Tabling is a technique that can get rid of infinite loops and redundant computations in the execution of recursive logic programs. The main idea of tabling is to memorize the answers to subgoals and use the answers to resolve their variant descendents. Tabling helps narrow the gap between declarative and procedural readings of logic programs. It not only is useful in the problem domains that motivated its birth, such as program analysis, parsing, deductive database, and theorem proving, but also has been found essential in several other problem domains such as model checking, learning, and data mining.

Early resolution mechanisms proposed for tabling such as OLDT rely on suspension and resumption of subgoals to compute fixpoints. Recently, a new resolution framework called linear tabling, envisioned by the proposer and several other researchers, has received considerable attention because of its simplicity, ease of implementation, and good space efficiency. The idea of linear tabling is to use depth-first iterative deepening rather than suspension to compute fixpoints. Linear tabling is still immature compared with OLDT and a great of potential remains to be exploited. The objective of this project is to qualitatively and quantitatively analyze possible strategies and propose effective optimization techniques to make it sustainable to large applications such as natural language and data mining applications.

**About My Collaborator:** Yi-Dong Shen is a Professor at the Software Institute of the Chinese Academy of Sciences in Beijing, a leading institution in China. He is an active researcher in logic programming theories and has published a number of papers in top-ranked journals such as Theoretical Computer Science, Theory and Practice of Logic Programming, New Generation Computing, and ACM Transactions on Computational Logic. I first met Prof. Dr. Shen in 1998 while both of us were visiting the Computing Science department of the University of Alberta, and have had opportunities to collaborate with him on the early tabling mechanisms. His research experience and expertise in logic programming theories and his enthusiasm in the research in tabling and data mining fit perfectly to the scope of the proposed research.

**Scientific Merits:** This research will strive to make both theoretical and practical contributions. The soundness and completeness of all the optimization techniques will be proved and the techniques will be implemented and experimentally compared. The results will be published in conference proceedings and journals, and the resulting system will be disseminated broadly to students, researchers, and users in industry.

**Broader Impacts:** Dr. Shen has a research group at the Software Institute and the PI has students including both undergraduate and PhD students working on applying tabled...
Prolog to bioinformatics, heuristics-learning for constraint solving, and protocol verification. The cooperation will enhance the connection between our research groups, help establish a formal relationship between CUNY Computer Science and the Software Institute, and also enhance the involvement of students in the application projects.

1 Introduction

1.1 Background

Recently there has been a growing interest of research in tabling because of its usefulness in a variety of application domains including program analysis, parsing, deductive database, theorem proving, model checking, and logic-based probabilistic learning [7, 11, 14, 17, 20, 23, 28, 29, 33]. The main idea of tabling is to memorize the answers to some subgoals and use the answers to resolve subsequent variant subgoals. This idea of caching previously calculated solutions, called memoization, was first used to avoid redundant evaluation of functions [13], and later was used to deal with left-recursion and redundancy in top-down descent parsers [11, 16, 29]. Tabling brings several powerful features of bottom-up evaluation such as completeness and termination into goal-directed top-down evaluation. Tabling has become a practical technique thanks to the availability of large amounts of memory in computers. It has become an embedded feature in a number of other logic programming systems such as ALS [9], B-Prolog [30, 35, 32, 34], Mercury, XSB [22], and YAP [21].

OLDT [25] is the first revised SLD resolution that accommodates the idea of tabling. In OLDT, a table area is used to record subgoals and their answers. When a subgoal (producer) is first encountered in execution, it is resolved by using program clauses just as in SLD resolution with the exception that the subgoal and its answers are recorded in the table. When a subgoal (consumer) is encountered that is subsumed by one of its ancestors, OLDT does not expand it as in SLD resolution but rather uses the answers in the table to resolve it. After the answers are exhausted, the computation of the consumer is suspended until the producer produces new answers into the table. The process is continued until the fixpoint is reached, i.e. when no answers are available for consumers and no producers can produce any new answers. Several other resolution formalisms including SLG [4] and SLS [3] have been developed for tabling that rely on suspension and resumption of subgoals to compute fixpoints. XSB is the first Prolog system that successfully supports tabling based on this idea [22].

