Abstract. Preference logic programming (PLP) is an extension of constraint logic programming for declaratively specifying problems requiring optimization or comparison and selection among alternative solutions to a query. PLP essentially separates the programming of a problem itself from the criteria specification of its solution selection. In this paper we give a precise formalization for the syntax and semantics of PLP based on the Herbrand model theory. This paper also presents an elegant and easy method of specifying and executing preference logic programs in terms of tabled Prolog. The method introduces a formal predicate mode declaration for designating certain predicates as optimization predicates, and stating the criteria for determining their optimal solutions via preference rules. A tabled Prolog, incorporated with the flexible mode declaration, provides an easy implementation vehicle for programming with preferences. Automatic transformation is applied to embed the preferences into the problem specification for efficient evaluation. We show that the procedural semantics of a preference logic program is equivalent to its declarative semantics.

1 Introduction

Constraint technology has found use in a wide variety of applications areas [13], e.g., electric circuit analysis, options trading, scheduling, DNA sequencing, interactive graphics. In particular, the paradigm of constraint logic programming (CLP) [16, 8, 9], wherein user-defined constraints are built into logic programs, has proven valuable for modeling many constraint problems. In many of these settings one is often interested in finding the optimal solutions to constraints with respect to some object functions. Recently, a promising extension of CLP called preference logic programming (PLP) [8, 9] has been introduced for declaratively specifying problems requiring optimization or comparison and selection among alternative solutions to a query.

The PLP paradigm essentially separates the constraints of a problem itself from the criteria specification of its solution selection or optimization. This kind of selection or optimization becomes a meta-level operation and therefore falls outside the standard constraint logic programming framework. The responsibility of how to find the optimal solution is shifted to the underlying logic programming system, in keeping with the spirit of logic programming as a declarative language. Preference logic programming has been shown useful by practical applications such as in artificial intelligence [2], data mining [4], document processing, and databases.
Preference logic programming was first proposed in [8, 9] as a new programming paradigm even though preference logics [21] has been studied back to early 1960’s. The main contribution of [8, 9] lies in showing how the concept of preferences provides a natural, declarative, and efficient means of specifying a host of practical problems using definite clauses. The declarative semantics of a preference logic program is given by its model-theoretic semantics [8, 9]. Reference [4] then extended part of Jayaraman’s work on preference logic grammars [14] to a three-valued preference logic grammars, and provided an implementation for specific applications on data standardization via normal logic programs using XSB [19]. Other efforts such as [15, 7, 20] have been done to incorporate optimization in a CLP framework; and [6, 17] addresses semantics for optimization predicates in a CLP framework.

In this paper we give a precise formalization for the syntax and semantics of PLP based on the Herbrand model theory. The specification of a general problem itself is separated from the preference specification of its solution selection. Their connection is established through a mode declaration scheme. Therefore, the semantics can be declared correspondingly as follows: the computation model of the general problem is defined as the least Herbrand model [12]; the semantics of preferences is defined as a partial order relation among solutions; and the incorporation of these two semantics components is made through a sequence of meta-level mapping operations over the least Herbrand model.

This paper also presents an elegant method of specifying and executing preference logic programs in the paradigm of tabled Prolog. We focus on the definite logic programs in this paper. Preferences on alternative solutions to a query are defined using a set of well-defined logic clauses, which are separated from the logic programs to the problem specification. The connection between preferences and the problem specification is made through a formal mode declaration for tabled predicates. The computation of preference logic programs are achieved in two steps. First, mode-directed automatic transformation is applied to embed the preferences into the problem specification for efficient evaluation. Second, the transformed program is then evaluated using tabled resolution, while the mode declaration provides selection mechanism among the alternative solutions. We show that the procedural semantics of preference logic programs is consistent to its declarative semantics.

We use a tabled Prolog system [19, 22, 10, 18] to implement preference logic programs, because a tabled Prolog can be thought of as an engine for efficiently computing fixed points, which is critical for finding the model-theoretic semantics of a preference program. A tabled Prolog is essential for extending traditional Prolog system with tabled resolutions. The main advantages of tabled resolution are that a tabled Prolog system terminates more often by computing fixed points, avoids redundant computation by memoing the computed answers, and keeps the declarative and procedural semantics consistent for pure logic programs with bounded-size terms.

A new mode-declaration scheme for tabled Prolog systems has recently been proposed in [11] to provide an attractive platform for making dynamic pro-
gramming simpler: there is no need to define the value of an optimal solution recursively, instead, defining the value of a general solution is enough. The optimal value as well as its associated solution, will be computed implicitly and automatically in a tabled Prolog system that uses the appropriate mode declaration. This mode-declaration scheme can be further extended for specifying and executing preference logic programs, since the mode-declaration scheme is indeed the intent of PLP, that is, tabled Prolog systems can selectively choose “better” answers for a given tabled predicate call guided by the declared mode, while PLP selectively chooses the “better” solutions based on preferences. In this paper, we focus on the optimization problems with a single best solution. Several examples are used to show how our declarative methods can be applied for solving practical optimization problems.

The main differences between the present paper and the original formulation of PLP [8, 9] are as follows:

- In the earlier formulation, optimization predicates are explicitly defined by using special rules called optimization clauses, and the clauses for the general problem are semantically independent from those optimization predicates. In this paper, an optimization predicate is simply specified by declaring modes on a given tabled predicate. As a result, the semantics of the clauses for the general problem will eventually be pruned based on the mode-directed preferences.

