5-1-2006

Logic Programming with Solution Preferences: A Declarative Method.

Miao Liu

Logic Programming with Solution Preferences: A Declarative Method

A thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Miao Liu

May 2006

Thesis Acceptance

Acceptance for the faculty of the Graduate College,

University of Nebraska, in partial fulfillment of the

requirements for the degree Master of Science,

University of Nebraska at Omaha.

Committee

Mahadevan Subramaniam

Haorong Li

[illegible]

Haifeng Guo

Chairperson

04-12-2006

Date

Abstract of the Thesis

Logic Programming with Solution Preferences: A Declarative Method

Miao Liu, Master of Science

University of Nebraska, 2006

Advisor: Professor Hai-feng Guo

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. This thesis presents a declarative method of specifying and executing preference logic programs based on a tabled Prolog system. 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 flexible mode declaration scheme is implemented in a tabled Prolog system, which provides an easy implementation vehicle for programming with preferences. Finally, experimental results and performance analysis demonstrate the effectiveness of the method.

Keywords: Preference Logic Programming, Tabled Prolog, Mode

to my parents

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First and foremost I wish to thank my advisor Hai-Feng Guo for his continuous support. Hai-Feng was always there to listen and to give advices. He taught me how to ask questions and express my ideas. Thank you for being there every step of the way.

The rest of my thesis committee: Zhengxin Chen, Haorong Li and Mahadevan Subramaniam for their encouragements, good questions, insightful comments and great suggestions.

Yinghua Zhu for her belief and support since day one. Without her school and work would not have been possible.

And my family for their understanding, unconditional support and encouragement to pursue my education. They have given up many things for me to become who I am today; for being there with me every single day of my life, thank you!

CHAPTER 1

Introduction

1.1 Overview

Traditionally when defining an optimization problem, we provide the objective function and specify the way of finding the optimal solutions. However the objective functions may be very difficult to define in problems such as ambiguity resolution. Moreover, since we have specified the way of finding the optimal solutions within the problem itself, the program is more difficult to maintain. An alternative approach is to declaratively define a program, and let the system effectively determine the optimal solutions for us. In ambiguity resolution problems for example, though the objective functions are hard to define, if we are given two potential solutions we know which one is more preferred than the other. We can describe such *solution preferences* using solution selection criteria and the system will determine the best solutions for us based on these criteria. This technique is an extension of constraint logic programming, and is referred to as *preference logic programming* or PLP [9, 10]. It 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 its optimization or selection criteria, and makes optimization or selection a meta-level operation.

There are two types of preferences related to logic programming. It is worthwhile to mention both of them here.

1. One type of preferences focuses on the constraints, which we refer to as *constraint preference* [4, 1]. Informally, it deals with problems where conflicts among the constraints occur. For example, one may have various constraints when buying a car, such as budget, safety, speed, etc. However, it is possible for the safety constraints to conflict with speed constraints. In such case, no car would satisfy all constraints and the customer has no car to select from. To resolve this problem, *constraint preference* is used to determine among these constraints which ones have higher preferences, and overrides others in order to resolve the problem. In our car buying example then, suppose safety constraints have the highest priority, the customer may make the ultimate decision based on safety concerns over speed. Thus, this type of preferences focuses on prioritizing *constraints.*

2. The other type of preferences focuses on the solutions obtained by the problem specification, and determines the optimum solutions based on user specified preference rules. This is referred to as *solution preference* problem.

For example, one have specified several routes from point a to point b, each with different costs. If the preference rule states that the route with the lowest cost is the most preferred, then the system should automatically determine the optimum solution(s) based on the cost of each route. Thus, this type of preferences focuses on the "ranking" of the *potential solutions.*

The term *preference logic programming* or simply PLP [9, 10] refers to the second type of preferences (i.e., solution preference), which is the focus of this thesis. On the other hand, since constraint preference does not focus on the same type of optimization problems we are interested in solving; it will not be further discussed. However, it is possible to combine both types of preference together since one does not conflict with another.

The PLP paradigm essentially separates the constraints of a problem itself from its optimization or selection criteria, and makes optimization or selection as meta-level operation. Preference logic programming has been shown useful in practical applications such as artificial intelligence [2], data mining [3], document processing and databases.

The original proposal of preference logic programming [9, 10] showed how the concept of preferences provides a natural, declarative, and efficient means of specifying a host of practical problems using definite clauses. Optimization predicates are explicitly defined by using special rules called optimization clauses, which makes these optimization predicates semantically independent from the clauses for the general problem. The declarative semantics of a preference logic program is given by its possible world semantics. To illustrate this, the following two examples show how PLP may be used in numeric and structural problem domains.

In Example 1, we are trying to search for a lowest-cost path, where predicate path(X,Y,C,D,L) denotes a path from X to Y with cost C, distance D and path route L.

Example 1 *PLP used in a numeric domain (best path problem).*

```
path(X, X, 0, 0, []).                                           (1)
path(X, Y, C, D, [e(X, Y)]) :-
    edge(X, Y, C, D).                                           (2)
path(X, Y, C, D, [e(X, Z) | P]) :-
    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),
    C is C1 + C2, D is D1 + D2.                                 (3)


edge(a,b,4,10).   edge(b,a,3,12).   edge(b,c,2,14).             (4)


path(X,Y,C1,D1,_) ≺ path(X,Y,C2,D2,_) :- C2 < C1.                (5)
path(X,Y,C1,D1,_) ≺ path(X,Y,C2,D2,_) :- C1 = C2, D2 < D1.       (6)
```

Clauses (1) to (4) make up the problem specification defining the path relation and a directed graph with a set of edges; clauses (5) and (6) states the solution selection criteria for optimizing the `path/5` predicate. Clause (5) can be interpreted as that a cheaper path is preferred, while clause (6) states that in case the cost of two paths are the same, then we take the shorter path. The symbol '$\prec$' is used as a preference symbol.

Now we've seen how PLP is beneficial to problems in numeric domain, the well-known *dangling else problem* is used to illustrate how PLP can be applied to problems in structural domain as well, as shown in Example 2.

Example 2 *PLP used in a structural domain (dangling else problem).*

```
ifstmt(if(C, T)) -->                                           (1)
   ['if'], cond(C), ['then'], stmt(T).


ifstmt(if(C, T, E)) -->                                        (2)
   ['if'], cond(C), ['then'], stmt(T), ['else'], stmt(E).

ifstmt(if(C, T, E)) ≺ ifstmt(if(C, NewT)) :-                   (3)
   combine(T, E, NewT).
```

Clauses (1) and (2) make up the problem specification defining the *if statement*; clause (3) specifies the solution selection criteria for this DCG by stating that if the *else* statement `E` can be combined into the *then* statement `T` to form a syntactically valid *new then* statement `NewT`, then the latter is a more preferred solution (a more detailed explanation is provided in Section 5.2 where typical PLP applications are discussed). Without clause (3), this syntax is ambiguous. Consider the following statement:

```
if (condA) then if (condB) then statementA else statementB;
```

Using the syntax specified by clauses (1) and (2) only, this statement can be interpreted either as:

```
1:   if (condA) {
        if (condB) statementA; else statementB;
     }
```

or as:

```
2:   if (condA) {
        if (condB) statementA;
     }
     else statementB;
```

To overcome this ambiguity issue, users may need to rewrite the entire grammar that the program was based on. As a result, many new terms may be introduced and the grammar may get very complicated. With PLP, on the other hand, a solution selection criterion (clause (3)) can be used to avoid ambiguity. In this case, clause (3) states that the "else-statement" is to be associated with the nearest conditional statement, i.e., interpretation 1 is more preferred than 2. With the combination of all three clauses shown in Example 2, the PLP system will parse the input stream using interpretation 1 automatically, while reporting interpretation 2 as false.

As we have seen from these two examples that the concept of PLP, which provides the separation of a problem definition and its selection criteria, really simplifies problem definitions. The following section presents what shall be addressed in this thesis in order to achieve it.

1.2 Problem Statement and Approach

The problem of this thesis is twofold: design of a method to efficiently support PLP, and implement the method into a Prolog system.

In order to support the above examples and preference programs in general, we propose an effective implementation for preference logic programming based on tabled programming. The method follows the PLP paradigm to separate the constraints of a problem itself from its optimization or selection criteria. Each preference logic program is defined using two disjoint sets of definite clauses, where one contains the specification of the constraints of the problem and the other defines the optimization or selection criteria. Connection between these two sets are established by a mode declaration scheme [12], which embeds optimization or selection criteria into the problem specification. Using this scheme, we do not need to define the value of an optimal solution recursively; but the value of a general solution suffices. The proposed tool is suitable for both numeric and structural preference problems.

The implementation of the PLP system is realized using C programming language on an UNIX environment. The system is a modification and extension of the tabled Prolog system presented in reference [11]. Although still based on *dynamic reordering of alternatives* (DRA), a different table management is implemented to support problems with multiple optimal solutions as well as argument reordering. These modifications will be discussed in detail in later chapters.

1.3 Significance of This Work

The proposed preference logic program implementation is the first complete PLP system constructed. It extends the tabled Prolog systems with mode-declaration scheme to support both numeric and structural solution preferences. New functions, such as removing a list of tabled solutions, are designed to improve tabling performance as well as the overall efficiency. A method of specifying and executing preference logic programming and preference logic grammars based on tabled Prolog systems is the main contribution of this work.

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 gives a brief introduction to the Prolog system, and a literature review of some relevant research works. Chapter 3 presents the declarative semantics of the PLP. The procedural semantics and the connection between solution selection criteria and the problem specification are discussed in Chapter 4. Then Chapter 5 discusses some of the typical applications using PLP. The implementation of the PLP system and experimental results are presented in Chapters 6 and 7, respectively. And finally, the conclusion and discussion on future extensions are given in Chapter 8 and 9.

CHAPTER 2

Background and Related Work

This chapter presents some background information regarding logic programming and Prolog, followed by a brief introduction to the *Warren Abstract Machine* (WAM), a virtual abstract machine that is normally used for implementing Prolog. Some major extensions to the traditional Prolog system, such as *tabled* Prolog and *PLP*, and some of the recent studies in these areas are also discussed in this chapter.

2.1 Prolog Systems

The name *Prolog* is derived from "***Pro***gramming in ***log***ic". This section briefly introduces the language of Prolog and how Prolog programs are evaluated.

2.1.1 Basics of the Prolog Language

A Prolog program is composed of a set of *Horn clauses.* Using Prolog's notation, each clause is a formula of the form

$$Head\texttt{:-}B_1, B_2, \ldots, B_n$$

where *Head* and $B_1, \ldots, B_n$ are atomic formulae and $n \geq 0$.[1] Each clause represents a logical implication of the form

$$\forall(B_1 \land \cdots \land B_n \rightarrow Head)$$

A separate type of clauses are those where *Head* is the atom `false`, which are simply written as

$$\texttt{:-}B_1, \ldots, B_n$$

These type of clauses are called *goals* or *queries.* Each atom in a goal is called a *subgoal.*

Each atomic formula is composed of a predicate applied to a number of arguments (also refered to as terms), and this will be denoted as $p(t_1, \ldots, t_n)$ – where p is the predicate name, and $t_1, \ldots, t_n$ are the terms used as arguments. Each term can be either a constant (c), a variable (X), or a complex term, such as $f(s_1, \ldots, s_m)$, where $s_1, \ldots, s_m$ are themselves terms and f is the *functor* of the term.

[1] If $n = 0$ then the formula is simply written as *Head* and called *fact.*

Execution of a Prolog program typically involves a program P and a goal $:\!-G_1, \ldots, G_n$. The objective is to verify whether there exists an assignment σ of terms to the variables in the goal such that $(G_1 \wedge \cdots \wedge G_n)\sigma$ is a logical consequence of P.[2] σ is called a *substitution*: a substitution is an assignments of terms to a set of variables (the *domain* of the substitution). If a variable X is assigned a term t by a substitution, then X is said to be *bound* and t is the (run-time) binding for the variable X. The process of assigning values to the variables in t according to a substitution σ is called *binding application.*

2.1.2 Execution of a Prolog Program

Prolog, as well as many other logic programming systems, make use of *SLD-resolution* to carry out program's execution. Execution of a program P w.r.t. a goal G proceeds by transforming a *resolvent* using a sequence of *resolution steps.* Each resolvent represents a conjunction of subgoals. The initial resolvent corresponds to the goal G. Each resolution step proceeds as follows:

1. Let us assume that $:\!-A_1, \ldots, A_k$ is the current resolvent. An element A_i of the resolvent is selected (*selected subgoal*) according to a predefined *computation rule.* In the case of Prolog, the computation rule selects the leftmost element of the resolvent.

[2]Following standard practice, the notation $e\sigma$ denotes the application of the substitution σ to the expression e, i.e., each variable X in e will be replaced by $\sigma(X)$.

2. If A_i is the selected subgoal, then the program is searched for a clause $Head\text{:-}B_1, \ldots, B_h$ whose head successfully unifies with A_i. Unification is the process which determines the existence of a substitution σ such that $Head\sigma = A_i\sigma$. If there are rules satisfying this property then one is selected (according to a *selection rule*) and a new resolvent is computed by replacing A_i with the body of the rule and properly instantiating the variables in the resolvent:

$$\text{:-}(A_1, \ldots, A_{i-1}, B_1, \ldots, B_h, A_{i+1}, \ldots, A_k)\sigma$$

   In the case of Prolog, the clause selected is the first one in the program whose head unifies with the selected subgoal.

3. If no clause satisfies the above property, then a failure occurs. Failures are cured using *backtracking*. Backtracking explores alternative execution paths by reducing one of the preceding resolvents with a different clause.

4. The computation stops either when a solution is determined (i.e., the resolvent contains zero subgoals) or when all alternatives have been explored without any success.

