Preference Datalog
(Summary)

Kannan Govindarajan * Bharat Jayaraman † Surya Mantha ‡

1 Motivation and Approach

The motivation for our work stems from the observation that traditional database query languages allow the user to express only the mandatory requirements on the data to be retrieved from a database. In many applications, it is more natural to express queries in terms of both mandatory, or hard, requirements as well as preferences, or soft requirements. Lacroix and Lavency [8] explore this idea with queries of the form

```
select R where H prefer S
```

whose result is the set of tuples from `R` that satisfy both `H` and `S`, if the set is nonempty; otherwise, i.e., if no tuple satisfies both `H` and `S`, the result is the set of tuples that satisfy just `H`. In other words, `S` is a preference, or a soft requirement. These authors also permit nested preferences and extreme-value preferences, and show that all such statements can still be translated into formulae in the domain relational calculus.

This paper explores the concept of preference in the setting of datalog, a framework that offers more expressiveness than the relational calculus by its ability to support transitive closures and general recursive queries. Just as datalog is a restriction of conventional logic programs, our proposed paradigm, called *preference datalog*, is a restriction of preference logic programs (PLPs), which we recently introduced in [3, 4] for specifying optimization and relaxation problems in a declarative manner. Preference datalog programs (PDPs) are preference logic programs (PLPs) without uninterpreted function symbols. PDPs extend datalog with constructs to specify which predicate is to be optimized and the criterion for optimization as well as which predicate is to be relaxed and the criterion for relaxation.

The contributions of this paper are two-fold:

1. At the language level, we show that preference datalog can directly encode all of the soft requirements in [8]. We also show that the concept of preference provides a modular and declarative (i.e., logical) means for formulating optimization and relaxation queries in deductive databases.

2. At the computation level, we describe bottom-up evaluation methods for preference datalog programs. (The evaluation mechanism for PLPs, on the other hand, uses a top-down scheme [3, 4].) We first develop a *naively-pruned bottom-up evaluation* procedure that is sound and complete for stratified PDP, and then we develop an extension of the *magic templates* method for datalog [11] that constructs an equivalent but more efficient program for bottom-up evaluation.

*Dept. of Computer Science, 226 Bell Hall, SUNY at Buffalo, NY 14260. E-mail: govin-k@cs.buffalo.edu
†Dept. of Computer Science, 226 Bell Hall, SUNY at Buffalo, NY 14260. E-mail: bharat@cs.buffalo.edu
‡Corporate Research & Technology, Xerox Corporation, Webster, NY 14580. E-mail: mantha@wrc.xerox.com

In earlier work, we have given a model-theoretic semantics for PLP using simple concepts from modal logic [3, 4]. Essentially we provided a possible-worlds semantics in which each world is a model for the first-order clauses of the program, and the ordering among the worlds is enforced by the preference clauses. We then gave a declarative semantics for optimization queries in terms of *preferential consequence*, or truth in strongly optimal worlds. (This is in constrast with *logical consequence* which refers to truth in all worlds.) We also gave the semantics of relaxation queries in terms of *relaxed preferential consequence*, i.e., truth in suitably-defined suboptimal worlds. All these concepts carry over to preference datalog programs as well. We refer the reader to the Appendix for a brief review of this material.

Our concept of optimization is closely related to the notion of extreme-value aggregate operations (such as *min* and *max*) in deductive databases. A program with such aggregate operations has an equivalent first-order formulation using negation. There has been a considerable interest recent years in providing a satisfactory semantics for aggregate operations [2, 7, 12, 14, 16]. The most noteworthy semantic difference in our approach is that the ordering among worlds, which is determined from the preferences, explicitly conveys the ordering among solutions, and this ordering is crucial to providing the semantics of relaxation queries.

The idea of hard and soft requirements has also been explored in HCLP (hierarchic constraint logic programming) [17] wherein a constraint may be optionally tagged with a weight, such as `strong`, `weak`, etc. This tag indicates the relative importance of a constraint and serves to organize all constraints into a linear hierarchy. The notion of a *comparator* is introduced in order to compare and order alternative solutions to the hard constraints by determining how well they satisfy the soft constraints. Given a constraint hierarchy, the solutions of interest are those that satisfy the hard constraints and are optimal according to the comparator. We have shown in [4] how HCLP can be directly translated into PLP, thereby showing that PLP is powerful enough to capture the notion of hard and soft constraints in HCLP. PLP is more powerful than HCLP because the latter does not provide a general support for optimization or relaxation queries.

Finally we mention possible avenues for further research. One direction would be to extend PDP by permitting in the bodies of clauses constraints as in CLP [5]. The resulting paradigm can be viewed as an extension of CQL [6] with preferences. For such a paradigm, a top-down evaluation scheme may be more suitable since it has been shown to be more efficient than bottom-up for constraint-based extensions of datalog [15]. Another would be to extend PDP with inductive aggregates, and to study relaxation issues in this extended language. We would also like to examine the complexity of PDP queries, with and without relaxation.

The rest of the paper is organized as follows: Section 2 introduces the syntax of preference datalog programs. Section 3 illustrates optimization and relaxation queries and shows how to translate the preference queries of Lacroix and Lavency into equivalent preference datalog programs. Section 4 describes bottom-up evaluation techniques preference datalog programs and also a magic rewriting technique that generates a more efficiently executable program.

2 Preference Datalog Programs

2.1 Syntax of Preference Datalog Programs

A PDP has two parts, a *first-order theory* without uninterpreted function symbols and an *arbiter*, as described below. The **first-order** part consists of clauses each of which can take one of two forms:

1. $H \leftarrow B_1, \ldots, B_n$, $(n \geq 0)$, *i.e.*, *definite clauses.* Each B_i is an atom that makes use of uninterpreted as well as interpreted predicates (for addition, comparison, etc.) as in datalog. In general, we permit any class of datalog programs for which there exist sound and complete bottom-up evaluation schemes with respect some canonical model.

2. $H \rightarrow G_1, \ldots, G_l \mid B_1, \ldots, B_m$, $(l, m \geq 0)$, *i.e., optimization clauses*[1]. $G_1, \ldots, G_l$ is called the *guard*, and each G_i is a literal that must be satisfied for this clause to be applicable to a goal. The intended meaning of such a clause is that the set of solutions to the head is some subset of that of the body.

Moreover, the predicate symbols appearing in a PDP can be partitioned into three disjoint sets, depending on the kinds of clauses used to define them:

1. *C-predicates* appear only in the heads of definite clauses and the bodies of these clauses contain only other *C-predicates* (C stands for *core*). The *C-predicates* define the mandatory requirements to be satisfied by each solution.