- In the papers [8, 9], the semantics for the constraints of the general problem are captured by its least Herbrand model; and the semantics of the optimization clauses are captured by a possible-worlds semantics: each world is a model for the constraints of the general problem, and an ordering over these worlds is enforced by preferences. In this paper, we give a new semantic definition for a preference logic program: the semantics for the constraints of the general problem are, as before, captured by its least Herbrand model; however, the semantics of the preferences is defined as a partial-order relation. How preferences affect the least Herbrand model is achieved through a sequence of meta-level mapping operations. The main advantage of this new semantics definition is to provide the link between the declarative and procedural semantics for a preference program.

- Additionally, we defines both syntax and semantics of preference logic programs through tabled Prolog programs, and provide a feasible implementation over a tabled Prolog system. In the papers [8, 9], a detailed implementation model for PLP was not provided; rather an abstract operational semantics for optimization and relaxation problems was outlined.

The rest of the paper is organized as follows: Section 2 gives a brief introduction on tabled Prolog systems and a new mode declaration scheme for tabled predicates. Section 3 presents the syntax and semantics of preference logic programs in terms of tabled Prolog programs, and how the mode declaration scheme in a tabled Prolog are extended for specifying and executing preference logic programs. Section 4 addresses the implementation issues of the declaration scheme
and presents the running performance on some optimization benchmarks. Finally, section 5 gives our conclusions.

2 Tabled Prolog with New Mode Declarations

2.1 Tabled Prolog

Traditional Prolog systems use SLD resolution [12] with the following computation strategy: subgoals of a resolvent are solved from left to right and clauses that match a subgoal are applied in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. That is, given any static computation strategy, one can always produce a program in which no answers can be found due to non-termination even though some answers may logically follow from the program. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

Tabled Prolog [19, 22, 10, 18] eliminates such infinite loops by extending logic programming with tabled resolution. The main idea is to memorize the answers to some calls and use the memorized answers to resolve subsequent variant calls. Tabled resolution adopts a dynamic computation strategy while resolving subgoals in the current resolvent against matched program clauses or tabled answers. It keeps track of the nature and type of the subgoals; if the subgoal in the current resolvent is a variant of a former tabled call, tabled answers are used to resolve the subgoal; otherwise, program clauses are used following SLD resolution. Thus, a tabled Prolog system can be thought of as an engine for efficiently computing fixed points.

In a tabled Prolog system, only tabled predicates are resolved using tabled resolution. Tabled predicates are explicitly declared as

:- table p/n.

where \( p \) is a predicate name and \( n \) is its arity. A global data structure \textit{table} is introduced to memorize the answers of any subgoals to tabled predicates, and to avoid recomputation. Consider the path program checking the existence of a path in Fig. 1(a). This program does not work in a traditional Prolog system. With the declaration of a tabled predicate \texttt{path/2} in a tabled Prolog system, it can successfully find the complete solutions due to the fixed point computation strategy.

The fixed point of a computing model may contain infinite number of solutions, which certainly affects the completion of the computation. Consider another path program searching the paths in Fig. 1(b). An extra argument is added for the predicate \texttt{path/3} to record the found path. However, this extra argument results in the fixed point of the computation infinite, since there are infinite number of paths from \( a \) to any node due to the cycle between \( a \) and \( b \). Therefore, a meta-level operation is useful to filter the infinite-size solution set to a finite one so that the computation can be completed.
2.2 New Mode Declarations for Tabled Predicates

This meta-level operation can be achieved by a mode declaration method in a tabled Prolog system. The new mode declaration for tabled predicates can be described in the form of

\[
\text{:- table}_{-}\text{mode } q(m_1,\ldots,m_n).\]

where \(q\) is a tabled predicate name, \(n \geq 0\), and each \(m_i\) has one of the forms as defined in Table 1.

<table>
<thead>
<tr>
<th>Modes</th>
<th>Informal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>an indexed argument;</td>
</tr>
<tr>
<td>−</td>
<td>a non-indexed argument;</td>
</tr>
<tr>
<td>min</td>
<td>a minimum non-indexed argument;</td>
</tr>
<tr>
<td>max</td>
<td>a maximum non-indexed argument;</td>
</tr>
<tr>
<td>last</td>
<td>a non-indexed argument recording the last answer;</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>a non-indexed argument controlled by user-defined preferences.</td>
</tr>
</tbody>
</table>

Table 1. Built-in Modes for Tabled Predicates

The mode declaration [11] was initially used to classify arguments as indexed (+) or non-indexed (−) for each tabled predicate. Only indexed arguments are used for variant checking during collecting new generated answers into the table. For each tabled call, any answer generated later for the same value of the indexed arguments is discarded because it is a variant (w.r.t. the indexed arguments) of a previously tabled answer. This step is crucial in ensuring that a fixed point is reached. Consider again the path program in Fig. 1(b). Suppose we declare the mode as “:- table_{mode} path(+,+,−)”; this means that only the first two arguments of the predicate path/3 are used for variant checking. As a result, the computation can be completed properly with three answers, that is, each reachable node from \(a\) has a simple path as an explanation.

