Recursive Query Rewriting by Transforming Logic Programs *

Jianguo Lu, Liren Chen, Katia Sycara[†] Jian Lu[‡]

Keywords: industrial application, transformation

1 Introduction

This paper proposes to rewrite database queries by logic program transformations. Query rewriting refers to the activity of determining if and how a query can be answered using a given set of resources, or, using a given set of materialized views[11]. Query rewriting is important because the base relations referred to in a query might be stored remotely and hence too expensive to access, or might be conceptual relations only and hence not existent physically. Query rewriting has applications in query optimization in centralized databases, query processing in distributed databases, and query answering in federated databases. With the widespread use of WWW-based information retrieval, the ability to answer queries using views becoming especially important in integrating semistructured information sources [1].

An example to integrate databases is:

Example 1 *Assume there are two databases (or web site) that provide flight and bus information. The first database provides information on cities connected by Greyhound with one stop-over, and pairs of cities between which Northwest Airlines has non-stop flights. The second database stores the cities that can be reached from Pittsburgh by one non-stop Greyhound bus and one non-stop flight and the airlines that offer these flights. We want to integrate these two databases so that users can ask arbitrary Datalog queries over the EDB predicate* $flight(From, To, Carrier)$*. The intended meaning of* $flight(pittsburgh, nyc, delta)$*, for example, is that Delta airline offers a non-stop flight from pittsburgh to New York City. The two databases we want to integrate can be seen as views over the predicate* $flight$ *and bus:*

```
view1(From,To):-bus(From,X,greyhound),bus(X,To,greyhound).
view1(From,To):-flight(From,To,northwest).
view2(To,Carrier):- bus(pittsburgh,X,greyhound),flight(X,To,Carrier).
```

Now assume a user is interested in the names of the cities that can be reached by plane or bus from Pittsburgh. He/she wishes to take the plane only once and only in the last stop, and do not care about what the carrier is. The following is the corresponding user query:

```
q(From, To, Carrier):-flight(From,To,Carrier).
q(From, To, Carrier):-bus(From,X), q(X, To, Carrier).
query(To):-q(pittsburgh, To, Carrier).
```

*Communications directed to: Jianguo LU, RI, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213. Email: jglu@cs.cmu.edu

[†]School of Computer Science,Carnegie Mellon University

[‡]Department of Computer Science, Nanjing University

Since only those two databases are available, we need to rewrite the query so that it only use the predicates *view*1 and *view*2. This is the problem our paper addresses.

Rewriting of conjunctive queries using conjunctive views is well understood, and shown to be NP-complete[2]. For a subclass of conjunctive queries, the acyclic conjunctive queries, there is a polynomial algorithm [9].

In the context of recursive queries, in general it is undecidable for the problem of whether a Datalog program can be rewritten into an equivalent program that only use views. Besides, in many cases, equivalent rewriting may not exist. Hence it is necessary to find a rewriting that approximates the original query. This study is also motivated by the fact that in applications such as integrated web searching using different sources, users will prefer having approximate results to having nothing returned. The approximate query could be either the generalization or the specialization of the original query, depending on the requirement of the application. [1] studied the the recursive query rewriting using conjunctive views(each view containing only one clause). In their study, a more specific Datalog program is returned as the result of rewriting.

This paper addresses the problem of recursive query rewriting using disjunctive views(each view may contain multiple clauses). As an application of generalization beyond logical implication[3], we provide a method to generalize the recursive query when equivalent rewriting is not obtainable. In the following, we first introduce some preliminary concepts of query rewriting. Secondly, we argue that generalization under the logical implication is not sufficient. Instead, generalization of logic programs beyond logical implication is required and a corresponding method is introduced. Finally we show how to use this method in rewriting recursive queries using disjunctive views.

2 Preliminaries

Our discussion is in the setting of Datalog programs, i.e., a set of function-free Horn clauses. A predicate is an *intensional database predicate*(IDB predicate) if it appears as the head of some rule. Predicates not appearing in any head are *extensional database predicates* (EDB). A query or a view is a Datalog program. A *conjuctive query*(or view) is a Datalog program with a single non-recursive Horn clause. A *disjunctive query*(or view) is a non-recursive Datalog program with multiple clauses. A *recursive query*(or view) is a recursive Datalog program.

Example 2 *Consider the Datalog program:*

```
Q:q(F,T):-flight(F,T).
  q(F,T):-bus(F,T).
  q(F,T):-bus(F,X),q(X,T).
```

Predicates flight, bus are EDB predicates, and q is IDB predicates. It is a recursive query, or a recursive view.

