996.

[27] P. Larson and H. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.

[28] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.

[29] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, pages 251–262, 1996.

[30] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external processors. In *Proc. PODS*, pages 227–37, 1996.

[31] A. Levy and M.-C. Rousset. CARIN: a representation language integrating rules and description logics. In *Proceedings of the European Conference on Artificial Intelligence*, Budapest, Hungary, 1996.

[32] A. Levy and D. Suciu. Deciding containment for queries with complex objects. In *Proc. PODS Conf.*, 1997.

[33] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, and Y. Papakonstantinou. Capability based mediation in TSIMMIS. In *Proc. SIGMOD Conf.*, 1998.

[34] Lotus. Lotus Notes. `http://www.lotus.com/notes`.

[35] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, Aug. 1986.

[36] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[37] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. PDIS Conf.*, 1996.

[38] A. Mendelzon and T. Milo. Formal models of the Web. In *Proc. PODS Conf.*, 1997.

[39] Y. Papakonstantinou. Query processing in heterogeneous information sources. Technical report, Stanford University Thesis, 1997. Available from www-cse.ucsd.edu/~yannis/papers/.

[40] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. VLDB Conf.*, 1996.

[41] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proc. ICDE Conf.*, pages 132–41, 1996.

[42] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[43] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS Conf.*, 1996.

[44] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. To appear, ICDE Conf. 1999.

[45] X. Qian. Query folding. In *Proc. ICDE*, pages 48–55, 1996.

[46] D. Quass, A. Rajaraman, S. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc. DOOD*, pages 319–44, 1995.

[47] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS Conf.*, pages 105–112, 1995.

[48] S. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27:633–55, 1980.

[49] T. Sellis, N. Roussopoulos, and R. Ng. Efficient Compilation of Large Rule Bases Using Logical Access Paths. *Information Systems*, 15(1):73–84, 1990.

[50] V. Subrahmanian et al. HERMES: A heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes/overview/paper.

[51] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the acedb data base manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, 1992.

[52] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems.* Computer Science Press, New York, NY, 1988.

[53] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies.* Computer Science Press, New York, NY, 1989.

[54] V. Vassalos and Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. In *Proc. VLDB Conf.*, pages 256–266, 1997.

[55] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms, 1998. Accepted for publication, Journal of Logic Programming.

[56] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, 1993.

[57] H. Z. Yang and P. . Larson. Query transformation for PSJ-queries. In *Proc. VLDB Conf.*, pages 245–254, 1987.

[58] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. SIGMOD Conference*, pages 316–327, 1995.

A Query composition

See Figure 4

B Mappings

The following algorithm returns the mapping, if there is one, of a single path condition c_1 to a single path condition c_2.

INPUT Two single path conditions c_1 and c_2
OUTPUT A mapping θ, if there is one, such that $\theta(c_1) \equiv c_2$
METHOD Run the function $\theta = \texttt{map1}(c_1, c_2, [])$

function $\texttt{map1}(c_1, c_2, \theta^i)$
  apply θ^i to $c_1.oid$ and $c_2.oid$

%Mappings of oid's and labels are not described in detail because they are well-known

  if there is no mapping θ_{oid} of $c_1.oid$ to $c_2.oid$
    return no mapping
  else apply $\theta^i \circ \theta_{oid}$ on $c_1.label$ and $c_2.label$
  if there is no mapping θ_{label} from $c_1.label$ to $c_2.label$
    return no mapping
  else apply $\theta^i \circ \theta_{oid} \circ \theta_{label}$ on $+c_1.value$ and $c_2.value$
  if $c_1.value$ and $c_2.value$ are terms and they have a mapping θ_{value}
    return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ \theta_{value}$
  else if $c_2.value$ is a set of the form $\{r_1, \ldots, r_m\}$ and $c_1.value$ is a variable then
    if there is already a mapping of $c_1.value$ to a set $\{s_1, \ldots, s_n\}$
      return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ [c_1.value \mapsto \{s_1, \ldots, s_n, r_1, \ldots, r_m\}]$
    else
      return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ [c_1.value \mapsto \{r_1, \ldots, r_m\}]$
  else return no mapping

Next, we describe a brute force algorithm for discovering the mappings from a set of single path conditions to another set of single path conditions.

INPUT Two sets $\{c^1_1, \ldots, c^1_n\}$ and $\{c^2_1, \ldots, c^2_m\}$ of single path conditions
OUTPUT All mappings θ such that for every condition $c^1_i, i = 1, \ldots, n$ there is a $c^2_j, j = 1, \ldots, m$ such that $\theta(c^1_i) \equiv c^2_j$
METHOD Run the function $\texttt{mapmany}(\{c^1_1, \ldots, c^1_n\}, \{c^2_1, \ldots, c^2_m\})$
function $\texttt{mapmany}(\{c^1_1, \ldots, c^1_n\}, \{c^2_1, \ldots, c^2_m\})$

for every function f from $\{1,\ldots,n\}$ to $\{1,\ldots,m\}$ do
    $\theta_0 \leftarrow []$
    for $i = 1,\ldots,n$
        if there is not a $\theta_i = \texttt{map1}(c_i^1, c_{f(i)}^2, \theta_{i-1})$
            exit inner loop
    if $theta_n$ was found, add θ_n to mappings
return

Notice that there may be up to m^n mappings between the two sets. The algorithm described above takes no more than exponential time.

