Query Reformulation under Incomplete Mappings[1]

Nam Huyn[2]
Stanford University
huyn@cs.stanford.edu

Abstract

This paper focuses on some of the important new translatability issues that arise in the problem of interoperation between two database schemas when mappings between these schemas are inherently more complex than traditional views or pure Datalog programs can capture. In many cases, sources cannot be redesigned, and mappings among them exhibit some form of incompleteness under which the question of whether a query can be translated across different schemas is not immediately obvious. The notion of query we consider here is the traditional one, in which the answers to a query are required to be definite: answers cannot be disjunctive or conditional and must refer only to domain constants. In this paper, mappings are modeled by Horn programs that allow existential variables, and queries are modeled by pure Datalog programs. We then consider the problem of eliminating functional terms from the answers to a Horn query where function symbols are allowed. We identify a class of Horn queries called "term-bounded" that are equivalent to pure Datalog queries. We present an algorithm that rewrites a term-bounded query into an "equivalent" pure Datalog query. Equivalence is defined here as yielding the same function-free answer.

1. This work was originally written in October 1994 but has never been published before taking the current form of a technical report. Earlier in 1994, I shared the ideas in this work with Xiaolei Qian, whose work in [9] embodies some of these main ideas. The ideas from [9] were in turn picked up by Oliver Duschka and used in [10] to derive what is essentially equivalent to our result.

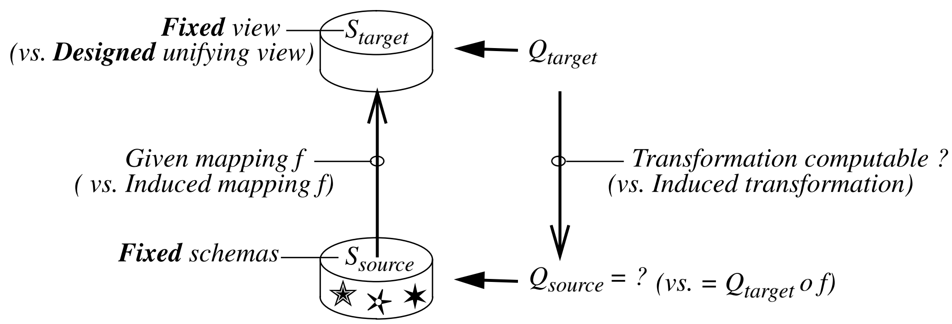
2. Work supported by ARO grant DAAH04-95-1-0192

1. Motivation

Query mediation vs. schema design

Much work on heterogeneous information systems interoperation is couched in terms of schema and view integration. These studies revolve around methodologies for designing good unifying schemas and for restructuring schemas [4]. In many situations however, the opportunity for designing or reengineering schemas is just not present. Yet, we are required to interoperate among these heterogeneous data sources. This new situation is contrasted with the traditional one in Figure 1 for the task of query translation. This new challenge has been found in the design of mediators [8]. Mediators are often designed to provide information within a narrowly scoped domain, for instance, to answer queries over some schema specific to that domain.

FIGURE 1. Query translation vs. Schema design



For instance, a mediator can be designed to answer commonly asked questions about documents. The mediator does not store document data itself. Its purpose is to provide a consistent informational interface for queries about documents while hiding all the complexities and heterogeneities among the underlying data sources it accesses. An appropriate vocabulary (or schema) about documents needs to be developed independently of the actual sources and exported to all potential users. The assumption here is that at the time the mediator is designed, the precise configuration of sources is unknown, let alone their schemas.

This assumption represents a major departure from traditional work on integration. “Mounting” a new data source to the mediator essentially consists of providing the mediator with a mapping between the mediator schema and the new source schema. From the mediator’s point of view, a new source schema may present itself with arbitrary degree of mismatch.

- **Mismatch between domains:** The data source uses zipcodes to represent location but the mediator publishes locations in terms of cities. Knowing the zipcode of a location does not necessarily tell you precisely which city. But it does narrow down to a small set of possible cities [1].
- **Mismatch between relations:** Different sources use different “cuts” at relating individuals in the same collection, these cuts being not totally mutually independent. For instance, a data source only captures the “`project-peer`” relationship but the mediator advertises the “`employee-manager`” relationship. Knowing that two individuals are peers on one project does not tell more than that they are managed by the same individual whose identity remains unknown nevertheless.
- **Difference in abstraction levels:** A data source records the partial order of events but the mediator exports the total order of events (e.g. exact time points at which events occurs).

