Linear Tabling Strategies and Optimizations

NENG-FA ZHOU
CUNY Brooklyn College & Graduate Center
TAISUKE SATO
Tokyo Institute of Technology
and
YI-DONG SHEN
Chinese Academy of Sciences

Recently there has been a growing interest of research in tabling in the logic programming community 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. The main idea of tabling is to memorize the answers to some subgoals and use the answers to resolve subsequent variant subgoals. Early resolution mechanisms proposed for tabling such as OLDT and SLG rely on suspension and resumption of subgoals to compute fixpoints. Recently, a fundamentally different resolution framework called *linear tabling*, envisioned by the authors and some other researchers, has received considerable attention because of its simplicity, ease of implementation, and good space efficiency. Linear tabling is a framework from which different methods can be derived based on the strategies used in handling looping subgoals. One decision concerns when answers are consumed and returned. This paper describes two strategies, namely, *lazy* and *eager* strategies, and compare them both qualitatively and quantitatively. The results indicate that, while the lazy strategy has good locality and is well suited for finding all solutions, the eager strategy is comparable in speed with the lazy strategy and is well suited for programs with cuts. Linear tabling relies on depth-first iterative deepening rather than suspension to compute fixpoints. Each cluster of inter-dependent subgoals as represented by a top-most looping subgoal is iteratively evaluated until no subgoal in it can produce any new answers. Naive re-evaluation of all looping subgoals, albeit simple, may be computationally unacceptable. In this paper, we introduce semi-naive optimization, an effective technique employed in bottom-up evaluation of logic programs to avoid redundant joins of answers, into linear tabling. We give the conditions for the technique to be safe (i.e. sound and complete) and propose an optimization technique called *early answer promotion* to enhance its effectiveness. Benchmarking in B-Prolog demonstrates that with this optimization liner tabling compares favorably well in speed with the state-of-the-art implementation of OLDT.

The preliminary results of this article appear in ACM PPDP'03 and PPDP'04.
Neng-Fa Zhou is supported in part by RF-CUNY and CUNY Software Institute, Taisuke Sato is supported in part by CREST, and Yi-Dong Shen is supported in part by the National Natural Science Foundation of China.
Authors' addresses: N.F. Zhou, Department of Computer and Information Science, CUNY Brooklyn College,2900 Bedford Avenue, Brooklyn, NY 11210-2889; email:zhou@sci.brooklyn.cuny.edu; T. Sato, Department of Computer Science, Tokyo Institute of Technology, 2-12-1 Ookayama Meguro-ku Tokyo Japan 152-8552; email:sato@cs.titech.ac.jp; Y.D. Shen, Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences in Beijing, China; email: ydshen@ios.ac.cn.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Prolog, Semi-naive optimization, Abstract machine, Recursion, Tabling, Memoization, Linear tabling

1. INTRODUCTION

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 cluttered programs.

Tabling [Tamaki and Sato 1986; Warren 1992] is a technique that can get rid of infinite loops for bounded-term-size programs and redundant computations in the execution of recursive 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 [Dawson et al. 1996], parsing [Eisner et al. 2004; Johnson 1995; Warren 1999], deductive database [Liu 1999; Ramakrishnan and Ullman 1995; Sagonas et al. 1994], and theorem proving [Nielson et al. 2004; Pientka 2003], but also has been found essential in several other problem domains such as model checking [Ramakrishnan 2002] and logic-based probabilistic learning[Sato and Kameya 2001; Zhou et al. 2003]. This idea of caching previously calculated solutions, called *memoization*, was first used to speed up the evaluation of functions [Michie 1968]. OLDT [Tamaki and Sato 1986] is the first resolution mechanism that accommodates the idea of tabling in logic programming and XSB is the first Prolog system that successfully supports tabling [Sagonas and Swift 1998]. 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 [Guo and Gupta 2001], B-Prolog [Zhou et al. 2000; Zhou et al. 2004], Mercury, and YAP [Rocha et al. 2005].

