New Generation Computing, 19(2001)57-86
Ohmsha, Ltd. and Springer-Verlag

Preference Queries in Deductive Databases

Kannan GOVINDARAJAN
E-speak Operation
Hewlett-Packard Company
Cupertino, CA 95014, U.S.A.
kannang@hpl.hp.com

Bharat JAYARAMAN
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260, U.S.A.
bharat@cse.buffalo.edu

Surya MANTHA
Communications and Software Services Group
Pittiglio Todd Rabin & McGrath
1000, Potomac Street, N.W.
Washington, D.C. 20007, U.S.A.
smantha@prtm.com

Received 4 February 1998
Revised manuscript received 30 March 2000

Abstract Traditional database query languages such as datalog and SQL allow the user to specify only mandatory requirements on the data to be retrieved from a database. In many applications, it may be natural to express not only mandatory requirements but also preferences on the data to be retrieved. Lacroix and Lavency[10] extended SQL with a notion of preference and showed how the resulting query language could still be translated into the domain relational calculus. We explore the use of preference in databases in the setting of datalog. We introduce the formalism of preference datalog programs (PDPs) as preference logic programs without uninterpreted function symbols for this purpose. PDPs extend datalog not only with constructs to specify which predicate is to be optimized and the criterion for optimization but also with constructs to specify which predicate to be relaxed and the criterion to be used for relaxation. We can show that all of the soft requirements in Reference[10] can be directly encoded in PDP. We first develop a *naively-pruned bottom-up evaluation* procedure that is sound and complete for computing answers to normal and relaxation queries when the PDPs are stratified, we then show how the evaluation scheme can be extended to the case when the programs are not necessarily stratified, and finally we develop

an extension of the *magic templates* method for datalog[14] that constructs an equivalent but more efficient program for bottom-up evaluation.

Keywords: Database Query Language, Datalog, Preferences and Constraints, Relaxation Queries, Bottom-Up Evaluation.

§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[10] 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 References[5,6] 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. These criteria are stated in terms of preference for one kind of solution over another. Furthermore, when optimal solutions are impossible to obtain, and we may be interested in finding suboptimal solutions by performing some relaxation. We introduce the notion of a relaxation query for this purpose that allows the user to specify the predicate to be relaxed and the criterion to be used 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 Reference[10]. We also show that the concept of preference provides a modular and declarative (i.e., logical) means for formulating optimization as well as relaxation queries in deductive databases. It should be noted that while several approaches for optimization have been proposed in the literature, it is not clear whether any of them can account for our formulation of relaxation queries.

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[5,6].) We first develop a *naively-pruned bottom-up evaluation* procedure that is sound and complete for stratified PDP, we show how the evaluation scheme can be extended to the case when the programs are not necessarily stratified, and finally we develop an extension of the *magic templates* method for datalog[14] that constructs an equivalent but more efficient program for bottom-up evaluation.

In earlier work, we have given a model-theoretic semantics for PLP using simple concepts from modal logic.[5,6] 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. The use of modal logic for this purpose is not surprising because optimization and relaxations are meta-level non-monotonic notions; an area where modal logics have found use in the past.[1,12] Furthermore, since we are computing preferential consequences as opposed to logical consequences, we do not incur the cost of theorem proving in a general modal logic.

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 considerable interest recent years in providing a satisfactory semantics for aggregate operations.[2,9,15,19,21] Ganguly *et al.*[2] considered first-order aggregates and showed that under certain monotonicity conditions, the first-order equivalent program has a total well-founded model[22] that can be computed using a greedy fixed-point procedure. Kemp and Stuckey[9] examined programs with recursion through aggregation. To give semantics for programs with aggregation, they extended two well-known semantics for programs with negation, namely, well-founded models and stable models.[3] Ross and Sagiv[15] provide semantics for aggregation where the domain over which the aggregation is performed is a complete lattice and the program is monotonic. By Tarski's theorem, we are guaranteed the existence of a least fixed-point for the aggregate operation. Sudarshan *et al.*[18,19] provide semantics for a class of aggregate operators using *valid models* for normal programs. Compared with these methods, 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. Furthermore, our model theory also has the desirable property of associating a unique intended preference model with every preference logic program—a feature that is not necessarily guaranteed by the negation-based approaches.

In a different setting, the idea of hard and soft requirements has also been explored in HCLP (hierarchic constraint logic programming)[23] 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 Reference[6] 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.

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. Section 5 describes a magic rewriting technique that generates a more efficiently executable program. Finally, section 6 provides conclusions and directions for further research.

§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 to 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*. Variables appearing only on the RHS of the $\rightarrow$ clause are existentially quantified. $G_1, \ldots, G_l$ 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 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*. We disallow mutual recursion between an *O-predicate* and a *D-predicate*. (*D* stands for *derived from O-predicates.*) In other words, if a predicate is defined using definite clauses and does not depend on an *O-predicate*, it is classified as a *C-predicate*.

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. Note that the *C-predicates* are different from the *D-predicates* because the *D-predicates* are eventually defined in terms of *O-predicates*. This, however, is not the case with *C-predicates* because they are defined in terms of other *C-predicates* only. One can think of the *C-predicates* as specifying the arbitrary constraints that are to be satisfied by any potential solution. The *D-predicates* on the other hand, are defined in terms of *O-predicates*.

Given any preference datalog program, we can order 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 construct the predicate call graph amongst the *O-predicates* and topologically sort it assigning equal ordinals to all the *O-predicates* in a cycle. Note that this construction is possible for an arbitrary preference datalog program. The ordering $\geq$ has the property that the equality holds between the levels of two *O-predicates* O_1 and O_2 if and only if they are defined in terms of each other. 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*. If a *D-predicate* is defined in terms of multiple *O-predicates*, the level of the *D-predicate* is the level of the highest *O-predicate* used in its definition.

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. The right hand side of a such a preference clause provides the justification for preferring one solution over another. By allowing only *C-predicates* on the right hand sides of arbiter clauses, we make justification for the preferences uniform across the worlds in the possible worlds semantics. Since the programmer has complete flexibility in defining the *C-predicates*, this requirement is not restrictive.

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

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

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'). 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.

The use of the term 'relaxation' may at first seem a bit counter-intuitive since the effect of a goal `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$ is to treat $c(\bar{u})$ as an *additional constraint* on the feasible solutions to $p(\bar{t})$. The relaxation is in the sense that the optimality of p will have to be relaxed in order to satisfy c.

Finally we note that 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

We now briefly illustrate a few query paradigms within the PDP framework. We first show how preferences in the relational calculus can be captured within the PDP framework and we also consider relaxation queries.