2. *O-predicates* appear in the heads of only optimization clauses (O stands for *optimization*). For each ground instance of an optimization clause, the instance of the *O-predicate* at the head is a candidate for the optimal solution provided the corresponding instance of the body of the clause is true. Also, for simplicity, we assume that the clause-heads and guards of the $\rightarrow$ clauses defining an *O-predicate* are such that, if any two heads unify, the conjunction of constraints enforced by the respective guards and clause-heads are unsatisfiable. We illustrate this requirement by an example in section 3.

3. *D-predicates* appear in the heads of only *definite* clauses and at least one goal in the body of at least one such clause is either an *O-predicate* or a *D-predicate*. (D stands for *derived from O-predicates.*)

The first order theory $\mathcal{T}$ can therefore be divided into two disjoint parts, $\mathcal{T}_C$ and $\mathcal{T}_O$. The definitions of the *C-predicates* make up the core program, $\mathcal{T}_C$, and the definitions of the *O-predicates* and the *D-predicates* make up the optimization program, $\mathcal{T}_O$. A preference datalog program $\mathcal{P}$ can be viewed as a 3-tuple $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$, where $\mathcal{T}_C$ and $\mathcal{T}_O$ together form $\mathcal{T}$ and $\mathcal{A}$ is the arbiter.

Given any preference datalog program, we can stratify the *O-predicates* into levels so that, for any optimization ($\rightarrow$) clause, the level of the *O-predicate* in its head is $\geq$ the level of any *O-predicate* in its body. We can then define the level of the *D-predicates* in the program as the level of the *O-predicate* used in the definition of the *D-predicate*. Furthermore, no *D-predicate* p_1 should be defined in terms of an *O-predicate* p_2 such that an instance of p_1 appears in a $\rightarrow$ clause defining p_2.

The **arbiter part** of a preference datalog program has clauses of the following form:

$$p(\bar{t}) \preceq p(\bar{u}) \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where p is an *O-predicate* and each L_i is some literal (positive or negative atom). In essence this form of the arbiter states that $p(\bar{t})$ *is less preferred than* $p(\bar{u})$ if $L_1, \ldots, L_n$. Each L_i is a literal, i.e., an atom whose head is a *C-predicate* or the negation of such an atom.

Finally, we introduce a **relaxation query** which has the form

`?- RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$,

where p is an *O-predicate* and c is a *C-predicate* or an interpreted predicate. The predicate p is called a *relaxable predicate* and $c(\bar{u})$ is said to be the *relaxation criterion* (`WRT` is read as 'with respect to').[2] The intended meaning of the relaxation goal is as follows: If the optimal solutions to $p(\bar{t})$ satisfies $c(\bar{u})$, then those are the intended solutions to the relaxation goal. Otherwise, the intended solutions are got by restricting the feasible solution space of $p(\bar{t})$ by treating $c(\bar{u})$ as an additional constraint and then finding the optimal solutions in this restricted space.

[1] Variables appearing only on the RHS of the $\rightarrow$ clause are existentially quantified.

[2] In general, a relaxation goal may also appear in the body of an *O-predicate* or a *D-predicate*, but we do not consider such goals in this paper.

3 Query Paradigms in Preference Datalog

3.1 Relaxation Queries

Consider the predicates `person(Name, Sex, DateOfBirth)` and `ancestor(Child,Ancestor)` from the well-known family database example. We can give a logical specification for the oldest ancestors as follows:

```
oldest_anc(X,Y) → ancestor(X,Y).
oldest_anc(X,Y1) ⪯ oldest_anc(X,Y2) ← person(Y1,_,DOB1), person(Y2,_,DOB2), DOB2 < DOB1.
```

The first clause introduces the optimization predicate `oldest_anc`, whose optimal solutions are a subset of the solutions for `ancestor` (hence the use of a $\rightarrow$ clause). The arbiter clause states the criterion for optimization: given two solutions for `oldest_anc`, the one with the smaller `DOB` is preferred.

To illustrate the relaxation goal, suppose that we want to use the above definition of `oldest_anc` to find the oldest female ancestors. First note that the query

```
?- oldest_anc(X,Y), person(Y,female,_)
```

is not correct: If all the oldest ancestors are male, the above query computes no answer. We can use the relaxation goal to solve our stated problem, as follows.

```
?- RELAX oldest_anc(X,Y) WRT person(Y,female,_).
```

This goal works by restricting the feasible solution space of `oldest_anc(X,Y)` by treating `person(Y,female,_)` as an additional constraint and then finding the optimal solutions in this restricted space.

We now provide an example of hierarchic optimization and relaxation. Given a base predicate `edge(X, Y, Cost, Distance)`, representing the travel cost and distance from a city `X` to `Y`, we want to find the best "get-away spots" from given a source, where these are defined as those destinations whose shortest distance from the source have the largest value. To solve this problem, we first give a dynamic-programming formulation for the minimum-distance problem as follows:

```
min_dist(X,X,N,0,0).
min_dist(X,Y,1,C,D) →  X <> Y | edge(X,Y,C,D).
min_dist(X,Y,N,C,D) →  N > 1, X <> Y |
                       min_dist(X,Z,1,C1,D1),
                       N1 = N - 1, min_dist(Z,Y,N1,C2,D2),
                       C = C1 + C2, D = D1 + D2.

min_dist(X,Y,_,_,D1) ⪯ min_dist(X,Y,_,_,D2) ← D2 < D1.
```

This formulation expresses the *optimal sub-problem property* of the minimum-distance problem: each call to `min_dist` uses only the *optimal* solutions to subsequent recursive calls on `min_dist`. Thus pruning of sub-optimal solutions occurs at each recursive call. Note that, had we replaced the $\rightarrow$ clauses in the above definition by $\leftarrow$ clauses, the `min_dist` predicate would no longer be an optimization predicate and the desired pruning would not occur. This example also shows the need for the guard: the conditions `X <> Y` and `N > 1, X <> Y` should be read as antecedents of the implication.