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 [Sagonas and Swift 1998] or copying stack segments into a different area [Demoen and Sagonas 1999] before backtracking takes place. Linear tabling is an alternative effective tabling scheme [Shen et al. 2001; Zhou et al. 2000; Zhou and Sato 2003; Zhou et al. 2004]. The main idea of linear tabling is to use iterative computation of looping subgoals rather than suspension and resumption of them as is done in OLDT to compute fixpoints. The DRA method proposed in [Guo and Gupta 2001] is based on the same idea but employs different strategies for handling looping subgoals and clauses. In linear tabling, a cluster of inter-dependent subgoals as represented by a *top-most looping subgoal* is iteratively evaluated until no subgoal in it can produce any new answers. Linear tabling is relatively easy to implement on top of a stack machine thanks to its linearity, and is more space efficient than OLDT since the states of subgoals need not be preserved.

Linear tabling is a framework from which different methods can be derived based on the strategies used in handling looping subgoals. One decision concerns when answers are consumed and returned. The *lazy* strategy postpones the consumption of answers until no answers can be produced. It is in general space efficient because of its locality and is well suited for all-solution search programs. The *eager* strategy, in contrast, prefers answer consumption and return to production. It is well suited for programs with cuts. This paper gives a comprehensive analysis of these two strategies and compares their performance experimentally.

Linear tabling relies on iterative evaluation of top-most looping subgoals to compute fixpoints. Naive re-evaluation of all looping subgoals may be computationally expensive. *Semi-naive optimization* is an effective technique used in bottom-up evaluation of Datalog programs [Bancilhon and Ramakrishnan 1986; Ullman 1988]. It avoids redundant joins by ensuring that the join of the subgoals in the body of each rule must involve at least one new answer produced in the previous round. The impact of semi-naive optimization on top-down evaluation had been unknown before [Zhou et al. 2004]. In this paper, we also propose to introduce semi-naive optimization into linear tabling. We have made efforts to properly tailor semi-naive optimization to linear tabling. In our semi-naive optimization, answers for each tabled subgoal are divided into three regions as in bottom-up evaluation, but answers are consumed sequentially not incrementally so that answers produced in a round are consumed in the same round. We have found that incremental consumption of answers does not fit linear tabling since it may require more rounds of iteration to reach fixpoints. Nevertheless, consuming answers incrementally may cause redundant consumption of answers. We further propose a technique called *early promotion* of answers to reduce redundant consumption of answers. Our benchmarking shows that this technique gives significant speed-ups to some programs.

An efficient tabling system has been implemented in B-Prolog,[1] in which the lazy strategy is employed by default but the eager strategy can be used for subgoals that are in the scopes of cuts or are not required to return all the answers. Our tabling system not only consumes an order of magnitude less stack space than XSB for some programs but also compares favorably well with XSB in speed.

The remainder of the paper is structured as follows: In the next section we define the terms used in this paper. In Section 3 we give the linear tabling framework and the two answer consumption strategies. In Section 4 we introduce semi-naive optimization into linear tabling and prove its completeness. In Section 5 we describe the implementation of our tabling system and also show how to implement cuts.

[1] www.bprolog.com

In Section 6 we compare the tabling strategies experimentally, evaluate the effectiveness of semi-naive optimization, and also compare the performance of B-Prolog with XSB. In Section 7 we survey the related work and in Section 8 we conclude the paper.

2. PRELIMINARIES

In this section we give the definitions of the terms to make this paper as much self-contained as possible. The reader is referred to [Lloyd 1988] for a description of SLD resolution.