The mode directive tabled_{mode} makes it very easy and efficient to extract explanation for tabled predicates. In fact, our strategy of ignoring the explanation argument (mode ‘−’) during variant checking results in only the first explanation for each tabled answer being recorded. Subsequent explanations are filtered by our modified variant checking scheme. Essentially, if we regard a tabled
predicate as a function, then all the non-indexed arguments are uniquely defined by the instances of indexed arguments. This feature also ensures that those generated explanations are concise and that cyclic explanations are guaranteed to be absent. More importantly, the meta-level mode declaration is especially useful to reduce an infinite computation model to a finite one for some practical uses.

The mode directive `table_mode` can be further extended to associate a non-indexed argument of a tabled predicate with some optimum constraint. With the mode `~`, a non-indexed argument for each tabled answer only records the very first instance. This “very first” property can actually be generalized to any optimum, e.g., the minimum value with mode `min` (or the maximum with mode `max`), in which case the global table will record answers with the value of that argument as small (or great) as possible. That is, a tabled answer can be dynamically replaced by a new one with smaller (or greater) value during the computation.

Contrary to the mode `~`, the mode `last` is useful for finding the last answer from a solution set; the mode `<<<` is used to support user-defined preferences. The incorporating uses of the modes `last` and `<<<` provides an elegant interface for preference logic programming, which will be addressed in the next section.

Notice that the new mode declaration method has been implemented in the tabled Prolog system TALS [10]. It can easily be applied on other tabled Prolog systems, since essentially only the variant checking operation is modified during collecting the tabled answers. Additionally, only two built-in optimum modes, `min` and `max`, are implemented in TALS, since these two optimums are popularly used in optimizations problems over the numeric domain; other optimums can be defined through the preference mode `<<<` as shown in the next section.

### 3 Preference Logic Programming

In this section, we formally addresses the syntax and semantics of preference logic programming in terms of tabled Prolog programs.

#### 3.1 Syntax

In optimization problems, we are often interested in comparing alternative solutions to a set of constraints and choosing the “best” one. In general there are two components to an optimization problem: specification of the constraints of the problem, and specification of what and how to be optimized. The intent of preference logic programming is to separate these two components and declaratively specify such applications.

**Definition 1 (Preference Logic Programs).** A (definite) preference logic program $P$ can be defined as a pair $<P_{\text{core}}, P_{\text{pref}}>$, where $P_{\text{core}}$ and $P_{\text{pref}}$ are two disjoint sets of definite clauses defined as follows: $P_{\text{core}}$ specifies the constraints of the general problem; $P_{\text{pref}}$ defines the predicate to be optimized with a mode declaration, and denotes the optimization criteria using a set of preference clauses (or preferences) in a form of:
\( T_1 \lll T_2 \text{ :- } B_1, B_2, \ldots, B_n. \quad (n \geq 0) \)

where two terms \( T_1 \) and \( T_2 \) are any numerical or structural terms, each \( B_i \) \((1 \leq i \leq n)\) is an atom as defined in standard logic programming, and the preference operator \( \lll \) is a special binary predicate.

The informal semantics of \( T_1 \lll T_2 \text{ :- } B_1, B_2, \ldots, B_n \) is that the term \( T_1 \) is less preferred than the one \( T_2 \) if \( B_1, B_2, \ldots, B_n \) are all true. Thus, if \( n > 0 \), a preference clause is conditional. On the other hand, a unit clause \( T_1 \lll T_2 \) is unconditional.

We abbreviate “preference logic program” to “preference program” and “tabled prolog program” to “tabled program” throughout.

**Example 1** Consider the following preference program searching for a shortest distance path, where \( \text{path}(X, Y, D, L) \) denotes a path from \( X \) to \( Y \) with the distance \( D \) and the path route \( L \).

\[
\begin{align*}
&\text{:- table path/4.} \quad (1) \\
&\text{path}(X, X, 0, []). \quad (2) \\
&\text{path}(X, Y, D, [e(X, Y)]) \text{ :- edge}(X, Y, D). \quad (3) \\
&\text{path}(X, Y, D, [e(X, Z) | P]) \text{ :-} \\
&\quad \text{edge}(X, Z, D1), \text{path}(Z, Y, D2, P), D \text{ is } D1 + D2. \quad (4) \\
&\text{edge}(a, b, 4). \quad \text{edge}(b, a, 3). \quad \text{edge}(b, c, 2). \quad (5) \\
&\text{:- table_mode path(+, +, \lll, -).} \quad (6) \\
&D1 \lll D2 \text{ :- } D2 < D1. \quad (7)
\end{align*}
\]

Clauses (1) to (5) make up the core program \( P_{\text{core}} \) defining the path relation and a directed graph with a set of edges; clause (6) and (7), the preference clauses \( P_{\text{pref}} \), specifies the predicate \( \text{path}/4 \) to be optimized and gives the criteria how to optimize the \( \text{path}/4 \) predicate, that is, the path for each pair of reachable nodes (according to the first two indexed arguments in \( \text{path}/4 \)) should be optimized based on the definition of \( \lll \): the shorter path is preferred.