OLDT is non-linear in the sense that the state of a consumer must be preserved before execution backtracks to its producer. This non-linearity requires freezing stack segments [22] or copying stack segments into a different area [6] before backtracking.
takes place.

Recently, another formalism, called linear tabling\textsuperscript{1}, has become an effective alternative tabling mechanism [24, 36, 9]. The main idea of linear tabling is to use depth-first iterative deepening rather than suspension to compute fixpoints. A significant difference between linear tabling and OLDT lies in the handling of variant descendents of a subgoal. In linear tabling, after a descendent consumes all the answers, it either fails or turns into a producer, producing answers by using the alternative clauses of the ancestor. A subgoal is called a \textit{looping subgoal} if a variant occurs in its evaluation. The evaluation of looping subgoals must be iterated to ensure the completeness of evaluation.

Linear tabling is relatively easy to implement on top of a WAM-like abstract machine thanks to its linearity. Linear tabling is more space efficient than OLDT since the states of subgoals need not be preserved. Nevertheless, linear tabling may be slower than OLDT because of iteration of looping subgoals, especially when programs contain a large number of interdependent recursive predicates. Two Prolog systems, namely B-Prolog [30, 31] and ALS [9], have been extended to support linear tabling.

1.2 Motivation and Objective

Prolog with Tabling has found its ways into many application areas ranging from deductive database, program analysis, natural language processing, model checking, to data mining and learning. We have been developing two application systems using our B-Prolog system: a protocol verifier and a statistical learning system called PRISM [33]. We have been applying the PRISM system to several machine learning tasks such as biosequence analysis and heuristics-learning for constraint solving. Efficient fixpoint computation is of crucial importance for our protocol verifier to find pitfalls in subtle and complex protocols, and for the learning system to learn probability distributions from large amounts of sample data.

Linear tabling is a framework from which different methods can be derived: each is a combination of strategies for handling looping subgoals in forward execution, backtracking, and iteration. For example, for the forward execution of a looping subgoal that is a descendent of a variant ancestor, one strategy is to fail the subgoal after it consumes all the available answers, and another strategy is to let the descendent produce answers by using the alternative clauses of the ancestor. The decisions on how to handle looping subgoals in forward execution, backtracking, and iteration are orthogonal, and different methods can be designed by combining different strategies.

\textsuperscript{1}Notice that the word \textit{linear} here has nothing to do with complexity or liner logic. It has the same meaning as \textit{L} in SLD: a derivation is made up of a sequence of goals $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_k$ such that $G_{i+1}$ is derived from $G_i$. 
The objective of this project is to qualitatively and quantitatively analyze possible strategies.

Current tabling systems incur considerable overhead to the execution of programs. Tamaki and Sato states the following in [25]: “The storage requirement [of tabling] can be too demanding in some cases and the overhead of table manipulation can be too large.” This situation has not changed since. The second objective of this project is to develop optimization techniques to tackle this problem. In concrete, we will investigate the following optimization techniques: (1) deferred tabling, deferring tabling to reduce the overhead of over-tabled programs; (2) semi-naive evaluation, an algorithm employed in bottom-up evaluation of Datalog programs [27] to avoid redundant joins; and (3) controlling search in tabled Prolog.

2 Prior Research Results on Tabling

The SLD resolution used in Prolog may not be complete or efficient for programs in the presence of recursion. For example, for a recursive definition of the transitive closure of a relation, a query may never terminate under SLD resolution if the program contains left-recursion or the graph represented by the relation contains cycles even if no rule is left-recursive. For a natural definition of the Fibonacci function, the evaluation of a subgoal under SLD resolution spawns an exponential number of subgoals, many of which are variants. The lack of completeness and efficiency in evaluating recursive programs is problematic: novice programmers may lose confidence in writing declarative programs that terminate and real programmers have to reformulate a natural and declarative formulation to avoid these problems, resulting in less declarative and less readable programs.

Tabling [25, 28] is a technique that can get rid of infinite loops for bounded-term-size programs and redundant computations in the execution of recursive Prolog programs. The main idea of tabling is to memorize the answers to some subgoals and use the answers to resolve subsequent variant subgoals. This idea of caching previously calculated solutions, called memoization, was first used to speed up evaluation of functions [13].