We next define the best get-away spots as those destinations that are farthest away from the source (where n below is the number of nodes in the graph):

```
best_spots(X, Y, Cost, Dist) → min_dist(X, Y, n, Cost, Dist).
best_spots(X, _, _, D1) ⪯ best_spots(X, _, _, D2) ← D1 < D2.
```

Suppose now that we want to find the best get-away spots within a certain budget (assume that travel cost is linearly proportional to travel distance). Since all of the best get-away spots may exceed our budget, we compute suboptimal solutions to `best_spots` with respect to our travel budget, as follows:

```
?- RELAX best_spots(X, Y, Cost, _) WRT Cost ≤ Budget.
```

3.2 Preferences in Relational Calculus

We now show that the kinds of queries expressible in the framework of [8] can be encoded in the framework of PDP. We take the liberty to stream-line their syntax in this paper. We build on queries of the form `select R where P`, whose corresponding relational calculus expression for the query is $\{\bar{r} \in R | P(\bar{r})\}$. We can translate queries in the domain relational calculus into datalog in the most natural way. A query with a simple preference clause has the form

`select X where Q prefer P1.`

Operationally, we first try to satisfy $\mathtt{Q} \wedge \mathtt{P1}$. If the resulting answer set is empty, we drop `P1` and just report the tuples that satisfy `Q`. The set of solutions to the query can be expressed as follows:

$$\{\bar{x} \in X | Q(\bar{x}) \wedge [\exists \bar{y}[Q(\bar{y}) \wedge P1(\bar{y})] \Rightarrow P1(\bar{x})]\}$$

The equivalent PDP query is $\mathtt{X1}(\bar{u})$, where `X1` is an optimization predicate defined as follows:

$$\mathtt{X1}(\bar{t}) \rightarrow \mathtt{Q}(\bar{t}),\ \mathtt{X}(\bar{t}).$$

$$\mathtt{X1}(\bar{t}_1) \preceq \mathtt{X1}(\bar{t}_2) \leftarrow \mathtt{P1}(\bar{t}_2),\ \neg\ \mathtt{P1}(\bar{t}_1).$$

Suppose that no solution to `Q` satisfies `P1`. The arbiter is then not applicable and all solutions to `Q` are reported as solutions to the query, as desired. However, any solution for `Q` that satisfies `P1` will be preferred over (i.e., will prune) solutions for `Q` that do not satisfy `P1`. This is precisely the effect that is specified by the preference clause in the original query.

Nested Preference Queries: In many situations, some soft requirements are more important than others. This can be captured in [8] by a nested preference clause of the form:

`select X where Q prefer P1 then P2.`

Essentially, tuples in `X` that satisfy `Q` as well as `P1` and `P2` are returned as answers. However, if there are no such tuples, then among the tuples that satisfy `Q`, we pick those that satisfy `P1`. However, if the latter set is empty, but there are some that satisfy `Q` and `P2`, we return them instead. Formally, the answer set is:

$$\begin{array}{l}\{\bar{x} \in X | Q(\bar{x}) \wedge [\exists \bar{y} Q(\bar{y}) \wedge P1(\bar{y}) \wedge P2(\bar{y}) \Rightarrow P1(\bar{x}) \wedge P2(\bar{x})] \\ \quad \wedge \quad [\neg \exists \bar{y}[Q(\bar{y}) \wedge P1(\bar{y}) \wedge P2(\bar{y})] \wedge \exists \bar{y}[Q(\bar{y}) \wedge P1(\bar{y})] \Rightarrow P1(\bar{x})] \\ \quad \wedge \quad [\neg \exists \bar{y}[Q(\bar{y}) \wedge P1(\bar{y}) \wedge P2(\bar{y})] \wedge \neg \exists \bar{y}[Q(\bar{y}) \wedge P1(\bar{y})] \wedge \exists \bar{y}[Q(\bar{y}) \wedge P2(\bar{y})] \Rightarrow P2(\bar{x})]\}\end{array}$$

The equivalent query in PDP makes use of hierarchic optimization. The translated query is $\mathtt{X2}(\bar{u})$ where the optimization predicate `X2` is defined as follows:

$$\mathtt{X2}(\bar{t}) \rightarrow \mathtt{X1}(\bar{t}).$$

$$\mathtt{X2}(\bar{t}_1) \preceq \mathtt{X2}(\bar{t}_2) \leftarrow \mathtt{P2}(\bar{t}_2),\ \neg\ \mathtt{P2}(\bar{t}_1).$$

$$\mathtt{X1}(\bar{t}) \rightarrow \mathtt{X}(\bar{t}),\ \mathtt{Q}(\bar{t}).$$

$$\mathtt{X1}(\bar{t}_1) \preceq \mathtt{X1}(\bar{t}_2) \leftarrow \mathtt{P1}(\bar{t}_2),\ \neg\ \mathtt{P1}(\bar{t}_1).$$

Equally-Important Preferences: In certain situations, it is important to enforce multiple preferences that have the same importance. These were formulated in the framework of [8] as follows:

`select X where Q prefer P1 or P2`

Informally, the effect of this query is to pick from `X` all tuples that satisfy `Q`. Furthermore, if the solutions to `Q` satisfy both `P1` and `P2`, then those are the answers to the query. However, if the solutions to `Q` do not satisfy both `P1` and `P2`, the intended solutions are those that satisfy at least one of `P1` and `P2`. However, if none of the solutions of Q satisfied either P1 or P2, all the solutions to Q are reported as answers. Formally, the answer set is:

$$\{\bar{x}|Q(\bar{x}) \wedge [\exists\bar{y}[Q(\bar{y}) \wedge P1(\bar{y}) \wedge P2(\bar{y})] \Rightarrow P1(\bar{x}) \wedge P2(\bar{x})] \\ \wedge [\neg\exists\bar{y}[Q(\bar{y}) \wedge P1(\bar{y}) \wedge P2(\bar{y})] \wedge \exists\bar{y}[Q(\bar{y}) \wedge [P1(\bar{y}) \vee P2(\bar{y})]] \Rightarrow [P1(\bar{x}) \vee P2(\bar{x})]]\}$$

The equivalent PDP is as follows. Note that we have taken the liberty to use complex boolean formulae in the antecedents of arbiter clauses below, recognizing that such formulae can be translated into equivalent PDP clauses.

$$\mathtt{X1}(\bar{t}) \rightarrow \mathtt{X}(\bar{t}),\ \mathtt{Q}(\bar{t}).$$

$$\mathtt{X1}(\bar{t}_1) \preceq \mathtt{X1}(\bar{t}_2) \leftarrow \mathtt{P1}(\bar{t}_2),\ \mathtt{P2}(\bar{t}_2),\ \neg\ (\ \mathtt{P1}(\bar{t}_1) \wedge \mathtt{P2}(\bar{t}_1)).$$

$$\mathtt{X1}(\bar{t}_1) \preceq \mathtt{X1}(\bar{t}_2) \leftarrow (\mathtt{P1}(\bar{t}_2) \vee \mathtt{P2}(\bar{t}_2)),\ \neg\ (\ \mathtt{P1}(\bar{t}_1) \vee \mathtt{P2}(\bar{t}_1)).$$