Definition 1 *Let V be a set of views(* $V = \{v1, ..., vn\}$*). A query Q' is an equivalent rewriting of Q that uses the views V if*

- *Q and Q' are equivalent(i.e., produce the same answers for any given database),*
- *Q' contains only IDB predicates of V.*

Example 3 *Consider the following query Q and view V:*

```
Q: q(X,Y):-flight(X,Y).          q(X,Y):-bus(X,Y).
   q(X,Y):-flight(X,U),q(U,Y).   q(X,Y):-bus(X,U),q(U,Y).
V: v(X,Y):-flight(X,Y).          v(X,Y):-bus(X,Y).
```

The rewriting of Q using V is

```
Q': q(X,Y):-v(X,Y).        q(X,Y):-v(X,U),q(U,Y).
```

Definition 2 *A query Q' is a generalizing rewriting of Q that uses the views V if*

- *Q' is more general than Q (i.e., Q' produces more answers for any given database than Q),*
- *Q' contains only IDB predicates of V.*

Example 4 *Consider the following query Q and view V:*

```
Q: q(X,Y):-flight(X,Y).  q(X,Y):-flight(X,U),q(U,Y).
V: v(X,Y):-flight(X,Y).  v(X,Y):-bus(X,Y).
```

The generalizing rewriting of Q using V is

```
Q': q(X,Y):-v(X,Y).      q(X,Y):-v(X,U),q(U,Y).
```

3 Generalization beyond logical implication

Since the seminal paper on generalization of clauses based on θ subsumption[7], there are various extensions in this area. Especially in inductive logic programming, people are using various methods that approximate logical implication, such as inverse resolution and inverse implication to generalize clauses [5]. However, there are many cases that generalization under logical implication relation is not adequate. To illustrate this, we have the following very simple example:

Example 5 *Suppose we have two programs Q1, Q2:*

```
Q1: grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-grandparent(X,Y).
Q2: grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-parent(X,V), parent(V,Y).
```

We can see that Q2 does not logically imply Q1. Hence it is impossible to generalize from Q1 to Q2 under the logical implication. However, under the least Herbrand semantics, Q2 and Q1 are equivalent.

As this example illustrates, it is often more adequate to do generalization based on semantics of our descriptional language itself(i.e., the logic program semantics) than pure logic semantics. In other words, we need to do generalization not restricted by the implication ordering. Instead, we need to go beyond logical implication.

In the following subsections we will first define three kinds of generalization orderings between programs, and introduce the ordering $\succeq_S$ on which our generalization method is based.

3.1 Generalizations

Generalizations are based on some kinds of orderings. Different orderings will result in different generalizations. Before presenting our method of generalization, it is necessary to introduce various notions of orderings between programs so that we can know the ordering on which our generalization is based and its relationship with other orderings.

In the discussion we assume the language has potentially sufficient number of constant symbols. The immediate consequence operator T_P maps Herbrand interpretations to Herbrand interpretations. It denotes one-step deduction using program P. The function corresponding to deductions of any number of steps is denoted by $[\![P]\!]$, and is defined by $[\![P]\!](X) = \cup_{i=0}^{\infty}(T_P+Id)^i(X)$, where Id is the identity function and $(f+g)(X) = f(X) \cup g(X)$. The success set SS(P) is { A: A has a successful SLD-derivation for P }. HB denotes the Herbrand base. It is known that $[\![P]\!](\phi) = SS(P) = lfp(T_P)$.

As for the notions of generalizations in logic programs, there are three layers: generalization between clauses without background theory, between clauses with background theory, and between programs.

They can be defined from either proof theoretic or model theoretic approach. In each layer there are some kinds of orderings, two of the most fundamental are based on *θ subsumption*, $\succeq_\theta$, and *logical implication*, $\succeq_I$. They are usually defined for clauses with or without background theory. In the domain of logic programming, we have to study the ordering between programs. Some orderings between programs can be defined as follows:

Definition 3 *For any two programs P_1 and P_2,*

1. $P_1 \succeq_{T_P} P_2$ *if* $T_{P_1}(X) \supseteq T_{P_2}(X)$ *for every* $X \subseteq HB$.
2. $P_1 \succeq_+ P_2$ *if* $[\![P_1]\!](X) \supseteq [\![P_2]\!](X)$ *for every* $X \subseteq HB$.
3. $P_1 \succeq_S P_2$ *if* $[\![P_1]\!](\phi) \supseteq [\![P_2]\!](\phi)$.