Let P be a program. Tabled predicates in P are explicitly declared and all the other predicates are assumed to be non-tabled. A subgoal of a tabled predicate is called a *tabled subgoal*. Tabled predicates are transformed into a form that facilitates execution: each rule ends with a dummy subgoal named $memo(H)$ where H is the head, and each tabled predicate contains a dummy ending rule whose body contains only one subgoal named *check_completion(H)*. For example, given the definition of the transitive closure of a relation,

```
p(X,Y):-p(X,Z),e(Z,Y).
p(X,Y):-e(X,Y).
```

The transformed predicate is as follows:

```
p(X,Y):-p(X,Z),e(Z,Y),memo(p(X,Y)).
p(X,Y):-e(X,Y),memo(p(X,Y)).
p(X,Y):-check_completion(p(X,Y)).
```

A table is used to record subgoals and their answers. For each subgoal and its variants, there is an entry in the table that stores the state of the subgoal (e.g., complete or not) and an answer table for holding the answers generated for the subgoal. Initially, the answer table is empty.

Definition 1. Let $G = (A_1, A_2, ..., A_k)$ be a goal. The first subgoal A_1 is called the *selected subgoal* of the goal. G' is *derived* from G by using a tabld *answer* F if there exists a unifier θ such that $A_1\theta = F$ and $G' = (A_2, ..., A_k)\theta$. G' is *derived* from G by using a rule "$H : -B_1, ..., B_m$" if $A_1\theta = H\theta$ and $G' = (B_1, ..., B_m, A_2, ..., A_k)\theta$. A_1 is said to be the *parent* of B_1, ..., and B_m. The relation *ancestor* is defined recursively from the parent relation.

Definition 2. A tabled subgoal that occurs first in the construction of an SLD tree is called a *pioneer*, and all subsequent variants are called *followers* of the pioneer. Let G_0 be a given goal, and $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ be a *derivation* where each goal is derived from the goal immediately preceding it. Let $G_i \Rightarrow \ldots \Rightarrow G_j$ be a sub-sequence of the derivation where $G_i = (A...)$ and $G_j = (A'...)$. The sub-sequence forms a *loop* if A and A' are variants. The subgoals A and A' are called *looping subgoals*. In particular, A is called the *pioneer looping subgoal* and A' is called the *follower looping subgoal* of the loop.

Notice that the pioneer and follower looping subgoals are not required to have the ancestor-descendent relationship, and thus a loop may not be a real loop [Shen et al. 2001]. Consider, for example, the goal "$p(X), p(Y)$" where p is defined by facts.

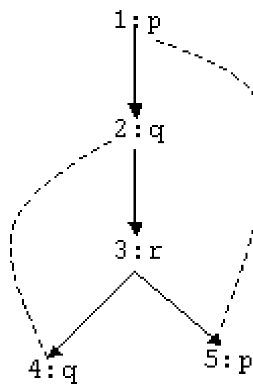

Fig. 1. A top-most looping subgoal.

The derivation "$p(X), p(Y)$" $\Rightarrow$ $p(Y)$ is treated as a loop although the selected subgoal $p(Y)$ in the second goal is not a descendant of $p(X)$.

Definition 3. A subgoal A is said to be *dependent* on another subgoal A' if A' occurs in a derivation of A. Two subgoals are said to be *inter-dependent* if they occur in each other's derivations. Inter-dependent subgoals constitute a *cluster*. A subgoal in a cluster is called the *top-most* subgoal of the cluster if none of its ancestors is included in the cluster.

Unless a cluster contains only a single subgoal, its top-most subgoal must also be a looping subgoal. For example, the subgoals at the nodes in the SLD tree in Figure 1 constitute a cluster and the subgoal `p` at node 1 is the top-most looping subgoal of the cluster.