While mappings in the traditional integration setting are essentially functional (views or Datalog programs), the possibility now of severe schema mismatch has made mappings inherently more complex. A class of mappings we are considering are *incomplete mappings*. A mapping (between a source schema and a target schema) is incomplete when an instance I_{Source} of a source schema does not always map into a unique instance I_{Target} of the target schema. That is, more than one I_{Target} is possible but there is not enough information to resolve the ambiguity. It is also possible that the "instance" I_{Target} contains existential variables that denote objects whose identities cannot be inferred.

Example 1.1

A travel advisor mediator publishes a direct flight schema `dflight(City1,City2)`. Unfortunately the only relevant data source available is a database containing information on package deals offered by some travel agency, e.g. the relation `package(Source,Dest,Cost)`. The problem is that traveling with these packages might or might not involve making intermediate connections, and there is no way to know which is the case. The database might not provide information about direct flights, but can be useful for answering other queries such as reachability questions. Thus, instead of giving up on the database, we must try to exploit whatever mapping we have, even if it is only a partial one:

$$(\forall X)(\forall Y)(\forall C)(\mathtt{package}(X,Y,C) \Rightarrow \mathtt{reach}(X,Y)).$$

$$(\forall X)(\forall Y)(\exists Z)(\mathtt{reach}(X,Y) \Rightarrow \mathtt{dflight}(X,Y) \vee \mathtt{dflight}(X,Z) \wedge \mathtt{reach}(Z,Y)).$$

Note the existential variable `Z` and the disjunction in the second rule. They prevent certain queries from being answered definitely, such as questions on direct flights. Nevertheless, not all queries against the `dflight` schema have uninteresting empty answers. For instance, whether there is a flight out of `X` can be reformulated as whether there is a package starting from `X`. The transitive closure of the `dflight` relation (i.e. a reachability question) can be rewritten as the transitive closure of the `package` relation (ignoring costs). Even though a reformulated query is not semantically equivalent to the original query, it is equivalent in the sense that it does not miss any answer that can be logically deduced to the best of our incomplete knowledge. ∎

Accomodating dynamic configuration

The precise configuration of information sources usually cannot be predicted in advance and is often not under the mediator's control. Under this new operating requirement, traditional approaches to integration typically do not scale well. Mounting a new data source generally involves the creation of new semantic relationships between the new schema and all existing schemas. However, requiring all possible mappings for all possible schema combinations to be specified is neither practical nor desirable. This can be a nuisance, especially if there are many existing schemas or if they have sizable vocabularies. Furthermore, some of these existing data sources may be subsequently unmounted, rendering many of the mappings just created effectively useless. The logical solution is to modularize mappings. That is, mappings are specified between the mediator schema and each source schema individually. If a source schema does not have an informationally complete set of relations, the resulting mapping is bound to contain partial information. This approach leaves implicit the relationships that result from any potential synergistic interactions among sources to be discovered by the mediator.

Example 1.2

A book mediator exports a view `bookinfo(bookid,title,place)`. A "locator" source has the relation `pub(bookid,publisher)`, while a "reference" source contains information about books in the relation `document(bookid,title)`. The "locator" source is mapped with:

$$(\forall B)(\forall P)(\exists T)\ (pub(B,P) \Rightarrow bookinfo(B,T,P)).$$

while the “reference” source is mapped with:

$$(\forall B)(\forall T)(\exists P)\ (document(B,T) \Rightarrow bookinfo(B,T,P)).$$

Assume that the semantics of `bookinfo` requires that a book has at most a title and a publisher:

`bookinfo.bookid` *functionally determines* `bookinfo.title, bookinfo.publisher`

Without requiring the database administrator to specify cross database mapping, namely that `bookinfo` is the join of `pub` and `document`, the system should have sufficient information to infer that the query `bookinfo(B,T,P)` can be logically reformulated as a join

`document(B,T) & pub(B,P).`

In fact, knowing `document(b,t)` and `pub(b,p)`, one can deduce `bookinfo(b,t,`$?_1$`)` and `bookinfo(b,`$?_2$`,p)`. Since “b” uniquely determines the other components, $?_1$ can be identified with “p” and $?_2$ with “t”. ∎

Information sources often interact semantically. Combining multiple sources has the potential of yielding more information than simply taking the union of these sources considered separately. The question then is how to fully exploit incomplete mappings in the task of query mediation.

Mediation with incomplete mappings

Mediators operate in environments that are essentially more stringent than much traditional work on integration has assumed. Mappings among schemas now assume a level of complexity that goes beyond simple views or Datalog programs. Incompleteness in mappings exemplifies this added complexity. Without the ability of exploiting complex mappings, mediation would be severely hampered. A technical challenge is how to mediate queries among sources under incomplete mappings without losing too much information.