Note that $P_1 \succeq_S P_2$ iff $SS(P_1) \supseteq SS(P_2)$. An important relationship between two orderings is their relative strength. We say an ordering $\succeq_X$ is stronger than $\succeq_Y$ if whenever $P \succeq_X Q$ then $P \succeq_Y Q$, where X and Y denote arbitrary subscripts. $\succeq_X$ is strictly stronger than $\succeq_Y$ if $\succeq_X$ is stronger than $\succeq_Y$ and $\succeq_Y$ is not stronger than $\succeq_X$. A weaker ordering means more programs are involved in the generalization hierarchy, hence it will generate more specific generalization. So, generally speaking, the weaker the ordering, the more desirable of the corresponding generalization.

Theorem 1 *$\succeq_{T_P}$ is strictly stronger than $\succeq_+$, and $\succeq_+$ is strictly stronger than $\succeq_S$.*

As for the correspondence between the usual notions of orderings and the above notions, we can summarize with the following theorem.

Theorem 2 *The arrows go from a stronger ordering to a weaker ordering:*

$$\begin{tikzcd}
\succeq_\theta \arrow[r] & \succeq_{T_P} \arrow[d] \\
\succeq_I \arrow[r, leftrightarrow] & \succeq_+ \arrow[d] \\
 & \succeq_S
\end{tikzcd}$$

Figure 1

A question that naturally arises is: what is the more desirable ordering based on which we generalize programs? As illustrated by the above figure and *Example 5*, we argue that the notion of set inclusion ordering between semantics of logic programs (i.e., $\succeq_S$) should be used as a basis for generalization.

3.2 Rules

Following [6], we view the generalization as a program transformation process. Given a program P_0, by successively applying one of the following transformation rules, a transformation sequence $P_0, ..., P_n$ is generated.

In the set of rules presented below, both deduction (unfolding) and induction (folding) operations are used. This is the key difference between our method and the other approaches in inductive logic programming. The use of this kind of restricted form of deduction operation can be justified by that although it goes down the implication chain, it preserves the semantics of the logic program. This can be depicted in the following:

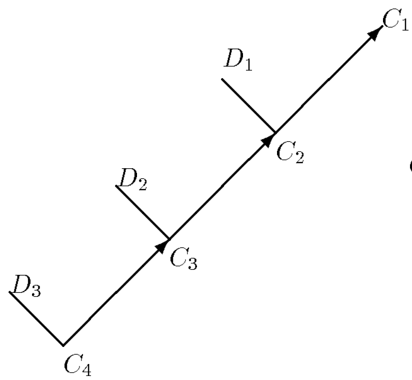

Figure 2

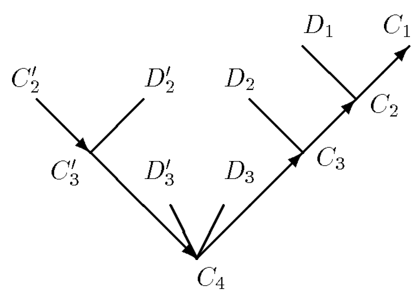

Figure 3

In the pictures above, each $C_{i+1}(C'_{i+1})$ is the result of resolution from $C_i(C'_i)$ and $D_i(D'_i)$. Current approaches perform generalizations in a manner as illustrated in figure 2: They go bottom-up along the inverse of the resolution chain (For instance, along the arrows from C_4 to C_1 in figure 2). They will not be able to tell the relationship between clauses C'_2 and C_1 in figure 3, although it may be true that $C_1 \succeq_S C'_2$. Our method allows going down the resolution chain when necessary (along the arrows from C'_2 to C_4 in figure 3), and then going up(from C_4 to C_1).

Now we are ready to give the $unfolding$ and $folding$ rules to produce generalization beyond implication, similar to the rules in [8].

Rule 1 *(Unfolding) Let P_k be the program $\{E_1, ..., E_r, C, E_{r+1}, ..., E_s\}$, and let C be the clause H:-F,A,G, where A is a positive literal and F and G are (possibly empty) sequences of literals. Suppose*

1. *$\{D_1, ..., D_n\}$ are all the clauses in P_j with $0 \le j \le k$, such that A is unifiable with $hd(D_1), ..., hd(D_n)$, with most general unifier $\theta_1, ..., \theta_n$, respectively, and*
2. *C_i is the clause $(H : -F, bd(D_i), G)\theta_i$, for $i \in \{1, 2, ..., n\}$.*

If we unfold C wrt A using $D_1, ..., D_n$ in P_j, we derive the clauses $C_1, ..., C_n$, and we get the new program $\{E_1, ..., E_r, C_1, ..., C_n, E_{r+1}, ..., E_s\}$.