C Normal form decomposition is not enough for TSL equivalence

We try to come up with a simpler syntactic condition for equivalence of two single-rule queries Q_1 and Q_2, based on the idea of the normal form for TSL queries. We will attempt to derive an equivalence condition as follows: we will normalize the query heads (and bodies) of Q_1 and Q_2 and we will attempt to show a modified version of Theorem 4.2.

Conjecture C.1 Two sets $S_1 = \{P_1,\ldots,P_n\}$ and $S_2 = \{T_1,\ldots,T_m\}$ of normal-form queries (heads and bodies are normalized) are equivalent if and only if for each P_i there exists a *mapping* to it from some T_j and for each T_i there exists a mapping to it from some P_j.

The following example shows that this is not correct; we conclude that normalizing the query heads does not allow us to come up with a simple syntactic characterization of query equivalence.

Example C.2 Consider the following queries; each one consists of two rules, and in order to make our point clear we have let all the queries have the same body. Notice that the first rule of query (Q24) creates an `l` object for every `a` object and an `n'` object for every `c` object, while the second rule creates an `l'` object for every `a` and an `n` object for every `c`. Then, notice that query (Q25), though it has different single path conditions from query (Q24), creates exactly the same result. The intuition is that the different query heads create different "parts" of the same answer graph for (Q24) and (Q25).

```
(Q24) <l(X) l {<f(Y) m {<n'(Z) n' V>}>}> :- <X a {<Y b {<Z c V>}>}>
      <l'(X) l' {<f(Y) m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
(Q25) <l(X) l {<f(Y) m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
      <l'(X) l' {<f(Y) m {<n'(Z) n' V>}>}> :- <X a {<Y b {<Z c V>}>}>
```

Notice that queries (Q24) and (Q25) have their heads and bodies in normal form. Also notice that there are no mappings from either rule of (Q24) to either rule of (Q25) and vice versa. □

To solve problems such as those posed by the above example we have to further normalize TSL rules, as explained in Section 4. The following rules are the graph component queries of (Q24).

```
top(l(X)) :- <X a {<Y b {<Z c V>}>}>
member(l(X),f(Y)) :- <X a {<Y b {<Z c V>}>}>
member(f(Y),n′(Z)) :- <X a {<Y b {<Z c V>}>}>
<l(X) l {}> :- <X a {<Y b {<Z c V>}>}>
<f(Y) m {}> :- <X a {<Y b {<Z c V>}>}>
<n′(Z) n′ V> :- <X a {<Y b {<Z c V>}>}>
```

INPUT A single path condition c, and a normal-form rule head r

OUTPUT A set S of unifiers θ such that $\theta(r)$ contains $\theta(c)$

METHOD Run the function $S =$`unify(`c`,`r`,`$[]$`)`

function `unify(`c`,`r`,`θ`)` *returns sets of unifiers*

  apply θ to $c.oid$ and $r.oid$

  if term unification of $c.oid$ and $r.oid$ results in θ_{oid} *unify object-id's*

    apply $\theta oid \circ \theta$ to labels of c and r

  else return empty set

  if unification of $c.label$ and $r.label$ results in θ_{label} *unify labels*

    apply $\theta_{label} \circ \theta_{oid} \circ \theta$ to values of c and r

  else return empty set

  if $c.value$ and $r.value$ are terms and their term unification results in θ_{value}

if atomic objects then unify the term values

    return $\theta_{value} \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is a variable and $r.value$ is a set

    return $[c.value \mapsto r.value] \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is a set $\{c_1 \ldots c_d\}$ and $r.value$ is a variable

    if there is already definition of the form $r.value \mapsto \{s_1\ ldotss_p\}$

      return $[r.value \mapsto \{c_1, \ldots, c_d, s_1, \ldots, s_p\}]¡rc\theta_{label} \circ \theta_{oid} \circ \theta$

    else

      return $[r.value \mapsto \{c_1, \ldots, c_d\}] \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is the empty set and $r.value$ is a set

    return $\theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ has a subobject condition c' and $r.value$ is a set

    for each subobject r_i or $r.value$

      $S_i =$`unify(`c'`, `r_i`, `$\theta_{label} \circ \theta_{oid}\ circ\theta$`)`

    return the union $\cup_i S_i$ of the results of the `unify` calls above

  else return empty set

Figure 4: The unification algorithm