If we characterize these mappings as first order theories, incompleteness would take the forms of disjunctions and existentially quantified variables, among others. Even if we restrict these theories to be Horn (with possible existential variables), whether a target query can be translated to an equivalent source query is not immediately obvious. The notion of query we consider here is the traditional one, in which answers to a query are required to be definite: they cannot be disjunctive or conditional and must refer only to domain constants. Because of incompleteness, some queries will admit no definite answers, yet other queries are answerable either partially or completely. The issue is how to maximize the completeness of the answer to a query, relative to what we know (mappings). Partial knowledge is not totally useless. In many cases, a solution to a query can be deduced if one is able to reason carefully around the incomplete information to reach a logical definite conclusion.

To summarize, in the problem we study here, we assume that adequate mappings between two fixed schemas S_1 and S_2 are given. How these mappings are created is a separate subject outside the scope of this paper. They may be more expressive than simple traditional views or Datalog programs. Furthermore, we assume some class of queries against S_1 that is permissible and some class of queries against S_2 that can be handled. Without requiring our data model to be extended, the question we like to study is, under some class of mappings between S_1 and S_2, whether a given query through S_1 can be reformulated as an equivalent query over S_2, and whether there are efficient algorithms to carry out the reformulation task. Definite answers to queries are assumed throughout.

Related work

One approach to query reformulation between semantically heterogeneous data sources consists of extending the data model to match the expressive power of the query language enriched to capture the more complex mappings. DeMichiel's work [1] on extended relational model and operations exemplifies this approach where indefinite answers to queries are allowed.

In spirit, Qian's query mediation approach to semantic interoperation [5] is the most related to ours, in which source and target queries are expressed in a traditional first order query language and are required to return definite answers. However, translatability issues and computational aspects of query reformulation are not addressed in her work.

Paper outline

This paper is organized as follows. In the next section , the general technical problem of query translation is outlined. Section 3 defines the class of "term-bounded" queries for which a translation algorithm is presented in Section 4. Proof of correctness of the algorithm is given in Section 5, along with the main theorem. Section 6 concludes the paper.

2. Problem Formalized

We will study the query reformulation problem in a relational model/deductive framework [7]. In this preliminary study, we assume both target and source accept any Datalog query against their own schema. The source EDB relations may contain only constant symbols but no functional terms.

Let S be a schema consisting of a set of predicates. A Datalog query against S, denoted $Q(S)$, is a Datalog program whose EDB predicates belong to S and that has a distinguished IDB predicate *"answer"* (the query predicate). A mapping $M(S_t,S_s)$ between a target schema S_t and a source schema S_s is a first order theory that relates predicates in S_t and S_s. We study the question of whether there is a query $Q(S_s)$ that is "equivalent" to the original query $Q(S_t)$ under $M(S_t,S_s)$. If the question can be decided, we would like to find algorithms that compute $Q(S_s)$ given $Q(S_t)$ and $M(S_t,S_s)$.

The notion of equivalence needs to be clarified here. First, query answers are required to be definite: if $answer(X_1,...,X_n)$ is the query predicate, we are only interested in the set of all tuples $(a_1,...,a_n)$ where a_i are domain constants and such that $answer(a_1,...,a_n)$ can be deduced. In other words, we rule out indefinite answers such as disjunctions of answer facts and sentences about the answer predicate involving existential variables. We then say that $Q(S_s)$ is equivalent to $Q(S_t)$ under $M(S_t,S_s)$ when the answer generated by $Q(S_s)$ (automatically definite) is identical to the definite answer deducible from $Q(S_t)$ and $M(S_t,S_s)$, for all extensions of S_s.

If we restrict $M(S_t,S_s)$ to be a pure Datalog program, it is easy to translate an original query $Q(S_t)$ to an equivalent query $Q(S_s)$ that can be processed by the source database. Indeed, $Q(S_s)$ is simply the union of $Q(S_t)$ and $M(S_t,S_s)$ where the S_t predicates are simply relabeled as IDB. However, in the general case where mappings are expressed as an arbitrary first order theory, there is no efficient way to either solve $Q(S_t)$ or to reformulate it into an equivalent $Q(S_s)$.

Fortunately, we have identified a class of mappings that arise frequently in practice and that seem to admit a more tractable solution. We will concentrate on mappings that are Horn programs, i.e. Datalog programs augmented with function symbols. These function symbols are created as a result of skolemizing existential variables. In other words, while pure Datalog programs consist of Horn clauses having no function symbols and where variables are universally quantified, we allow some variables to be existentially quantified. Within this class of programs, we identified a subclass we called *term-bounded* for which we will present an algorithm that eliminates functional terms.