**Example 2** Consider the following preference program searching for a lowest-cost path, where \( \text{path}(X, Y, C, D, L) \) denotes a path from \( X \) to \( Y \) with the cost \( C \), the distance \( D \) and the path route \( L \).

\[
\begin{align*}
&\text{:- table path/5.} \quad (8) \\
&\text{path}(X, Y, 0, 0, []). \quad (9) \\
&\text{path}(X, Y, C, D, [e(X, Y)]) \text{ :- edge}(X, Y, C, D). \quad (10) \\
&\text{path}(X, Y, C, D, [e(X, Z) | P]) \text{ :-} \\
&\quad \text{edge}(X, Z, C1, D1), \text{path}(Z, Y, C2, D2, P), \\
&\quad D \text{ is } D1 + D2, C \text{ is } C1 + C2. \quad (11) \\
&\text{:- table_mode path(+, +, \lll, \lll, -).} \quad (12) \\
&(C1, D1) \lll (C2, D2) \text{ :- } C2 < C1. \quad (13) \\
&(C1, D1) \lll (C2, D2) \text{ :- } C1 = C2, D2 < D1. \quad (14)
\end{align*}
\]
In this example, the preferences are defined over two arguments in path/5. Preference rule (13) tells that a lower-cost path is preferred, and rule (14) tells that if the costs of two paths are same, then a shorter path is preferred.

3.2 Declarative Semantics
The declarative semantics of a preference program is based on the Herbrand model theory [5, 12]. The preferences are essentially interpreted as a sequence of meta-level mapping operation over the least Herbrand model for the core program. We use the following notational conventions in the rest of paper: \( P \) is used to denote a preference logic program \( \langle P_{\text{core}}, P_{\text{pref}} \rangle \), \( B \) to denote the Herbrand base of \( P_{\text{core}} \), \( 2^{B_{\text{core}}} \) to denote the set of all Herbrand interpretations of \( P_{\text{core}} \), \( \omega \) a Herbrand atom to denote an atom in \( B_{\text{core}} \), \( o \) arbitrary atom used to denote a preference logic program.

Definition 3 (A Preference Relation). Let \( q/n \) be the predicate to be optimized with a mode declaration \( q(m_1, m_2, \cdots, m_n) \) in a preference program \( P \); let \( m_{i1}, m_{i2}, \cdots, m_{ik} \) \((0 \leq k \leq n)\) be all the modes ‘+’ such that \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \); let \( m_{i1}, m_{i2}, \cdots, m_{ij} \) \((0 \leq j \leq n)\) be all the modes ‘<<<’ such that \( 1 \leq o_1 < o_2 < \cdots < o_j \leq n \). We define two functions \( K_{q/n} \) and \( O_{q/n} \) as follows: given an arbitrary atom \( q(a_1, a_2, \cdots, a_n) \),

\[
K_{q/n}(q(a_1, a_2, \cdots, a_n)) = (a_{i1}, a_{i2}, \cdots, a_{ik});
O_{q/n}(q(a_1, a_2, \cdots, a_n)) = (a_{o1}, a_{o2}, \cdots, a_{oj}).
\]

Given an atom of \( q/n \), the purposes of the above functions \( K_{q/n} \) and \( O_{q/n} \) are to return a sequence of indexed arguments and optimized arguments, respectively, in a left-to-right order. We say two atoms of \( q/n \) comparable if and only if these two atoms have the same indexed arguments but comparable optimized arguments in terms of ‘<<<’. Thus, we have a preference relation defined as follows.

Definition 3 (A Preference Relation). Let \( P \) be a preference program and \( q/n \) be the optimized predicate. A preference relation of \( P_{\text{pref}} \) is an ordered relation \( \prec_P \) s.t. for any two Herbrand atoms \( A_1 \) and \( A_2 \) of the predicate q/n, \( A_1 \prec_P A_2 \) if \( K_{q/n}(A_1) = K_{q/n}(A_2) \) and \( O_{q/n}(A_1) \lll O_{q/n}(A_2) \).

For instance, consider the Example 1. Its preference relation \( \prec_P \) is the set

\[
\{ \text{path}(a, a, 1, \_ \prec_P \text{path}(a, a, 0, \_), \cdots, \text{path}(a, a, i, \_ \prec_P \text{path}(a, a, j, \_), \cdots, \\
\text{path}(a, b, 1, \_ \prec_P \text{path}(a, b, 0, \_), \cdots, \text{path}(a, b, i, \_ \prec_P \text{path}(a, b, j, \_), \cdots, \\
\text{path}(c, c, 1, \_ \prec_P \text{path}(c, c, 0, \_), \cdots, \text{path}(c, c, i, \_ \prec_P \text{path}(c, c, j, \_), \cdots \} ,
\]

where we assume that the distance is defined over the natural numbers, and ‘\_’ is used to represent any ground term from the Herbrand universe.
Definition 4 (Model and Intended Model). Let $P$ be a preference program, $q/n$ be the optimized predicate, and $I$ be an interpretation for $P_{\text{core}}$. We say $I$ is a model for $P$ if

a). For any atom $A$ in $I$, there exists a ground instance, $A : - B_1, \cdots, B_n$, of a clause in $P$ s.t. $\{B_1, \ldots, B_n\} \subseteq I$;

b). For any two atoms $A_1$ and $A_2$ of $q/n$ in $I$, neither $A_1 \prec_P A_2$ nor $A_2 \prec_P A_1$ is true.