The operational semantics of a logic based language is determined by the choice of computation rule (selection of the subgoal in the resolvent) and the choice of selection rule (selection of the clause to compute the new resolvent). In the case of Prolog, the computation rule selects the leftmost subgoal in the resolvent, while the selection rule selects the first clause in the program which successfully unifies with the selected subgoal.

2.1.3 Extra-logical Predicates

Many logic languages (e.g., Prolog) introduce a number of *extra-logical predicates*, used to perform tasks such as:

1. input/output (e.g., read and write files);
2. add a limited form of control to the execution (e.g., the cut (`!`) operator, used to remove some unexplored alternatives from the computation);
3. perform meta-programming operations; these are used to modify the structure of the program (e.g., `assert` and `retract`, which adds or removes clauses from the program, respectively), or query the status of the execution (e.g., `var` and `nonvar`, used to test the binding status of a variable).

An important aspect of many of these extra-logical predicates is that their behavior is *order-sensitive*, meaning that they can produce a different outcome depending on when they are executed. In particular, this means that they can potentially produce a different result if a different selection rule or a different computation rule is adopted.

Since most of Prolog implementations are based on the WAM, In addition, the PLP system implemented in this thesis is realized by extending a WAM-based

Prolog system. Therefore, a brief introduction to the WAM is given in the next section.

2.2 Implementation of Prolog and WAM

Implementations of Prolog follow the familiar activation record model that is traditionally used for implementing standard programming languages such as C, Java, etc. Thus, most Prolog's implementations have a stack where an activation record (called an environment, in Prolog's parlance) is allocated, a heap where dynamic data resides, plus other standard memory areas (code space, registers, etc.). Because Prolog uses unification to instantiate parameters during a call, parameter passing is a little more involved since the unification algorithm has to be invoked to pass parameters. Likewise, because Prolog has backtracking, two additional modifications are needed: First, a new data-structure called a *choice point* has to be allocated on the environment stack where information about the state that existed at that point is recorded, so that this state can be restored if execution needs to backtrack to this point. The choice-points correspond to branch points in the search tree that a Prolog execution builds and traverses in a depth-first manner. Second, a new stack called the trail stack is used to store the addresses of all the variables that need to be reset to unbound status upon backtracking; the trail stack stores those variables that are created before branching takes place in the search tree but are bound to a possibly different value in each of the branches.

Thus, most Prolog implementations compile Prolog programs to an abstract machine, called the *Warren Abstract Machine (WAM)* [20, 21]. The WAM, which has become a de-facto standard for sequential implementations of Prolog and Logic Programming languages, defines an abstract architecture whose instruction set is designed to:

1. allow an easy mapping from Prolog source code to WAM instructions;
2. be sufficiently low-level to allow an efficient emulation and/or translation to native machine code.

Most (sequential and parallel) implementations of Prolog currently rely either directly on the WAM, or an architecture similar to the WAM.

The WAM is a stack-based architecture, sharing some similarities with imperative languages implementation schemes (e.g., use of `call/return` instructions, use of frames for maintaining procedure's local environment), but extended in order to support the features peculiar to Logic Programming, namely *unification* and *backtracking* (and some other variations, like the need to support dynamic type checking). At any instance, the state of the machine is defined by the content of its memory areas (illustrated in Fig. 2.1). The state can be subdivided into *internal* and *external* state.

1. *Internal State:* it is described by the content of the machine registers. The

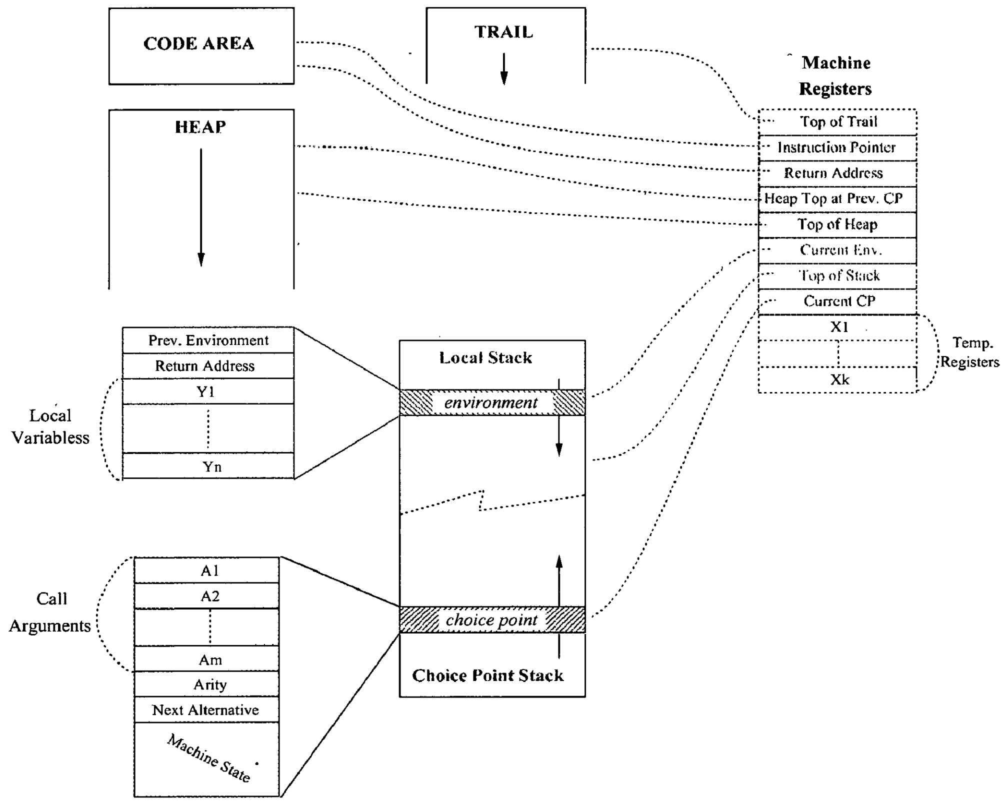


Figure 2.1: Organization of the WAM

purpose of most of the registers is described in Fig. 2.1.

2. *External State:* it is described by the content of the logical data areas of the machine:

   (a) *Heap:* data areas in which complex data structures (lists and Prolog's compound terms) are allocated.

   (b) *Local Stack:* (also known as *Control Stack* or *Environment Stack*) it serves the same purpose as the control stack in the implementation of imperative languages – it contains control frames, called *environments* (akin to the activation records used in implementation of imperative languages), which are created upon entering a new clause (i.e., a new "procedure") and are used to store the local variables of the clause and the control information required for "returning" from the clause.

(c) *Choice Point Stack:* choice points encapsulate the execution state for backtracking purposes. A choice point is created whenever a call having multiple possible solution paths (i.e., more than one clause successfully match the call) is encountered. Each choice point should contain sufficient information to restore the status of the execution at the time of creation of the choice point, and should keep track of the remaining unexplored alternatives. In some implementations, the choice-points are also placed in the local stack, and so there is no separate choice-point stack.

(d) *Trail Stack:* during an execution variables can be instantiated (they can receive bindings). Nevertheless, during backtracking these bindings need to be undone, to restore the previous state of execution. In order to make this possible, bindings that can be affected by this operation are registered in the trail stack. Each choice point records the point of the trail where the undoing activity needs to stop.

Prolog is a dynamically typed language; hence it requires type information to be associated with each data object. In the WAM, Prolog terms are represented as *tagged words*: each word contains:

1. a *tag* describing the type of the term (atom, number, list, compound structure, unbound variable);

2. a *value* whose interpretation depends on the tag of the word; e.g., if the tag indicates that the word represents a list, then the value field will be a pointer to the first node of the list.[3]

Prolog programs are compiled in the WAM into a series of abstract instructions operating on the previously described memory areas. In a typical execution, whenever a new subgoal is selected (i.e., a new "procedure call" is performed), the following steps are taken:

1. The arguments of the call are prepared and loaded into the temporary registers $X_1, \ldots, X_n$ – the instruction set contains a family of instructions, the "`put`" instructions, for this purpose.

2. The clauses matching the subgoal are detected and, if more than one is available, a choice point is allocated (using the "`try`" instructions);

3. The first clause is started: after creating (if needed) the environment for the clause ("`allocate`"), the execution requires *head unification* – i.e., unification between the head of the clause and the subgoal to be solved – to be performed (using "`get/unify`" instructions). If head unification is successful (and assuming that the rule contains some user-defined subgoals),

[3]Lists in Prolog, as in Lisp, are composed of *nodes*, where each node contains a pointer to an element of the list (the *head*) and a pointer to the rest of the list (the *tail*).

then the body of the clause is executed, otherwise backtracking to the last choice point created takes place.

4. Backtracking involves extracting a new alternative from the choice point that is topmost on the stack ("`retry`" will extract the next alternative, assuming this is not the last one, while "`trust`" will extract the last alternative and remove the exhausted choice point), restoring the state of execution associated with such choice point (in particular, the content of the topmost part of the trail stack is used to remove bindings performed after the creation of the choice point), and restarting the execution with the new alternative.

After the basic introduction to Prolog and its implementation, we now discuss some extensions to the traditional Prolog system and related research works.

2.3 Extensions to the Traditional Prolog System

The system presented in this thesis utilizes and optimizes some extensions of the traditional Prolog system, namely *tabled* Prolog and *PLP*. Hence some background information regarding these extensions is presented in this section, followed by relevant research works in preference logic programming.

2.3.1 Tabled Prolog

One of a major extension to the traditional Prolog system is the tabled Prolog. As discussed in Section 2.1, traditional Prolog systems use SLD resolution [14] with the computation strategy that 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 [23, 24, 11, 19] 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.

The tabled resolution employed in this thesis is named *dynamic reordering of alternatives* (DRA) [11]. Other tabled resolutions, such as SLG [23], SLDT [24], etc. perform similarly toward the computation of a fixed point. The DRA resolution computes a fixed point in a very similar way as bottom-up execution of logic programs [14]. Its main idea is to dynamically identify *looping alternatives* from the program clauses, and then repetitively apply those alternatives until no more answers can be found. A looping alternative refers to a clause that matches a tabled call and will lead to a resolvent containing a recursive variant call.

Example 3 is used to illustrate the effectiveness of tabling. This program does not work properly in a traditional Prolog system due to left-recursion defined in clause (2). With the declaration of a tabled predicate `reach/2` in a tabled Prolog system, it can successfully find a set of complete solutions due to the fixed-point computation strategy, which is illustrated in Figure 2.2.

Example 3 *A tabled logic program defining a reachability relation:*

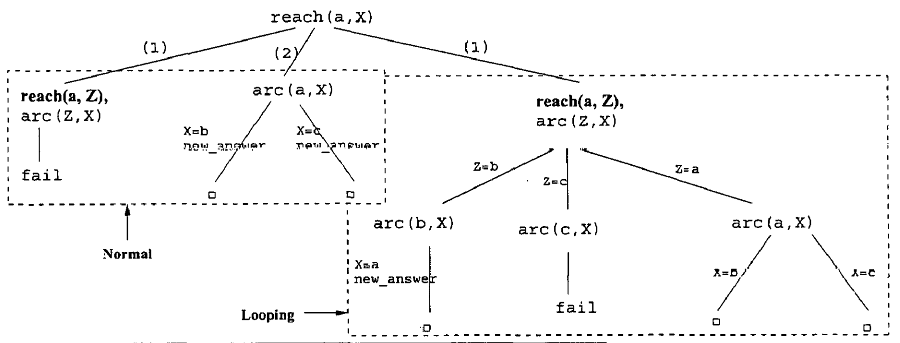
```
:- table reach/2.                                     (1)

reach(X,Y) :- reach(X,Z), arc(Z,Y).                   (2)

reach(X,Y) :- arc(X,Y).                               (3)


arc(a,b).     arc(a,c).     arc(b,a).                 (4)


:- reach(a,X).                                        (5)
```



Tabled Subgoals	Answers	Looping Alternatives

reach(a, X)	reach(a, b)
	reach(a, c)
	reach(a, a)
		(1)

Figure 2.2: DRA Resolution for Example 3

As shown in Figure 2.2, the computation of `reach(a,X)` is divided into three stages: *normal*, *looping* and *complete*. The purpose of the normal stage is to find all the looping alternatives (clause (2) leading to a variant subgoal `reach(a,Z)`) and record all the answers generated from the non-looping alternatives (clause (3)) into the table. The `new_answer` label indicates that the new answer generated from that successful path should be added into the table. Then, in the looping stage only the looping alternative (clause (2)) is performed repeatedly to consume new tabled answers until a fixed point is reached, that is, no more answers for `reach(a,X)` can be found. Afterwards, the complete stage is reached. As a result, the query `:- reach(a,X)` returns a complete answer set `X=b`, `X=c` and `X=a`, albeit the predicate is defined left-recursively.

2.3.2 Mode Declaration for Tabled Predicates

Although tabled Prolog systems can successfully determine the solution set for Example 3, just memorizing the answers in the table still has some potential problems. Such problem could happen when a computing model contains infinite number of solutions. Consider Example 4, which searches the paths for reachable nodes and records the paths. Notice that the paths information is infinite, since there are infinite number of paths from a to any node due to the cycle between a and b. Suppose that we are only interested in finding one possible path from one node to another reachable node, hence only one path information is required to be recorded in the table for each pair of reachable nodes. 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. This meta-level operation is also useful to narrow a big finite-size solution set down to a smaller one. An example is to filter out from the general solution set the suboptimal ones in order to derive our optimal solutions.