3. Term-Bounded Horn Queries

A Horn program with function symbols is *term-bounded* if there is an upper bound on the size of derivable tuples, independent of the EDB relations. Again, these EDB relations contain no function symbols. The size of a tuple is the height of the tree that represents terms and subterms in the tuple. In this class of programs, unbounded growth of functional terms within derivable facts never happens.

Example 3.1: The following program

```
answer :- integer(X).
integer(succ(X)) :- integer(X).
integer(X) :- base(X).
```

is excluded from the class since the minimum model contains `integer(succ`$^{\mathtt{i}}$`(c))` for arbitrary `i`, given the extension {`base(c)`}. ∎

Example 3.2: The following program:

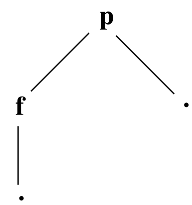
```
answer :- ancestor(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z) & ancestor(Z,Y).
parent(X,f(X,Y)) :- grandparent(X,Y).
parent(f(X,Y),Y) :- grandparent(X,Y).
```

computes the transitive closure of the `parent` relationship from a `grandparent` EDB. The function symbol `f` was introduced as a result of skolemizing the existential variable. This program is term-bounded since the only derivable facts are necessarily of the form `parent(a,f(b,c))`, `parent(f(a,b),c)`, `ancestor(a,b)`, `ancestor(a,f(b,c))`, `ancestor(f(a,b),c)`, or `ancestor(f(a,b),f(c,d))`, where `a`,`b`,`c` and `d` stand for domain constants. ∎

Pattern classes

Each of these forms instantiates what we call a *function nesting pattern class* or simply, *a pattern class*. A pattern class is defined by a particular nesting of function symbols: all variables and other constant symbols in an atom play no role in determining its class. If predicate and function symbols have fixed arity (which we assume here), then the atoms `p(f(X),Y)`, `p(f(X),X)`, `p(f(U),V)`, and `p(f(a),X)`, for example, are instances of the same pattern class. This class is depicted in Figure 2.

FIGURE 2. Pattern class p(f(_),_)



On the other hand, even though `p(X)` subsumes `p(f(X))`, they belong to two different pattern classes, as depicted in Figure 3. The reason behind this definition of pattern class will be clear in a moment. But essentially, a pattern class is used to make explicit the exact function nesting patterns of all the ground facts that could be generated by an atom belonging to that class.

FIGURE 3. Pattern classes p(_) and p(f(_))

Characterization

While *safety* of Horn programs (that their answers are always finite) has been shown to be undecidable [6], it is not clear whether stronger notions of safety such as *effective computability* [3] and *term-boundedness* are decidable, even though sufficient tests exist for effective computability.

Similarly, while term-boundedness is not obviously decidable, there are non trivial sufficient conditions that can be easily checked. For instance, if function symbols occur only the non recursive portion of a Horn program (i.e. if the mutually recursive rules do not have function symbols), it is easy to see that the Horn program is term-bounded. A more refined test that keeps track of how functional terms flow between IDBs, using a device similar to *term circulation graph* [2], can be used to characterize a larger subclass of term-bounded Horn programs. However, let us emphasize that testing for term-boundedness is not an end by itself. It is only useful to assure termination of our procedure that eliminates functional terms.

4. Eliminating Functional Terms

Informally, the algorithm works on each rule all of whose variables are known to be bound to domain constants. The starting point obviously consists of those rules whose bodies only refer to EDB predicates. The exact form of tuples (a.k.a. pattern class) each such rule can generate can be precisely predicted. The rule head is then “flattened”, i.e. replaced with a “flat” head by means of a new predicate (a.k.a. flat predicate) that represents the pattern class. All rules having subgoals unifiable with the head will have an arbitrary number of these subgoals replaced by flat predicate subgoals. This is reminiscent of a bottom-up symbolic evaluation where pattern classes are propagated from the base predicates toward the query predicate. All pattern classes that can potentially be generated are made explicit and encoded as flat predicates. At the end, all remaining rules that cannot be flattened are discarded. Also, those top level rules whose heads represent pattern classes for the predicate "answer" involving function symbols are discarded as well.