Further, we say $I$ is an Intended model for $P$ if

c). For any atom $A$ of $q/n$ in $I$, if there exists a Herbrand atom $A_1$ s.t. $A \prec_P A_1$, then $A_1$ is not in the least Herbrand model for $P_{\text{core}}$. We call $A$ an optimized atom in $I$.

Note that the model for $P_{\text{core}}$ follows the standard model definition [5, 12] for definite clauses, which is different from Def. 4 for a preference program $P$. Def. 4(c) tells that no better-preferred atom $A_1$ than the optimized atom $A$ can be found in the least Herbrand model $P_{\text{core}}$, otherwise, $A$ cannot be an optimized atom. However, there may exist a Herbrand atom $A_1 \notin P_{\text{core}}$ better-preferred than $A$.

We wish to obtain the link between the models of $P$ and $P_{\text{core}}$ so that we can find out how preferences affect the semantics of a general program. For this we need to introduce two new meta-level mappings defined over Herbrand interpretations.

Definition 5. Let $P$ be a preference program, $M$ be a Herbrand model for $P_{\text{core}}$, $q/n$ be the predicate to be optimized, and $M_1$ be a subset of $M$ containing all the atoms with $q/n$. We define a meta-level mapping $\phi_P : 2^{B_{\text{core}}} \rightarrow 2^{B_{\text{core}}}$ as follows:

$$\phi_P(M) = M - \{A \in M_1 : \exists A_1 \in M_1 \text{ s.t. } A \prec_P A_1\}.$$ 

Definition 6. Let $P$ be a preference program. We define a meta-level mapping $\pi_P : 2^{B_{\text{core}}} \rightarrow 2^{B_{\text{core}}}$ as follows: $\pi_P(M) = \{A \in M : A : - A_1, \cdots, A_n \text{ is a ground instance of a clause in } P_{\text{core}} \text{ and } \{A_1, \cdots, A_n\} \subseteq M\}$.

The above two mappings provides the link between the declarative and procedural semantics of a preference program. The mapping $\phi_P$ filters non-optimal atoms from the model according to the preference relation; the mapping $\pi_P$ filters those atoms depending on the removed non-optimal atoms from the model. It is obvious that $\pi_P$ is monotonic $\pi_P(I) \subseteq I$ for any given Herbrand interpretation. Thus, we come to a major result of the theory as shown in the next two theorems.

Theorem 1. Let $P$ be a preference program and $M_{\text{core}}^P$ be the least Herbrand model for $P_{\text{core}}$. Then $M_P = \pi_P \uparrow \omega(\phi_P(M_{\text{core}}^P))$ is an intended model for $P$. 

9
**Proof**: We show how $M_P$ satisfies the properties (a), (b), and (c) as defined in the Def. 4 for an intended model:

1. Based on the definition of $\phi_P$, it is clear that $\phi_P(M_P^{\mathsf{core}})$ satisfies the property (b); Since $\pi_P$ is a monotonic s.t. $\pi_P(I) \subseteq I$ for any given Herbrand interpretation, $\pi_P \uparrow \omega(\phi_P(M_P^{\mathsf{core}}))$ satisfies the property (b) too.

2. We associate a complete lattice with the program $P$. $2^{B_{\mathsf{core}}}$, the set of all Herbrand interpretations of $P_{\mathsf{core}}$ and $P$, is a complete lattice under the partial order of set inclusion $\subseteq$, where the top element is $B_P$ and the bottom element is $\emptyset$. Thus, $M_P = \pi_P \uparrow \omega(\phi_P(M_P^{\mathsf{core}}))$ must be a fixed point of $\pi_P$ over the lattice, that is, $\pi_P(M_P) = M_P$. Therefore, $M_P$ satisfies the property (a), and hence it is a model for $P$.

3. Let $q/n$ be the optimized predicate and $A$ be one atom of $q/n$ in $M_P$. Assume that there exists a Herbrand atom $A_1 \in M_P$ s.t. $A \prec_P A_1$. According to the definition of $\phi_P$ in Def. 5, $A \notin \phi_P(M_P^{\mathsf{core}})$, and hence $A \notin M_P$, which is a contradiction to the fact that $A \in M_P$. Therefore, $M_P$ satisfies the property (c).

Thus, $M_P$ is an intended mode for $P$. 

**Theorem 2.** Let $P$ be a preference program. Then $M_P$ exists and is unique.

**Proof**: Both the existence and uniqueness of $M_P$ are determined respectively by those of $M_P^{\mathsf{core}}$, the least Herbrand model for $P_{\mathsf{core}}$. For each definite logic program, $M_P^{\mathsf{core}}$ exists and is unique [5, 12]. The proof is therefore completed. 