3. LINEAR TABLING AND ANSWER CONSUMPTION STRATEGIES

Linear tabling takes a transformed program and a goal, and tries to find a path in the SLD tree that leads to an empty goal. The primitive *table_start*(A) is executed when a tabled subgoal A is encountered. Just as in SLD resolution, linear tabling explores the SLD tree in a depth-first fashion, taking special actions when *table_start(A)*, *memo(A)*, and *check_completion(A)* are encountered. Backtracking is done in exactly the same way as in SLD resolution. When the current path reaches a dead end, meaning that no action can be taken on the selected subgoal, execution backtracks to the latest previous goal in the path and continues with an alternative branch. When execution backtracks to a top-most looping subgoal of a cluster, however, we cannot fail the subgoal even after all the alternative clauses have been tried. In general, the evaluation of a top-most looping subgoal must be iterated until its fixpoint is reached. We call each iteration of a top-most looping subgoal a *round*.

Various linear tabling methods can be devised based on the framework. A linear tabling method comprises strategies used in the three primitives: *table_start(A)*, *memo(A)*, and *check_completion(A)*. In linear tabling, a pioneer subgoal has two roles: one is to produce answers into the table and the other is to return answers to its parent through its variables. Different strategies can be used to produce and return answers. The *lazy strategy* gives priority to answer production and the *eager strategy* prefers answer consumption to production. In the following we define the three primitives in both strategies.

3.1 The lazy strategy

The lazy strategy postpones the consumption of answers until no answers can be produced. In concrete, for top-most looping subgoals no answer is returned until they are complete, and for other pioneer subgoals answers are consumed only after all the rules have been tried.

3.1.1 *table_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, meaning that it is encountered for the first time in the current path, then different actions are taken depending on A's state. If A's state is *visited* meaning that A has occurred before in a different path during the current round, then it is resolved by using answers. Otherwise, if A has never occurred before during the current round, it is resolved by using rules. In this way, a pioneer subgoal needs to be evaluated only once in each round.

If A is a follower of some ancestor A_0, meaning that a loop has been encountered,[2] then it is resolved by using the answers in the table. After the answers are exhausted, A fails. Failing A is unsafe in general since it may have not returned all of its possible answers. For this reason, the top-most looping subgoal of the cluster of A needs be iterated until no new answer can be produced.

3.1.2 *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 just fail; otherwise fail after the answer is added into the table. The failure of *memo* postpones the return of answers until all rules have been tried.

3.1.3 *check_completion*(A). This primitive is executed when the subgoal A is being resolved by using rules and the dummy ending rule is being tried. If A has never occurred in a loop, then A's state is set to *complete* and A is failed after all the answers are consumed.

If A is a top-most looping subgoal, we check if any new answers are produced during the last round of evaluation of the cluster under A. If so, A is re-evaluated by calling *table_start*(A) after all the dependent subgoals's states are initialized. Otherwise, if no new answer is produced, A is resolved by using answers after its state and all its dependent subgoals' states are set to *complete*. Notice that a top-most looping subgoal does not return any answers until it is complete.

If A is a looping subgoal but not a top-most one, A will be resolved by using answers after its state is set to *visited*. Notice that A's state cannot be set to *complete* since A is contained in a loop whose top-most subgoal has not been completely

[2]As to be discussed later, A_0 must be an ancestor of A under the lazy strategy.

evaluated. For example, in Figure 1, `q` reaches its fixpoint only after the top-most looping subgoal `p` reaches its fixpoint.

As described in the definition of *table_start*(A), a *visited* subgoal is never evaluated using rules again in the same round. This optimization is called *subgoal optimization* in [Zhou and Sato 2003]. If evaluating a subgoal produces some new answers then the top-most looping subgoal will be re-evaluated and so will the subgoal; and if evaluating a subgoal does not produce any new answer, then evaluating it again in the same round would not produce any new answers either. Therefore, the subgoal optimization is safe.