Extreme-Value Preferences: Preference clauses with a maximum or a minimum preference are formulated in the framework of [8] as follows:

```
select X where Q prefer maximum T wrt P
```

Essentially, we want to pick from the relation X those tuples that satisfy Q and that are mapped by the mapping relation P to the highest possible value. Formally the solution set is:

$$\{\bar{x} \in X|Q(\bar{x}) \wedge [\exists\bar{y}\exists t Q(\bar{y}) \wedge P(\bar{y},t) \Rightarrow P(\bar{x},t1) \wedge t1 = \mathbf{max}\{t|\exists\bar{y}[P(\bar{y},t) \wedge Q(\bar{y})]\}]\}$$

where P is the relation that links the $\bar{x}$'s to the t's. The equivalent PDP program as before introduces a new *O-predicate* X1 as before with the following clauses:

$$\mathtt{X1}(\bar{t},u) \rightarrow \mathtt{X}(\bar{t}),\ \mathtt{Q}(\bar{t}),\ \mathtt{P}(\bar{t},\ u).$$

$$\mathtt{X1}(\bar{t}_1,u_1) \preceq \mathtt{X1}(\bar{t}_2,u_2) \leftarrow u_1 < u_2.$$

Thus PDP can express all of the preference clauses introduced in [8]. Note that we have only made use of optimization clauses in the translated programs, i.e., relaxation queries are not needed in these translations.

4 Bottom-up Evaluation for Preference Datalog

We first consider bottom-up evaluation of stratified PDPs and then extend the technique for computing answers to relaxation queries and non-stratified preference datalog programs. Finally, we present a magic rewriting technique for PDP so that the rewritten program does not make inferences that are not relevant to the query.

4.1 Naively Pruned Evaluation for PDP

A preference datalog program is stratified if the level l_1 of an *O-predicate* O_1 appearing in the body of a clause defining another *O-predicate* O_2 of level l_2 is such that $l_1 < l_2$. Suppose the given preference datalog program has k levels; the naively pruned bottom-up evaluation of the program has $k+1$ stages as follows:

1. Compute bottom-up the canonical supported model of the core program, and let this be M_0.

2. Consider the definitions of the *O-predicates* at level 1. Treating the $\rightarrow$ clauses just as $\leftarrow$ clauses and starting from M_0, compute bottom-up the canonical model of the clauses defining *O-predicates* at level 1. Let this set be O_1^{up} (*up* stands for *unpruned*). Suppose S is a set of ground atoms and $\mathcal{A}$ is a set of arbiter clauses. The result of applying the arbiter $\mathcal{A}$ on the set S is a set T defined as follows:

$T = \{p(\bar{t}) \in S|$ there does not exist an arbiter clause $A \in \mathcal{A}$ of the form $p(\bar{u}_1) \preceq p(\bar{u}_2) \leftarrow c_1(\bar{v}_1), \ldots, c_n(\bar{v}_n)$, an atom $p(\bar{t}_1) \in S$ and a substitution θ such that θ is the most general unifier of the set of equations $\{p(\bar{t}) = p(\bar{u}_1), p(\bar{t}_1) = p(\bar{u}_2)\}$ and $c_1(\bar{v}_1)\theta, \ldots, c_n(\bar{v}_n)\theta$ is true. The set T is said to be the *result* of applying the arbiter on the set S.

Suppose the result of applying the arbiter on the set O_1^{up} is O_1. Now consider the definitions of *D-predicates* of level 1. Starting with the set O_1, we construct bottom-up the canonical model of program whose clauses are the definitions of the *D-predicates* of level 1. Let this set be M_1.

3. For each $i \geq 2$, we start from M_{i-1} and construct M_i as in step 2.

We now state the soundness and completeness results for the bottom-up evaluation technique and provide proof sketches. We start with the following lemma which provides the justification for interpreting the $\rightarrow$ clauses as $\leftarrow$ clauses in the bottom-up evaluation of the program. This lemma is used in the soundness and completeness proofs for the evaluation technique.

Lemma 1 *Suppose a set $S \cup S_p$ of atoms is a supported model for a clause of the form $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ where S contains instances of goals in the bodies of the clause and S_p contains instances of the head of the clause. Then, for any S'_p such that $S'_p \subseteq S_p$, the set $S \cup S'_p$ is a supported model for a clause of the form $p(\bar{t}) \rightarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ where the variables that appear only on the right hand side of the $\rightarrow$ are existentally quantified. Furthermore, no supported model for the $\rightarrow$ clause can be a proper superset of $S \cup S_p$.*

Proof Sketch: Consider the clause of the form:

$$p(\bar{t}) \rightarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$$

Since the variables that appear only on the right hand side of the $\rightarrow$ clause are existentially quantified, any set M of atoms such that the set of instances of the head of the clause in M is some subset of the set of instances of the goals in the body of the clause is a model for the clause. Since two supported models of the same normal program cannot be proper subsets of each other, we have our requisite result. ■

Theorem 1 (Soundness and Completeness) *Given a stratified preference datalog program with n levels, an atom A belongs to the set M_n constructed by the naively pruned bottom-up evaluation procedure if and only if A is a preferential consequence of the program.*

Proof Sketch: The proof is by induction on the level of the predicate that appears at the head p of the atom $A = p(\bar{t})$. It makes use of soundness and completeness of the bottom-up evaluation for datalog. The proof for the case where p is a *C-predicate* follows from the soundnesss and completeness of the bottom-up evaluation technique being used. Suppose p is an *O-predicate* at level 1. As a result of the lemma 1 and the fact that every ground instance of an *O-predicate* is supported by atmost one $\rightarrow$ clause, we can treat the $\rightarrow$ clause just as a $\leftarrow$ clause for the purposes of bottom-up evaluation. Treating the $\rightarrow$ clause as a $\leftarrow$ clause allows us to compute all possible solutions some of which get pruned by the arbiter. Therefore, bottom-up evaluation is sound and complete for *O-predicates* at level 1. The inductive case is similar. ■