Example 4.1: This simple example briefly illustrates the algorithm.

```
[r1] a(B,f(B,N)) :- d(B,N).
[r2] p(f(B,N),N) :- d(B,N).
[r3] answer(B,N) :- a(B,P) & p(P,N).
```

`r1`'s head is flattened into `a1(B,B,N)` by defining `a1(X,Y,Z)≡a(X,f(Y,Z))`, and `r1` is replaced by:

```
[r4] a1(B,B,N) :- d(B,N).
```

`r1`'s head unifies with `r3`'s first subgoal, creating a new rule:

```
[r5] answer(B,N) :- a1(B,B,N) & p(f(B,N),N).
```

Note that the functional term has propagated through `r5`'s body into the `p` subgoal and the first subgoal in `r3` has had predicate `a` replaced by `a1`. Next, `r2`'s head is flattened into `p1(B,N,N)` by letting `p1(X,Y,Z) ≡ p(f(X,Y),Z)`, and `r2` is replaced with:

```
[r6] p1(B,N,N) :- d(B,N).
```

`r2`'s head unifies with `r3`'s second subgoal and `r5`'s second subgoal, creating two new rules:

```
[r7] answer(B,N) :- a(B,f(B,N)) & p1(B,N,N).
[r8] answer(B,N) :- a1(B,B,N) & p1(B,N,N).
```

Now that the flattening process is done, `r3`, `r5` and `r7` can be discarded since they remain unflattenable. The result, a pure Datalog program equivalent to the original program:

```
[r4] a1(B,B,N) :- d(B,N).
[r6] p1(B,N,N) :- d(B,N).
[r8] answer(B,N) :- a1(B,B,N) & p1(B,N,N).
```

can further be simplified into `answer(B,N) :- d(B,N)` by expanding the subgoals of `r8`. ∎

Algorithm 4.2: Eliminating functional terms.

INPUT: A term-bounded Horn program with the query predicate "answer".

OUTPUT: An equivalent Datalog program using no functional terms.

METHOD:

- Initialize INPUT-SET to the set of rules in the program, OUTPUT-SET to the empty set. Mark all EDB predicates "flat".
- While there is a rule in INPUT-SET that is "flattenable" (i.e. all of whose subgoals have been marked "flat"), remove it from INPUT-SET, "flatten" it and put the result in OUTPUT-SET. This may cause new rules to be added to INPUT-SET.
- At the end, OUTPUT-SET contains the result. Its query predicate is the new predicate that represents the pattern class for "answer" involving no function symbols.

To "flatten" a rule `r` with head $p(t_1, ..., t_n)$:

1. We first check if `r`'s head is an instance of a pattern class that has been recorded so far.
2. If the check is negative, we record the pattern class for the head, create a new predicate `p'` (marked "flat") that becomes the representative for that class, and rewrite `r`'s head in terms of `p'`. Note that the rewritten head is now flat, and is also recorded as a "flat head pattern". The arity of `p'` is the number of leaves in the tree that represents the pattern class (the choice of which argument of `p'` corresponds to which leaf is not important as long as the correspondence is maintained). We continue in step 4.
3. If the check is positive, we retrieve the predicate `p'` that is the representative for the existing pattern class. We then rewrite `r`'s head in terms of `p'`. If the new head is subsumed by an existing flat head pattern, we are done. Otherwise we record the new flat head pattern and continue in step 4.
4. For every rule in INPUT-SET that has `p`-subgoals in its body, create copies in each of which a different subset of the `p`-subgoals is selected to unify with `r`'s head and rewrite in terms of `p'`. Note that some of these copies may be discarded because there is no variable substitution that makes all the `p`-subgoals in the subset unify with `r`'s head. A small but important detail about unification is in order here. Unification, in addition to being a standard one that favors the subgoal's variables, is also sensitive to "flat" variables (in a rule, a variable is flat if it is an argument of a flat predicate). Namely, a flat variable cannot be bound to a functional term. When unifying `r`'s head with a subgoal, some of the subgoal's variables are flat since they are shared with some other flat subgoal, and all `r`'s head variables are flat since all `r`'s variables are flat. All the resulting rule copies are added back to INPUT-SET. ∎

Example 4.3

Consider the "grandparent" program from Example 3.2, which involves recursion. Let us trace the application of the algorithm to translate the program, which we repeat here for convenience:

```
[r1]answer(X,Y) :- ancestor(X,Y).
[r2]ancestor(X,Y) :- parent(X,Y).
[r3]ancestor(X,Y) :- parent(X,Z) & ancestor(Z,Y).
[r4]parent(X,f(X,Y)) :- grandparent(X,Y).
[r5]parent(f(X,Y),Y) :- grandparent(X,Y)
```

Initially:

FLAT = {`grandparent`}, INPUT-SET = {`r1`,`r2`,`r3`,`r4`,`r5`}.