3.1.4 *Example.* Consider the following program, where `p/2` is tabled, and the query `p(a,Y0)`.

```
p(X,Y):-p(X,Z),e(Z,Y),memo(p(X,Y)). (p1)
p(X,Y):-e(X,Y),memo(p(X,Y)).        (p2)
p(X,Y):-check_completion(p(X,Y)).   (p3)

e(a,b).
e(b,c).
```

The following shows the steps that lead to the production of the first answer:

```
1: p(a,Y0)
     ⇓apply p1
2: p(a,Z1),e(Z1,Y0),memo(p(a,Y0))
     loop found, backtrack to goal 1
1: p(a,Y0)
     ⇓ apply p2
3: e(a,Y0),memo(p(a,Y0))
     ⇓ apply e(a,b)
4: memo(p(a,b))
     ⇓ add answer p(a,b)
```

After the answer `p(a,b)` is added into the table, `memo(p(a,b))` fails. The failure forces execution to backtrack to `p(a,Y0)`.

```
1: p(a,Y0)
     ⇓ apply p3
5: check_completion(p(a,Y0))
```

Since `p(a,Y0)` is a top-most looping subgoal which has not been completely evaluated yet, `check_completion(p(a,Y0))` does not consume the answer in the table but instead starts re-evaluation of the subgoal.

```
1: p(a,Y0)
     ⇓apply p1
6: p(a,Z1),e(Z1,Y0),memo(p(a,Y0))
     ⇓use answer p(a,b)
7: e(b,Y0),memo(p(a,Y0))
     ⇓apply e(b,c)
8: memo(p(a,c))
```

When the follower p(a,Z1) is encountered this time, it consumes the answer p(a,b). The current path leads to the second answer p(a,c). On backtracking, the goal numbered 6 becomes the current goal.

```
6: p(a,Z1),e(Z1,Y0),memo(p(a,Y0))
   ⇓use answer p(a,c)
9: e(c,Y0),memo(p(a,Y0))
```

Goal 9 fails. Execution backtracks to the top goal and tries the clause p3 on it.

```
1: p(a,Y0)
   ⇓ apply p3
10: check_completion(p(a,Y0))
```

Since the new answer p(a,c) is produced in the last round of evaluation, the top-most looping subgoal p(a,Y0) needs to be evaluated again. The next round of evaluation produces no new answer and thus the subgoal's state is set to *complete*. After that the top-most subgoal returns the answers p(a,b) and p(a,c).

3.1.5 *Properties of the lazy strategy.* Under the lazy strategy, answers are not returned immediately after they are produced but are returned via the table. No answer is returned for a top-most looping subgoal until the subgoal is complete.

All loops are guaranteed to be real: for any loop $G_i = (A \ldots) \Rightarrow \ldots \Rightarrow G_j = (A' \ldots)$ where A and A' are variants, A must be an ancestor of A'. Because each cluster of inter-dependent subgoals is completely evaluated before any answers are returned to outside of the cluster, the lazy strategy has good locality and is thus suited for finding all solutions. For example, when the subgoal $p(Y)$ is encountered in the goal "p(X),p(Y)", the subtree for p(X) must have been explored completely and thus needs not be saved for evaluating p(Y).

The cut operator cannot be handled efficiently under the lazy strategy. The goal "$p(X), !, q(X)$" produces all the answers for $p(X)$ even though only one is needed.

The linear tabling framework inherits the main idea from [Shen et al. 2001], i.e., iterating the evaluation of top-most looping subgoals until no new answer is produced, and the lazy stratey adopted does not affect its soundness and completeness.

3.2 The eager strategy

The eager strategy prefers answer consumption and return to production. For a pioneer, answers are used first and rules are used only after all available answers are exhausted, and moreover a new answer is returned to its parent immediately after it is added into the table. The following describes how the three primitives behave under the eager strategy.

3.2.1 *table_start*(A). Just as in the lazy strategy, A is registered if it is not registered yet. A is resolved by using the tabled answers if A is complete or A is a follower of some former variant subgoal. If A is a pioneer, being encountered for the first time in the current round, it is resolved by using answers first, and then rules after all existing answers are exhausted.

3.2.2 *memo*(A). If the answer A is already in the table, then this primitive fails; otherwise, this primitive succeeds after adding the answer A into the table.

Notice that A is returned immediately after it is added into the table. If A is not new, then it must have been returned before.