The unfolding rule is essentially a deduction operation. However, it is a semantics preserving operation (i.e., $P_{k+1} \simeq_S P_k$) due to the requirement in condition 1 that the $D_1, ..., D_n$ are all the clauses that define the predicate A.

Rule 2 *(Folding) Let P_k be the program $\{E_1, ..., E_r, C_1, ..., C_n, E_{r+1}, ..., E_s\}$ and let $\{D_1, ..., D_n\}$ be a subset of clauses in program P_j, Suppose that there exists a positive literal A such that, for $i \in \{1, ..., n\}$,*

1. *$hd(D_i)$ is unifiable with A via most general unifier θ_i,*

2. *C_i is the clause $(H:-F, bd(D_i), G)\theta_i$, where F and G are sequences of literals,*

3. *$\{D_1, ..., D_n\} \cap \{C_1, ..., C_n\} = \phi$.*

If we fold $C_1, ..., C_n$ using $D_1, .., D_n$ in P_j, we derive the clause H :- F,A,G, call it C, and we get the new program $P_{k+1} \equiv \{E_1, ..., E_r, C, E_{r+1}, ..., E_s\}$.

This rule differs the usual folding rule in program transformation. Here we omit the condition that for any clause D of P_k not in $\{D_1, ..., D_n\}$, $hd(D)$ is not unifiable with A. Hence, it is a generalization operation, essentially the same as the absorption in [10]. Here we use a more complicated form than absorption because this rule is also intended to fold multiple clauses at the same time. Condition 3 is necessary to ensure $P_{k+1} \succeq_S P_k$. A simple instance is that, without this restriction, self-folding may occur, and will result in a more specific program.

A folding is *exact* if for any clause D of P_k not in $\{D_1, ..., D_n\}$, $hd(D)$ is not unifiable with A. Otherwise, we call the folding is *generalizing*. An *exact* folding preserves the semantics of the program. A folding or a sequence of foldings is complete if only view predicates remain in the program.

Example 6 *An example of one step of exact folding is folding C1 and C2 in Q using D1 and D2 obtaining Q':*

```
Q: q(X,Y):-flight(X,Y).          (C1)
   q(X,Y):-bus(X,Y).             (C2)
   q(X,Y):-flight(X,U),q(U,Y).
   q(X,Y):-bus(X,U),q(U,Y).
V: v(X,Y):-flight(X,Y).          (D1)
   v(X,Y):-bus(X,Y).             (D2)
Q':q(X,Y):-v(X,Y).
   q(X,Y):-flight(X,U),q(U,Y).
   q(X,Y):-bus(X,U),q(U,Y).
```

Here the literal A is v(X,Y). Since there is no other clauses in V, it is an exact folding.

Example 7 *An example of generalizing folding is folding C1 in Q using D1 obtaining Q'.*

```
Q: q(X,Y):-flight(X,Y).          (C1)
   q(X,Y):-flight(X,U),q(U,Y).
V: v(X,Y):-flight(X,Y).          (D1)
   v(X,Y):-bus(X,Y).             (D2)
Q':q(X,Y):-v(X,Y).
   q(X,Y):-flight(X,U),q(U,Y).
```

Here the literal A is v(X,Y). Since there is another clause D2 in V whose head is v(X,Y) and unifiable with A, this folding is a generalizing folding. A complete folding of Q using V is

```
Q'': q(X,Y):-v(X,Y).
     q(X,Y):-v(X,U),q(U,Y).
```

Theorem 3 *(Correctness) Let $P_0, ..., P_n$ be a transformation sequence of definite programs constructed by using the unfolding and folding rules. We have $P_i \succeq_S P_0$, for $i \in \{1, 2, ..., n\}$. And in general, $P_i \not\succeq_I P_0$, $P_i \not\succeq_\theta P_0$.*

The correctness of the system follows directly from the results of program transformations.

A program is generalized by interleaved applications of the unfolding and folding rules. Following the usual practice in program transformation, we also take the "rules + strategies" approach to generalize programs.

The general strategy is that for a query Q and a set of views V,

1. Try folding Q using clause(s) in V first. If there are complete and exact foldings then stop. We get an equivalent rewriting.
2. When there is no complete folding, unfolding Q as much as possible until complete folding is possible.
3. Unfolding process stops when a compplete generalizing folding is equivalent to some previously existing complete folding.

4 Example

It is easy to see that now we can solve the problem in example 1.