Example 4 *A tabled logic program defining a reachability relation with path information:*

```
:- table reach/3.
reach(X,Y,E) :- reach(X,Z,E1), arc(Z,Y,E2), append(E1,E2,E).
reach(X,Y,E) :- arc(X,Y,E).
```

```
arc(a,b,[(a,b)]).      arc(a,c,[(a,c)]).      arc(b,a,[(b,a)]).

:- reach(a,X,E).
```

This meta-level operation can be achieved by a mode declaration [12] for tabled predicate, which is described in the form of

$$\texttt{:- table } q(m_1, ..., m_n).$$

where q/n is a tabled predicate name, $n \geq 0$, and each m_i has one of the forms as defined in Table 2.1.

Modes	Informal Semantics
$+$	an indexed argument
$-$	a non-indexed argument

Table 2.1: Original Built-in Modes for Tabled Predicates

The mode declaration [12] 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 newly generated answers into the table. For a 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 program in Example 4. Suppose we declare the mode as "`:- table reach(+,+,-)`"; this means that only the first two arguments of the predicate `reach/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.

While the original mode declaration solves the problem presented above, there are two limitations associated with it. First, the terms with mode '+' are required to appear prior to the terms with mode '-' (e.g. "`:- table reach(+,-,+)`" would not be supported). This is due to the way the table data structure is managed. This limitation is addressed in the thesis by implementing a new management system in order to support flexible ordering of mode declarations. Second, the available built-in modes are limited to either indexed or non-indexed. This thesis extends the mode directive **`table`** to associate a non-indexed argument of a tabled predicate with some optimum constraint. User defined preference symbols, such as '<<<' used throughout this thesis, may be used to support user-defined preferences. The types of preferences are flexible, varying from numerical minimization/maximization to structural comparisons. The tabled data structure and its management, as well as the extended modes are discussed in detail in Chapter 6, where specific implementation issues are addressed.

2.3.3 Preference Logic Programming

Another important extension on the traditional Prolog systems is PLP, which was first proposed in [9, 10] as a new programming paradigm even though preference logics has been studied back to early 1960's. However, most of the works carried out were not implementation-oriented, and an effective PLP system was lacking.

Some of the early works such as [8] focused solely on numeric values and minimum/maximum based on the problem specifications. Cui and Swift [3] extended part of earlier work on preference logic grammars [15] to a three-value logic, and provided an implementation for specific applications on data standardization via normal logic programs using XSB [23]. Other efforts such as [16, 8, 22] have been undertaken to incorporate optimization in a Constraint Logic Programming (CLP) framework; and [6, 17] addresses semantics for optimization predicates in a CLP framework.

Reference [7] proposed an approach for describing preferred criteria in CLP as a problem of relational optimization. This approach allows the users to define a preference relation which indicates when a solution is better than another solution. However, the limitation of this approach is that its operational semantics for the optimization require the underlying structures to be strict total orders. Therefore, it would not work if the problem contains potential solutions that are not comparable with each other.

Reference [13] gives 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 Herbrand model [5, 14] and fixed-point theory; the semantics of preferences is defined as a strict partial order relation[4] among solutions. With this mode declaration scheme, some problems associated with a simple tabled Prolog system aforementioned could be resolved.

The PLP system presented in this thesis further extends the approach presented in [13]. Some of the major extensions and modifications are highlighted as follows:

1. *The connection between problem specification and preference criteria is simplified.* The original connection was established using transformation, which essentially transforms a user defined program P into P', and automatically create a new predicated to realize the connection. This transformation procedure is not needed in the new system.

2. *The requirement of a strict partial order relation among solutions is relaxed.* E.g. one could specify as a preference criteria that `solution1` $\prec$ `solution2` and `solution2` $\prec$ `solution1`. The PLP system, during evaluation, will determine that neither `solution1` nor `solution2` is an optimal solution among the potential solutions to the problem specificatoin, and report accordingly back to the user.

[4]An ordering which is irreflexive, antisymmetric and transitive.

We are, however, more interested in the problems with solutions forming partial order relations.

3. *A new table management system is designed and implemented.* This new system supports not only `insert` (which was supported in the original system), but also `remove`, which is crucial for maintaining user defined preference relations among the tabled solutions.

4. And as mentioned in Section 2.3.2, more built-in modes are supported. More importantly, *argument reordering based on mode declarations* is realized.

The necessities and realizations of these extensions and modifications are discussed in detail in the following chapters. Prior to this discussion, we first present the declarative and procedure semantics of the PLP.

CHAPTER 3

Declarative Semantics

This chapter defines the syntax and declarative semantics of PLP. A user defined preference symbol will be used in the actual preference programs to replace '$\prec$'. The symbol — '<<<' — is the choice of this thesis. Provided with a user defined symbol, we now formally define the syntax and semantics of PLP.

3.1 Syntax

There are two components of an optimization problem: (i) specification of the constraints in the problem; and, (ii) specification of what the optimal solution is and how it can be selected. The main idea of preference logic programing is to separate these two components and declaratively specify such applications.

We have already discussed a meta-level operation using mode declaration, which is of the following form:

$$\texttt{:- table } q(m_1, ..., m_n)\texttt{.}$$

where q/n is a tabled predicate name, $n \geq 0$, and each m_i has one of the forms as defined in Table 2.1. Let $m_{i1}, m_{i2}, \cdots, m_{ik}$ $(0 \leq k \leq n)$ be all the modes '+' such that $1 \leq i1 < i2 < \cdots < ik \leq n$; We define one operator $\mathcal{K}$ as follows: given an arbitrary atom $q(a_1, a_2, \cdots, a_n)$, $\mathcal{K}(q(a_1, a_2, \cdots, a_n)) = (a_{i1}, a_{i2}, \cdots, a_{ik})$, which is a sequence of indexed arguments in a left-to-right order. Now we can formally define Preference Logic Programs.

Definition 1 (Preference Logic Programs)

A (definite) preference logic program P can be defined as a pair <P_{core}, P_{pref}>, where P_{core} and P_{pref} are two disjoint sets of clauses defined as follows: P_{core} specifies the constraints of the problem as a set of definite clauses; P_{pref} defines the optimization criteria using a set of preference clauses (or preferences) of the form:

$$p(T_1) <<< p(T_2) \text{ :- } B_1, B_2, ..., B_n. \qquad (n \geq 0)$$

where $\mathcal{K}(p(T_1)) = \mathcal{K}(p(T_2))$ and each B_i $(1 \leq i \leq n)$ is an atom defined in P. p is referred to as an ***optimization predicate****.*

The informal semantics of $p(T_1)$ <<< $p(T_2)$:- B_1, B_2, ..., B_n is that the atom $p(T_1)$ is less preferred than $p(T_2)$ if B_1, B_2, ..., and B_n are all true. Note that the two atoms being compared have the same predicate symbol p. Also, $\mathcal{K}(p(T_1)) = \mathcal{K}(p(T_2))$ states that only two atoms with the same corresponding indexed arguments (mode '+') are comparable. We abbreviate "preference logic program" to "preference program" and "tabled prolog program" to "tabled program" throughout.

Example 5 *The following is the preference program searching for a lowest-cost path. It corresponds to Example 1 presented earlier.*

```
path(X, X, 0, 0, []).                                        (1)
path(X, Y, C, D, [e(X, Y)]) :-
    edge(X, Y, C, D).                                        (2)
path(X, Y, C, D, [e(X, Z) | P]) :-
    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),
    C is C1 + C2, D is D1 + D2.                              (3)


edge(a,b,4,10).   edge(b,a,3,12).   edge(b,c,2,14).          (4)


:- table path(+, +, <<<, <<<, -).                            (5)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C2 < C1.                                                 (6)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C1 = C2, D2 < D1.                                        (7)
```

Clauses (1) to (4) make up the core program P_{core} defining the path relation and a directed graph with a set of edges; clauses (5) to (7), the preference clauses P_{pref}, specify the predicate path/5 that is to be optimized and give the criteria for optimizing the path/5 predicate. That is, the path for each pair of reachable nodes (according to the first two indexed arguments in path/5) should be optimized based on the definition of <<<: the shorter path is more preferred.

Example 5 shows an optimization problem with compound objectives. This type of problems are difficult to solve directly using traditional constraint programming with objective functions. Instead, a two-step selection procedure is usually involved, where, first, only the cost criterion is used to find all the lowest-cost paths, and secondly the optimal path is selected by comparing distances among the lowest-cost paths. However, the preference logic program, as shown in Example 5, is intended to specify and solve this problem directly in a declarative method. That is, it separates the constraints of a problem itself from the criteria for selecting the optimal solutions. 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 paradigm.

3.2 Semantics

The semantics is being provided in order to facilitate a proof of correctness of the implementation. The declarative semantics of a preference program is based on the Herbrand model theory [5, 14]. The preferences are essentially interpreted as a sequence of meta-level mapping operations over the least Herbrand model for the core program. We use the following notational conventions: P is used to denote a preference logic program $<P_{core}, P_{pref}>$, B_P to denote the Herbrand base of P, B^P_{core} to denote the Herbrand base of P_{core}, $2^{B^P_{core}}$ to denote the set of all Herbrand interpretations of P_{core}, a Herbrand atom to denote an atom in B^P_{core},

ω is the first infinite ordinal, and $\mathcal{F} \uparrow n(x)$ to denote applying the mapping $\mathcal{F}$ n times as $\overbrace{\mathcal{F}(\mathcal{F}(\cdots \mathcal{F}}^{n}(x)\cdots))$.

Definition 2 (A Preference Relation) *Let P be a preference program and q/n be an optimization predicate. A* preference relation *over q/n is an ordered relation $\prec_{(q/n)}$ s.t. for any two Herbrand atoms A_1 and A_2 of the predicate q/n, $A_1 \prec_{(q/n)} A_2$ if either of the followings is true:*

- $A_1 <\!\!<\!\!< A_2 \in \psi_P \uparrow \omega(M^P_{core})$
- *$\exists$ an atom A_3 s.t. $A_1 \prec_{(q/n)} A_3$ and $A_3 \prec_{(q/n)} A_2$.*

where M^P_{core} is the least Herbrand model for P_{core}, and $\psi_P : 2^{B_P} \longrightarrow 2^{B_P}$ is defined as follows: $\psi_P(M) = M \;\cup\; \{A_1 <\!\!<\!\!< A_2 :\; A_1 <\!\!<\!\!< A_2$:- $B_1, \cdots, B_n$ is a ground instance of a clause inP and $\{B_1, \cdots, B_n\} \subseteq M\}$.

We abbreviate the preference relation $\prec_{(q/n)}$ to $\prec$ whenever the optimization predicate is obvious from the context. For instance, consider the Example 5. Its preference relation $\prec$ is the set

$$\begin{aligned}
\{ \quad & \text{path}(a,a,7,22,\_) \prec \text{path}(a,a,0,0\_), \\
& \text{path}(a,a,14,44,\_) \prec \text{path}(a,a,0,0,\_), \\
& \text{path}(a,a,14,44,\_) \prec \text{path}(a,a,7,22,\_), \quad \cdots \\
& \text{path}(a,b,11,32,\_) \prec \text{path}(a,b,4,10,\_), \\
& \text{path}(a,b,18,54,\_) \prec \text{path}(a,b,4,10,\_), \\
& \text{path}(a,b,18,54,\_) \prec \text{path}(a,b,11,32,\_), \quad \cdots \\
& \cdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}$$

where the numbers $0, 7, 11, ...$ are the possible costs, $0, 22, 44, ...$ are the possible distances, and '_' means any ground term from the Herbrand universe.

Definition 3 (Model and Intended Model) *Let P be a preference program, and I be an interpretation for P_{core}. We say I is a* model *for P if for any optimization predicate q/n in P, we have*

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. $\{B_1, ..., B_n\} \subseteq I$;*

b). *for any two atoms A_1 and A_2 of q/n in I, neither $A_1 \prec A_2$ nor $A_2 \prec A_1$ 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. $A \prec A_1$,* *then A_1 is not in the least Herbrand model for P_{core}. We call A* an optimized atom *in I.*

Note that the model for P_{core} follows the standard model definition [14] for definite clauses, which is different from Def. 3 for a preference program P. Def. 3(c) says that no better-preferred atom A_1 than the optimized atom A can be found in the least Herbrand model of P_{core}, otherwise, A cannot be an optimized atom. However, there may exist a Herbrand atom $A_1 \notin P_{core}$ that is better-preferred than A.

We wish to obtain the link between the models of P and P_{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 4 *Let P be a preference program, M be a Herbrand model for P_{core}, and M_1 be a subset of M containing all the atoms of any optimization predicate. We define a meta-level mapping $\phi_P : 2^{B^P_{core}} \longrightarrow 2^{B^P_{core}}$ as follows:*

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

Definition 5 *Let P be a preference program. We define a meta-level mapping $\pi_P : 2^{B^P_{core}} \longrightarrow 2^{B^P_{core}}$ as follows: $\pi_P(M) = \{A \in M : A \text{ :- } A_1, \cdots, A_n$ is a ground instance of a clause in P_{core} 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 ϕ_P filters suboptimal atoms from the model according to the preference relation; the mapping π_P filters those atoms depending on the removed suboptimal atoms from the model. It is obvious that $\pi_P(I) \subseteq I$ for any given Herbrand interpretation I. Thus, we come to a major result of the theory as shown in the next theorem.

Theorem 1 *Let P be a preference program and M^P_{core} be the least Herbrand model for P_{core}. Then*

$$M_P = \pi_P \uparrow \omega(\phi_P(M^P_{core}))$$

is an intended model for P.