Relaxation Queries: We now describe a bottom-up procedure to compute the answers to relaxation queries. Given a preference datalog program without relaxation goals in the bodies of clauses and a relaxation query to the program, we first rewrite the program with the relaxation query into a preference datalog program and a new query. Given a preference datalog program P and a relaxation query of the form `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$, the set $relax(p(\bar{t}), c(\bar{u}))$ of **relaxed clauses** for p introduces a new *O-predicate* $relax\_p$, the relaxed version of p, into the program. The clauses for $relax\_p$ are obtained from the clauses for p by modifying every clause for p that is applicable to $p(\bar{t})$ in the relaxation goal is by adding $c(\bar{u})$ to the body;

clauses for p that are not applicable to $p(\bar{t})$ are not modified. If some instance of a clause for p is applicable to $p(\bar{t})$ then we obtain two clauses in the relaxed version, one that is applicable to $\bar{t}$ with the body containing $c(\bar{u})$, and the other that is not applicable to $\bar{t}$ with the body as before. The arbiter clauses for $relax\_p$ are obtained from the arbiter clauses for p in a similar manner. The reader is referred to the appendix for an example. Furthermore, the **relaxed query** corresponding to the relaxation query RELAX $p(\bar{t})$ WRT $c(\bar{u})$ is $relax\_p(\bar{t})$. The soundness of the translation is established by the following theorem.

Theorem 2 (Correctness of Translation) *Given a preference datalog program P and a relaxable goal $G =$ RELAX $p(\bar{t})$ WRT $c(\bar{u})$, $p(\bar{t})\theta \wedge\ c(\bar{u})\theta$ is a relaxed preferential consequence of P and G if and only if $relax\_p(\bar{t})\theta$ is a preferential consequence of the program $P \cup relax(p(\bar{t}), c(\bar{u}))$.*

Proof Sketch: Suppose there is an instance $relax\_p(\bar{t})\theta$ in a world in the intended preference model for $P \cup relax(p(\bar{t}), c(\bar{u}))$. Clearly, the instance $p(\bar{t})\theta$ belongs to a world in the relaxed intended preference model of P and G as $c(\bar{u})\theta$ is true. Similarly, we can show that if an instance $p(\bar{t})\sigma$ occurs in any world in the relaxed intended preference model of P and G, then the instance $relax\_p(\bar{t})\sigma$ occurs in some world in the intended preference model of $P \cup relax(p(\bar{t}), c(\bar{u}))$. The arbiter clauses for $relax\_p$ enforce the same ordering when the solutions to p satisfy c. ■

The naively-pruned bottom-up evaluation technique for a stratified preference datalog program P and a relaxation query G consists of the following steps:

1. Augment P with the relaxed clauses for the relaxable predicate in G to produce program P'. Let G' be the relaxed query corresponding to the relaxation query G.
2. Perform naively-pruned bottom-up evaluation of the program P'.

Theorem 3 (Soundness and Completeness) *Given a stratified preference datalog program P and a relaxation query $G =$ RELAX $p(\bar{t})$ WRT $c(\bar{u})$, an instance $relax\_p(\bar{t})\theta$ is computed by the naively pruned bottom-up evaluation technique applied to the program $P \cup relax(p(\bar{t}), c(\bar{u}))$ if and only if $p(\bar{t})\theta$ is a relaxed preferential consequence of P and G.*

Proof Sketch: The proof follows from the correctness of the translation and soundness and completeness of naively-pruned bottom-up evaluation for stratified preference datalog programs. ■

We have also developed a technique for bottom-up evaluation of general preference datalog programs, i,e., those that are not necessarily stratified. In this case, the bottom-up evaluation mechanism keeps track of a pair of sets of the form $\langle M_i^p, M_i^{up}\rangle$.In the beginning, the pair of sets of interest is $\langle M_0, M_0\rangle$ where M_0 is the canonical model of the core program. Given a pair $\langle M_i^p, M_i^{up}\rangle$ at any stage i, the set M_i^p is said to be the **pruned set** at stage i and the set M_i^{up} is said to be the **unpruned set** at stage i. At each stage i, the unpruned set O_i^{up} of the *O-predicates* at level i are generated starting from the pruned set at stage $i-1$, i.e., M_{i-1}^p. However, we also generate the complete set of instances of *O-predicates* at level i N_i^{up} starting from the unpruned set M_{i-1}^{up}. We now prune the set O_i^{up} using the set N_i^{up} to produce the pruned set of *O-predicates* at level i, O_i^p. The first and second components of the pair of sets $\langle M_i^p, M_i^{up}\rangle$ are constructed bottom-up from the set O_i^p and the set N_i^{up} respectively. If the preference datalog program P has n levels, the pair of sets $\langle M_n^p, M_n^{up}\rangle$ is the result of the naively pruned bottom-up evaluation of P.

Theorem 4 (Soundness) *Given a preference datalog program P with n levels, suppose the naively pruned bottom up evaluation produces a pair $\langle M_n^p, M_n^{up}\rangle$ then any atom $A \in M_n^p$ is a preferential consequence of P.*

Proof Sketch: The proof is by induction on the number of levels of *O-predicates* in the program. Since we also use the unpruned sets to perform pruning, we can show that any atom that survives pruning is indeed a preferential consequence. ■

Naively pruned bottom-up evaluation is not complete for arbitrary non-stratified preference datalog programs. This is because the $\rightarrow$ clauses and the arbiter clauses for some non-stratified *O-predicate* may be such that the $\preceq$ relation among the models is reflexive. In particular, the optimal worlds may be related to themselves, and according to the arbiter, one optimal solution may be preferred over another even though both solutions exist in the same world. However, naively-pruned bottom-up evaluation does not keep track of the worlds when applying the arbiter so as not to incur the complexity of modal theorem proving [1, 13].

We can also extend the bottom-up evaluation technique presented here to the case when the preference datalog program has relaxation goals in the bodies of clauses. Essentially, we consider the unpruned model for the *O-predicate* in the relaxation goal and use the relaxation criterion of the relaxation goal to filter out instances of the relaxable predicate that do not satisfy the relaxation criterion. We then prune this filtered set using the arbiter clauses for the *O-predicate*.

4.2 Magic Rewriting for PDPs

The bottom-up evaluation mechanism outlined above is inefficient in that it may compute the set of all preferential consequences of the program in order to answer a given query. In this subsection, we describe an improvement based on the magic rewriting presented in [10, 11]. We first consider queries to preference datalog programs that are not relaxation queries and later describe the technique for making the evaluation of relaxation queries more efficient by magic rewriting.

Let P be a PDP program and a query $Q = q(\bar{s})$. Intuitively, for each derived predicate p in P, we define a new predicate *magic_p* such that the bottom-up evaluation of the program generates a fact $magic\_p(\bar{u})$ if while solving for $q(\bar{s})$ the top-down evaluation had to solve for $p(\bar{u})$. Let P^m be the program that is the result of performing the magic template transformation on the program-query pair $\langle P, Q\rangle$. Initially P^m is empty. Furthermore, we assume that left to right *sips* [11] is used.