Tabling in Prolog not only is useful in the problem domains that motivated its birth, such as program analysis, parsing, deductive database and theorem proving but also has been found essential in several other problem domains such as model checking [20] and probabilistic logic learning [19, 23, 33].

OLDT [25] is the first revised SLD resolution that accommodates the idea of tabling. In OLDT, a table area is used to record subgoals and their answers. When a subgoal (producer) is first encountered in execution, it is resolved by using program clauses just as in SLD resolution with the exception that the subgoal and its answers
are recorded in the table. When a subgoal (consumer) is encountered that is subsumed
by one of its ancestors, OLDT does not expand it as in SLD resolution but rather
uses the answers in the table to resolve it. After the answers are exhausted, the
computation of the consumer is suspended until the producer produces new answers
into the table. The process is continued until the fixpoint is reached, i.e. when no
answers are available for consumers and no producers can produce any new answers.
Several extensions of OLDT including SLG [4] and SLS [3] have been developed for
evaluating general logic programs with negation. XSB is the first Prolog system that
successfully supports tabling [22].

OLDT is non-linear in the sense that the state of a consumer must be preserved
before execution backtracks to its producer. This non-linearity requires freezing stack
segments [22] or copying stack segments into a different area [6] before backtracking
takes place. Recently, another formalism, called linear tabling\(^2\), has emerged as an
alternative tabling method [24, 35, 9]. The main idea of linear tabling is to use
iterative computation rather than suspension to compute fixpoints. A significant
difference between linear tabling and OLDT lies in the handling of variant descendents
of a subgoal. In linear tabling, after a descendent consumes all the answers, it either
fails or turns into a producer, producing answers by using the alternative clauses of
the ancestor. A subgoal is called a looping subgoal if a variant occurs as a descendent
in its evaluation. The evaluation of looping subgoals must be iterated to ensure the
completeness of evaluation.

Linear tabling is a framework from which different methods can be derived based
on the strategies used in handling looping subgoals in forward execution, backtracking,
and iteration. The framework can be defined by three primitives on tabled subgoals:

\(\text{start}(A), \text{memo}(A), \text{and check}\_\text{completion}(A)\).

- \(\text{start}(A)\) This primitive is executed when a tabled subgoal \(A\) is encountered.
The subgoal \(A\) is registered into the table if it is not registered yet. If \(A\)'s state
  is complete meaning that \(A\) has been completely evaluated before, then \(A\) is
  resolved by using the answers in the table.

  If \(A\) is a pioneer of the current path, meaning that it is encountered for the first
time, then it is resolved by using program clauses.

  If \(A\) is a follower of some ancestor \(A_0\), meaning that a loop has been encountered,
then it is first resolved by using the answers in the table. After the answers are
exhausted, two possible actions can be taken: One is to fail \(A\). This strategy
is called \(FDF\) (Fail Descendent Followers). The other is to resolve \(A\) by using

\(^2\)Notice that the word \emph{linear} here has nothing to do with complexity or linear logic. It has the
same meaning as \(L\) in SLD: a derivation is made up of a sequence of goals \(G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_k\)
such that \(G_{i+1}\) is derived from \(G_i\).
the alternative clauses of its ancestor $A_0$. This strategy is called SAC (Steal Alternative Clauses).

- **memo($A$)** This primitive is executed when an answer is found for the tabled subgoal $A$. If the answer $A$ is already in the table, then fail; otherwise the answer is added into the table. Two different strategies can be used after the answer is added. One is called *eager* consumption strategy, which lets the subgoal succeed and consume the answer. The other strategy is called *lazy* consumption strategy, which fails the primitive and forces the system to backtrack to find the next answer.

- **table_completion($A$)** This primitive is executed when the subgoal $A$ is being resolved by using program clauses and all the clauses have been tried. If $A$ has never occurred in a loop, then $A$’s state can be set to *complete* and $A$ can be failed after all the answers are consumed. If $A$ is a top-most looping subgoal, then different actions should be taken depending on whether this is $A$’s normal evaluation or re-evaluation. These actions depend again on what strategies are used by **start($A$)** and **memo($A$)**. In any case, if no new answer is produced in the last round of re-evaluation of $A$, then $A$’s state can be set to *complete* and $A$ can be failed after all the answers are consumed.