3.1 Preferences in Relational Calculus

We now show that the kinds of queries expressible in the framework of Lacroix and Lavency[10] 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 $\{\bar{r} \in R | P(\bar{r})\}$. Essentially, this query selects out the tuples from the relation `R` that satisfy the property `P`. We can translate queries in the domain relational calculus into datalog in the most natural way. Each relation in the database is represented by an EDB predicate and the properties are either constraints or are user-defined predicates. A query with a simple preference clause has the form

```
select X where Q prefer P1.
```

Operationally, we first try to satisfy `Q` $\wedge$ `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 no solution to Q satisfies P1, the arbiter is 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 is captured in Reference[10] 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:

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

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 Reference[10] as follows:

```
select X where Q prefer P1, 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 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 solutions. 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.

$$\texttt{X1}\ (\bar{t}) \rightarrow \texttt{X}\ (\bar{t}),\ \texttt{Q}\ (\bar{t}).$$
$$\texttt{X1}\ (\bar{t}_1) \preceq \texttt{X1}\ (\bar{t}_2) \leftarrow \texttt{P1}\ (\bar{t}_2),\ \texttt{P2}\ (\bar{t}_2), \neg\ (\texttt{P1}\ (\bar{t}_1) \wedge \texttt{P2}\ (\bar{t}_1)).$$
$$\texttt{X1}\ (\bar{t}_1) \preceq \texttt{X1}\ (\bar{t}_2) \leftarrow (\texttt{P1}\ (\bar{t}_2) \vee \texttt{P2}\ (\bar{t}_2)), \neg\ (\texttt{P1}\ (\bar{t}_1) \vee \texttt{P2}\ (\bar{t}_1)).$$

Extreme-Value Preferences: Preference clauses with a maximum or a minimum preference are formulated in the framework of Reference[10] 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:

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

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:

$$\texttt{X1}\ (\bar{t},u) \rightarrow \texttt{X}\ (\bar{t}),\ \texttt{Q}\ (\bar{t}),\ \texttt{P}\ (\bar{t},u).$$
$$\texttt{X1}\ (\bar{t}_1,u_1) \preceq \texttt{X1}\ (\bar{t}_2,u_2) \leftarrow u_1 < u_2.$$

Thus PDP can express all of the preference clauses introduced in Reference[10]. Note that we have only made use of optimization clauses in the translated programs, i.e., relaxation queries are not needed in these translations.

3.2 Relaxation Queries

Consider the `person(Name, Sex, DOB)` and `ancestor(Child,Ancestor)` predicates from the well-known family database example:

```
father(Child,Father).
mother(Child,Mother).
parent(Child,Parent).
parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
ancestor(X,Y) ← parent(X,Y).
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y).
```

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 re-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 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.

§4 Bottom-up Evaluation for Preference Datalog

In earlier work,[5,6] we introduced a top-down query evaluation mechanism for preference logic programs. In many datalog programs a bottom-up evaluation mechanism is more efficient as it can avoid unnecessary re-computation through memoization. Furthermore, since bottom-up evaluation is better than top-down for datalog,[20] we are interested in studying bottom-up evaluation techniques for preference datalog. We first consider bottom-up evaluation for stratified and locally stratified PDP. Finally, we present a magic rewriting technique for PDP so that the bottom-up evaluation of the rewritten program does not make inferences that are not relevant to the query.

4.1 Stratified Programs

Traditionally, stratification is defined with every respect to all the predicates in the program. In PLP, however, we are interested in stratification as it applies to the *O-predicates* only.

Definition 4.1

A preference logic program P is said to be *O-stratified* if the following holds: There is a mapping f from the set of *O-predicates* to the set $\{1, \ldots, n\}$, for some least n, such that if an instance of an *O-predicate* P_1 appears in the body of a $\rightarrow$ clause defining an *O-predicate* P_2, then $f(P_1) < f(P_2)$. For any *O-predicate* P, its *rank* is defined to be $f(P)$.

We first consider O-stratified preference datalog programs. 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. This may be computed incrementally using the semi-naive iteration.
2. Consider the definitions of the *O-predicates* at level 1. Treating the $\rightarrow$ clauses just as $\leftarrow$ clauses (the conditions before the guard in the $\rightarrow$ clause are treated as ordinary goals in the $\leftarrow$ clause) 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. We refer the reader to the Appendix for a brief description of the model theory. The following lemma states the correctness of interpreting $\rightarrow$ clauses as $\leftarrow$ clauses.

Lemma 4.1
Suppose $\mathcal{T}$ is a O-stratified collection of clauses containing a clause $\mathcal{C}$ of the form:

$$p(\bar{t}) \leftarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m), p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$$

Suppose the set S is a supported model for $\mathcal{T}$. Now, consider the theory $\mathcal{T}' = (\mathcal{T} \setminus \{\mathcal{C}\}) \cup \mathcal{C}'$ where $\mathcal{C}'$ is a clause of the form:

$$p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) \mid p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$$

where all the variables that occur only on the right hand side are existentially quantified and the c_i's are treated as antecedents of the implication. Any supported model S' of $\mathcal{T}'$ is such that $S' \subseteq S$. Indeed, S itself is a supported model for $\mathcal{T}'$.

Proof
Consider the clause of the form:

$$p(\bar{t}) \rightarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m) \mid p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$$

Note that the variables that appear only on the right hand side of the $\rightarrow$ clause are existentially quantified. A set of atoms S will model such a clause if it has the following property. Suppose a ground instance $p(\bar{t})\theta$ of the head of the clause is in S, that is θ satisfies $c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m)$, then there must be some assignment σ for the variables that occur only on the right hand side such that $\{p_1(\bar{t}_1)\theta\sigma, \ldots, p_n(\bar{t}_n)\theta\sigma\} \subseteq S$.
Now consider a clause of the form:

$$p(\bar{t}) \leftarrow c_1(\bar{u}_1), \ldots, c_m(\bar{u}_m), p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$$

Note that for such a clause, every variable that occurs in the clause is universally quantified. Suppose a set of atoms S_1 is a supported model for this clause. This means that for every substitution η such that $\{c_1(\bar{u}_1)\eta, \ldots, c_m(\bar{u}_m)\eta, p_1(\bar{t}_1)\eta, \ldots, p_n(\bar{t}_n)\eta\} \subseteq S_1$, we have that $p(\bar{t})\eta \in S_1$. Clearly each set S that models the $\rightarrow$ clause is a subset of the set S_1 that models the corresponding $\leftarrow$ clause. This is because we can decide not to include a particular instance of the head of the $\leftarrow$ clause into the set S and still model the $\rightarrow$ clause. The set S_1 itself is a model for the $\rightarrow$ clause as long as the variables that appear only on the right hand side of the $\rightarrow$ clause are existentially quantified. Therefore, by treating the $\rightarrow$ clause as a $\leftarrow$ clause, we can obtain all the possible substitutions for the goal with an *O-predicate* p at its head that can model the clause for the p. ■

The consequence of the above lemma is that solutions to any *O-predicate* goal G can be computed by treating $\rightarrow$ clauses as $\leftarrow$ clauses. In what follows, a *world* is a collection of instances of the predicates in the program that is a model for the clauses of the program.

Theorem 4.1 (Soundness and Completeness)
Given a O-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
The proof is by induction on the level of *O-predicates* in the program. Clearly, we are interested in the case when the head of the atom is an *O-predicate*, since the correctness of the other kinds of atoms will follow from that.
Base case: Consider an atom $A = p(\bar{t})$ such that the level of the *O-predicate* p is 1. Since the program is O-stratified, each model for the $\rightarrow$ clause is a subset of the corresponding $\leftarrow$ clause. From this it follows that any instance of p that appears in any world is a subset of the set O_1^{up}. Furthermore, if the arbiter applies between two instances of p then the corresponding worlds in the intended preference model are related. Since the program is O-stratified, the strongly optimal worlds are not related to themselves (for proof see Reference[4]). Therefore, the set of instances of p that remain after the pruning step are precisely the set of preferential consequences. Since the *D-predicates* are defined in terms of *O-predicates* via $\leftarrow$ clauses, if we start with the pruned set of *O-predicates* and compute bottom-up the least fixed point of the clauses defining the *D-predicates* we would get the declarative semantics of the program at level 1.

Inductive Hypothesis: Given a preference logic program, the set M_k is the declarative semantics of the program at level k.
Inductive Step: Consider the definitions of the *O-predicates* at level $k+1$. By our inductive hypothesis, the set M_k is the declarative semantics of the program up to level k. Recall that the worlds in the intended preference model at level $k+1$ were constructed by extending the set M_k with instances of *O-predicates* and *D-predicates* so that each world models the clauses for the predicates at level $k+1$. We can use the lemma from the previous chapter to conclude that each instance of an *O-predicate* that occurs in any world is present in the set O^{up}_{k+1}. Furthermore, after pruning, the set of instances *O-predicates* are precisely those that are true in some strongly optimal world. Extending the set O_{k+1} to the set M_{k+1} (using the definitions of the D-predicates at level $k+1$), gives us the declarative semantics for the program at level $k+1$ as bottom-up evaluation is complete for $\leftarrow$ clauses defining the *D-predicates*. ■

If the program is not O-stratified, a naively-pruned bottom-up evaluation is not complete. For instance, consider the following non-stratified program:

```
q(X) ← p(X).
p(b) → p(a), r(a).
p(a) → p(b), r(b).
p(a) ≺ p(b).
p(b) ≺ p(a).
r(a).
r(b).
```

The query `q(X)` has two correct optimal answers: `X = a` and `X = b` since `q(a)`, and `q(b)` are both preferential consequences of the program (see appendix for definition). The reader may note that standard model theoretic techniques such as well-founded models would not classify these as correct answers. The possible worlds semantics that we associate with preference programs that defines these to be the correct answers. Since the least model of the set of clauses obtained by treating the $\rightarrow$ clauses as $\leftarrow$ clauses is $\{\texttt{r(a)}, \texttt{r(b)}\}$, the naively-pruned bottom-up evaluation will not compute either answer for this query. This example also illustrates why top-down evaluation may be more suited for preference logic programs than bottom-up evaluation. Since bottom-up evaluation works with ground instances, it is more difficult to guarantee soundness and completeness. In top-down evaluation, the requirement that the invoked goals be sufficiently non-ground was crucial in guaranteeing soundness. In bottom-up evaluation, however, that requirement is difficult to enforce. If the program is locally O-stratified, bottom-up evaluation succeeds as lemma in showed that the various models for the $\rightarrow$ clause are subsets of the canonical model for the corresponding $\leftarrow$ clause. Non-stratification, however, had a profound effect on the top-down operational semantics. The search tree emanating from the goal was infinite which is not the case for bottom-up evaluation.

4.2 Locally Stratified Programs

We now consider programs that are not necessarily O-stratified and show

how the bottom-up evaluation can be extended to those programs.

Definition 4.2
A preference logic program P is said to be *locally O-stratified* if the following two conditions hold:

1. There is a mapping f from the set of *O-predicates* to $\{1, \ldots, n\}$ for the least n such that if an instance of an *O-predicate* P_1 appears in the body of a $\rightarrow$ clause defining an *O-predicate* P_2, then $f(P_1) \leq f(P_2)$. In addition, for any *O-predicate* P, $f(P)$ is the *rank* of P.
2. There is a well-founded ordering $\prec_k$ over the set of ground instances of all *O-predicates* of *rank* k, defined as follows: (i) ground instances of base facts of *O-predicates* of rank k all map to the $\perp$ of the ordering $\prec_k$; and (ii) ground instances of optimization clauses have the property that each instance of an *O-predicate* of rank k that appears in the body is $\prec_k$ the instance of the *O-predicate* of rank k that appears in the head.
   Furthermore, the ordering $\prec_k$ is defined by considering only those argument positions of a ground instance that are not used by the arbiter clauses to prefer one solution over another.

When the program was O-stratified, we were able to perform a bottom-up evaluation of the program level by level. However, when the program is not O-stratified, we cannot do the same. The technique works for programs that are O-stratified because by the time we are evaluating the program at level k, we accurately know the preferential consequences of predicates upto level $k-1$. This however is not the case when the program is not O-stratified.

Consider the following dynamic-programming formulation for the minimum-distance problem as follows:

```
min_dist(X,X,0,0).
min_dist(X,Y,1,C)          →   X <> Y | edge(X,Y,C).
min_dist(X,Y,N+1,C1+C2)    →   N > 0, X <> Y |
                               min_dist(X,Z,1,C1),
                               min_dist(Z,Y,N,C2).
min_dist(X,Y,N,C1) ≼ min_dist(X,Y,N,C2) ← C2 < C1.
```

The *optimal subproblem property* of the minimum-distance problem is expressed well by the above formulation: each call to `min_dist` uses only the *optimal* solutions to subsequent recursive calls on `min_dist`. Note that this program has function symbols such as $+$. However, for the purposes of the discussion here, we will treat such function symbols as being pre-interpreted and the resulting program is a constraint datalog program.[8]

The technique outlined in the previous subsection will work for such locally O-stratified programs, since we can prove a lemma similar to lemma 4.1 and thereby treat $\rightarrow$ clauses as $\leftarrow$ clauses for the purpose of generating solutions. However, it is possible that non-optimal versions of some O-predicates may be used to generate facts that will then get pruned later. Therefore, the level by level bottom-up evaluation will generate unnecessary facts. We need to

make sure that when using a recursive rule for an *O-predicate* p, we only consider instances of the p in the body that are known to be optimal.

A similar problem arises in bottom-up evaluation of programs with negation. There, when using a recursive rule for a predicate p_1 that depends negatively on itself, we need to make sure that we have enough information to actually infer the negative instance in the body. Ross[16] discusses a technique for bottom-up evaluation of such programs that addresses this problem. We now show how a similar technique can be used for bottom-up evaluation of preference datalog programs.

In keeping with the standard terminology, we refer to the predicates defined using base facts Extensional Database predicates (EDB predicates) and those defined using rules Intensional Database predicates (IDB predicates). Following Ross,[16] we introduce the following auxiliary predicates:

1. A *depends* predicate (abbreviated as *depo*) that keeps a record of which atoms depend on optimal instances of other atoms.
2. A unary *un-depends* predicate (abbreviated as *depo'*) that keeps a record of atoms whose optimality has been decided.
3. A currently depends predicate (abbreviated as *dd*). This relation is precisely *depo* − *depo'*.
4. An extra modal operator $\circ$. Intuitively, $\circ p(\bar{t})$ holds if at the previous fixed point, $p(\bar{t})$ is known to be optimally true. That is, no other instance of p is preferred over $p(\bar{t})$ at the previous fixed point.

The relations *depo*, *depo'*, and *dd* are maintained by the bottom-up evaluation procedure. We only provide intuitive definitions for them. However, they could be implemented a much more efficiently than described here. The key intuition behind using the modal operator $\circ$ is that we want to postpone the firing of some rules until optimal instances of the goals on right hand side have been computed. Its effect on the bottom-up computation is that the bottom-up evaluation reaches many fixed points. At each fixed point, some new optimal instances may be inferred and that enables some more new rules to fire. This goes on till at some fixed point no more new optimal instances are inferred.

Furthermore, we also use meta variables that can unify with arbitrary atoms. Given a rule r, the *opt-replacement* of r is the rule obtained by replacing each O-predicate goal $q(\bar{t})$ in the body of r with $\circ q(\bar{t})$.

Definition 4.3
Given a preference datalog program P, we define the *opt-rewritten* version P^{opt-R} of P as follows:

1. For every rule r, the opt-replacement of r is in P^{opt-R}.
2. For every rule of the form $p(\bar{t}) \rightarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ (or $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$), and for i from 1 to n, if $p_i(\bar{t}_i)$ is an O-predicate goal, we add the opt-replacement of the rule:
   $depo(p(\bar{t}), p_i(\bar{t}_i)) \leftarrow p_1(\bar{t}_1), \ldots, p_{i-1}(\bar{t}_{i-1})$
3. For every rule of the form $p(\bar{t}) \rightarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ (or $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$), and for i from 1 to n, if $p_i(\bar{t}_i)$ is not an O-predicate goal or an EDB

predicate, we add the opt replacement of the rule:
$depo(p(\bar{t}), X) \leftarrow p_1(\bar{t}_1), \ldots, p_{i-1}(\bar{t}_{i-1}), dd(p_i(\bar{t}_i), X)$

4. We also add the following rules to P^{opt-R}

$$depo'(Q) \leftarrow dd(Q, R), R.$$
$$depo'(Q) \leftarrow dd(Q, R), \circ R.$$

5. There is a meta-rule in P^{opt-R}.

$$\circ P \leftarrow \forall Q(depo(P, Q) \Rightarrow depo'(Q)), optimal(P).$$

The meta-rule is defined in terms of an embedded implication $\Rightarrow$. Basically, it says that a fact of the form $\circ P$ can be inferred if for each fact Q such that $depo(P, Q)$ is true at the current fixed point, it is the case that $depo'(Q)$ is true. Furthermore, P is optimal in that there is no other instance of the predicate at the head of P that is true at the current fixed point that is preferred according to the arbiter. The meta-rule succintly captures the computation that has to be performed by the bottom-up evaluation procedure at each fixed point.

Now, consider the program for `min-dist` from before. The opt-rewritten version of that program looks as follows:

```
min_dist(X,X,0,0).
min_dist(X,Y,1,C) → X <> Y | edge(X,Y,C).
min_dist(X,Y,N+1,C1+C2) → N > 1, X <> Y |
                          ∘ min_dist(X,Z,1,C1),
                          ∘ min_dist(Z,Y,N,C2).
min_dist(X,Y,N,C1) ≼ min_dist(X,Y,N,C2) ← C2 < C1.
depo(min_dist(X,Y,N+1,C1+C2),min_dist(X,Y,1,C1)) ←
                    N > 0, X <> Y.
depo(min_dist(X,Y,N+1,C1+C2),min_dist(Z,Y,N,C2)) ←
      N > 0, X <> Y, ∘ min_dist(X,Z,1,C1).
```

The *naively-pruned* bottom-up evaluation now proceeds along using program P^{opt-R}.

1. First compute the least fixed point of the clauses defining the core predicates in the program. Let this set be M_0.
2. Now consider the clauses that define the O-predicates and D-predicates at level 1. Apply the clauses at level 1 till you reach a fixed point. At this point, we can infer some instances of $\circ p(\bar{t})$ for some O-predicates. Note that the predicates *depo*, *depo'*, and *dd* will get modified as this computation proceeds. Use the optimal ($\circ$) instances currently computed to restart the bottom-up evaluation of the level 1 of the program. We continue until a particular pass of the bottom-up evaluation does not produce any more new facts. The set produced at this point is the set M_1, the set of preferential consequences at level 1.
3. Repeat step 2 for each level of the program resulting in the set M_n where n is the number of levels in the program.

In the example above, the first step will result in the production of a set

that contains all the instances of the edge predicate in the program. When the rules making up the predicate min_dist are considered, the first fixed point is reached by using only the base fact and the rule for min_dist that depends on edge. This is because the other rules have ○ goals in their bodies. Now, we will be able to infer facts of form ○ min_dist(X,Y,1,C) for various X and Y. This is because min_dist(X,Y,1,C) instances depend only on the EDB predicate edge. The meta rule will be applicable because the embedded implication is vacuously true. The only instances that survive will be the ones that actually correspond to the shortest edges between any pair of nodes. When the next fixed point is reached, we will be able to infer facts of the form ○ min_dist(X,Y,2,C) and so on. This essentially mimics the dynamic programming algorithm for shortest path.

Note that we have used auxiliary predicates to indicate how the bottom-up evaluation should proceed. We can come up with efficient implementations for maintaining the relations *depo*, *depo'*, and *dd* so that the generation of the ○ facts using the meta rule is reasonably efficient.

Lemma 4.2
Given a preference datalog program P with n levels, suppose for some $0 < i < n$, the bottom up evaluation procedure outlined above produces the set M_i then, when executing the procedure for level $i+1$ of the program, if at any fixed point, an instance of the form $\circ p(\bar{t})$ is inferred using the meta rule, then the atom $p(\bar{t})$ is a member of the set M_{i+1}.

Proof
Essentially, all we want to show is that once an atom has been deemed to be optimal, its optimality is not rescinded. This follows from the observation that the ordering that determines local stratification is defined using the argument positions that are the same in both instances of the O-predicate in the head of an arbiter clause and not the argument positions that are used by the body of the arbiter to choose one instance over the other. Therefore, if at a fixed point, the optimality of $p(\bar{t})$ depends on the optimality of $p(\bar{u})$, because a $\rightarrow$ clause is applicable with $p(\bar{t})$ at the head and $p(\bar{u})$ in the body, then no arbiter clause is applicable between any instance of $p(\bar{t})$ and $p(\bar{u})$. ■

Theorem 4.2
Given a preference datalog program that is locally O-stratified with n levels, an atom A belongs to the set M_n constructed by the bottom-up procedure outlined above if and only if A is a preferential consequence of the program.

Proof
The proof is by induction over the levels of the program. For level 1, suppose an atom A belongs to the set M_1 that is constructed at the end of iteration at level 1. Suppose further that A is an instance of an O-predicate at level 1. If A belongs to the set M_1, it must have been introduced during some iteration. Suppose that it was the j^{th} iteration and the set at the $(j-1)^{th}$ fixed point the set of atoms deemed to be true was $M_{1.(j-1)}$ and after the j^{th} iteration, the set was $M_{1.j}$. We want to show that A is a preferential consequence of the program.

This however, follows from the observation that since the atom A survived the pruning that created the set M_{1_j}. Furthermore, it was created by using the optimal instances of all the O-predicates that it depended on. Therefore, the instance of A is optimal. Furthermore, by the previous lemma, we also know that once A has been deemed to be optimal, it indeed is optimal.

A similar proof holds for each level in the program. ■

4.3 Relaxation Queries

We now describe a bottom-up procedure to compute the answers to relaxation queries. To provide the semantics for RELAX goals, we first note that, for truth in the *optimal* world, both constraints and preferences must be satisfied. If we consider only the worlds that contain instances of both the relaxable predicate and the relaxation criterion to determine the best solution, we effectively relax the preferences that made the worlds without instances of the relaxation criterion better.

Definition 4.4
Given a preference logic program P and a relaxable query $G = \texttt{RELAX}\ p(\bar{t})\ \texttt{WRT}\ c(\bar{u})$, 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 $c(\bar{u})$.

Definition 4.5
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.

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.

Definition 4.6
Given a relaxable query $G = \texttt{RELAX}\ p(\bar{t})\ \texttt{WRT}\ c(\bar{u})$, 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. $relax\_p(\bar{t}, \bar{u}) \rightarrow \bar{x} = \bar{t} \mid p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n), c(\bar{u})$.
2. $relax\_p(\bar{x}, \bar{v}) \rightarrow \langle \bar{x}, \bar{v} \rangle \neq \langle \bar{t}, \bar{u} \rangle \mid p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$

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. $relax\_p(\bar{t}\theta_1, \bar{u}\theta_1) \preceq relax\_p(\bar{t}\theta_2, \bar{u}\theta_2) \leftarrow L_1\sigma_1\sigma_2, \ldots, L_n\sigma_1\sigma_2$, $c(\bar{u}\theta_1), c(\bar{u}\theta_2)$, 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. $relax\_p(\bar{t}_1, \bar{v}_1) \preceq relax\_p(\bar{t}_2, \bar{v}_2) \leftarrow L_1, \ldots, L_n, (\bar{t} \neq \bar{t}_1 \vee \bar{t} \neq \bar{t}_2), (\bar{u} \neq \bar{v}_1 \vee$

$\bar{u} \neq \bar{v}_2$).

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

In essence, 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 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 appendix B for an example. Note that we have made use of ordered pairs in the translation for the sake of convenience. An equivalent translation without ordered pairs is easily devised. Furthermore, the *relaxed query* corresponding to the relaxation query RELAX $p(\bar{t})$ WRT $c(\bar{u})$ is $relax\_p(\bar{t}, \bar{u})$. The soundness of the translation is established by the following theorem.

Theorem 4.3 (Correctness of Translation)
Given a preference datalog program P and a relaxation 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 $relax\_p(\bar{t}, \bar{u})\theta$ is a preferential consequence of the program $P \cup relax(p(\bar{t}), c(\bar{u}))$.

Proof
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 $p(\bar{t}, \bar{u})\sigma$ occurs in any world in the relaxed intended preference model of P and G, then the instance $relax\_p(\bar{t}, \bar{u})\sigma$ 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 O-stratified preference datalog program P and a relaxation query G consists of the following steps: (i) 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. (ii) Perform naively-pruned bottom-up evaluation of the program P'.