1. For every rule R in P of the form $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$, add to P^m the rule:
$$p(\bar{t}) \leftarrow magic\_p(\bar{t}), p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n).$$
2. For every rule R in P of the form $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ and for each literal in the body $p_i(\bar{t}_i)$, where p_i is a derived predicate symbol, add to P^m the rule:
$$magic\_p_i(\bar{t}_i) \leftarrow magic\_p(\bar{t}), p_1(t_1), \ldots, p_{i-1}(\bar{t}_{i-1})$$
3. For every rule R in P of the form $p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) | p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$, add to P^m the rule:
$$p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) | magic\_p(\bar{t}), p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n),$$
4. For every rule R in P of the form $p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) | p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ and for each literal in the body $p_i(\bar{t}_i)$, add to P^m the rule:
$$magic\_p_i(\bar{t}_i) \leftarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m), magic\_p(\bar{t}), p_1(t_1), \ldots, p_{i-1}(\bar{t}_{i-1})$$
5. For every rule R in P of the form $p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) | p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ and for each literal in the body $p_i(\bar{t}_i)$, where p_i is not an *O-predicate*, add to P^m the rule:
$$magic\_p_i(\bar{t}_i) \leftarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m), magic\_p(\bar{t}), p_1(t_1), \ldots, p_{i-1}(\bar{t}_{i-1})$$
6. Create a seed fact $magic\_q(\bar{s})$ from the query.

The reader is referred to the appendix for an example of the application of the magic rewriting technique.

Theorem 5 *Given a PDP P and a normal query $Q = q(\bar{t})$, suppose P^m is the result of applying magic rewriting to the program-query pair $\langle P, Q \rangle$; then P and P^m are equivalent with respect to the answers to the query $q(\bar{t})$.*

Proof Sketch: The proof is based on the correctness of interpreting the $\rightharpoonup$ clauses as $\leftharpoonup$ clauses and correctness of magic rewriting for $\leftharpoonup$ clauses [10]. ■

We now briefly outline the technique for computing answers to relaxation queries to preference datalog programs. Given a preference datalog program P and a relaxation query Q, we first augment P with the relaxed clauses for the relaxable predicate in Q to produce the program P'. Furthermore, suppose the relaxed query corresponding to Q is Q'. Note that P' is a preference datalog program program without any relaxation goals and Q' is a normal query, not a relaxation query. We can then perform magic rewriting on the program query pair $\langle P', Q' \rangle$ and by the correctness of the translation, we can prove the above theorem for relaxation queries too. We are investigating extensions of magic rewriting for preference datalog programs that have relaxation goals in the bodies of clauses.

References

[1] B.F. Chellas. *Modal Logic, An Introduction.* Cambridge University Press, 1980.

[2] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and Maximum Predicates in Logic Programming. In *Proc. 10th ACM Symp. on Principles of Database Systems*, pages 154–163, 1991.

[3] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming. In *Proc. 12th Intl. Conf. on Logic Programming*, pages 731–745. MIT Press, 1995.

[4] K. Govindarajan, B. Jayaraman, and S. Mantha. Optimization and Relaxation in Constraint Logic Languages. 1996. To appear in *Proc. 23rd ACM Symposium on Principles of Programming Languages.* Also available as Department of Computer Science, SUNY at Buffalo Technical Report TR 95-37.

[5] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[6] P. C. Kannelakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. In *Proc. ACM Symp. on Principles of Database Systems*, pages 299–313, 1990.

[7] D. B. Kemp and P. J. Stuckey. Semantics of Logic Programs with Aggregates. In *Proc. International Logic Programming Symposium*, 1991.

[8] M. Lacroix and P. Lavency. Preferences: Putting More Knowledge into Queries. In *Proc. 13th Intl. Conf. on Very Large Data Bases*, pages 217–225, 1987.

[9] S. Mantha. *First-Order Preference Theories and their Applications.* PhD thesis, University of Utah, November 1991.

[10] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. 5th Joint Intl. Conf. and Symp. on Logic Programming*, pages 140–159, 1988.

[11] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering.* Kluwer Academic Publishers, 1992.

[12] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proc. ACM Symp. on Principles of Database Systems*, pages 114–126, 1992.

[13] E. Spaan. *Complexity of Modal Logics.* PhD thesis, CWI, University of Amsterdam, 1993.

[14] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the Well-Founded and Valid Semantics for Aggregation. In *Proc. International Logic Programming Symposium*, pages 590–608, 1993.

[15] David Toman. Top-down Beats Bottom-up for Constraint Based Extensions of Datalog. 1995. To appear in Proc. Intl. Logic Programming Symposium.

[16] A. Van Gelder. The Well-founded Semantics of Aggregation. In *Proc. 11th ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[17] M. Wilson and A. Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16:277–318, 1993.

Appendix A

A.1 Model Theory of Preference Logic Programs

We start the description of the model theory with a brief review of the logic of preference. The syntax of the logic of preference introduced by Mantha [9] extended the syntax of first-order logic by introducing a unary modal operator $\mathcal{P}_f$ with the associated rule of formation: If F is a formula, then so is $\mathcal{P}_f F$.

Preference logic programs are to be viewed as theories in this logic by translating each *definite preference clause* $p(\bar{t}) \preceq p(\bar{u}) \leftarrow L_1, \ldots, L_n$ into a clause of the form

$$p(\bar{t}) \rightarrow \mathcal{P}_f(p(\bar{u}) \ \wedge \ L_1 \ \wedge \ \ldots \ \wedge \ L_n)$$

A **preference model** $\mathcal{M}$ is a triple $\langle \mathcal{W}, \preceq, \mathcal{V} \rangle$, where $\mathcal{W}$ is a non-empty set of possible worlds, $\preceq$ is a binary relation over $\mathcal{W}$, and $\mathcal{V}$ is a valuation function that determines the truth of atomic formulae at individual worlds. Boolean connectives such as $\wedge, \vee, \neg, \rightarrow, etc.$ have the standard interpretation. The semantics of preference formulae of the form $\mathcal{P}_f F$ is given as follows[3]:

$$\models^w_{\mathcal{M}} \mathcal{P}_f F \ \text{ iff } \ (\forall v \in \mathcal{W}) \ [(\models^v_{\mathcal{M}} F) \rightarrow (w \preceq v)].$$

Informally, $\mathcal{P}_f F$ is true in a world w in a preference model iff every world v where F is true is related to w by the relation $w \preceq v$. If $\mathcal{P}_f F$ is true at a world w, then F is said to be a **preference criterion** at world w. In other words, any world v where F is true is at least as good as w. A preference model $\mathcal{M}$ is said to be **supported** if and only if, for any two worlds w and v, if $w \preceq v$ then, there is a formula $\mathcal{P}_f A$ such that $\models^w_{\mathcal{M}} \mathcal{P}_f A$ and $\models^v_{\mathcal{M}} A$. A supported preference model is also the preference model that minimizes the relation $\preceq$. Given a preference model $\mathcal{M} = \langle \mathcal{W}, \preceq, \mathcal{V} \rangle$, a world $w \in \mathcal{W}$ is said to be **strongly optimal** if and only if there is no world w' different from w such that $w \preceq w'$.