For a pioneer in a loop that is not a top-most looping subgoal, two different strategies can be used after the subgoal exhausts all the clauses and answers. One is to fail the subgoal. This means that under this strategy re-evaluation starts only at top-most looping subgoals. The other strategy is to iterate the evaluation of the subgoal until it reaches a temporary fixpoint. The fixpoint is temporary since the top-most subgoal on which this subgoal depends has not been completely evaluated.

Linear tabling is relatively easy to implement on top of a WAM-like abstract machine thanks to its linearity. Linear tabling is more space efficient than suspension-based methods since the states of subgoals need not be preserved. In [32] and [34], we have investigated the lazy consumption strategy and proposed several optimization techniques including *subgoal optimization* and *semi-naive optimization* that can significantly improve the performance.

### 3 Proposed Research

As described above, linear tabling is a framework from which different methods can be derived: each is a combination of strategies for handling looping subgoals in forward
OISE: Zhou

execution, backtracking, and iteration. For example, for the forward execution of a looping subgoal that is a descendent of a variant ancestor, one strategy is to fail the subgoal after it consumes all the available answers, and another strategy is to let the descendent produce answers by using the alternative clauses of the ancestor. The decisions on how to handle looping subgoals in forward execution, backtracking, and iteration are orthogonal, and different methods can be designed by combining different strategies. Different strategies, called scheduling strategies, have been studied in the context of OLDT [8], but no work has been done in linear tabling. One of the objectives of this project is to qualitatively and quantitatively analyze possible strategies and perfect a method that is most efficient in terms of both space and time.

No tabling method could lead to an efficient system without optimization. Another objective of this project is to investigate several optimization techniques. We will also be investigating the method for handling cuts under various kinds of strategies.

3.1 Deferred tabling

Current tabling systems incur significant overhead to the execution of programs. Tamaki and Sato states the following in [25]: “The storage requirement [of tabling] can be too demanding in some cases and the overhead of table manipulation can be too large.” This situation has not much changed since. For example, for the naive reverse program both XSB and B-Prolog slow down by over 30 times if the predicates are declared tabled.

The overhead problem of tabling is a common problem to all current tabling systems. We propose a technique, called deferred tabling to tackle this problem.

In current tabling systems, a table-declared subgoal is tabled the first time when it is encountered regardless of whether the answers can be reused or not. For a predicate for which no subgoal occurs twice, tabling it is a waste. For a predicate for which even if some subgoals may occur multiple times, most subgoals may occur only once. Tabling those single-occurring subgoals is a waste. This technique amounts to defer tabling a subgoal until a sufficiently close subgoal occurs again in execution.

The deferred tabling technique is analogous to the buffering technique used in operating systems. One of the key components in the design of this technique is to define the closeness of subgoals. We will rely on subgoal abstraction to check the closeness of subgoals and explore several definitions for subgoal abstraction.

3.2 Semi-naive evaluation in linear tabling

Linear tabling relies on iterative evaluation of top-most looping subgoals to compute fixpoints. Blind re-computation of all subgoals and clauses is not computationally
acceptable. A system should re-evaluate only those subgoals and should use only those clauses and answers that can contribute to the generation of new answers.

We have proposed a framework for incorporating the semi-naive evaluation into linear tabling [34]. The semi-naive algorithm used in bottom-up evaluation for Datalog programs [2, 27] can be used to avoid redundant joins in linear tabling. In concrete, in each round of evaluation, the join of the subgoals in the body of each rule must involve at least one new answer produced in the previous round. Let “H:−A_1, ..., A_k, ..., A_n” be a clause where A_k is the last subgoal in the body that may depend on H (i.e., all the subgoals to the right of A_k that have occurred in early rounds must be already complete when H is re-evaluated). For each combination of the answers of A_1, ..., and A_{k−1}, if the combination does not contain any new answers then A_k should consume new answers only.

In semi-naive evaluation, answers can be consumed either *sequentially* or *incrementally*. In sequential consumption, for each subgoal either all the available answers or only new answers are consumed. In *incremental consumption*, in contrast, answers produced in a round are not consumed until the next round. These two schemes need to be thoroughly investigated, and the conditions for them to be complete need to be identified and proved.