- Flatten [`r4`] and [`r5`], defining `parent1(X,Y,Z)` ≡ `parent(X,f(Y,Z)` and `parent2(X,Y,Z)` ≡ `parent(f(X,Y),Z)`:

```
[r41-r4]parent1(X,X,Y) :- grandparent(X,Y).[1]
[r6+r2]ancestor(X,f(X,Y)) :- parent1(X,X,Y).[2]
[r7+r3]ancestor(X,Y) :- parent1(X,X,Z) & ancestor(f(X,Z),Y).
[r51-r5]parent2(X,Y,Y) :- grandparent(X,Y).
[r8+r2]ancestor(f(X,Y),Y) :- parent2(X,Y,Y).
[r9+r3]ancestor(f(X,Z),Y) :- parent2(X,Z,Z) & ancestor(Z,Y).
```

FLAT = {`grandparent`, `parent1`, `parent2`}, INPUT-SET = {`r1`,`r2`,`r3`,`r6`,`r7`,`r8`,`r9`}.

1. [ri-rj] denotes ri being the result of flattening rj, and therefore replacing rj.
2. [ri+rj] denotes ri being a copy of rj where some subgoals are replaced with a flat predicate just defined.

- Flatten [r6] and [r8], defining ancestor1(X,Y,Z) ≡ ancestor(X,f(Y,Z)), and ancestor2(X,Y,Z) ≡ ancestor(f(X,Y),Z):

```
[r61-r6]ancestor1(X,X,Y) :- parent1(X,X,Y).
[r10+r1]answer(X,f(X,Y)) :- ancestor1(X,X,Y).
[r11+r3]ancestor(X,f(Z,Y)) :- parent(X,Z) & ancestor1(Z,Z,Y).
[r12+r9]ancestor(f(X,Z),f(Z,Y)) :- parent2(X,Z,Z) & ancestor1(Z,Z,Y).
[r81-r8]ancestor2(X,Y,Y) :- parent2(X,Y,Y).
[r13+r3]ancestor(X,Y) :- parent(X,f(Z,Y)) & ancestor2(Z,Y,Y).
[r14+r7]ancestor(X,Y) :- parent1(X,X,Y) & ancestor2(X,Y,Y).
```

FLAT = {grandparent, parent1, parent2, ancestor1, ancestor2}, INPUT-SET = {r1,r2,r3,r7,r9,r10,r11,r12,r13,r14}.

- Flatten [r10], [r11], [r12] and [r14], defining answer1(X,Y,Z) ≡ answer(X,f(Y,Z)), ancestor3(X,U,Y,V) ≡ ancestor(f(X,U),f(Y,V)), ancestor4(X,Y) ≡ ancestor(X,Y):

```
[r101-r10]answer1(X,X,Y) :- ancestor1(X,X,Y).
[r111-r11]ancestor1(X,Z,Y) :- parent(X,Z) & ancestor1(Z,Z,Y).
[r15+r1]answer(X,f(Y,Z)) :- ancestor1(X,Y,Z).
[r16+r3]ancestor(X,f(Y,U) :- parent(X,Z) & ancestor1(Z,Y,U).
[r17+r9]ancestor(f(X,Z),f(Y,U)) :- parent2(X,Z,Z) & ancestor1(Z,Y,U).
[r121-r12]ancestor3(X,Z,Z,Y) :- parent2(X,Z,Z) & ancestor1(Z,Z,Y).
[r18+r1]answer(f(X,Z),f(Z,Y)) :- ancestor3(X,Z,Z,Y).
[r19+r3]ancestor(X,f(U,Y)) :- parent(X,f(Z,U)) & ancestor3(Z,U,U,Y).
[r20+r7]ancestor(X,f(Z,Y)) :- parent1(X,X,Z) & ancestor3(X,Z,Z,Y).
[r141-r14]ancestor4(X,Y) :- parent1(X,X,Y) & ancestor2(X,Y,Y).
[r21+r1]answer(X,Y) :- ancestor4(X,Y).
[r22+r3]ancestor(X,Y) :- parent(X,Z) & ancestor4(Z,Y).
[r23+r9]ancestor(f(X,Z),Y) :- parent2(X,Z,Z) & ancestor4(Z,Y).
```

FLAT = {grandparent, parent1, parent2, ancestor1, ancestor2, answer1, ancestor3, ancestor4}, INPUT-SET = {r1,r2,r3,r7,r9,r13,r15,r16,r17,r18,r19,r20,r21,r22,r23}.