[3] We write $\models^w_{\mathcal{M}} F$ to mean that the formula F is assigned the truth value *true* by $\mathcal{V}$ in the world w in the preference model $\mathcal{M}$.

A.1.2 Models for Preference Logic Programs

We build models for preference logic programs in stages. We stratify the *O-predicates* into levels so that, for any optimization ($\rightarrow$) clause, the level of the *O-predicate* in its head is $\geq$ the level of any *O-predicate* in its body. This can be done by constructing the predicate call-graph among the *O-predicates* in the program and topologically sorting the graph to obtain the ordinal levels of the *O-predicates*. First we consider programs that have atmost one level of *O-predicates*. We shall henceforth assume in this subsection that a preference logic program P satisfies this requirement. At the end of this subsection, we will extend the semantics to programs with any number of levels *O-predicates*.

The **pre-interpretation** I of interest to us interprets functions such as $+$ over the appropriate domain (as in CLP) and leaves all other function symbols uninterpreted (as in Herbrand interpretations). For the rest of the paper we fix this pre-interpretation I. We are interested in a canonical model for the core program derived from the pre-interpretation I, as it specifies the constraints to be satisfied. For definite programs, this is given by the least I-based model for the program.

Following Mantha [9], we give a possible-worlds semantics for preference logic programs. Given a preference logic program, its preference model is constructed by first by defining the worlds (the valuation function $\mathcal{V}$) in the model Essentially, each world is constructed by extending the canonical model for $\mathcal{T}_C$ by including instances of *O-predicates* so that it becomes a model for $\mathcal{T}_C \wedge \mathcal{T}_O \wedge \mathcal{A}$. The instances of *D-predicates* (and *O-predicates*) at each world have to be *supported*, i.e. if $p(\bar{t})$ is a ground instance of a *D-predicate* (or an *O-predicate*), then there is a clause in the program whose head unifies with $p(\bar{t})$ and the instances of the goals in the body are present in the world. Satisfying $\mathcal{A}$ enforces the ordering among the worlds in the preference model.

Definition 1 *Given a preference logic program P, the* **intended preference model** $\mathcal{M}$ *is the preference model* $\langle \mathcal{W}, \preceq, \mathcal{V} \rangle$ *that maximizes the number of worlds in* $\mathcal{W}$ *and minimizes the relation* $\preceq$ *(i.e. is supported) and is such that* $\mathcal{V}$ *assigns different interpretations to different worlds.*

Consider the following formulation of shortest path:

```
path(X,Y,C,[e(X,Y)])      ←  edge(X,Y,C).
path(X,Y,C,[e(X,Z)|L1]) ←  edge(X,Z,C1), path(Z,Y,C2,L1),
                             C = C1 + C2.

sh_dist(X,Y,C,L)          →  path(X,Y,C,L).
sh_dist(X,Y,C1,L)         ⪯  sh_dist(X,Y,C2,L1) ← C2 < C1.
```

We use the above program to illustrate the model theory. Consider a directed graph with the following edges {`edge(a,b,5), edge(b,c,10), edge(a,c,25)`} . The canonical model M for $\mathcal{T}_C$ of interest to us here is the set:

{`edge(a,b,5), edge(b,c,10), edge(a,c,25)`} $\cup$
{`path(a,b,5,[e(a,b)]), path(b,c,10,[e(b,c)]), path(a,c,25,[e(a,c)]),`
`path(a,c,15,[e(b,c),e(a,b)])`}.

The following is a fragment of the intended preference model for the example program considered above and is used to illustrate how the ordering among worlds is enforced. For brevity, we have considered worlds that have instances of `sh_dist(a,b,_,_)` and `sh_dist(b,c,_,_)` since there is only one path between the associated vertices in the graph. These worlds are shown below:

1. $M \cup$ {`sh_dist(a,b,5,[e(a,b)])`, `sh_dist(b,c,10,[e(b,c)])`,
   `sh_dist(a,c,25,[e(a,c)])`, `sh_dist(a,c,15,[e(b,c),e(a,c)])`}
   $\cup$ { `15 < 25`, $\mathcal{P}_f$(`sh_dist(a,c,15,_)`, `15 < 25`) }.

2. $M \cup$ {`sh_dist(a,b,5,[e(a,b)])`, `sh_dist(b,c,10,[e(b,c)])`,
   `sh_dist(a,c,25,[e(a,c)])`}$\cup$ { `15 < 25`, $\mathcal{P}_f$(`sh_dist(a,c,15,_)`, `15 < 25`) }.

3. $M \cup$ {`sh_dist(a,b,5,[e(a,b)])`, `sh_dist(b,c,10,[e(b,c)])`,
   `sh_dist(a,c,15,[e(b,c),e(a,c)])`, `15 < 25` }.

The arbiter is satisfied by the following binary relation $\preceq$ on the set of worlds: $\{1 \preceq 1,\ \ 1 \preceq 3,\ \ 2 \preceq 1,\ \ 2 \preceq 3\}$.

Definition 2 *Given a preference logic program P, and a negation-free formula F, let $\mathcal{M}$ denote the intended preference model for P. F is said to be a* **preferential consequence** *of P if F is true in some strongly optimal world in $\mathcal{M}$.*

For instance, in the example above, world 3 is an optimal world. Note that only `sh_dist(a,c,15,_)` and no other `sh_dist(a,c,X,_)` for any `X` is true in an optimal world (model 3). Suppose there is a world w where `sh_dist(a,c,X,_)` is true for some `X` $>$ `15`, then the formula (`sh_dist(a,c,15,_)` $\wedge$ `15` $<$ `X`) becomes a *preference criterion* at w, making world 3 above better than w. Given a preference logic program P whose Herbrand Base is B_P and an atom A if A is a preferential consequence of the program, we write $P \models\!\!\approx A$. The **declarative semantics**, D_P, is defined to be the set $\{A \in B_P \ \mid\ P \models\!\!\approx A\}$.