Proof: We show how M_P satisfies the properties (a), (b), and (c) as defined in the Def. 3 for an intended model:

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

2). We associate a complete lattice with the program P. $2^{B^P_{core}}$, the set of all Herbrand interpretations of P_{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, $M_P = \pi_P \uparrow \omega(\phi_P(M^P_{core}))$ must be a fixed point of π_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 an optimization 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. $A \prec A_1$. According to the definition of ϕ_P in Def. 4, $A \notin \phi_P(M^P_{core})$, 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. $\square$

If we reconsider the preference program in the Example 5. Its least Herbrand model M^P_{core} and $\phi_P(M^P_{core})$ are shown below.

M^P_{core} ={

$\text{edge}(a, b, 4, 10), \text{edge}(b, a, 3, 12), \text{edge}(b, c, 2, 14),$

$\text{path}(a, a, 0, 0, []), \text{path}(a, a, 7, 22, [(a, b), (b, a)]), \cdots$

$\text{path}(a, b, 4, 10, [(a, b)]), \text{path}(a, b, 11, 32, [(a, b), (b, a), (a, b)]),$

$\dots \quad \dots$

$\text{path}(c, c, 0, 0, [])$

}

$\phi_P(M^P_{core})$ = {

$\text{edge}(a, b, 4, 10), \text{edge}(b, a, 3, 12), \text{edge}(b, c, 2, 14),$

$\text{path}(a, a, 0, 0, []), \text{path}(a, b, 4, , 10, [(a, b)]),$

$\text{path}(a, c, 6, 24, [(a, b), (b, c)]), \text{path}(b, a, 3, 12, [(b, a)]),$

$\text{path}(b, b, 0, 0, []), \text{path}(b, c, 2, 14, [(b, c)]),$

$\text{path}(c, c, 0, 0, [])$

```
    }
```

We also have $\pi_P \uparrow \omega(\phi_P(M^P_{core})) = \phi_P(M^P_{core})$ for this program. However, if we add an extra clause

```
shortest(X,Y,C,D,P) :- path(X,Y,C,D,P).,
```

then $\pi_P \uparrow \omega(\phi_P(M^P_{core}))$ is different from $\phi_P(M^P_{core})$. e.g.,

shortest$(a, a, 7, 22, [(a, b), (b, a)]) \in \phi_P(M^P_{core})$, but

shortest$(a, a, 7, 22, [(a, b), (b, a)]) \notin \pi_P \uparrow \omega(\phi_P(M^P_{core}))$.

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_{core}, the least Herbrand model for P_{core}. For each definite logic program, M^P_{core} exists and is unique [5, 14]. The proof is therefore completed. □

Example 6 *Consider the following preference program with contradictory preferences:*

```
q(a).

q(b).

:- table q(<<<).

q(a) <<< q(b).

q(b) <<< q(a).
```

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 an 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: This is shown based on the Def. 3(c) since M_P is an intended model for P. □

CHAPTER 4

Procedural Semantics

A preference logic program cannot be executed directly on any existing tabled Prolog system. The main reason is that the mode-controlled table manipulation is implemented at the system level, whereas preferences are defined at the Prolog programming level. Therefore the optimization criteria P_{pref} needs to be embedded into the general problem specification P_{core} during execution. This embedding procedure needs to be carefully designed to ensure that the procedural semantics is consistent with the declarative semantics of the preference program.

This chapter first discusses the previous embedding procedure and how it is improved in the new approach. Then we illustrate how it is realized in the system followed by some preference program examples. Finally correctness of the new approach is proven.

4.1 Embedding Approaches

In this section we first briefly discuss the cmbedding approach employed in [13]. The reasons for improvement are identified, and a new approach is designed to address these shortcomings. An overview of the new approach is then followed.

4.1.1 Previous Approach and Its Shortcomings

In the previous approach [13], the embedding is achieved using *transformation.* The idea is to take the user defined program P as its input, and introduces (inserts) a *new* predicate as a wrapper to form the transformation output P'. This procedure is illustrated using the following example. Consider the transformed tabled program of the program in Example 5.

Example 7 *Previous approach: transformed program of Example 5.*

```
%%% RENAMED PREDICATE %%%

pathNew(X, X, 0, 0, []).                                      (1)

pathNew(X, Y, C, D, [e(X, Y)]) :-

    edge(X, Y, C, D).                                         (2)

pathNew(X, Y, C, D, [e(X, Z) | P]) :-

    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),

    C is C1 + C2, D is D1 + D2.                               (3)


edge(a,b,4,10).   edge(b,a,3,12).   edge(b,c,2,14).           (4)
```

```
% mode 'last' will be explained below %

:- table path(+, +, last, last, -).                              (5)

path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :- C2 < C1.              (6)

path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-                       (7)

    C1 = C2, D2 < D1.


%%% NEW PREDICATE %%%

path(X, Y, C, D, P) :-                                           (8)

    pathNew(X, Y, C, D, P),

    (path(X, Y, C1, D1, P1)

        -> path(X, Y, C1, D1, P1) <<< path(X, Y, C, D, P)

        ;  true

    ).
```

The differences between P and P' are: (i) The original predicate path/5 in Example 5 is replaced by a new predicate pathNew/5 to emphasize that this predicate generates a new preferred path candidate from X to Y. (ii) The predicate path/5, given a new definition in clause (8), represents the way for identifying a preferred answer. The meaning of clause (8) is the following: given a path candidate A by pathNew(X,Y,C,D,P), we need to check whether there already exists a tabled answer, if so, they are compared with each other to keep the preferred one in the table; otherwise, the candidate is recorded as a first tabled answer. The mode '`last`' declared in clause (5) is used to record the last answer (of the corresponding argument) from a solution set. This procedure ensures that the optimal solution is the last one left in the table after all the computation and comparisons are done. Mode '`last`' however is obsolete and is not necessary in the new system.

Although this transformation procedure is effective and can be fully automated, there are some shortcomings that can be improved. Firstly, the transformation alters the original program dramatically. Not only a new predicate declared in clause (8) is introduced, but also the existing predicates are being renamed. This process and the transformed program could become very complex depending on the optimization criteria involved. Secondly, this approach only supports one optimal solution. Hence it requires a strict partial ordering among the potential solutions. To address these issues, a new approach is introduced.

4.1.2 Overview of the New Approach

The approach taken in this thesis addresses the embedding issue differently than the previous transformation. A prolog level predicate `updateTable(QueryName, Criteria)` is introduced to manage the tabled solutions. It is invoked every time a new solution is obtained. Essentially it determines amongst the obtained *new solution* (referred to as `CurrentSolution`) and the solutions reside in the table, which ones shall be kept and discards the rest. There are two arguments to the predicate `updateTable/2`: `QueryName` specifies the name of the query that it needs to update; and `Criteria` specifies the optimization criteria to be used (e.g. `'<<<'`). The procedure starts by retrieving all of the tabled solutions for the specified query. Then it uses the optimization criteria to compare the `CurrentSolution` against each of the tabled solution following the pseudo-code presented below:

```
foreach tabled solution T in table
    if T <<< CurrentSolution
        flag T for removal from the table
    if CurrentSolution <<< T
        flag CurrentSolution as discard
endfor
```

Figure 4.1: Pseudo-code: Managing Tabled Solutions

The pseudo-code states that: *any* tabled solution that is less preferred than `CurrentSolution` is removed from the table; on the other hand, `CurrentSolution` is added to the table *only if* none of the tabled solutions is more preferred than it. Note that the two **if** conditions in the pseudo-code are examined independently from each other. Therefore, if contradiction exists in the preference declarations, e.g. both `sol1 <<< sol2` and `sol2 <<< sol1` can be derived, then neither `sol1` nor `sol2` are recorded in the table. On the other hand, if two solutions are not comparable, they do not eliminate each other. Hence, multiple optimal solutions are supported by this approach.

It will be proven later in this chapter that `updateTable/2` guarantees the solutions reside in the table (if any) for the specified query are the best solutions obtained so far based on the specified optimization criteria. Since it is invoked on all solutions, the tabled solutions obtained upon termination are the optimal solutions for the particular query. If no tabled solution exists, it implies that the preference program contains no solution.

4.2 Predicates Supporting the New Approach

New built-in predicates are introduced in order to support the new approach. The definition of the predicate `updateTable/2` strictly follows the description of the approach presented in Section 4.1.2. The formal definition of this predicate is presented in Figure 4.2, and detailed discussion of this process is followed.

Line 1 gives the top level definition of `updateTable/2`. When it is invoked, it first (line 2) obtains a list of all tabled solutions for the tabled predicate (`TabSol`) and the new solution (`CurrentSol`). `TabSol` as well as `CurrentSol` are then examined (line 3) using the specified optimization criteria. This would determine how the table should be updated in order to record the best solutions found so far.

```
updateTable(QueryName, Criteria) :-                                  (1)

    allTabledSolutions(QueryName, TabSol, CurrentSol),               (2)

    updateTabledSolutions(TabSol, CurrentSol, Criteria).             (3)


updateTabledSolutions( TabSol, Sol, Criteria ) :-                    (4)

    compareToTabled( Criteria, TabSol, Sol, Flist, SolFlag ),        (5)

    removeTabledSolutions( Flist ),                                  (6)

    SolFlag == 1.                                                    (7)


compareToTabled(_, [], _, [], Flag) :-                               (8)

    (var(Flag) -> Flag = 1 ; true ), !.                              (9)

compareToTabled(Criteria, [T|Ttail], Sol, [F|Ftail], Flag) :-        (10)

    Term1 =.. [Criteria, T, Sol], % T less preferred than Sol?       (11)

    Term2 =.. [Criteria, Sol, T], % Sol less preferred than T?       (12)

    (Term1 -> F = 1 ; F = 0),                                        (13)

    (Term2 -> Flag = 0 ; true),                                      (14)

    compareToTabled(Criteria, Ttail, Sol, Ftail, Flag), !.           (15)
```

Figure 4.2: Definition of updateTable/2

We break down the predicate `updateTabledSolutions/4` even further. In line 5, `compareToTabled/5` compares the new solution (`Sol`) against each tabled solution and generates a list of flags (`Flist`) indicating which of these tabled solutions shall be removed. A one (1) at position i of `Flist` indicates the tabled solution at position i of `TabSol` shall be removed; zero (0) otherwise. Line 6 thus removes the tabled solutions based on `Flist`. `SolFlag` is also determined in line 5 which indicates if `Sol` is more preferred than all the currently tabled solutions. If it is, line 7 will evaluate to be true, and `Sol` is added to the table; otherwise, `Sol` is simply discarded. Predicate `compareToTabled/5` follows directly from the pseduo-code presented in Figure 4.1, and its definition is given in lines 8 thru 15.

With these predicates defined, we can now embed P_{core} into P_{pref}. This is illustrated using examples in the next section.

4.3 Embedding Preference Programs and Examples

This section presents the embedding procedure for a given PLP containing P_{core} and P_{pref}. The embedding is deterministic and can be fully automated by simply invoke `updateTable/2` at the end of every optimization predicates definitions. The two arguments of `updateTable/2` may be determined automatically: `QueryName` is the same as the predicate which invoked `updateTable/2`; `Criteria` is specified in the mode declaration of this tabled predicate. This procedure is formally presented in the Figure 4.3.

```
foreach optimization predicate Name/N in $P_{core}$
    foreach definition "Head :- Body." of Name/N
        replace "Body."
        with "Body, updateTable(Name, PreferenceSymbol)."
    endfor
endfor
```

Figure 4.3: Pseudo-code: Embedding of P_{core} and P_{pref}

We further illustrates the embedding using concrete examples in this section. These examples also show us how various types of preference programs are supported by the PLP system.

4.3.1 Shortest Path

The following example shows the embedded program of Example 5. The difference between the embedded program and the original one is the addition of the sub-goals (`updateTable/2`) in the optimization predicate `path/6` in clauses (2) and (3). These sub-goals invoke the update table action every time a new solution is obtained. Therefore the solutions selection criteria defined in P_{pref} are successfully applied to the problem specification P_{core}, and the desired embedding is realized.

Example 8 *Current approach: embedded program of Example 5.*

```
%%% P_core %%%

path(X, X, 0, 0, []) :-

    updateTable(path, '<<<').                                   (1)

path(X, Y, C, D, [e(X, Y)]) :-

    edge(X, Y, C, D),

    updateTable(path, '<<<').                                   (2)

path(X, Y, C, D, [e(X, Z) | P]) :-

    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),

    C is C1 + C2, D is D1 + D2,

    updateTable(path, '<<<').                                   (3)


edge(a,b,4,10).   edge(b,a,3,12).   edge(b,c,2,14).             (4)


:- table path(+, +, <<<, <<<, -).                               (5)


%%% P_pref %%%

path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-

    C2 < C1.                                                    (6)

path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-

    C1 = C2, D2 < D1.                                           (7)
```

4.3.2 Multiple Optimal Solutions

We have pointed out earlier that the new approach is able to handle preference rules that allow multiple optimal solutions. Consider Figure 4.4 below. Each node represents a solution and each arc represents a preference relation where the preference decreases downwards. i.e., solution a and d are the most preferred and p is the least preferred. Since a and d are not comparable among themselves, this preference yields two optimal solutions, namely a and d.

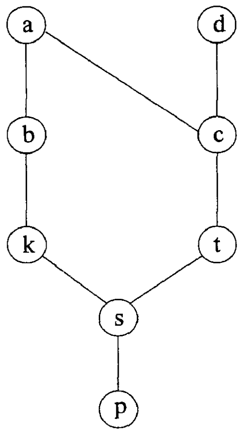


Figure 4.4: A Solution Set with Multiple Optimal Solutions

The corresponding embedded preference program of Figure 4.4 is presented in Example 9.