If we reconsider the preference program in the Example 1. Its least Herbrand model $M_P^{\mathsf{core}}$ and $\phi_P(M_P^{\mathsf{core}})$ are shown below.

$$M_P^{\mathsf{core}} = \{ \text{edge}(a, b, 4), \text{edge}(b, a, 3), \text{edge}(b, c, 2),$$
$$\text{path}(a, a, 0, []), \text{path}(a, a, 7, [(a, b), (b, a)]), \ldots$$
$$\ldots \text{path}(c, c, 0, []) \}$$

$$\phi_P(M_P^{\mathsf{core}}) = \{ \text{edge}(a, b, 4), \text{edge}(b, a, 3), \text{edge}(b, c, 2),$$
$$\text{path}(a, a, 0, []), \text{path}(a, b, 4, [(a, b)]), \text{path}(a, c, 6, [(a, b), (b, c)]),$$
$$\text{path}(b, a, 3, [(b, a)]), \text{path}(b, b, 0, []), \text{path}(b, c, 2, [(b, c)]),$$
$$\text{path}(c, c, 0, []) \}$$

We also have $\pi_P \uparrow \omega(\phi_P(M_P^{\mathsf{core}})) = \phi_P(M_P^{\mathsf{core}})$ for this program. However, if we add an extra clause “$\text{shortest}(X, Y, D, P) \leftarrow \text{path}(X, Y, D, P)$.”, then $\pi_P \uparrow \omega(\phi_P(M_P^{\mathsf{core}}))$ is different from $\phi_P(M_P^{\mathsf{core}})$, e.g., shortest($a, a, 7, [(a, b), (b, a)]$) $\notin \phi_P(M_P^{\mathsf{core}})$, but shortest($a, a, 7, [(a, b), (b, a)]$) $\notin \pi_P \uparrow \omega(\phi_P(M_P^{\mathsf{core}}))$.

**Example 3** Consider the following preference program with contradictory preferences:
The intended model of this program is an empty set, since $M^P_{core} = \{q(a), q(b)\}$ and $\phi_P(M^P_{core}) = \emptyset$; 
'<<<'$/2$ is tabled to avoid the non-termination because it
has been cyclically defined.

**Corollary 3** Let $P$ be a preference program and $q/n$ be its optimized predicate.

$A$ is an atom of $q/n$ and $A \in M_P$ if and only if $A$ is an optimized atom in $M^P_{core}$.

**Proof:** It is shown based on the Def. 4(c) since $M_P$ is an intended model for $P$. ⊓⊔

### 3.3 Procedural Semantics

A preference program $P$ can be automatically transformed to a new tabled program by incorporating the general problem specification $P_{core}$ and the optimization criteria $P_{pref}$. The optimization criteria are then elegantly embedded to filter the unoptimal answers. Thus, the procedural semantics of a preference logic program is dependent on that of a tabled program.

**Example 4** Consider the following transformed tabled program from the program in Example 1.

```prolog
:- table q/1.
q(a).
q(b).
:- table_mode q(<<<).
:- table '<<<'/2.
a <<< b.
b <<< a.

A is an atom of $q/n$ and $A \in M_P$ if and only if $A$ is an optimized atom in $M^P_{core}$.

**Proof:** It is shown based on the Def. 4(c) since $M_P$ is an intended model for $P$. ⊓⊔

### 3.3 Procedural Semantics

A preference program $P$ can be automatically transformed to a new tabled program by incorporating the general problem specification $P_{core}$ and the optimization criteria $P_{pref}$. The optimization criteria are then elegantly embedded to filter the unoptimal answers. Thus, the procedural semantics of a preference logic program is dependent on that of a tabled program.

**Example 4** Consider the following transformed tabled program from the program in Example 1.

```prolog
:- table path/4.
pathNew(X, X, 0, []). (1)
pathNew(X, Y, D, [e(X, Y)]) :- edge(X, Y, D). (2)
pathNew(X, Y, D, [e(X, Z) | P]) :-
  edge(X, Z, D1), path(Z, Y, D2, P), D is D1 + D2. (3)
edge(a,b,4). edge(b,a,3). edge(b,c,2). (5)

D1 <<< D2 :- D2 < D1. (7)