3.2.3 *check_completion*(A). If A is a top-most looping subgoal, just as in the lazy strategy, we check whether any new answers are produced during the last round of evaluation of A. If so, A is evaluated again by calling *table_start*(A). Otherwise, if no new answer is produced, this primitive fails after A's and all its dependent subgoals' states are set to *complete*. If A is a looping subgoal but not a top-most one, this primitive fails after A's state is set to *visited*. A *visited* subgoal is never evaluated using rules again in the same round. Notice that unlike under the lazy strategy, the primitive *check_completion*(A) never returns any answers under the eager strategy. As described above, all the available answers must have been returned by *table_start*(A) and *memo*(A) by the time *check_completion*(A) is executed.

3.2.4 *Example.* Because of the need to re-evaluate a top-most looping subgoal, redundant solutions may be observed for a query. Consider, for example, the following program and the query "`p(X),p(Y)`".

```
p(1):-memo(p(1)).              (r1)
p(2):-memo(p(2)).              (r2)
p(X):-check_completion(p(X)). (r3)
```

The following derivation steps lead to the return of the first solution `(1,1)` for `(X,Y)`.

1: $\underline{\texttt{p(X)}}$`,p(Y)`
$\Downarrow$ use r1
2: $\underline{\texttt{memo(p(1))}}$`,p(Y)`
$\Downarrow$ add answer p(1)
3: $\underline{\texttt{p(Y)}}$
$\Downarrow$ loop found, use answer p(1)

When the subgoal `p(Y)` is encountered, it is treated as a follower and is resolved using the tabled answer `p(1)`. After that the first solution `(1,1)` is returned to the top query. When execution backtracks to `p(Y)`, it fails since it is a follower and no more answer is available in the table. Execution backtracks to `p(X)`, which produces and adds the second answer `p(2)` into the table.

1: $\underline{\texttt{p(X)}}$`,p(Y)`
$\Downarrow$ use r2
4: $\underline{\texttt{memo(p(2))}}$`,p(Y)`
$\Downarrow$ add answer p(2)
5: $\underline{\texttt{p(Y)}}$
$\Downarrow$ use answer p(1)

When `p(Y)` is encountered this time, there are two answers `p(1)` and `p(2)` in the table. So the next two solutions returned are `(2,1)` and `(2,2)`. When execution backtracks to goal 1, the dummy ending rule is applied.

```
1: p(X),p(Y)
   ⇓ use r3
6: check_completion(p(X)),p(Y)
```

Since new answers are added into the table during this round, the subgoal `p(X)` needs to be evaluated again, first using answers and then using rules. The second round produces no answer but returns the four solutions `(1,1)`, `(1,2)`, `(2,1)` and `(2,2)` among which only `(1,1)` has not been observed before.

3.2.5 *Properties of the eager strategy.* Since answers are returned eagerly, a pioneer and a follower may not have an ancestor-descendant relationship. Because of the existence of fake loops and the necessity of iterating the evaluation of top-most looping subgoals, redundant solutions may be observed. In the previous example, the solutions `(1,1)`, `(2,1)` and `(2,2)` are each observed twice. Provided that the top-most looping subgoal `p(X)` did not return the answer `p(1)` again in the second round, the solution `(1,2)` would have been lost.

The earger strategy is more suited than the lazy strategy for single-solution search. For certain applications such as planning it is unreasonable to find all answers either because the set is infinite or because only one answer is needed. For these applications the eager strategy is more effective than the lazy one. As described below in Subsection 5.3, cuts are handled more efficiently with the eager strategy.

The eager strategy may require more re-computation to reach fixpoints, but its adoption does not affect the soundness and completeness of linear tabling [Shen et al. 2001].