Example 9 *Support for multiple optimal solutions.*

```
    :- table letter(<<<).                                    (1)
```

```
%%% P_core %%%

letter(p):-

    updateTable(letter, '<<<').                                   (2)

letter(X) :-

    member(X, [s, p, b, k, t, c, a, d]),

    updateTable(letter, '<<<').                                   (3)


%%% P_pref %%%

letter(b) <<< letter(a).    letter(c) <<< letter(a).              (4)

letter(c) <<< letter(d).    letter(k) <<< letter(b).              (5)

letter(t) <<< letter(c).    letter(s) <<< letter(k).              (6)

letter(s) <<< letter(t).    letter(p) <<< letter(s).              (7)
```

Clause (2) sets the initial solution of `letter/1` to be p. Clause (3) determines the set of letters to be considered, and also embeds the selection criteria into the problem specification. Clauses (4) thru (7) defines the preference relations among the letters as shown in Figure 4.4.

When the program executes, despite letter p being the initial solution of `letter/1`, the preference rules will replace the tabled solutions with the best solution found so far. Thus when letter a is unified with variable X by predicate `member/2`, all the previously tabled solutions are eliminated and a is added into the table. However, the next letter, d, does not eliminate a, nor is eliminated by a. Therefore, both a and d are tabled upon termination; and both of them are indeed optimal solutions of this preference program.

4.3.3 Optimal Substructure Problems

A problem is said to have optimal substructure if its optimal solution can be constructed efficiently from optimal solutions to its sub-problems. Problems that have such property include matrix multiplication, alignment, etc. Optimal substructure of a problem may be defined in the solution selection criteria when applies, and it could simplify the preference declaration of the problem. This is illustrated in Example 10 below.

In this example, we define the right association rule of '+' (addition). The optimal substructure property holds because in order for `a+(b+(c+d))` to be the optimal association for `a+b+c+d`, its sub-problem, e.g., `b+(c+d)` needs to be the optimal association for `b+c+d`. The preference program is defined as follows.

Example 10 *Support for problems with optimal substructures.*

```
:- table rightAsso(<<<,+,+).                                    (1)


%%% P_core %%%

rightAsso(T1+T2,L1,L2) :-                                       (2)

    rightAsso(T1,L1,L3), L3=[+|L4], rightAsso(T2,L4,L2),

    updateTable(rightAsso, '<<<').

rightAsso(X,L1,L2) :-                                           (3)
```

```
    L1=[X|L2], member(X, [a,b,c,d]),

    updateTable(rightAsso, '<<<').


%%% P_pref %%%

rightAsso(T1,L1,L2) <<< rightAsso(T2,L1,L2) :-                    (4)

    lesspref(T1,T2).

rightAsso(T1,L1,L2) <<< rightAsso(T2,L1,L2) :-                    (5)

    lesspref(T1,T3),

    rightAsso(T3,L1,L2) <<< rightAsso(T2,L1,L2).


lesspref((T1 + T2) + T3, T1 + (T2 + T3)).                         (6)

lesspref(T1 + T2, T3 + T2) :- lesspref(T1, T3).                   (7)


:- rightAsso(P, [a,+,b,+,c,+,d], []).                             (8)
```

The first argument of `rightAsso/3` represents the solutions that we are expecting. The second and the third arguments form a difference list. Difference list is used to represent the information about grammatical categories not as a single list, but as the difference between two lists. E.g., "`[a,+,b,+,c,+d], []`" is equivalent to "`[a,+,b,+,c,+,d,+], [+]`", which is the list representation of `a+b+c+d` for our input. The usage of the difference list here is the same as the last two hidden arguments in DCG rules.

Clause (2) and (3) construct possible associations of `[a,+,b,+,c,+,d]`, and embed the selection criteria into the problem specification. The preference predicate is defined in clause (4) and (5), where clause (5) emphasizes on the transitivity rule. In the definition of predicate `lesspref/2`, clause (6) states that right association is more preferred than left association on the same level; and the optimal substructure property is defined by clause (7). This preference program yields `a+(b+(c+d))` when clause (8) is executed.

Notice that the preference rule

```
lesspref(T1 + T2, T1 + T3) :- lesspref(T2, T3).
```

is not a part of the program because it is implied. The reason is that `T1`, `T2` and `T3` are optimal solutions to the sub-problems. Adding an optimally structured expression to the *right* hand side of '+' would not violate the *right* associativity of the expression as a whole. Clause (7) in the program however is required because it adds the expression to the *left* hand side of the '+', which may cause violation.

4.4 Proof of Correctness

We prove the correctness of the procedural semantics of the PLP system in this section. Definition 6 states the procedural semantics of a embedded program.

Definition 6 *Let P and B_P be a program and its Herbrand base. We define a meta-level procedure $T_P : 2^{B_P} \longrightarrow 2^{B_P}$. Given a Herbrand interpretation I, $T_P(I)$ performs:*

```
$I_0 \leftarrow \emptyset$;
**foreach** *ground instance* $A$ :- $A_1, \cdots, A_m$ *of a clause in* $P$ *where*
$\{A_1, \cdots, A_n\} \subseteq I$
    **if** $\exists\ i \in I_0$ *s.t.* $A \prec i$
        $I_0 \leftarrow I_0 - \{\ i \mid i \in I_0,\ i \prec A\ \}$
    **else**
        $I_0 \leftarrow (I_0 - \{\ i \mid i \in I_0,\ i \prec A\ \}) \cup \{A\}$
**return** $I_0$
```

Thus, the fixed point semantics of P can be described as $T_P \uparrow \omega(\emptyset)$.

We can state the following theorem regarding the soundness and completeness of embedded preference programs based on the mapping defined in Definition 6.

Theorem 4 (Soundness and Completeness) *Let P be a preference program, $\rho(P)$ be the embedded program of P, and q/n be an optimization 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_{core}.*

Proof: This is shown based on the definition of `updateTable/2`. $\forall$ atoms of q/n, atom A_1 of q/n is pruned if $\exists$ an atom A_2 of q/n s.t. $A_1 \prec A_2$. Therefore, A is an atom of q/n and $A \in T_{\rho(P)} \uparrow \omega(\emptyset) \Leftrightarrow A$ is an optimized atom in M^P_{core} $\square$

Finally, Theorem 5 shows the equivalence between the declarative semantics of a preference program and its procedural semantics over a transformed tabled program.

Theorem 5 (Correctness) *Let P be a preference program, and $\rho(P)$ be the embedded program of P. Then $\pi_P \uparrow \omega(\phi_P(M^P_{core})) = T_{\rho(P)} \uparrow \omega(\emptyset)$, where $M^P_{core} = T_{P_{core}} \uparrow \omega(\emptyset)$.*

Proof: Let A be a Herbrand atom. The proof is based on the following two cases:

(i) If A is an atom of an optimization predicate,

$$\begin{array}{lll} & A \in \pi_P \uparrow \omega(\phi_P(M^P_{core})) & \\ \Leftrightarrow & A \text{ is an optimized atom in } M^P_{core}. & \textit{Corollary } 3 \\ \Leftrightarrow & A \in T_{\rho(P)} \uparrow \omega(\emptyset) & \textit{Theorem } 4 \end{array}$$

(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_{core}.

$$\begin{array}{lll} & A \in \pi_P \uparrow \omega(\phi_P(M^P_{core})) & \\ \Leftrightarrow & A \text{ is an instance of a fact in } P_{core}. & \text{Def. 5} \\ \Leftrightarrow & A \text{ is an instance of a fact in } \rho(P). & \text{Embedding} \\ \Leftrightarrow & A \in T_{\rho(P)} \uparrow \omega(\emptyset) & \text{Def. 6} \end{array}$$

Inductive case: Assume that for any ground instance

A :- $B_1, \cdots, B_n$, of a clause in P_{core}, we have $\{B_1, ..., B_n\} \subseteq \pi_P \uparrow \omega(\phi_P(M^P_{core})) \Leftrightarrow \{B_1, ..., B_n\} \subseteq T_{\rho(P)} \uparrow \omega(\emptyset)$, where $n > 0$.

	$A \in \pi_P \uparrow \omega(\phi_P(M^P_{core}))$	
$\Leftrightarrow$	$\exists$ a ground instance: A :- $B_1, \cdots, B_n$	
	s.t. $\{B_1, ..., B_n\} \subseteq \pi_P \uparrow \omega(\phi_P(M^P_{core}))$.	Def. 5
$\Leftrightarrow$	$\{B_1, ..., B_n\} \subseteq T_{\rho(P)} \uparrow \omega(\emptyset)$	Hypothesis
$\Leftrightarrow$	$A \in T_{\rho(P)} \uparrow \omega(\emptyset)$	Def. 6

$\square$

CHAPTER 5

Typical PLP Applications

We have shown some examples in Section 4.3 to help illustrating the embedded preference programs. Typical applications utilizing preference declarations may be more complicated than these examples. In this chapter we present a few of these applications as representatives of the problems that can take advantages of the PLP system.

5.1 Dynamic Programming

We have mentioned that problems with optimal substructure properties are supported by the PLP system. Dynamic programming is a typical type of algorithms that is based on this property. In this sectoin we use the *matrix-chain multiplication* problem as an example to illustrate how preferences can be used for simplifying the specification of dynamic programming problems in logic programming.

A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

Thus, the matrix-chain multiplication problem may be stated as follows:

Problem 1 *Given a chain $\langle A_1, A_2, ..., A_n \rangle$ of n matriccs, where for $i = 1, 2, ..., n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 ... A_n$ in a way that minimizes the number of scalar multiplications.*

For example we are given three matrices, $M1$, $M2$ and $M3$. Suppose the dimensions of these matrices are as follows: $M1 = 10 \times 100$, $M2 = 100 \times 5$ and $M3 = 5 \times 50$. If we multiply $M1$ and $M2$ first followed by $M3$, i.e., $(M1 \times M2) \times M3$ the number of scalar multiplications is $(10 \times 100 \times 5) + (10 \times 5 \times 50) = 7500$. However, if we use the following computation, $M1 \times (M2 \times M3)$, the number of scalar multiplications is increased to $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 75000$. Hence, it is important to find the optimal way(s) to multiply matrices.

To solve this problem by dynamic programming, we need to define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, which denotes a sub-chain of matrices $A_i A_{i+1} ... A_j$ for $1 \leq i \leq j \leq n$. Thus, our recursive formula for the minimum cost of parenthesizing the product $A_{i..j}$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

This formula shows that the programmer has to find the optimal value by comparing all possible multiplication costs explicitly. In fact, for a general optimization problem, the definition of an optimal solution could be quite complicated due to heterogeneous solution construction. Then, comparing all possible solutions explicitly to find the optimal one can be tricky and error-prone.

This explicit comparison can be avoided by utilizing preference declarations. With preferences, the programmer is only required to define *what* a general solution is, while *how* to search for the optimal solution is left to the logic programming system. For the matrix-chain multiplication, instead of defining the cost of an optimal solution, we only need to specify what the cost for a general solution is. The recursive definition for the cost of parenthesizing $A_{i..j}$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j \end{cases}$$

where $i \leq k < j$. Therefore, we have a preference program as follows:

Example 11 *A preference program that finds the optimal way to multiply matrices.*

```
:- table matrix(+, <<<, -).                                 (1)


%%% P_core %%%

matrix([D1, D2], 0, (D1, D2)).                              (2)

matrix([D1, D2, D3 | Dr], V, (E1 * E2)) :-                  (3)
```

```
        break([D1, D2, D3 | Dr], DL1, DL2, Dk, Dn),
        matrix(DL1, V1, E1),
        matrix(DL2, V2, E2),
        V is V1 + V2 + D1 * Dk * Dn,
        updateTable(matrix, '<<<').


%%% P_pref %%%
matrix(D, V, _) <<< matrix(D, V1, _) :- V1 < V.                   (5)
```

In this program, predicate `matrix(D,V,E)` is given `D` as a list of dimensions. Both `V` and `E` are the output of this program indicating the optimal scalar multiplication and its corresponding parenthesization, respectively. Predicate `break/5` is used to break a list of dimensions into two parts, say `L1` and `L2`, at the point of `Dk`. `Dn` represents the last dimension in the list `L`.

The output of this preference program is presented below. When we query the system using `matrix([10, 100, 5, 50], V, E)`, we obtain the following unique solution:

```
V=7500
E=(10, 100)*(100, 5)*(5, 50)
```

Notice that we have assigned a non-indexed mode to the third argument (the evidence), which tells the table to only record the very first instance of this argument. In case we want to record multiple evidences for a preferred solution, we may assign the preference mode to this argument instead (`:- table matrix(+, <<<, <<<)`). With this declaration, if there exist more than one optimal way of multiplying specified matrices, all optimal solutions are recorded. For example the query `matrix([10, 10, 10, 10, 10], V, E)` returns four optimal solutions as follows:

```
V=3000
E=(10, 10) * ((10, 10) * ((10, 10) * (10, 10))) ;
V=3000
E=(10, 10) * ((10, 10) * (10, 10) * (10, 10)) ;
V=3000
E=(10, 10) * (10, 10) * ((10, 10) * (10, 10)) ;
V=3000
E=(10, 10) * ((10, 10) * (10, 10)) * (10, 10) ;
V=3000
E=(10, 10) * (10, 10) * (10, 10) * (10, 10) ;
```

5.2 Ambiguity Resolution

Another important application that can take advantage of PLP is ambiguity resolution. It is critical to the language processing for precise syntax and semantics. We have presented an example of syntactic ambiguity using the *dangling else* problem in Chapter 1. And we have mentioned that to avoid such an ambiguity it is often necessary to change grammar itself, which sacrifices the clarity of the original grammar. In this section we present the preference program that is used to solve the ambiguity problem with out modifying the grammar.