path(X, Y, D, P) :-
  pathNew(X, Y, D, P),
  ( path(X, Y, D1, P1) -> D1 <<< D; true ). (8)
```

Three major changes have been made in this transformation by taking advantage of the unique global table in the system: (i) The original predicate path/4 in Example 1 is replaced by a new predicate pathNew/4 to emphasize that this
predicate is to generate a new preferred path candidate from \( X \) to \( Y \). (ii) The predicate \( \text{path/4} \), given a new definition as the clause (8), represents the way how to identify a preferred answer. The meaning of the clause (8) is the following: given a path candidate \( A \) by \( \text{pathNew}(X,Y,D,P) \), we need to check whether there already exists a tabled answer, if so, they are compared to each other to keep the preferred one in the table; otherwise, the candidate is recorded as a first tabled answer. By this way, the optimal solution must be the last one added into the table. (iii) The table mode \( <<< \) for \( \text{path/4} \) is hence replaced by \( \text{last} \) to catch the last and optimal tabled answer.

**Definition 7 (**\( \rho \)-Transformation).** Let \( P \) be a preference program and \( \text{q/n} \) be the optimized predicate. The transformation to a new tabled program \( P' = \rho(P) \) can be formalized as follows:

1. For each clause defining \( \text{q/n} \) in \( P \), the predicate \( \text{q/n} \) in its head (the part to the left of \( :- \)) is renamed to a new predicate \( \text{q}'/n \);
2. A new clause definition for \( \text{q/n} \) is introduced in the form of:

\[
\text{q}(a_1, \cdots, a_n) :- \text{q}'(a_1, \cdots, a_n), \not \text{not } D_2 <<< D_2, \\
(q(b_1, \cdots, b_n) \rightarrow D_1 <<< D_2; \text{true}).
\]

where \( a_1, \cdots, a_n \) and \( b_1, \cdots, b_n \) are all variables; for \( 1 \leq i, j \leq n \), \( a_i \) and \( b_i \) use a same variable if and only if \( a_i \in K_{\text{q/n}}(a_1, \cdots, a_n) \) (i.e., all the corresponding indexed arguments are the same variable); \( a_i \) and \( a_j \) use two different variables if \( i \neq j \); \( D_1 = O_{\text{q/n}}(b_1, \cdots, b_n) \) and \( D_2 = O_{\text{q/n}}(a_1, \cdots, a_n) \); and \( \not \text{not } \) gives negation by failure;
3. In the mode declaration of \( \text{q/n} \), the first preference mode \( <<< \) is changed into \( \text{last} \), and the rest preference modes (if any) are changed into the standard non-indexed mode \( - \);
4. For all the rest clauses in \( P \), make a same copy into \( P' \).

We call this transformation \( \rho \)-transformation.

The purpose of \( \not \text{not } D_2 <<< D_2 \) is to check whether \( D_2 \) is involved into any contradictory preferences. It is worthwhile to be mentioned that for the consideration of efficiency, \( \not \text{not } D_2 <<< D_2 \) is omitted in practice unless the predicate \( <<< /2 \) itself is tabled, which gives a signal that contradictory preferences might exist. In the general case, the detection of contradictory preferences is an undecidable problem; even for the programs with bounded-size terms (i.e., finite domain), the detection is an NP complexity problem.

The procedural semantics of a tabled program is dependent on tabled resolution [3, 22, 10]. In spite of having different tabled resolution, a tabled Prolog can be thought of as an engine for efficiently computing the least fixed points. The procedure of computing fixed points of a definite logic program mimics the bottom-up computation strategy [10].
Definition 8. Let $P$ and $B_P$ be a program and its Herbrand base. We define a meta-level mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ as follows. Let $I$ be a Herbrand interpretation. Then $T_P(I) = \{ A \in B_P : A \vdash A_1, \ldots, A_n \text{ is a ground instance of a clause in } P \text{ and } \{A_1, \ldots, A_n\} \subseteq I \}.$

Definition 9. Given a tabled program $P$, its fixed point semantics can be described as $T_P \uparrow \omega(\emptyset)$.

Def. 8 was previously given in [12, 5]; The mapping $T_P$ is continuous and monotonic [12, 5]. Thus, we have the following major results. Theorem 5 shows the equivalence between the declarative semantics of a preference program and its procedural semantics over a transformed tabled program.

Proposition 4 Let $P$ be a preference program and $q/n$ be its optimized predicate. $A$ is an atom of $q/n$ and $A \in T^{\rho(P)} \uparrow \omega(\emptyset)$ if and only if $A$ is an optimized atom in $M^{P}_{\text{core}}$.

Proof: It is shown based on the transformed definition of $q/n$ in Def. 7.

Theorem 5. Let $P$ be a preference program. Then $\pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})) = T^{\rho(P)} \uparrow \omega(\emptyset)$, where $M^{P}_{\text{core}} = T^{P}_{\text{core}} \uparrow \omega(\emptyset)$.

Proof: Let $A$ and $q/n$ be a Herbrand atom and the optimized predicate, respectively. The proof is based on the following two cases:

(i) If $A$ is an atom of $q/n$,

\[
A \in \pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})) \iff A \text{ is an optimized atom in } M^{P}_{\text{core}}. \quad \text{Corollary 3}
\]

\[
\iff A \in T^{\rho(P)} \uparrow \omega(\emptyset) \quad \text{Proposition 4}
\]

(ii) If $A$ is not an atom of $q/n$, then the proof is a structural induction on the definition of the predicate for $A$:

Base case: Let $A$ be a ground instance of a fact clause in $P_{\text{core}}$.

\[
A \in \pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})) \iff A \text{ is an instance of a fact in } P_{\text{core}}. \quad \text{Def. 6}
\]

\[
\iff A \text{ is an instance of a fact in } \rho(P). \quad \text{Def. 7}
\]

\[
\iff A \in T^{\rho(P)} \uparrow \omega(\emptyset) \quad \text{Def. 8 and Def. 9}
\]

Inductive case: Assume that for any ground instance $A \vdash B_1, \ldots, B_n$, of a clause in $P_{\text{core}}$, we have $\{B_1, \ldots, B_n\} \subseteq \pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})) \iff \{B_1, \ldots, B_n\} \subseteq T^{\rho(P)} \uparrow \omega(\emptyset)$, where $n > 0$.

\[
A \in \pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})) \rightarrow \exists \text{ a ground instance } A \vdash B_1, \ldots, B_n \quad \text{Def. 6}
\]

\[
s.t. \{B_1, \ldots, B_n\} \subseteq \pi_P \uparrow \omega(\phi_P(M^{P}_{\text{core}})). \quad \text{Induction hypothesis}
\]

\[
\iff \{B_1, \ldots, B_n\} \subseteq T^{\rho(P)} \uparrow \omega(\emptyset) \quad \text{Def. 6 and Def. 9}
\]

\[
\iff A \in T^{\rho(P)} \uparrow \omega(\emptyset)
\]

\[
\square
\]
4 Implementation and Experiments