4. SEMI-NAIVE OPTIMIZATION

The basic linear tabling framework described in the previous section does not distinguish between new and old answers. The problem with this naive method is that it redundantly joins answers of subgoals that have been joined in early rounds. Semi-naive optimization [Ullman 1988] reduces the redundancy by ensuring that at least one new answer is involved in the join of the answers for each rule. In this section, we introduce semi-naive optimization into linear tabling and identify the necessary conditions for it to be complete. We also propose a technique called *early answer promotion* to further avoid redundant consumption of answers. This optimization works with both the lazy and eager strategies.

4.1 Preparation

To make semi-naive optimization possible, we divide the answer table for each tabled subgoal into three regions as depicted below:

old	*previous*	*current*

The names of the regions indicate the rounds during which the answers in the regions are produced: *old* means that the answers were produced before the previous round, *previous* the answers produced during the previous round, and *current* the answers produced in the current round. The answers stored in *previous* and *current* are said to be *new*. Before each round of evaluation is started, answers are promoted accordingly: *previous* answers become *old* and *current* answers become *previous*.

In our optimization, answers are consumed *sequentially*. For a subgoal, either all the available answers or only new answers are consumed. This is unlike in bottom-up evaluation where answers are consumed *incrementally*, i.e., answers produced in a round are not consumed until the next round. As will be discussed later, incremental consumption of answers as is done in bottom-up evaluation does avoid certain redundant joins but does not fit linear tabling since it may require more rounds to reach fixpoints.

For a given program, we find a level mapping from the predicate symbols in the program to the set of integers that represents the *call graph* of the program. Let m be a level mapping. We extend the notation to assume that $m(p(\ldots)) = m(p)$ for any subgoal $p(\ldots)$.

Definition 4. For a program, a level mapping m represents the *call graph* if for each rule "$H{:}{-}A_1, A_2, ..., A_n$" in the program, $m(H) > m(A_i)$ iff the predicate of A_i does not call either directly or indirectly the predicate of H, and $m(H) = m(A_i)$ iff the predicates of H and A_i occur in a loop in the call graph.

The level mapping as defined divides predicates in a program into several strata. The predicate at each stratum depends only on those on the lower strata. The level mapping is an abstract representation of the dependence relationship of the subgoals that may occur in execution. If two subgoals A and A' occur in a loop, them $m(A) = m(A')$.

Definition 5. Let "$H{:}{-}A_1, ..., A_k, ..., A_n$" be a rule in a program and m be the level mapping that represents the call graph of the program. A_k is called the *last depending subgoal* of the rule if $m(A_k) = m(H)$ and $m(H) > m(A_i)$ for $i > k$.

The last depending subgoal A_k is the last subgoal in the body that may depend on the head to become complete. Thus, when the rule is re-executed on a subgoal, all the subgoals to the right of A_k that have occurred before must already be complete.

Definition 6. Let "$H{:}{-}A_1, ..., A_n$" be a rule in a program and m be a level mapping that represents the call graph of the program. If there is no depending subgoal in the body, i.e., $m(H) > m(A_i)$ for $i = 1, ..., n$, then the rule is called a *base rule*.

4.2 Semi-naive optimization

THEOREM 1. Let "$H{:}{-}A_1, ..., A_k, ..., A_n$" be a rule where A_k is the last depending subgoal, and C be a subgoal that is being resolved by using the rule in a round of evaluation of a top-most looping subgoal T. For a combination of answers of A_1, $\cdots$, and A_{k-1}, if C has occurred in an early round and the combination does not contain any new answers, then it is safe to let A_k consume new answers only.

Proof: Let $A_{k_{old}}$ and $A_{k_{new}}$ be the *old* and *new* answers of the subgoal A_k, respectively. For a combination of answers of A_1, $\cdots$, and A_{k-1}, if the combination does not contain new answers then the join of the combination and $A_{k_{old}}$ must have been done and all possible answers for C that can result from the join must have been produced during the previous round because the subgoal C has been encountered before. Therefore only new answers in $A_{k_{new}}$ should be used. □

COROLLARY 1. Base rules need not be considered in the re-evaluation of any subgoals.