The semi-naive algorithm, once fully incorporated, will have as great an impact on linear tabling as it has on bottom-up evaluation. With this algorithm, linear tabling can constantly beat the magic-sets method [27] since instantiation information of goals are passed down automatically in top-down evaluation and linear tabling enjoys far better space efficiency than the OLDT and bottom-up evaluation methods.

### 3.3 Single-solution search and cuts

The lazy consumption strategy is suited for finding all answers. For certain applications such as planning and theorem proving it is unreasonable to find all answers either because the set is infinite or because the search space is huge. The cut operator in Prolog can be used to prune unnecessary branches in a search tree. Nevertheless, under the lazy consumption strategy the goal “p(X),!,q(X)” produces all the answers for p(X) even though only one is needed.

At first glance, the eager consumption strategy seems to be amenable to cuts, but it causes other problems. In particular, more re-computation may be needed to reach fixpoints. For instance, in the goal “p(X),p(Y)” p(Y) may be started before p(X) is complete under the eager consumption strategy, and for each instance of p(X) the join “p(X), p(Y)” needs to be performed more than once to guarantee that no solution is lost. The effect can be disastrous if there is a heavy computation to be done between these two variant subgoals.

We propose a novel and effective approach to dealing with cuts. We first replace
each chunk of subgoals $A$ that resides in the scope of a cut with $\text{once}(A)$ and generate a program from the original program for each such call of $\text{once}$. The key idea of the approach to implementing $\text{once}$ is to translate clauses into continuation-passing style (CPS) clauses. For a CPS program that is required to return only one solution, we can prove that continuations do not need to be tabled.

The transformation from a standard clause to a CPS clause is straightforward[26, 5]. In a CPS program, each subgoal takes an extra argument that explicitly represents the continuation computation of the subgoal. Let $H(\text{Cont})$ be the CPS atomic formula of $H$. The clause

$$H : -B_1, B_2, \ldots, B_n$$

is translated into the following CPS clause:

$$H(\text{Cont}) : -B_1(B_2(\ldots B_n(\text{Cont}))\ldots).$$

If the body of the original clause is empty, then body of the CPS clause calls the continuation:

$$H(\text{Cont}) : -\text{call}(\text{Cont}).$$

CPS binary programs have the following property: A solution is found for a query if any clause in the transformed CPS predicates succeeds. For a call of $\text{once}$, a solution should be returned immediately after it is found.

### 3.4 An efficient tabling system for PRISM

Tabling is used in the construction of explanation graphs for observed samples in PRISM [23, 33]. Using tabling resembles using dynamic programming in the machine learning algorithms such as the Baum-Welch algorithm for the HMM [18] and the Inside-Outside algorithm for PCFG [1]. The graphical EM learning algorithm adopted in PRISM first finds an explanation graph and then repeatedly walks it until it converges. Therefore, a compact and efficient representation for explanation graphs is crucially important. To this end, we propose two techniques, namely auto-tabling and graph compression.

The auto-tabling technique amounts to devising a standard for automatically tabling predicates. It is daunting for programmers to declare what predicates need to be tabled. Users’ declarations can be inaccurate. Programs may slow down significantly because of over-tabling or may fall into infinite loops because some predicates are not tabled. Furthermore, table declarations affect directly the structure of the resulting explanation graph. Therefore, such a standard should take the following into consideration: (1) the overhead of over-tabling, (2) the cost of re-computation...
because of lack of tabling; (3) the possibility of infinite loops; and (4) the compactness of the resulting explanation graph.

The resulting explanation graph from a program may not be optimal even if a wise standard is used in selecting table predicates. The graph compression technique reduces the size of the graph further by eliminating redundant nodes and edges, and by combing possible subgraphs.

4 Scientific Merits

The result of this project will be an efficient linear tabling method that is an optimal combination of possible strategies. The method will be implemented in our B-Prolog system and will be made widely available to the public. Our protocol verification and statistical learning systems will be used to evaluate different methods in addition to the de facto standard benchmark suite. Our application programs will be the direct beneficiaries of the research result. B-Prolog is being used by thousands of users in a number of projects including several ones that require tabling. For example, Oege de Moor of Oxford University is using tabled Prolog in a program analysis project and Douglas R. Miles of Teknowledge, a Stanford University based AI company, is using tabled Prolog in a natural language processing project. All these projects will benefit from our research result.