- Flatten [r15], [r17], [r18], [r20], [r21] and [r23], defining answer2(X,U,Y,V) ≡ answer(f(X,U),f(Y,V)), and answer3(X,Y) ≡ answer(X,Y):

```
[r151-r15]answer1(X,Y,Z) :- ancestor1(X,Y,Z).
[r171-r17]ancestor3(X,Z,Y,U) :- parent2(X,Z,Z) & ancestor1(Z,Y,U).
[r24+r1]answer(f(X,Z),f(Y,U)) :- ancestor3(X,Z,Y,U).
[r25+r3]ancestor(X,f(Y,V)) :- parent(X,f(Z,U)) & ancestor3(Z,U,Y,V).
[r26+r7]ancestor(X,f(Y,U)) :- parent1(X,X,Z) & ancestor3(X,Z,Y,U).
[r181-r18]answer2(X,Z,Z,Y) :- ancestor3(X,Z,Z,Y).
[r201-r20]ancestor1(X,Z,Y) :- parent1(X,X,Z) & ancestor3(X,Z,Z,Y).
[r211-r21]answer3(X,Y) :- ancestor4(X,Y).
[r231-r23]ancestor2(X,Z,Y) :- parent2(X,Z,Z) & ancestor4(Z,Y).
[r27+r1]answer(f(X,Z),Y) :- ancestor2(X,Z,Y).
[r28+r3]ancestor(X,Y) :- parent(X,f(Z,U)) & ancestor2(Z,U,Y).
[r29+r7]ancestor(X,Y) :- parent1(X,X,Z) & ancestor2(X,Z,Y).
```

FLAT = {`grandparent`, `parent1`, `parent2`, `ancestor1`, `ancestor2`, `answer1`, `ancestor3`, `ancestor4`, `answer2`, `answer3`},
INPUT-SET = {`r1,r2,r3,r7,r9,r13,r16,r19,r22,r24,r25,r26,r27,r28,r29`}.

- Flatten [`r24`], [`r26`], [`r27`] and [`r29`], defining `answer4(X,Y,Z)` ≡ `answer(f(X,Y),Z)`:

```
[r241-r24]answer2(X,Z,Y,U) :- ancestor3(X,Z,Y,U).
[r261-r26]ancestor1(X,Y,U) :- parent1(X,X,Z) & ancestor3(X,Z,Y,U).
[r271-r27]answer4(X,Z,Y) :- ancestor2(X,Z,Y).
[r291-r29]ancestor4(X,Y) :- parent1(X,X,Z) & ancestor2(X,Z,Y).
```

FLAT = {`grandparent`, `parent1`, `parent2`, `ancestor1`, `ancestor2`, `answer1`, `ancestor3`, `ancestor4`, `answer2`, `answer3`, `answer4`},
INPUT-SET = {`r1,r2,r3,r7,r9,r13,r16,r19,r22,r25,r28`}.

No more rule in INPUT-SET can be flattened. Since `answer3` is the new query predicate, useless rules can be dropped, namely {`r101`, `r151`, `r181`, `r241`, `r271`, `r61`, `r111`, `r121`, `r171`, `r201`, `r261`}, to yield the final program:

```
[r41] parent1(X,X,Y) :- grandparent(X,Y).
[r51] parent2(X,Y,Y) :- grandparent(X,Y).
[r81] ancestor2(X,Y,Y) :- parent2(X,Y,Y).
[r141] ancestor4(X,Y) :- parent1(X,X,Y) & ancestor2(X,Y,Y).
[r211] answer3(X,Y) :- ancestor4(X,Y).
[r231] ancestor2(X,Z,Y) :- parent2(X,Z,Z) & ancestor4(Z,Y).
[r291] ancestor4(X,Y) :- parent1(X,X,Z) & ancestor2(X,Z,Y).    ∎
```

5. Proof Of Correctness

We first prove that if the algorithm terminates, the resulting program will computes the same set of tuples as the original program. We then prove that for a term-bounded input program, the algorithm always terminates. And finally, as a corollary, we state our theorem.

5.1 Equivalence

To prove equivalence, we use induction on the number of "flatten" steps. Consider the following invariant: at each step, INPUT-SET ∪ OUTPUT-SET is equivalent to the original program. Note that equivalence here means that the two programs compute the same least fixpoint where tuples with function symbols are allowed.

Initially, the invariant trivially holds. Now we want to show that if the invariant holds before a "flatten" step, it still holds after. Let's denote P_1 (resp. P_2) the program INPUT-SET ∪ OUTPUT-SET before (resp. after) the "flatten". P_2 cannot generate more tuples than P_1, since new rules added to P_1 are just instances of some of P_1's rules. P_2 cannot generate less, since all possible invocations of the rule to be flattened are considered, each possibility resulting in a rule copy to be added to P_1. Therefore, in terms of the original IDB predicates, P_1 and P_2 generate the same sets of tuples.

Assume the algorithm terminates. After the last "flatten" step, all rules in INPUT-SET cannot be flattened, i.e. they each contain a goal whose predicate is not flat. These rules are useless, since none can start generating tuples (for this, a rule must have all its subgoals with a flat predicate). Therefore, INPUT-SET can be dropped without affecting the program's answer.