The above discussion provides the declarative semantics for programs with exactly one level of *O-predicates*. We now briefly describe how the semantics is extended to the case when there are many levels of *O-predicates* in stages. Essentially, given the level-k declarative semantics for $k \geq 1$, we construct the $k+1$ declarative semantics as follows: Each world in the level-$k+1$ intended preference model is an extension of the level-k declarative semantics to the *O-* and *D-predicates* in level $k+1$. The ordering among the worlds is enforced using the arbiter clauses relevant to *O-predicates* in level $k+1$. Thus the declarative semantics of a preference logic program with n levels of *O-predicates* is the level-n declarative semantics as outlined above.

A.2 Model Theory for Relaxation Queries

Definition 3 *Given a preference logic program P and a relaxable query G =* `RELAX` *$p(\bar{t})$* `WRT` *$c(\bar{u})$, the* **relaxed intended preference model** *for P and G is a sub-frame M_r of the intended preference model M for P such that M_r contains all the worlds in M such that the only instances of $p(\bar{t})$ that appear in each world correspond to substitutions that are solutions of $c(\bar{u})$.*

Proposition 4.1 *Given a preference logic program P and a relaxable query G, the relaxed intended preference model exists and is unique up to an isomorphism.*

Definition 4 *Given a preference logic program P, an atom A that depends on a relaxable goal G is said to be a* **relaxed preferential consequence** *of P and G if it is true in some strongly optimal world in the relaxed intended preference model for P and G.*

For example, consider the shortest path program from the previous section and the relaxable query `?- RELAX sh_dist(a,b,Cost,Path) WRT notinpath(c,Path)`. The worlds in the relaxed intended model will contain only those instances of `sh_dist(a,b,Cost,Path)` such that `notinpath(c,Path)` is also true. The instance of `sh_dist(a,b,Cost,Path)` will correspond to the shortest path between `a` and `b` that does not pass through `c`.

A.3 Relaxed Clauses for Relaxation Goals

Definition 5 *Given a relaxable query* G = `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$*, the function* $relax(p(\bar{t}), c(\bar{u}))$ *returns the set of* **relaxed clauses** *for* p *by including, for every clause* $p(\bar{x}) \rightarrow p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$ *for* p*, the following pair of clauses:*

1. $relax\_p(\bar{t}) \rightarrow p_1(\bar{x}_1)\theta, \ldots, p_n(\bar{x}_n)\theta, c(\bar{u})$
2. $relax\_p(\bar{x}) \rightarrow \bar{x} \neq \bar{t} \mid p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$

where θ *is the unifier of* $\bar{x}$ *and* $\bar{t}$*. Furthermore, for every arbiter clause of* p *of the form* $p(\bar{t}_1) \preceq p(\bar{t}_2) \leftarrow L_1, \ldots, L_n$*,* $relax(p(\bar{t}), c(\bar{u}))$ *includes the following arbiter clauses:*

1. $relax\_p(\bar{t}\theta_1) \preceq relax\_p(\bar{t}\theta_2) \leftarrow L_1\sigma_1\sigma_2, \ldots, L_n\sigma_1\sigma_2, c(\bar{u}\theta_1), c(\bar{u}\theta_2)$*, where* θ_1 *and* θ_2 *are variable renaming substitutions such that* $\bar{t}\theta_1$ *and* $\bar{t}\theta_2$ *do not share any common variables. The substitutions* σ_1 *is the most general unifier of* $\bar{t}_1$ *and* $\bar{t}\theta_1$*, and* σ_2 *is the most general unifier of* $\bar{t}_2$ *and* $\bar{t}\theta_2$*.*
2. $relax\_p(\bar{t}_1) \preceq relax\_p(\bar{t}_2) \leftarrow L_1, \ldots, L_n, (\bar{t} \neq \bar{t}_1 \vee \bar{t} \neq \bar{t}_2)$*.*

The **relaxed query** *corresponding to* G *is* $relax\_p(\bar{t})$*.*

For example, given the query `?- RELAX sh_dist(a,b,Cost,Path) WRT notinpath(c,Path)`, $relax$(`sh_dist(a, b, Cost, Path), notinpath(c, Path)`) returns the following set of clauses:

1. `relax_sh_dist(a,b,Cost,Path)` $\rightarrow$ `path(a,b,Cost,Path), notinpath(c,Path).`
2. `relax_sh_dist(X,Y,Cost,Path)` $\rightarrow$ (X,Y) $\neq$(a,b) $\mid$ `path(X,Y,Cost,Path).`
3. `relax_sh_dist(a,b,C1,P1)` $\preceq$ `relax_sh_dist(a,b,C2,P2)` $\leftarrow$ `C2 < C1, notinpath(c,P1), notinpath(c,P2).`
4. `relax_sh_dist(X,Y,C1,P1)` $\preceq$ `relax_sh_dist(X,Y,C2,P2)` $\leftarrow$ `C2 < C1, (X, Y)` $\neq$ `(a, b).`

A.4 Magic Rewriting for Preference Datalog

For example consider the following PDP:

```
path(X,Y,C)          ←  edge(X,Y,C).
path(X,Y,C)          ←  edge(X,Z,C1), path(Z,Y,C2), C = C1 + C2.

naive_sh_path(X,Y,C) →  path(X,Y,C).

naive_sh_path(X,Y,C1) ⪯ naive_sh_path(X,Y,C2) ← C2 < C1.
```

and suppose the query to the program is `?- naive_sh_path(a,b,X)`. The magic rewriting described above would produce the following program:

```
path(X,Y,C)          ←  magic_path(X,Y,C), edge(X,Y,C).
path(X,Y,C)          ←  magic_path(X,Y,C), edge(X,Z,C1), path(Z,Y,C2), C = C1 + C2.
magic_path(X,Z,C1)   ←  magic_path(X,Y,C), edge(X,Z,C1).
magic_path(X,Y,C)    ←  magic_naive_sh_path(X,Y,C).
magic_naive_sh_path(a,b,X).

naive_sh_path(X,Y,C) →  magic_naive_sh_path(X,Y,C), path(X,Y,C).

naive_sh_path(X,Y,C1) ⪯ naive_sh_path(X,Y,C2) ← C2 < C1.
```