Example 8 *Suppose the the set of view is V. The initial query is Q1.*

```
V:  v1(F,T):-bus(F,X),bus(X,T).     (C1)
    v1(F,T):-flight(F,T).           (C2)
    v2(F,T):-bus(F,T),flight(F,T).  (C3)

Q1: q(F,T):-flight(F,T).            (D1)
    q(F,T):-bus(F,X), q(X,T).       (D2)
```

By unfolding D2 using D1 and D2, we have Q2:

```
Q2: q(F,T):-flight(F,T).
    q(F,T):-bus(F,X), flight(X,T).
    q(F,T):-bus(F,X), bus(X,Y), q(Y,T).
```

By folding `q(F,T):-flight(F,T)` using C2, we have `q(F,T):-v1(F,T)`.
By folding `q(F,T):-bus(F,X), flight(X,T)` using C3, we have`q(F,T):-v2(F,T)`.
By folding `q(F,T):-bus(F,X), bus(X,Y), q(Y,T)` using C1, we have `q(F,T):-v1(F,T),q(Y,T)`.
All together we have a complete generalizing folding Q3:

```
Q3: q(F,T):-v1(F,T).
    q(F,T):-v2(F,T).
    q(F,T):-v1(F,T),q(Y,T).
```

We can see that Q3 is a generalizing rewriting of Q1 using views V.

If we unfold Q2 further, one of the unfoldings that has complete folding is:

```
Q4 : q(F,T):-flight(F,T).
     q(F,T):-bus(F,X), flight(X,T).
     q(F,T):-bus(F,X), bus(X,Y), flight(Y,T).
     q(F,T):-bus(F,X), bus(X,Y), bus(Y,Z), flight(Z,T).
     q(F,T):-bus(F,X), bus(X,Y), bus(Y,Z), bus(Z,W), q(W,T).
```

The complete folding of Q4 is:

```
Q5 : q(F,T):-v1(F,T).
     q(F,T):-v2(F,T).
     q(F,T):-v1(F,X), v1(X,T).
     q(F,T):-v1(F,X), v2(X,T).
     q(F,T):-v1(F,X), v1(X,Y), q(Y,T).
```

Since Q5 is equal to Q3, the unfolding process is stopped here, and the minimal generalization of Q1 is Q3.

5 Conclusions

This paper introduces an active application area, i.e., recursive query rewriting in information integration, in which the techniques in logic program transformation can play an important role.

This work is also related with inductive logic programming. Unlike traditional inductive logic programming that generalizes a set of programs or clauses, we address the problem of generalizing Datalog programs with respect to a given set of views. In dealing with this problem, we found that generalizing under logical implication is not sufficient. Hence, the notion of generalization beyond logical implication is introduced, and a corresponding method is proposed.

Comparing the results in the query rewriting community, rewriting recursive queries is a recognized difficult problem, especially when views are not restricted as conjunctive[1]. We contributed to this area in that we can deal with not only conjunctive views, but also disjunctive views.

References

[1] Oliver M. Duschka, Query planning and optimization in information intehration, Ph.D. dissertation, Stanford University, December 1997.

[2] A.Y. Levy, A.O.Mendelzpn, Y.Sagiv, and D. Srivastava, Answering queries using views. In Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems, 1996, page 95-104.

[3] Jianguo Lu, Jun Arima, Inductive logic programming beyond logical implication, Proceedings of the 7th International Workshop on Arithmetic Learning Theory, 1996 Lecture Notes in Artificial Intelligence 1160.

[4] Maher, M.J., Equivalence of logic programs, Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988.

[5] Muggleton, S., L. De Raedt. Inductive logic programming: theory and methods. Journal of Logic Programming, 19,20:629–679, 1994.

[6] Muggleton, S., Inverting the resolution principle. In Machine Intelligence 12. Oxford University Press, 1991.

[7] Plotkin, G. D., A note on inductive generalization, Machine Intelligence 5, Edinburgh University Press 1970, pp. 153-163.

[8] Pettorossi, A., M. Proietti, Transformation of logic programs: foundations and techniques, J. Logic programming, 1994, 19(20), pp. 261-320.

[9] X. Qian. "Query folding". In Proceedings of the Twelfth International Conference on Data Engineering, pages 48-55. February 1996.

[10] Rouveirol, C., Jean Francois Puget, Beyond inversion of resolution, in Bruce W. Porter and Ray J. Mooney(eds.) Machine learning: Proceedings of the seventh international conference on machine learning, 1990. Morgan Kaufmann. pp. 122-130.

[11] Jeffrey D. Ullman, Information Integration Using Logical Views, Invited paper for ICDT '97.