Our research will also have much broader impact on logic programming and deductive database research. As far as logic programming is concerned, the need to extend Prolog to narrow the gap between declarative and procedural readings of programs has been urged long before [15]. Although tabling has been perceived as necessary to narrow the gap, it has not become a standard feature. One primary reason may be the lack of an easy-to-implement and efficient tabling method. Our tabling method will accelerate the acceptance of tabling as a standard feature of logic programming. Deductive database used to be a very hot research area. A number of bottom-up evaluation methods have been invented for evaluating recursive Datalog rules [2, 10, 27], and several systems have been developed [12]. Deductive database was expected to replace relational database as the next generation database model. Nevertheless, deductive database research failed to meet the early expectations. One reason for this failure is that Datalog does not provided the flexibility needed in the representation of data and control in many application domains. Our system does not suffer from this limitation. Our linear tabling method has the potential to revitalize the research and development of deductive database systems and applications.

The lack of completeness and efficiency in evaluating non-tabled recursive programs is problematic: novice programmers may lose confidence in writing declarative programs that terminate and real programmers have to reformulate a natural and
declarative formulation to avoid these problems, resulting in less declarative and less readable programs. Therefore, our research will not only enhance the productivity of logic programming but also improve the friendliness of logic programming to beginners.

5 Broader Impacts of Research

Our research will also have broad impacts on society and education. CUNY is striving to establish its reputation as a leading research institution in the region, and some programs, e.g. The Honors Program, are attracting some of the best computer science students in the greater New York region, especially from minority groups. The PI has students including both undergraduate and PhD students working on applying tabled Prolog to bioinformatics, heuristics-learning for constraint solving, and protocol verification. Our research will seek to involve and develop the academic skills of the students including minority students. Our research results will be disseminated through several channels to society, one of which is the CUNY Institute for Software Development (CISDD). CISDD serves as a hub that connects software industries and institutions in the greater New York region. We will make use of office space for students and visitors provided by CISDD and through CISDD establish a close relationship with industry.

The Software Institute of the Chinese Academy of Sciences is a leading institution in China. The cooperation between Dr. Shen and the PI will enhance the connection between our research groups, and help establish a formal relationship between CUNY Computer Science and the Software Institute.

6 A Schedule of Planned Travel

It transpires from the above that our research objectives require both theoretical proofs and experimentation, and the cooperation between Prof. Yi-Dong Shen, a theorist, and the PI, a system architect, can help advance the research. Our cooperation will be carried out in various ways. We will make frequent use of the Internet to exchange our ideas and developments on the subject. We will also visit each other at least once a year to work closely on diverse topics of mutual interest.

7/1/2005-7/15/2005: Visit Prof. Shen at the Institute of Software of the Chinese Academy of Sciences in Beijing to discuss optimization techniques and prove their completeness.
1/2/2006-1/15/2006: Visit Prof. Shen to discuss the possibility of having a set of conditions for semi-naive evaluation to achieve the full effect of the semi-naive algorithm used in bottom-up evaluation. Investigate possible methods for controlling search in tabled programs.

5/1/2006-5/7/2006: Host the visit of Prof. Shen at CUNY. Discuss the implementation status of the optimization techniques. Invite him to give a talk at the CUNY Computer Science Colloquium.

12/15/2006-12/30/2006: Visit Prof. Shen to finalize some research papers.

7 Experience and Capabilities of the PI

The PI, Dr. Neng-Fa Zhou, has been an active researcher in programming language systems for over ten years. He has authored over thirty papers on programming language, constraint-solving, graphics, and machine learning systems including over fifteen papers published in top journals (ACM TOPLAS, Journal of Logic Programming, Theory and Practice of Logic Programming, Journal of Functional and Logic Programming, and Software Practice and Experience) and major conferences. His papers on compilation of logic programs, constraint solving, and tabling have received a number of citations. He is the main developer of the B-Prolog system, a cutting-edge constraint logic programming system which has tens of thousands users worldwide in both academia and industry.
References