Example 12 *A preference program that solves dangling else problem.*

```
:- table stmt(<<<, +, +).                                    (1)


%%% P_core %%%
stmt(A, B, C) :-                                             (2)
    cond(A, B, C),
    updateTable(stmt, '<<<').
stmt(if(A, B), [if|C], D) :-                                 (3)
    cond(A, C, E), E = [then|F], stmt(B, F, D),
    updateTable(stmt, '<<<').
stmt(if(A, B, C), [if|D], E) :-                              (4)
    cond(A, D, F),
    F = [then|G], stmt(B, G, H),
    H = [else|I], stmt(C, I, E),
    updateTable(stmt, '<<<').
cond(tt, [tt|A], A).         cond(ff, [ff|A], A).            (5)


%%% P_pref %%%
stmt(if(A, B, C), L1, L2) <<< stmt(if(A, D), L1, L2) :-      (6)
```

```
    combine(B, C, D).

combine(if(A, B), C, if(A, D)) :-                             (7)

    combine(B, C, D), !.

combine(if(A, B, C1), C, if(A, B, C2)) :-                     (8)

    combine(C1, C, C2), !.

combine(if(A, B), C, if(A, B, C)).                            (9)
```

Clauses (2) thru (5) make up the core program specifying the grammar. Note that this could be further simplified using DCG rules, which would look similar to Example 2 in Chapter 1. The emphasis here is the declarations of preference clauses (6) thru (9). The recursively defined predicate `combine/3` is utilized to ensure that no *if* statements without *else* clauses can appear inside the *then* part of an *if-then-else* statement. Using this preference, the *else* will bind to the last *if* statement, and still allows chaining. Since this preference is recursively defined, the optimal substructure property of this grammar is expressed.

The output of this preference program is presented below. When we query the system using `stmt(L, [if, tt, then, if, ff, then, tt, else, ff], []).`, the unique solution is obtained as follows:

```
L = if(tt,if(ff,tt,ff))
```

This parsing result shows indeed that the dangling else is paired with the last *if* statement.

CHAPTER 6

Implementation of the PLP System

The PLP system is implemented on top of a tabled Prolog system utilizing DRA approach [11], and is realized using C programming language. It is the C level procedures that carry out the WAM instructions that correspond to a given Prolog program. Therefore all Prolog level predicates, both built-in and user defined, rely on the C level implementation.

This chapter discusses the implementation of the PLP system at C level. The data structure that is used to record the tabled solutions is shown first. Then we discuss how the tabled solutions are managed. This is the C level correspondence of the `updateTable/2` predicate defined in Chapter 4. Following the table management, another aforementioned improvement – argument reordering is then discussed.

6.1 Data Structure of the Tabled Solutions

The solutions that reside in the table should be easily retrievable during execution. Therefore, the *trie data structure* [18], designed for data retrieval purposes is the natural choice. The term trie comes from "retrieval". It is an ordered tree data structure that is used to store associative arrays. This concept may be better illustrated using an example. Suppose we have three tabled solutions for tabled predicate `q/3`, say `q(a,c,t)`, `q(a,k,v)`, and `q(a,k,s)`. These solutions are stored using trie structure as shown in Figure 6.1. Symbol '$\wedge$' is used to indicate grounded pointers. Notice that each element contains three portions: 1) the actual data, or a pointer to the data if it is complex; 2) a child pointer (possibly *null*) pointing to the next argument of the predicate; and 3) a sibling pointer (possibly *null*) pointing to the alternative solutions that differs from this solution starting at this argument.

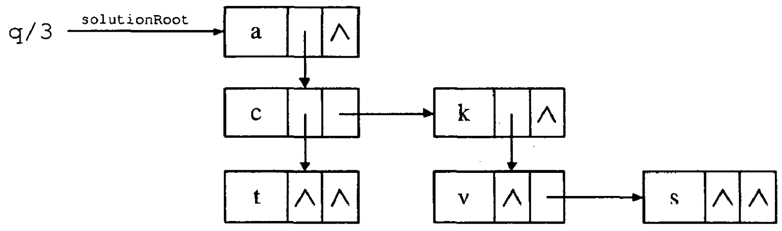


Figure 6.1: Tabled Solutions in the Trie Structure

Retrieving solutions from the tire structure is straightforward. Consider Figure 6.1, to retrieve a solution for `q/3`, we start the traversal at the first element that the `solutionRoot` pointer of `q/3` points to and extract its data. Hence the value of the first argument is obtained. We continue extracting the value of the second argument by following the child pointer of the current element. Now the sequence $\langle a, c\rangle$ is obtained. This process continues utill the value of the last argument is extracted. In our case, sequence $\langle a, c, t\rangle$ is finally obtained, and therefore we have obtained `q(a,c,t)` as a solutions for `q/3`.

To retrieve additional solutions for `q/3`, we would take the sibling pointers instead of child pointers, and then follow the same procedure to obtain the rest of the sequences. However, this would require us to record the choice points where alternative traversals were taken, which could be a potential performance problem. To overcome this, we added one additional level of elements to the bottom of the trie structure which explicitly indicates the traversal information. An additional pointer, `solutionStart`, is introduced for quick access to the elements at this level. These modifications are shown in Figure 6.2 below.

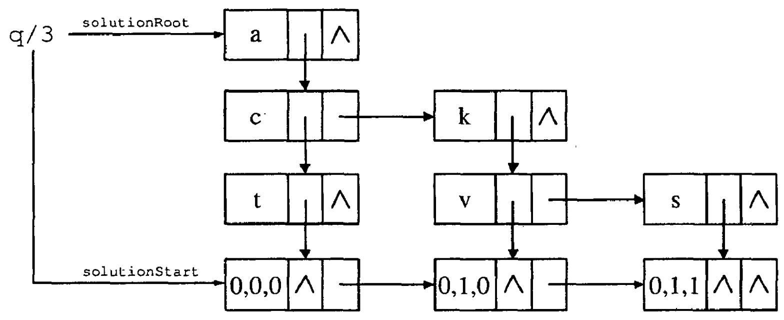


Figure 6.2: Tabled Solutions in the Trie Structure with Path Information

The path information recorded at the bottom level of the structure indicates how to obtain the solution expressed by the element that is pointing to it. The i-th value in a path represents the number of siblings that need to be offset in order to obtain the value of the i-th argument for the predicate. For example path $\langle 0, 1, 0 \rangle$ indicates that traveling from the first element that `q/3` is pointing to, offset 0 sibling to get the value of the first argument and move down to the second level. Then offset 1 sibling to get the value of the second argument and move down to the third level. Finally offset 0 sibling to get the value of the third argument. This traversal yields $\langle a, k, v \rangle$, which is indeed the solution that points to path $\langle 0, 1, 0 \rangle$. Therefore, rather than recording the choice points at run time, we may simply look up this information and retrieve the desired solution accordingly. Notice that all bottom elements that contain traversal information are linked together. This feature further simplifies the process of retrieving multiple tabled solutions for a predicate.

```
typedef struct Trie_Answer *TAptr;

typedef struct Trie_Answer

{

     TAptr child;       // a pointer to its child

     TAptr sibling;     // a pointer to its sibling

     PWord atom;        // abstract data type records the atom

}
```

Figure 6.3: Type Define: Trie Answer

Figure 6.3 defines the trie answer data type **`Trie_Anser`** as defined in the system. Its definition follows directly from the structure of trie. Type PWord is an abstract data type that is used to represent the data, or a pointer to the data, that is associated with the current element. The same data type is used for both elements carrying data as well as elements recording traversal information at the bottom level of trie.

```
typedef struct Tabled_Pred *TABLEptr;

typedef struct Tabled_Pred

{

    /* for retrieving tabled solutions */

    TAptr solutionRoot;      // a pointer to the root of solutions

    TAptr solutionStart;     // a pointer to the first solution path

    TAptr solutionEnd;       // a pointer to the last solution path

    int   arity;             // the arity of the tabled predicate


    /* for implementing DRA */

    short     pIndexState;   // 1 = NORMAL, 2 = LOOPING, 3 = COMPLETE

    CLAUSEptr loop;          // a pointer to looping Alternatives

    byte      constFlag;     // 1 = const, 0 = variable

    TABLEptr  next;          // a pointer to the next tabled predicate

}
```

Figure 6.4: Type Define: Tabled Predicate

Figure 6.4 defines the tabled predicate data type Tabled_Pred as defined in the system. Its definition is a little more complicated because it contains information needed to realize DRA as well. The implementation of DRA is out of scope of this thesis, thus we only concentrate on the data members that are associated with retrieving solutions from the table. These data members are: `solutionRoot`, `solutionStart`, `solutionEnd`, and `arity`. `solutionRoot` is a pointer pointing to the root of solutions. `solutionStart` is a pointer pointing to the first solution path, while `solutionEnd` points to the last solution path. `arity` simply records the arity of the tabled predicate. For easier understanding of these pointers, both `solutionRoot` and `solutionStart` are visually represented in Figure 6.2.

After defining the data types of `Trie_Answer` and `Tabled_Pred`, the next section discusses how the tabled solutions are managed in the trie. These operations corresponds to the predicates involved in the `updateTable/2` predicate defined in Figure 4.2.

6.2 Management of the Tabled Solutions

Three procedures dealing with the trie structure presented above are defined at C level in order to support the Prolog level predicate `updateTable/2` for its tabling operations. They are: *retrieve all* tabled solutions for a specified predicate, *remove* a set of solutions from the table, and *add* a new solution to the table. These operations are essential to the implementation of the PLP system. Thus each of them is discussed in detail in this section.

We will present these operations in the order of which they are invoked by updateTable/2. The first operation invoked is to retrieve a list of all tabled solutions of a specific predicate, hence it is discussed first. The retrieved list is then compared against the new solution obtained, and a set of tabled solutions to be removed are determine. Therefore we illustrate how solutions are removed from the table next, followed by a discussion on adding solutions to the table.

6.2.1 Retrieving All Tabled Solutions of a Table Predicate

The first procedure invoked by updateTable/2 is to list all tabled solutions of the specified predicate. This procedure is carried out by performing a traversal from the start of the solution to the end. Since the predicate is specified, we can locate the root of solutions and the first solution path using pointers solutionRoot and solutionStart, respectively (as shown in Figure 6.2). Then for each element at the path level, we retrieve the corresponding values based on the path information to form a solution. The result of this traversal on each element at the path level is the set of all tabled solutions for the specified predicate.

Figure 6.5 shows the C level implementation of this procedure. It illustrates at which positions we should extract the data from in order to reconstruct valid solutions from the a structure. The code for extracting the actual data is straightforward yet lengthy, thus has been omitted from this figure.

In this function definition, tptr is a pointer pointing to the specified tabled predicate. The two TAptr pointers defined are used for traversal purposes:

```
void allTabledSolutions(TABLEptr tptr)

{

     TAptr currentAnswerPtr;      // points to a path level element

     TAptr currentNodePtr;        // points to a data level element

     short *pathPtr;              // points to a path value


     if (tptr->solutionStart != NULL) {

          for (currentAnswerPtr = tptr->solutionStart;

               currentAnswerPtr != NULL;

               currentAnswerPtr = currentAnswerPtr->sibling)

          {

               /* get the current tabled answer */

               pathPtr = (short *)currentAnswerPtr->atom;

               currentNodePtr = tptr->solutionRoot;


               for (i = 1; i <= tptr->arity; i++) {

                    /* need to move to the sibling? */

                    if (*pathPtr >= 1)

                         for (k = 0; k < *pathPtr; k++)

                              currentNodePtr = currentNodePtr->sibling;


                    /* CODE FOR EXTRACTING DATA IS OMITTED */


                    /* move to the next argument */

                    currentNodePtr = currentNodePtr->child;

                    pathPtr++;

               }

          }

     }

}
```

Figure 6.5: Pseudo-code: Retrieving All Tabled Solutions of a Predicate

`currentAnswerPtr` which points to the path level of the trie, and `currentNodePtr` which points to the data level of the trie. Pointer `pathPtr` points to the path information recorded in a path level element. If no tabled solution exists for the specified predicate, no action is taken and the function terminates. Otherwise, `currentAnswerPtr` is set to point to the first path element, and its path information is assigned to `pathPtr`. Then `currentNodePtr` travels along the trie according to the path information, so a tabled solution is retrieved. This process is repeated until `currentAnswerPtr` points to *null*, i.e., all tabled solutions are obtained. Notice that this function does not have any return value because it sets the value of the argument `List` of the Prolog predicate `allTabledSolutions(List)` directly. Upon termination of the C function `allTabledSolutions`, `List` will contain the list of all tabled solutions of the tabled predicate, if any, currently stored; or empty otherwise.

6.2.2 Removing a Set of Solutions From the Table

After Prolog receives the list of all tabled solutions for the specified predicate, it constructs a flag list indicating the set of tabled solutions to be removed. To remove this set of from the table, we just need to remove the desired path level elements from the trie structure. This is similar to the standard linked list removal procedure. However, it is a little more involved because we have multiple elements that need to be removed. Luckily, the flag list (`FList`) generated by `compareToTabled/5` at Prolog level (in Figure 4.2) corresponds directly to the elements in the path level. i.e., the i-th element in the path list is to be removed if the i-th flag in the flag list is turned on; otherwise it stays in the trie structure. This one-to-one correspondence allows us to accomplish the removal of multiple elements still in one traversal.