5.2 Termination

To prove termination, we show that there can only be a finite number of "flatten" steps. A step falls into one of the following three cases:

1. a new (flat) predicate (i.e. a new class pattern) gets created.
2. no new predicate gets created but the new head is not subsumed by any existing flat head pattern.
3. no new predicate gets created and the new head is subsumed by some existing flat head pattern.

First, we show there can only be a finite number of steps in case 1. The key point is that the algorithm only generates pattern classes that are possible. That is, for any pattern class generated, there is always an EDB extension that causes a tuple of the corresponding form to be generated (to see this, just consider the OUTPUT-SET, all rules in that set are well founded). Now, if there were an infinite number of pattern classes generated, there must be an EDB extension that causes an infinite number of tuples to be generated. This is not possible since the input program is assumed to be term-bounded. So the number of steps of case 1 is bounded. Second, for a given class pattern, there can only be a finite number of different flat head patterns consistent with the class pattern. Therefore, the number of steps of case 2 is also bounded. As a result, finally, the total number of rule copies the algorithm generates (by means of steps of the first two cases) is bounded, and therefore the number of steps of case 3 is bounded.

As a corollary, the existence of an algorithm for eliminating functional terms from a term-bounded Horn query is a constructive proof of the following theorem:

Theorem 4.4: *Any term-bounded Horn query has an equivalent pure Datalog query.* ∎

6. Concluding Remarks

Mappings between different schemas can be arbitrarily complex. We are interested in those mappings under which query reformulation admits reasonably efficient algorithms. The major contributions of this paper are twofold. It sets an important new research direction that will have significant impact on practical interoperation among heterogeneous systems. It also presents an algorithm that rewrites some Horn queries with function symbols into pure Datalog queries, an important first step in that direction.

Let us emphasize again that the safety problem of Horn queries is not the same as the problem of determining equivalence of Horn queries and pure Datalog queries where we want to eliminate those answers that contain functional terms and especially the ones with unbounded sizes. We are extending our results to larger classes of Horn queries that allow certain forms of unbounded growth of terms. These queries can be expressed as infinite unions of conjunctive subqueries such that after those subqueries involving functional terms are eliminated, the remainders are still equivalent to Horn queries but with no function symbols. Another observation is that queries that result from skolemizing variables in Datalog queries are special cases of general Horn programs. We are investigating ways of exploiting this restriction to extend our results.

While the problem is not entirely solved for Horn queries, there are some essentially non Horn constructs that look interesting. Functional dependencies are a special case of clauses using equality. When represented as Horn clauses (i.e. by substituting predicate literals for equality literals), an interesting question arises as to what extent completeness is still preserved.

This paper focuses on the case of source and target queries in Datalog and the strict requirement of their equivalence. It is worth pointing out other cases of interest that are no less important in practice: source and target queries can range over various classes of language, completeness is no longer strictly required and where mediation now is extended from a query translation task to the added task of coordinating queries over sources.

7. Acknowledgements

We thank Prof. Jeff Ullman for his valuable comments regarding both technical contents and presentation of this material.

References

[1] L. DeMichiel. An approach to performing relational operations over mismatched domains. *IEEE Trans. on Knowledge and Data Engineering*, Vol. 1 No. 4, Dec. 1989, pp. 485-493.

[2] M. Kifer and E. L. Lozinskii. SYGRAPH: Implementing logic programs in a database style. *IEEE Trans. on Software Engineering,* vol. 14 No. 7, July 1988, pp. 922-935.

[3] R. Krishnamurthy, R. Ramakrishnan and O. Shmueli. A Framework for testing safety and effective computability of extended Datalog. *Proc. ACM Symp. on Management of Data,* Chicago, 1988, pp. 154-163.

[4] R. J. Miller, Y. E. Ioannidis and R. Ramakrishnan. The use of information capacity in schema integration and translation. *Proc. Int Conf. on Very Large Data Bases*, Dublin, Ireland, 1993, pp. 120-133.

[5] X. Qian. Semantic interoperation via intelligent mediation. *Proc. 3rd Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, April 1993, pp. 228-231.

[6] O. Shmueli. Decidability and expressiveness aspects of logic queries, *Proc. 6th ACM Symp. on Principles of Database Systems,* 1987, pp. 237-249.

[7] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. 1 and 2*, Computer Science Press, Rockwille MD, 1989.

[8] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, Vol. 25, No. 3, March 1992, pp. 38-49.

[9] X. Qian..Query folding. In *Proc. 12th Int. Conf. on Data Engineering*, 1996, pp. 48-55.

[10] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. Submitted for publication.