Theorem 4.4 (Soundness and Completeness)
Given a (locally) O-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 $p(\bar{t}, \bar{u})\theta$ is a relaxed preferential consequence of P and G.

Proof
If P is a (locally) O-stratified preference logic program and G = RELAX $p(\bar{t})$ WRT $c(\bar{u})$ is a relaxable goal then $P' = P \cup relax(p(\bar{t}), c(\bar{u}))$ is also a O-stratified program. Furthermore, from the correctness of translation theorem we know that the set of correct relaxed answers to the goal $relax\_p(\bar{t})$ with respect to the program P' are the answers of interest. Since P' is also (locally) O-stratified, naively pruned bottom-up evaluation is sound and complete and we have our desired result. ■

4.4 Magic Rewriting

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. We now describe an improvement based on the magic rewriting presented in Reference[13]. 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 $Q = q(\bar{s})$. Traditionally in magic rewriting, for each IDB 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 $p(\bar{u})$. However, in our setting, we will find it more convenient to introduce the magic predicates as meta predicates. In particular, we have two versions of the magic predicates. One for the O-predicates, $magic_o$, and one for the C-predicates and D-predicates in the program, *magic*. Furthermore, we will use the supplementary magic rewriting along the lines of the magic rewriting for datalog programs with negation presented in Reference[16]. Therefore, for each rule r_j in the program that has k goals in the body, we introduce $k+1$ supplementary predicates $sup_{j.i}$ for $i = 0, \ldots, k$. Essentially, $sup_{j.i}$ transmits the relevant variable bindings from the $(i-1)^{th}$ goal through the n^{th} goal to the head. Each predicate $sup_{j.i}$ can have as arguments all the variables that occur in the clause. However, this can be optimized by following the scheme in Reference[16] by retaining variables that either appear in the rule head or variables that occur in the $(i-1)^{th}$ subgoal that also occur the rest of the body. We assume that a left-to-right sideways information passing (sip)[13] is used.

In addition to the magic predicates, we have the following meta predicates:

1. $dep(P, Q)$, which means that P depends on Q in the normal sense.
2. $depo(P, Q)$, which means that P depends on an optimal instance of Q.
3. $depo'(Q)$ which is the complementary version of *depo* which means that the instances of *depo* are not useful anymore as the optimal status of Q is known.
4. We also use the $\circ$ operator as before.

The meta predicate $magic_o$ is special. An instance of the form $magic_o(p(\bar{t}))$ is a shortened notation for an ordered pair of the form $\langle magic_o(p(\bar{t}')), \bar{t} = \bar{t}' \rangle$. The term $\bar{t}'$ is the term $\bar{t}$ where all the positions in $\bar{t}$ that are used by the arbiter for p

are replaced by new variables. The constraint $\bar{t} = \bar{t}'$ is then enforced before the optimality of $p(\bar{t})$ is used to infer other facts. For instance, before facts of the form $depo'(p(\bar{t}))$ are inferred. This captures the non-groundness condition that the top-down semantics described in References[5,6] placed on the O-predicate goals that were encountered in the top-down computation. This is necessary because we want to be able to guarantee that all the solutions for the argument positions used by the arbiter are produced before pruning is performed.

Let P^m be the program that is the result of performing the magic template transformation on the program-query pair $\langle P, Q \rangle$. P^m has three components $\langle P_0^m, P_1^m, P_2^m \rangle$.

We now describe what is included in each component in turn.

The following is in P_0^m:

(0) If the initial query is ?- $p(\bar{t})$ where p is not an O-predicate, then include the rule:

$$magic(p(\bar{t})).$$

(0a) If the initial query is ?- $p(\bar{t})$ where p is an O-predicate, where the arguments positions that are used by the arbiter to choose amongst alternative solutions of p are unbound in $p(\bar{t})$, then include the rule:

$$magic_o(p(\bar{t})).$$

The following are included in P_1^m:

(1) If the head of rule r_j is $p(\bar{t})$, and p is not an O-predicate, we include the rule:

$$sup_{j.0}(\bar{u}) \leftarrow magic(p(\bar{t})).$$

(1a) If the head of rule r_j is $p(\bar{t})$, and p is an O-predicate, we include the rule:

$$sup_{j.0}(\bar{u}) \leftarrow magic_o(p(\bar{t})).$$

(2) If the head of rule r_j is $p(\bar{t})$, r_j has k goals in the body, and p is not an O-predicate, then include the rule:

$$p(\bar{t}) \leftarrow sup_{j.k}(\bar{u}).$$

(2a) If the head of rule r_j is $p(\bar{t})$, r_j has k goals in the body, and p is an O-predicate, then include the rule:

$$p(\bar{t}) \rightarrow sup_{j.k}(\bar{u})$$

(3) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j. If p is not an O-predicate and the arguments of $sup_{j.(i-1)}$ are $\bar{u}$, then include the rule:

$$magic(p(\bar{t})) \leftarrow sup_{j.(i-1)}(\bar{u}).$$

(3a) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j, and p is an O-predicate, then include the rule:

$$magic_o(p(\bar{t})) \leftarrow sup_{j.(i-1)}(\bar{u})$$

(4) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and r_j has k goals in the body. If p is not an O-predicate and $i \leq k$, include the rule:

$$sup_{j.i}(\bar{u}) \leftarrow sup_{j.(i-1)}(\bar{v}), p(\bar{t}).$$

(4a) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and r_j has k goals in the body. If p is an O-predicate and $i \leq k$, include the rule:

$$sup_{j.i}(\bar{u}) \leftarrow sup_{j.(i-1)}(\bar{v}), \circ p(\bar{t}).$$

(5) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and $q(\bar{s})$ is the head of r_j. If p is not an O-predicate and q is not an O-predicate, include the rules:

$$dep(q(\bar{s}), p(\bar{t})) \leftarrow magic(q(\bar{s})), sup_{j,(i-1)}(\bar{u}).$$
$$dep(P, p(\bar{t})) \leftarrow dep(P, q(\bar{s})), sup_{j.(i-1)}(\bar{u}).$$

(5a) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and $q(\bar{s})$ is the head of r_j. If p is an O-predicate and q is not an O-predicate, include the rules:

$$depo(q(\bar{s}), p(\bar{t})) \leftarrow magic(q(\bar{s})), sup_{j,(i-1)}(\bar{u}).$$
$$depo(P, p(\bar{t})) \leftarrow dep(P, q(\bar{s})), sup_{j.(i-1)}(\bar{u}).$$

(5b) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and $q(\bar{s})$ is the head of r_j. If p is an O-predicate and q is an O-predicate, include the rules:

$$depo(q(\bar{s}), p(\bar{t})) \leftarrow magic_o(q(\bar{s})), sup_{j,(i-1)}(\bar{u}).$$
$$depo(P, p(\bar{t})) \leftarrow depo(P, q(\bar{s})), sup_{j.(i-1)}(\bar{u}).$$

(5c) If $p(\bar{t})$ is the i^{th} goal in the body of rule r_j and $q(\bar{s})$ is the head of r_j. If p is not an O-predicate and q is an O-predicate, include the rules:

$$dep(q(\bar{s}), p(\bar{t})) \leftarrow magic_o(q(\bar{s})), sup_{j,(i-1)}(\bar{u}).$$
$$dep(P, p(\bar{t})) \leftarrow depo(P, q(\bar{s})), sup_{j.(i-1)}(\bar{u}).$$

(5d) Include the rule:

$$depo'(Q) \leftarrow magic_o(Q), \circ Q.$$

The following meta-rule is introduced in P_2^m:

$$\circ P \leftarrow magic_o(P), \forall Q(depo(P, Q) \Rightarrow depo'(Q)), optimal(P).$$

The meta rule in P_2^m essentially states that we infer a fact of the form $\circ P$ only when the optimal truth of all the facts that it depends on has been determined. This meta-rule is fired only at fixed points in the bottom-up computation using the rules in P_1^m. Therefore, the bottom-up computation algorithm proceeds in the following loop:

1. Assert the facts in P_0^m.
2. Using the facts in P_1^m, the bottom-up computation proceeds until a fixed point is reached.
3. At the fixed point, new instances of $\circ$ and $depo'$ are inferred using the meta rule P_2^m and the rule for $depo'$ until a fixed point is reached. If new

o instances were inferred in this iteration, go to step 2, else bottom-up computation is complete.

Example 5.1 Consider the following preference datalog program:

```
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: The program component P_0^m contains:
$magic_o$`(naive_sh_path(a,b,X)).`

The component P_1^m contains the following rules:

```
sup1.0(X,Y,C)            ←  magic(path(X,Y,C)).
sup2.0(X,Y,C)            ←  magic(path(X,Y,C)).
sup3.0(X,Y,C)            ←  magic_o(naive_sh_path(X,Y,C)).

path(X,Y,C)              ←  sup1.1(X,Y,C).
path(X,Y,C)              ←  sup2.3(X,Y,C,).
naive_sh_path(X,Y,C)     →  sup3.1(X,Y,C).

magic(path(X,Y,C2))      ←  sup2.1(X,Y,C,Z,C1,C2).
magic(path(X,Y,C))       ←  sup3.0(X,Y,C).

sup1.1(X,Y,C)            ←  sup1.0(X,Y,C), edge(X,Y,C).
sup2.1(X,Y,C,Z,C1)       ←  sup2.0(X,Y,C), edge(X,Z,C1).
sup2.2(X,Y,C,C1,C2)      ←  sup2.1(X,Y,C,Z,C1), path(Z,Y,C2).
sup2.3(X,Y,C)            ←  sup2.2(X,Y,C,C1,C2), C = C1 + C2.
sup3.1(X,Y,C)            ←  sup3.0(X,Y,C), path(X,Y,C).

dep(X, path(Z,Y,C2))     ←  dep(X, path(X,Y,C)),
                            sup2.1(X,Y,C,Z,C1).
dep(X, path(X,Y,C))      ←  depo(X, naive_sh_path(X,Y,C)),
                            sup3.0(X,Y,C).
dep(path(X,Y,C), path(Z,Y,C2)) ← magic(path(X,Y,C)),
                            sup2.1(X,Y,C,Z,C1).
dep(naive_sh_path(X,Y,C), path(X,Y,C)) ← magic(path(X,Y,C)),
                            sup3.0(X,Y,C).

depo'(X) ← magic_o(X), o X.

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

The program fragment P_2^m has the following meta-rule:

$$\begin{aligned} \circ\ \texttt{naive\_sh\_path(X,Y,C)} \leftarrow\ & \mathit{magic}_o(\texttt{naive\_sh\_path(X,Y,C)}), \\ & \forall Q(\mathit{depo}(\texttt{naive\_sh\_path(X,Y,C)}, Q) \Rightarrow \mathit{depo}'(Q)), \\ & \mathit{optimal}(\texttt{naive\_sh\_path(X,Y,C)}). \end{aligned}$$

The magic predicates, *magic* and $magic_o$, and all the other supplementary predicates are defined using just definite clauses. Therefore, no circular non-stratified dependencies are introduced by this kind of magic rewriting. In addition, the reader may note that the above rewriting technique requires local stratification amongst the *O-predicates*. This however does not place any restriction on the cyclic dependence amongst instances of *C-predicates*. Therefore, even if the graph in the above example has a cyclic edge relation, the bottom-up procedure will be able to terminate.

Theorem 4.5
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
We outline the proof of the theorem when q is an *O-predicate*. The other cases follow in a natural fashion. Furthermore, to show that this is true for *O-predicates*, we induct on the level of the *O-predicates*. In the base case, suppose the level of the *O-predicate* is 1. Since we can interpret the $\rightarrow$ clauses as $\leftarrow$ clauses, we can show that the bottom-up evaluation of the rewritten program generates a fact of the form $magic(p(\bar{u}))$ if the top-down evaluation starting from $q(\bar{t})$ had to solve for $p(\bar{u})$. The soundness of the rewriting technique requires that a predicate appearing at the head of an optimization clause must be invoked with unbound variables at certain argument positions—the values at these positions being determined by the body of the optimization clause and used by the arbiter to prefer one solution over another. Therefore, the query added to the program should be of the form $magic_o(q(\bar{t}))$ where some of the argument positions in $\bar{t}$ are unbound. If the goal G is not sufficiently uninstantiated, it could happen that the optimal instances are not computed by the bottom-up evaluation at all. This requirement is identical to the non-groundness requirement that the top down evaluation technique introduced by us in References[5,6]. Therefore, we can show that the rewritten program and the original program are equivalent with respect to the answers to the query for this case. The inductive case is similar. ■

The rewriting technique for computing answers for relaxation queries is a straightforward extension of the foregoing ideas. 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 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 $\langle P', Q' \rangle$. By the correctness of the translation, we can prove the correctness theorem for relaxation queries.

§5 Conclusions and Further Research

In this paper, we have shown that the notion of preference adds substantial power to deductive database query languages such as datalog by allowing one to express criterion for optimization and relaxation in a declarative and modular manner. We introduced preference datalog programs as preference logic programs[5,6] without any uninterpreted function symbols as the formalism in which to explore the use of preference in deductive databases. We proposed a bottom-up evaluation technique for evaluating answers to normal and relaxation queries as opposed to the top-down evaluation technique proposed in our earlier work.[5,6] We also developed a modification of magic rewriting technique so that the bottom-up evaluation of the rewritten program will not make any deductions that are not relevant to answering the query.

While we discussed the evaluation of relaxable queries, we did not describe the evaluation technique when relaxable goals appear in bodies of clauses defining various predicates. The usefulness of such a construct is made apparent by the following example that computes the path with the n^{th}-lowest cost between any two nodes in a graph:

```
n_sh_path(1,X,Y,C,P) ← naive_sh_path(X,Y,C,P).
n_sh_path(N+1,X,Y,C,P) ← n_sh_path(N,X,Y,D,_),
          RELAX sh_path(X,Y,C,P) WRT C > D.
```

The top-down evaluation technique presented in Reference[6] performed a program transformation and evaluated the rewritten program to answer such queries. We are investigating efficient ways of incorporating such a scheme in our bottom-up evaluation technique. The relaxation regime presented here allowed the user to specify how to modify the definition of the optimization predicates to obtain solutions when none exists. One can also envisage a similar modification of the preference clauses of any O-predicate.

Another direction for further research would be to extend PDP by permitting in the bodies of clauses constraints as in CLP.[7] The resulting paradigm can be viewed as an extension of CQL[8] with preferences. Just as this paper extended the bottom-up evaluation technique for datalog to preference datalog, we would have to extend the bottom-up evaluation technique for CQLs[8,17] to the extended paradigm.

We are also interested in extending the paradigm to incorporate inductive aggregates such as sum. This may be achieved by adding bags as a built-in data-type. The interaction of inductive aggregates and relaxation also provides interesting problems for research. The queries that we allow are first order queries and the bodies of arbiter clauses are not allowed to have preferential goals. A very natural kind of query to allow is whether some solution to a goal is preferred over another. It is also natural to have preferences about our preferences, and for certain preferences to depend on other preferences.

We believe that preferences will provide an important technique for querying the world-wide web for information. In traditional database systems, the data is standardized with respect to some data model, and there is a close relationship between the query langauge and the data model. However, when the amount of information being stored in the database is very large or there are data sources with dissimilar data formats, querying by traditional query languages can be problematic. Often, specifying a query purely in terms of constraints results in either to too many solutions or non at all. The problem is compounded when sources of data do not necessarily conform to any fixed data model as in the case of the world-wide web (WWW). Most web search engines today allow the queries that often result in very unintuitive answers. The number of answers returned by these search engines are either in the millions or none at all. And, sometimes an apparently more specific query provides more answers.

The query power and flexibility provided by preference datalog will be very relevant in such applications, as it will help the user better control the search. Preferences can be thought of as modular additions to current search engines. For instance, if one is interested in a web-page with a certain property, one can formulate a query with that property as a preference. This way, the web-pages that do not satisfy that property will not be reported as potential answers. However, in the event that no web-page has the property, the user will get whatever the search engine currently provides.

References

1) Brown, A., Mantha, S. and Wakayama, T., "Preference Logics: Towards a Unified Approach to Non-Monotonicity in Deductive Reasoning," *Annals of Mathematics and Artificial Intelligence, 10*, pp. 233–280, 1994.

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

3) Gelfond M. and Lifschitz, V., "The Stable Model Semantics for Logic Programming," in *Proc. of 5th Joint International Conference and Symposium on Logic Programming* (Kowalski, R. A. and Bowen, K. A., eds.), pp. 1081–1086, 1988.

4) Govindarajan, K., "Optimization and Relaxation in Logic Languages," *PhD thesis*, Dept. of Computer Science, SUNY-Buffalo, 1997.

5) Govindarajan, K., Jayaraman, B. and Mantha, S., "Preference Logic Programming," in *Proc. of 12th Intl. Conf. on Logic Programming*, pp. 731–745. MIT Press, 1995.

6) Govindarajan, K., Jayaraman, B. and Mantha, S., "Optimization and Relaxation in Constraint Logic Languages," in *Proc. of 23rd ACM Symposiun on Principles of Programming Languages*, pp. 91–103, 1996.

7) Jaffar, J. and Lassez, J. L., "Constraint Logic Programming," in *Proc. of 14th ACM Symp. on Principles of Programming Languages*, pp. 111–119, 1987.

8) Kannelakis, P. C., Kuper, G. M. and Revesz, P. Z., "Constraint Query Languages," in *Proc. of ACM Symp. on Principles of Database Systems*, pp. 299–313, 1990.

9) Kemp, D. B. and Stuckey, P. J., "Semantics of Logic Programs with Aggregates," in *Proc. of International Logic Programming Symposium*, 1991.

10) Lacroix, M. and Lavency, P., "Preferences: Putting More Knowledge into Queries," in *Proc. of 13th Intl. Conf. on Very Large Data Bases*, pp. 217–225, 1987.

11) Mantha, S., "First-Order Preference Theories and their Applications," *PhD thesis*, University of Utah, November, 1991.

12) Marek, V. W., Schwarz, G. F. and Truszczynski, M., "Modal Nonmonotonic Logics: Ranges, Characterization, Computation," *JACM, 40, 4*, pp. 963–990, September 1993.

13) Ramakrishnan, R., "Magic Templates: A Spellbinding Approach to Logic Programs," in *Proc. of 5th Joint Intl. Conf. and Symp. on Logic Programming*, pp. 140–159, 1988.

14) Ramakrishnan, R., Srivastava, D. and Sudarshan, S., "Efficient Bottom-up Evaluation of Logic Programs," in *the State of the Art in Computer Systems and Software Engineering* (Vandewalle, J., ed.), Kluwer Academic Publishers, 1992.

15) Ross, K. A. and Sagiv, Y., "Monotonic Aggregation in Deductive Databases," in *Proc. of ACM Symp. on Principles of Database Systems*, pp. 114–126, 1992.

16) Ross, K. A., "Modular Stratification and Magic Sets for Datalog Programs with Negation," *Journal of the ACM, 41, 6*, pp. 1216–1266, 1994.

17) Srivastava, D. and Ramakrishnan, R., "Pushing Constraint Selections," *Journal of Logic Programming, 16, 3–4*, pp. 361–414, 1993. A Preliminary Version Appeared in the Proc. ACM Symp. on Principles of Database Systems 1992.

18) Sudarshan, S. and Ramakrishnan, R., "Aggregation and Relevance in Deductive Databases," in *Proc. of the International Conference on Very Large Databases*, 1991.

19) Sudarshan, S., Srivastava, D., Ramakrishnan, R. and Beeri, C., "Extending the Well-Founded and Valid Semantics for Aggregation," in *Proc. of International Logic Programming Symposium*, pp. 590–608, 1993.

20) Ullman, J. D., "Bottom-up Beats Top-down for Datalog," in *Proc. of ACM Symp. on Principles of Database Systems*, 1989.

21) van Gelder, A., "The Well-founded Semantics of Aggregation," in *Proc. of 11th ACM Symposium on Principles of Database Systems*, pp. 127–138, 1992.

22) van Gelder, A., Ross, K. and Schlipf, J. S., "Unfounded Sets and Well-Founded Semantics for General Logic Programs," *JACM, 38, 3*, pp. 620–650, 1991.

23) Wilson, M. and Borning, A., "Hierarchical Constraint Logic Programming," *Journal of Logic Programming, 16*, pp. 277–318, 1993.

Appendix

We provide here a brief description of the model theory of preference logic programs; a full description may be found in Govindarajan's dissertation.[4] We begin with a a brief review of the logic of preference. The syntax of the logic of preference introduced by Mantha[11] 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*[1]:

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

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 $w \preceq w'$.

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 at most 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,[11] 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

*[1] 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 $\mathcal{M}$.

ordering among the worlds in the preference model.

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

Kannan Govindarajan, Ph.D.: He obtained his bachelors degree in Computer Science and Engineering from the Indian Institute of Technology, Madras, and he completed his Ph.D. degree in Computer Science from the State University of New York at Buffalo. His dissertation research was on optimization and relaxation techniques for logic languages. His interests lie in the areas of programming languages, databases, and distributed systems. He currently leads the trading community effort in the E-speak Operation in Hewlett Packard Company. Prior to that, he was a member of the Java Products Group in Oracle Corporation.

Bharat Jayaraman, Ph.D.: He is a Professor in the Department of Computer Science at the State University of New York at Buffalo. He obtained his bachelors degree in Electronics from the Indian Institute of Technology, Madras (1975), and his Ph.D. from the University of Utah (1981). His research interests are in programming languages and declarative modeling of complex systems. Dr. Jayaraman has published over 50 papers in refereed conferences and journals. He has served on the program committees of several conferences in the area of programming languages, and he is presently on the Editorial Board of the Journal of Functional and Logic Programming.

Surya Mantha, Ph.D.: He is a manager in the Communications and Software Services Group of Pittiglio Rabin Todd & McGrath (PRTM), a management consulting firm serving high technology industries. He obtained a bachelors degree in Computer Science and Engineering from the Indian Institute of Technology, Kanpur, an MBA in Finance and Competitive Strategy from the University of Rochester, and a Ph.D. in Computer Science from the University of Utah (1991). His research interests are in the modeling of complex business processes, inter-enterprise application integration, and business strategy. Dr. Mantha has two US patents, and has published over 10 research papers. Prior to joining PRTM, he was a researcher and manager in the Architecture and Document Services Technology Center at Xerox Corporation in Rochester, New York.