Figure 6.6 shows the C level implementation of this procedure. It illustrates how a path level element is determined to stay or discard depending on the flag information. In this function definition, `tptr` is a pointer pointing to the specified tabled predicate. `flagPtr` points to the starting location of flag list determined by Prolog. Like `addTabledSolutions` function, the remove function also has two `TAptr` pointers defined, but for different purposes. `currentPtr` points to the path level element that is currently being examined according to its corresponding flag list value. `prePtr` on the other hand points to either the last path level element that is determined to stay, or *null* if no such element has been determined yet. It is used so that we can update the `solutionEnd` information upon termination. Notice that the standard linked list removal procedure for one element is adapted in this function as indicated, but the actual code is omitted. Notice also that after removal, if `solutionStart` is pointing to *null*, i.e. all tabled solutions are removed, `solutionEnd` and `solutionRoot` are also set to *null*. Since this function operates directly on the trie structure that stores the tabled solutions, no return values is needed.

```
void removeListed(TABLEptr tptr, long* flagPtr)
{
    /* pointers to path level elements */
    TAptr currentPtr;     // element currently being examined
    TAptr prePtr;         // the last element determined to stay

    if (tptr->solutionStart != NULL)
    {
        /* initialize prePtr */
        prePtr->atom = 0;
        prePtr->sibling = tptr->solutionStart;

        for (currentPtr = tptr->solutionStart;
            currentPtr != NULL;
            currentPtr = currentPtr->sibling, flagPtr++)
        {
            if (*flagPtr)
                /* STANDARD LINKED LIST REVMOVAL IS OMITTED */
            else
                prePtr = currentPtr;
        }

        /* prePtr is the LAST node that is NOT removed */
        tptr->solutionEnd = prePtr;

        if (tptr->solutionStart == NULL)
            tptr->solutionEnd = tptr->solutionRoot = NULL;
    }
}
```

Figure 6.6: Pseudo-code: Removing a Set of Solutions From Table

6.2.3 Adding a Solution To the Table

The last operation needed to support `updateTable/2` is to add a new solution to the table. A solution can be added to the table if none of the tabled solution is more preferred than itself. The actual *adding* operation is triggered automatically at the Prolog level if `SolFlag == 1` is a true statement (line 7 in Figure 4.2). However, defining this operation at C level could be a little tricky, due to the usage of different modes for different arguments.

Mode declarations of tabled predicates are discussed in Section 2.3.2. It 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. An extension of the mode declaration is introduced and presented in Table 6.1, while keeping the semantics for '+' and '−' unchanged.

Modes	Informal Semantics
+	an indexed argument
−	a non-indexed argument
<<<	a user-defined pereference mode

Table 6.1: Extended Built-in Modes for Tabled Predicates

Mode '+' indicates the corresponding argument is indexed and will be used for variant checking during collecting new generated answers into the table. Mode

'-' indicates the corresponding argument is non-indexed and does not need to be recorded into the table except the very first instance. Mode '<<<' indicates whether or not the corresponding argument is to be inserted into the table depends on the defined preference rule '<<<'. As aforementioned, '<<<' is only the choice of preference symbol of this thesis, and it could vary from user to user. In anyways, it does not alter the semantics of the *preference mode* declared. Notice that modes such as '`max`' and '`min`' can all be expressed using mode '<<<', given that the preference rules are defined as such respectively. Therefore, we may consider the extended mode table as a generalization of mode declaration.

We use an example to help understanding the new mode. For example, given a predicate `q(+,<<<,-)` and assume that the optimization criteria is to *maximize* the second argument for a fixed first argument. Further assume that a solution exists in the table for `q/3`, say `q(a,5,3)`. If a new solution `q(a,9,2)` is generated and is to be added to the table, it replaces `q(a,5,3)` because $5 < 9$. Now another new solution `q(a,9,7)` is generated, but it does not get added to the table because the third argument is declared to be non-indexed. On the other hand, if the new solution were `q(b,9,7)`, both `q(a,9,2)` and `q(b,9,7)` would reside in the table because the first argument is indexed, and `a` is different from `b`.

It is obvious that the procedure for adding a solution into the table shall follow the declared argument modes. However, we need to make an assumption before discussing the adding procedure. The *assumption* states that the mode declaration of a tabled predicate q does not have mode '+' appearing after '<<<' and '−', and does not have mode '<<<' appearing after '−'. i.e., mode declaration such as `q(-, +, <<<)` would be inappropriate. The reason for this assumption is that with the above declaration, all instances of the first argument will have to be tabled for q in order to get to the rest of the arguments. Tabling all instances of this argument violates the semantics of non-indexing '−', hence not allowed. This shortcoming is addressed in the next section when *argument reordering* is discussed. Therefore the above assumption will no longer be necessary. However, for the discussion of adding solution to the table for now, let's assume all modes are declared in the desired order. This procedure still applies with the support of argument reordering, as will be discussed in Section 6.3.

Given the appropriate ordering of the argument modes, Figure 6.7 shows an abstract pseudo-code for the procedure of adding a solution into the table. As defined in the function, if the table was empty for the particular predicate prior to adding the solution, we can simply insert the values of all arguments into the trie accordingly, regardless of their modes. It is a little more complicated when the table contains some solutions already. In this case we need to compare the new solution against the tabled ones, and make decisions based on the mode declaration of the predicate. Notice that this comparison is *different* from the comparison using preference rules as predicate `compareToTabled/5` defined in Figure 4.2. The comparison process performed here is done *per argument basis*,

```
void addNewSolution(TABLEptr tptr, Argument arg)
{
     TAptr tabled = tptr->solutionStart;    // points to data level
     int   insertFlag = 0;                  // indicates insert or not


     if (tptr->solutionStart == NULL)       // empty table, insert all
          for (i=0; i<arg.count; i++) {
               /* INSERT arg INTO TRIE, CODE OMITTED*/
          }
     else // non-empty table, need to make comparisons based on modes
          for (i=0; i<arg.count; i++, arg++, tabled=tabled->child) {
               swith(arg.mode)
               {
                    case '-':
                         return;
                    case '+':
                    case '<<<':
                         if (arg.value != tabled->atom AND any of its sibling)
                              insertFlag = 1;
               }
               if (insertFlag) {
                    /* INSERT REST OF args INTO TRIE, CODE OMITTED */
                    break;  // break the for loop
               }
          }
}
```

Figure 6.7: Pseudo-code: Adding a New Solution to Table

and is solely depended on the mode declaration of the predicate. There are three possible modes that can be associated with each argument:

1. The mode of the current argument is '-', which implies that so are the rest of the arguments, if any. In this case, the new solution is discarded. This is because getting to this point implies the only differences between the new solution and the tabled solution occur at the non-indexed arguments, which would have instances tabled already.

2. The mode of the current argument is '+'. In this case, we compare the current argument against the corresponding argument in the table as well as all of its siblings. If no matching is found among the siblings, the new solution is marked as to be added as a *new sibling.* However if we do find a matching sibling, we move on to the next argument and repeat the comparison process again.

3. The mode of the current argument is '<<<'. Notice that when we get to `addNewSolution` function at C level, it implies that this solution has already been examined by `updateTable/2` against the preference specifications. Therefore, the procedure for this case is the same as for '+' described above.

Figure 6.7 is used to illustrate how to determine whether or not to insert a new solution, and its position to be inserted in trie structure. The procedure for inserting elements into a trie is as straightforward as to inserting elements into a linked list. Therefore, the code has been omitted from the figure.

6.3 Argument Reordering

After discussing the procedure to add a new solution to the table, we now get back to our assumption stated earlier that the mode declaration of a tabled predicate does not have mode '+' appearing after '<<<' and '-', and does not have mode '<<<' appearing after '-'. One might ask the following questions:

- What if the mode of the predicate have to be defined as `q(-, +, <<<)`?
- and how would it be recorded in the table while not violating the '-' mode declaration?

These concerns are addressed by a technique named *argument reordering*, and it is discussed in detail in this section. It is important to emphasis that the new order of the arguments in a tabled predicate *only* affects the order of the values of these arguments recorded in the table. Thus the rest of the system does not need to be aware of this new ordering.

In this section, we first discuss why such reordering is needed. Then we explain how the new order is determined and managed. Finally, we address the changes necessary to support reordering in both adding and retrieving procedures.

6.3.1 Need for Reordering

It is true that in most cases we can purposely order the arguments of a tabled predicate to follow the condition specified in the assumption. However, it could be cumbersome in practice. Predicates utilizing DCG declarations are a good example. Notice that in a DCG rule, two arguments expressing the difference list are hidden from the predicate declaration. The positions of these two arguments always appear after the argument to be parsed, and they ought to be indexed in the table. The argument to be parsed, on the other hand, shall be declared with mode '<<<' because we are interested in optimizing it. This yields a mode declaration as follows:

```
:- table predicate(<<<, +, +).
```

Obviously, this declaration violates the assumption. Therefore, a mechanism is needed to reorder the arguments based on their mode declarations. Notice again that this reordering mechanism *only* affects the order of the arguments recorded in the table. Therefore except adding and retrieving tabled solutions, all the rest of operations aforementioned are independent from this mechanism, thus will not be modified. Figure 6.8 below illustrates where the reordering takes place in the PLP system.

As the figure indicates, the new order of a tabled predicate is *not* known to the outside world except within the table. On the other hand, only the new

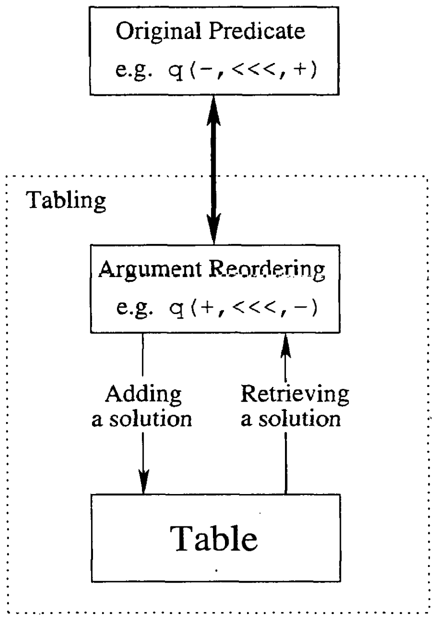


Figure 6.8: Argument Reordering Technique

order of the arguments is known to the table. The advantage of this design is that changes to the existing system are limited to be only within the table management procedures, adding and retrieving to be specific. Yet, as we will see in Section 6.3.3, even these changes are minor and can be easily implemented.

6.3.2 Reordering Based on Mode Declarations

As the title suggests, arguments are reordered based on their associated mode declarations. To put these arguments into the desired order, we assign priorities to each of the three modes as shown in Table 6.2 below, where the lower the number, the higher the priority:

Each argument in a tabled predicate has a mode associated with it, which in

Modes	Priority
+	1
<<<	2
−	3

Table 6.2: Priorities of Modes

turn has a priority assigned to it as well. With the priorities of the arguments determined, a sorting in increasing order of the priorities would put the arguments in the desired order. Therefore, all of the indexed arguments will appear at the beginning, followed by the arguments with preference mode. All of the non-indexed arguments are shifted to the rear end, and they will not be inserted into the table if the table is non-empty.

These arguments, though sorted, still have their *original positions* recorded. The original position information is necessary when retrieving a solution from the table, because we need to put the retrieved values back to the original positions they appear in the predicate definition.

6.3.3 Support for Reordered Arguments in Table

As we have already emphasized, only procedures for adding and retrieving tabled solutions might be affected by the new reordering mechanism. It turns out that, the procedure for adding a solution into the table *does not* need to be altered at all. The reason is that a tabled predicate is only known to the table with its arguments properly ordered, thus it can table the values of each arguments as it receives using the algorithm presented in Section 6.2.3. The procedure for retrieving a tabled solution, on the other hand, needs to be handled with a little extra attention.

Since the values of the arguments recorded for a tabled predicate is in the re-arranged order, we need an extra step to put them back to their original positions after retrieval. Luckily, each argument still has its original position information associated with it, so we can easily determine at which argument position of the tabled predicate the value should be unified with. This is illustrated using the pseudo-code presented in Figure 6.9.

```
foreach argument A in a tabled solution of predicate P
    value ← extract value of A from table
    n ← get A's original position in P
    unify value with the n-th argument of P
endfor
```

Figure 6.9: Pseudo-code: Retrieve Procedure Supporting Reordering

While the above procedure is invoked when retrieving one tabled solution, it is also a necessary step to be included in the pseudo-code for retrieving all tabled solutions presented in Figure 6.5. However, it is by no means more complicated than single solution retrieval. We can simply invoke this procedure at the data extraction stage. i.e., the line denoted as

```
/* CODE FOR EXTRACTING DATA IS OMITTED */
```

and it will make sure all solutions retrieved are rearranged back to their original orders.

6.3.4 Example of Argument Reordering

A very simple example is used to show the reordering of the arguments, and how a solution is stored in the table. Suppose we have a predicate declared as follows:

```
:- table q(-, <<<, +).
```

and suppose we have obtained the first solution of this predicate `q(5, 4, a)` and it is determined to be tabled. Since the mode declaration of this predicate is not in the desired order, this solution will be reordered as `q(a, 4, 5)` based on the priorities of the modes. Hence the trie structure of this solution looks as shown in Figure 6.10 (notice that the value of the non-indexed argument is tabled since it is the first instance).

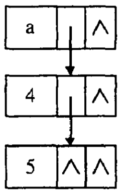


Figure 6.10: Reordered Solution in Trie

When we retrieve this solution from the table, 'a' is first obtained, and it is unified with the third argument of `q/3`, which yields `q(_, _, a)`. The value of the next argument is 4, and is unified with the second argument of `q/3`, which yields `q(_, 4, a)`. Finally, 5 is obtained and unified with the first argument. Solution `q(5, 4, a)` is hence successfully retrieved from the table.

CHAPTER 7

Experimental Results

The performance of the PLP system was tested under both numeric and structural domains using logic programs with and without preference declarations. The running times are compared and presented in this chapter followed by discussions. All tests were performed on an Intel Pentium 4 CPU 2.4GHz machine with 512M RAM running RedHat Linux 9.0.

7.1 Performance on Numeric Domain

For preference rules defined under numeric domain, five typical optimization examples were used: `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 `knapsack` is the knapsack problem. Table 7.1 compares the running time performance between the programs with and without preferences.