A tabled Prolog, incorporated with the flexible mode declaration, provides an easy implementation vehicle for programming with preferences. We have extended the TALS tabled Prolog [10] with the mode declaration scheme. Mode ‘+’ is used to denote an indexed argument, which is critical for variant checking during collecting answers into the table. Special modes such as ‘last’ and ‘<<<’ are provided to support user-defined preferences. No change is required on the tabled resolution to implement the mode declaration scheme; therefore, the same idea can also be applied to other tabled Prolog systems.

Two major changes to the global data structure table are needed to support mode declarations. First, each table predicate is associated with a new item mode, which is represented as a bit string in our system. The default mode for each argument in a table predicate is ‘-‘. Second, the answers to a tabled call are selectively recorded depending on its mode declaration. The declared modes essentially specify the user preferences or selection constraints among the answers. When a new answer to a tabled goal is generated, variant checking on indexed arguments is invoked to determine whether the answer is variant to a previously tabled one. If that is the case, declared modes on non-indexed arguments are used to select a better answer to table; otherwise, a new table entry is added to record the answer.

Our experimental benchmarks for preference logic programming include five typical optimization examples. matrix is the matrix-chain multiplication problem; lcs is longest common subsequence problem; obst finds an optimal binary search tree; apsp finds the shortest paths for all pairs of nodes; and knap is the knapsack problem. All tests were performed on an Intel Pentium 4 Mobile CPU 2.0GHz machine with 512M RAM running RedHat Linux 9.0.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Benchmark & matrix & lcs & obst & apsp & knap \\
\hline
without preferences & 12.78 (1.0) & 2.8 (1.0) & 48.39 (1.0) & 4.1 (1.0) & 29.35 (1.0) \\
with preferences & 9.68 (0.76) & 1.89 (0.68) & 33.95 (0.70) & 3.91 (0.95) & 30.06 (1.02) \\
\hline
\end{tabular}
\caption{Running time performance comparison: Seconds/(Ratio)}
\end{table}

Table 2 compares the running time performance between the programs with and without preferences. For the preference programs, the tabled system collects optimal answers implicitly by applying the predefined preferences; the programs without preferences adopt a traditional method – e.g., use the builtin predicate findall/3) – to collect all the possible answers explicitly and then locate the optimal one. The experimental data indicates, based on the running timings and their ratios in Table 2, that the programs with preference declaration are better than or comparable to those corresponding programs without preferences. The reasons are explained as follows.
The efficiency for preference programs are mainly credited to two factors. First, tabled Prolog systems with mode declaration provides a concise but easy-to-use interface for preference logic programming. The transformation procedure to incorporate the problem specification and preferences does not introduce any major overhead; the mode declaration are flexible and powerful to support user-defined preferences, and the mode functionality is implemented at the system level instead of the Prolog programming level. Second, tabled answers can be more efficiently organized due to the mode declaration. Indeed, if an indexed argument is instantiated in advance before a tabled goal is called, variant checking on this indexed argument can be avoided since its value is same for all the answers; furthermore, it is not necessary to record the pre-instantiated value with each tabled answer because the same value has already been stored in the tabled call entry. Those optimization leads to great running performance improvement.

Another important disadvantageous efficiency issue is the frequent retrieval or replacement of tabled answers. That is because the optimized answer is dynamically selected by comparing with old tabled answers according to the preferences. The retrieval of a tabled answer for comparison needs time overhead to locate each argument of the answer from the table. For replacing a tabled answer in the current TALS system, if a tabled subgoal only involves numerals as arguments, then the tabled answer will be completely replaced if necessary. If the arguments involve structures, however, then the answer will be updated by a link to the new answer. Space taken up by the old answer has to be recovered by garbage collection (the ALS Prolog’s garbage collector has not yet been extended by us to include table space garbage recovery). As a result, the retrieval and replacement of tabled answers causes performance overhead; the overhead will be minimal if the first tabled answer for each tabled call is optimal.

5 Conclusions

This paper provides a formal interpretation of preference logic programming: “Programming = Logic + Preferences + Control”. The programming is particularly suited to those constraint problems requiring optimization or comparison and selection among alternative solutions. It allows logic (specification of the problem) and preferences (the criteria specification of its solution selection) to be specified separately in a declarative fashion.

The declarative semantics of a preference logic program is defined as an intended model based on the Herbrand model theory. The preferences are essentially interpreted as a sequence of meta-level mapping operation over the least Herbrand model for the core program. We have shown that the intended model exists and is unique for a preference logic program.

This paper presents an elegant method of specifying and executing preference logic programs in the paradigm of a tabled Prolog. A preference logic program $P$ can be automatically transformed to a new tabled program by incorporating the general problem specification $P_{core}$ and the optimization criteria $P_{pref}$. The optimization criteria are elegantly embedded to filter the unoptimal answers.
Therefore, the procedural semantics of a preference logic program is reduced to that of a tabled logic program, which is equivalent to computing its least fixed point via a bottom-up computation strategy. We have shown that for a preference logic program, its declarative semantics is equivalent to its procedural semantics via the transformation to a tabled program.

A preference logic programming system has been successfully implemented in the TALS tabled Prolog system. No major changes are required to the Prolog engine and its tabled resolution scheme. The same idea can be easily applied to other tabled Prolog systems, since essentially only the variant checking operation during tabled answers collection needs to be modified due to the mode declaration.

References