The experimental results show that preferences provide a declarative approach

	matrix	**lcs**	**obst**	**apsp**	**knapsack**
with preferences	4.46	0.87	4.96	5.6	52.1
no preferences	7.26	1.47	22.69	6.8	79.0

Table 7.1: Numeric domain running time comparisons (in seconds)

without sacrificing efficiency of dynamic programming. In 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 built-in predicate `findall/3` – to collect all the possible answers explicitly and then locate the optimal one. The experimental data indicates that the programs with preference declaration are better than those corresponding programs without preference declaration.

7.2 Performance on Structural Domain

For preference rules defined under structural domain, two typical examples were used: `dangling` is the dangling else problem which we have used through out the thesis; and `parser` is a programming language parser supporting different operator precedence and left associativity. Table 7.2 compares the running time performance between the programs with and without preferences.

In the preference programs, like those defined under numeric domains, the

	parser (small)	parser (large)	dangling (small)	dangling (large)
with preferences	0.21	10.1	0.05	5.23
no preferences	0.01	0.02	0.01	0.01

Table 7.2: Structural domain running time comparisons (in seconds)

tabled system collects optimal answers implicitly by applying the predefined preferences; the programs without preferences utilizes DCG rules with extra terms defined explicitly to remove the ambiguities. Although the preferences provide a more declarative approach, longer time was spent in determining the optimal answer based on these rules by comparing potential solutions. The explicitly defined DCG rules on the other hand, generate the sole solution by backtracking, and no comparison of any kind was involved. Hence the latter outperforms the former.

7.3 Discussion on the Performances

The performance differences on numeric and structural domains seem to be shocking at first. However, a closer examination reveals the explanation.

The efficiency for preference programs are mainly credited to the mode declarations. Tabled Prolog systems with mode declaration provides a concise but easy-to-use interface for preference logic programming. Mode declarations are flexible and powerful for supporting user-defined preferences, and the mode functionality is implemented at the system level instead of the Prolog programming level.

Despite its gain by utilizing mode declarations, two important disadvantageous efficiency issues are the frequent access to the tabled answers, and comparisons using the preference rules every time a new solution is generated. The retrieval of a tabled answer for comparison incurs time overhead due to having to locate each argument of the answer in the table. For replacing a tabled answer only involves numerals as arguments, 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. In addition, preference rules defined on numeric domain only involve simple comparisons which can be performed extremely efficiently. The ones defined on structural domains, on the other hand, are much more involved and possibly recursively defined, which takes much longer to determine the more preferred solutions.

Therefore, preference logic programs can be used without sacrificing efficiencies on problems where preference comparisons can be evaluated relatively quickly. They are also very useful for specifying problems with dynamically changing preferences, as the preference criteria are separated from the problem specifications.

CHAPTER 8

Conclusion

The underlying philosophy of preference logic programming may be expressed using the equation: Program = Logic + Preferences + Control. This paradigm is particularly suited to those optimization problems requiring comparison and selection among alternative solutions. It allows logic (constraints of the problem) and preferences (the criteria for the optimal solutions) to be specified separately in a declarative fashion.

The declarative semantics of a preference logic program is based upon Herbrand models. Preference specifications essentially induce a strict partial order on ground atoms, and the intended model is defined in terms of the most preferred atoms according to this order. We show that this intended model can be characterized as the least fixed point of a natural meta-level mapping operation over the least Herbrand model of the core program. Furthermore, this semantics paves the way for a proof of correctness of our proposed implementation.

This thesis presented an implementation scheme on how to extend a tabled Prolog system with declaring and executing preferences. This implementation has several advantages over [13]: (1) A new table management system is designed and implemented, which enhanced the ability to manage user defined preference relations among the tabled solutions. (2) The new table management system also allows the connection between problem specification and preference criteria to be simplified; hence transformation is no longer needed. (3) The requirement of a strict partial order relation among solutions is relaxed. Contradictory preferences, as well as preferences with multiple optimums are now supported. (4) Argument reordering based on mode declarations is designed in efforts to support tabled predicate definitions with arguments not in the desired order.

A PLP system based upon mode-directed preferences has been successfully implemented in the TALS tabled Prolog system. No major changes are required to the Prolog engine and its tabled resolution scheme. Experimental results have shown that tabled Prolog programming with preferences provide a declarative yet efficient approach for generalized optimization. Additionally, the same implementation 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.

CHAPTER 9

Future Extensions

9.1 Improvements on Structural Domain

The results from our experiments in Chapter 7 indicated that though PLP has its advantages in solving problems in numeric domain, it could use some improvements in structural domain. We noticed that the majority of the problems in this domain, such as parsing, associativity, etc., have their solution sets forming partial order relations. Therefore, we have considered the improvements possible for problems having this property.

Given that the solution set always form a partial order relation, we may represent this relation using a lattice with the most preferred solution at the top and the less preferred solutions at lower levels. The idea is to quickly find a possible solution to the problem regardless of its preference, then keep going upwards on the lattice until the top is reached. The top solution is therefore the sole optimal solution we are expecting.

This approach could be realized by two steps. (1) First we solve the problem to get one solution without taking preference rules into consideration, and record it as the base solution. (2) Once the base solution is obtained, we then keep trying to find another solution that is more preferred than the base using *only* the defined preference rules. The new solution is then considered as the new base, and the process continues until no more solution can be produced. The solution recorded at the end is our optimal solution.

This approach would improve the efficiency because solving the problem initially does not require preference comparisons. In addition, walking along the lattice upwards only invokes the preference rules, so no re-solving is necessary.

9.2 Dynamic Preference and Incremental Computing

Preference logic programming provided with us the ability to separate the problem declaration from the solution selection criteria. This separation is important and beneficial because not only it makes the program more declarative, but also gives the users the ability to modify the preferences at run time. We refer to this as *dynamic preferences*, which is commonly practiced in scheduling problems where the requirements are changing rapidly.

Although the current PLP supports dynamic preferences (users can add/remove preferences as a Prolog program file), some additional improvements could make it more efficient. We are currently focusing on developing incremental solvers to handle the changing preferences. The idea is that instead of solving the PLP every time new preferences are added/removed from the program, the system can determine the part(s) of the program that need to be re-solved, while keeping the rest of the solved solutions unchanged. We can also take the advantage of tabling to retrieve previously generated solution more quickly than starting from scratch.

APPENDIX A

Sample Testing Programs

This appendix contains one pair of programs (with and without preferences) each for numeric and strucutral domain.

A.1 Numeric Domain - Matrix Chain Multiplication

Preference Logic Programming:

```
matrix([D1, D2], 0, D1, D2, (D1, D2)).
matrix([D1, D2, D3 | Dr], V, D1, Dn, (E1 * E2)) :-
    break([D1, D2, D3 | Dr], DL1, DL2, Dk),
    matrix(DL1, V1, D1, Dk, E1),
    matrix(DL2, V2, Dk, Dn, E2),
    V is V1 + V2 + D1 * Dk * Dn.

break([D1, D2, D3], [D1, D2], [D2, D3], D2).
break([D1, D2, D3, D4 | Dr], [D1, D2], [D2, D3, D4 | Dr], D2).
break([D1, D2, D3, D4 | Dr], [D1 | L1], L2, Dk) :-

    break([D2, D3, D4 | Dr], L1, L2, Dk).


:- table matrix(+, <<<, -, -, -).

matrix(D, V, D1, Dn, _) <<< matrix(D, V1, D1, Dn, _) :- V1 < V.
```

Traditional Prolog:

```
matrix([D1, D2], 0, D1, D2, (D1, D2)).

matrix([D1, D2, D3 | Dr], V, D1, Dn, (E1 * E2)) :-

    findall(

        (V, E1, E2),

        ( break([D1, D2, D3 | Dr], DL1, DL2, Dk),

          matrix(DL1, V1, D1, Dk, E1),

          matrix(DL2, V2, Dk, Dn, E2),

          V is V1 + V2 + D1 * Dk * Dn ),

        VL),

    minimal((V,E1,E2), VL).


minimal(V, [V]).

minimal((V,E1,E2), [(V1,E11,E12), (V2,E21,E22) | VL]) :-

    minimal((V3,E31,E32), [(V2,E21,E22) | VL]),

    ( V1 > V3

        -> (V,E1,E2) = (V3,E31,E32)

        ;  (V,E1,E2) = (V1,E11,E12)

    ).


break([P1, P2, P3], [P1, P2], [P2, P3], P2).
break([P1, P2, P3, P4 | Pr], [P1, P2], [P2, P3, P4 | Pr], P2).

break([P1, P2, P3, P4 | Pr], [P1 | L1], L2, Pk) :-

    break([P2, P3, P4 | Pr], L1, L2, Pk).
```

A.2 Structural Domain - Dangling Else

Preference Logic Programming:

```
stmt(A, B, C) :-
    cond(A, B, C).
stmt(if(A, then, B), [if|C], D) :-
    cond(A, C, E),
    E = [then|F], stmt(B, F, D).
stmt(if(A, then, B, else, C), [if|D], E) :-
    cond(A, D, F),
    F = [then|G], stmt(B, G, H),
    H = [else|I], stmt(C, I, E).


cond(tt, [tt|A], A).
cond(ff, [ff|A], A).


:- table stmt(<<<, +, +).
stmt(if(A, then, B, else, C), L1, L2) <<<
stmt(if(A, then, D), L1, L2) :-
    combine(B, C, D).
combine(if(A, then, B), C, if(A, then, D)) :-

    combine(B, C, D), !.

combine(if(A, then, B, else, C1), C, if(A, then, B, else, C2)) :-

    combine(C1, C, C2), !.

combine(if(A, then, B), C, if(A, then, B, else, C)).
```

Traditional Prolog:

```
stmt(A, B, C) :-
    cond(A, B, C).
stmt(if(A, then, B), [if|C], D) :-
    cond(A, C, E),
    E = [then|F],stmt(B, F, D).
stmt(if(A, then, B, else, C), [if|D], E) :-
    cond(A, D, F),
    F = [then|G],then_stmt(B, G, H),
    H = [else|I],stmt(C, I, E).


then_stmt(A, B, C) :-
    cond(A, B, C).
then_stmt(if(A, then, B, else, C), [if|D], E) :-
    cond(A, D, F),
    F = [then|G],
    then_stmt(B, G, H),
    H = [else|I],
    then_stmt(C, I, E).


cond(tt, [tt|A], A).                cond(ff, [ff|A], A).
```

REFERENCES

[1] G. Brewka: Well-Founded Semantics for Extended Logic Programs with Dynamic Preferences. *Journal of Artificial Intelligence Research*, 4:19-36, 1996.

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

[3] Baoqiu Cui and Terrance Swift: Preference Logic Grammars: Fixed Point Semantics and Application to Data Standardization. *Artificial Intelligence*, 138(1-2): 117–147, 2002.

[4] J. Delgrande, T. Schaub, and H. Tompits: Logic Programs with Compiled Preferences. *ECAI*, 464-468, 2000.

[5] M.H. van Emden and R.A. Kowalski: The Semantics of Predicate Logic as a Programming Language. *JAMC*, 23(4): 733–742, 1976.

[6] F. Fages: On the Semantics of Optimization Predicates in CLP Languages. In *Proc. 13th FST-TCS*, 1993.

[7] F. Fages, J. Fowler, and T. Sola: Handling Preferences in Constraint Logic Programming with Relational Optimization. In *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*,

261 - 276, 1993.

[8] S. Ganguly, S. Greco, and C. Zaniolo: Minimum and Maximum Predicates in Logic Programming. In *Proc. Tenth PODS*, 1991.

[9] K. Govindarajan, B. Jayaraman, and S. Mantha: Preference Logic Programming. In Proceedings of *International Conference on Logic Programming (ICLP)*, pages 731–745, 1995.

[10] K. Govindarajan, B. Jayaraman, and S. Mantha: Optimization and Relaxation in Constraint Logic Languages. *POPL* 1996: 91–103.

[11] Hai-Feng Guo and Gopal Gupta: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In Proceedings of *International Conference on Logic Programming (ICLP)*, pages 181–196, 2001.

[12] Hai-Feng Guo and Gopal Gupta: Simplifying Dynamic Programming via Tabling. *Practical Aspects of Declarative Languages (PADL)*, pages 163–177, 2004.

[13] Hai-Feng Guo and Bharat Jayaraman: Mode-directed Preferences for Logic Programs. *The 20th Annual ACM Symposium on Applied Computing*, Mar. 2005.

[14] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.

[15] Bharat Jayaraman, Kannan Govindarajan, and Surya Mantha: Preference Logic Grammars. *Computer Languages*, 24(3): pages 179–196, 1998.

[16] M.J. Maher and Peter J. Stuckey: Expanding Query power in Constraint Logic Programming Languages. Proceedings of NACLP, 1989.

[17] S. Parker: Partial Order Programming. In *Proc. 16th POPL*, 1989.

[18] P. Rao, I. V. Ramakrishnan, K. Sagonas, T. Swift, and D. S. Warren: Efficient table access mechanisms for logic programs. Journal of Logic Programming, 38(1):31-54, Jan. 1999.

[19] R. Rocha, F. Silva, and V. S. Costa: On a Tabling Engine That Can Exploit Or-Parallelism. In *ICLP* Proceedings, pages 43–58, 2001.

[20] Warren Abstract Machine: A Tutorial. MIT Press, 1991.

[21] David H. D. Warren: An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, 1983.

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

[23] XSB system. http://xsb.sourceforge.net

[24] Neng-Fa Zhou, Y. Shen, L. Yuan, and J. You: Implementation of a Linear Tabling Mechanism. *Journal of Functional and Logic Programming*, 2001(10), 2001.