Semi-naive optimization would be unsafe if it were applied to new subgoals that have never been encountered before. The following example, where all the predicates are assumed to be tabled, illustrates this possibility:

```
?- p(X,Y).

p(X,Y) :- p(X,Z),q(Z,Y).  (C1)
p(b,c) :- p(X,Y).         (C2)
p(a,b).                   (C3)

q(c,d) :- p(X,Y),t(X,Y).  (C4)

t(a,b).                   (C5)
```

In the first round of `p(X,Y)` the answer `p(a,b)` is added to the table by `C1`, and in the second round the rule `C2` produces the answer `p(b,c)` by using the answer produced in the first round. In the third round, the rule `C1` generates a new subgoal `q(c,Y)` after `p(X,Z)` consumes `p(b,c)`. If semi-naive optimization were applied to `q(c,Y)`, then the subgoal `p(X,Y)` in `C4` could consume only the new answer `p(b,c)` and the third answer `p(b,d)` would be lost.

4.3 Analysis

In semi-naive optimization described above, answers produced in the current round are consumed immediately rather than postponed to the next round as in the bottom-up version, and answers are promoted each time a new round is started. This way of consuming and promoting answers may cause certain redundancy.

Consider the conjunction (P, Q). Assume Q_o, Q_p, and Q_c are the sets of answers in the three regions (respectively, *old*, *previous*, and *current*) of the subgoal Q when Q is encountered in round i. Assume also that P had been complete before round i and P_a is the set of answers. The join $P_a \bowtie (Q_p \bigcup Q_c)$ is computed for the conjunction in round i. Assume Q'_o, Q'_p, and Q'_c are the sets of answers in the three regions when Q is encountered in round i+1. Since answers are promoted before round $i+1$ is started, we have:

$$Q'_o = Q_o \bigcup Q_p$$
$$Q'_p = Q_c \bigcup \alpha$$

where α denotes the new answers produced for Q after the conjunction (P, Q) in round i. When the conjunction (P, Q) is encountered in round $i+1$, the following join is computed.

$$P_a \bowtie (Q'_p \bigcup Q'_c) = P_a \bowtie (Q_c \bigcup \alpha \bigcup Qc')$$

Notice that the join $P_a \bowtie Q_c$ is computed in both round i and $i+1$.

We could allow last depending subgoals to consume answers incrementally as is done in bottom-up evaluation,[3] but doing so may require more rounds to reach fixpoints. Consider the following example, which is the same as the one shown above but has a different ordering of clauses:

```
?- p(X,Y).

p(a,b).                        (C1)
p(b,c) :- p(X,Y).              (C2)
p(X,Y) :- p(X,Z),q(Z,Y).       (C3)

q(c,d) :- p(X,Y),t(X,Y).       (C4)

t(a,b).                        (C5)
```

In the first round, `C1` produces the answer `p(a,b)`. When `C2` is executed, the subgoal in the body cannot consume `p(a,b)` since it is produced in the current round. Similarly, `C3` produces no answer either. In the second round, `p(a,b)` is moved to the *previous* region, and thus can be consumed. `C2` produces a new answer `p(b,c)`. When `C3` is executed, no answer is produced since `p(b,c)` cannot be consumed. In the third round, `p(a,b)` is moved to the *old* region, and `p(b,c)` is moved to the *previous* region. `C3` produces the third answer `p(b,d)`. The fourth round produces no new answer and confirms the completion of the computation. So in total four rounds are needed to compute the fixpoint. If answers produced in the current round are consumed in the same round, then only two rounds are needed to reach the fixpoint.

4.4 Early promotion of answers

As discussed above, sequential consumption of answers may cause redundant joins. In this subsection, we propose a technique called *early promotion* of answers to reduce the redundancy.

Definition 7. Let Q be a follower of some pioneer that first exhausts its answers in the current round.[4] Then all answers of Q in the *current* region are promoted to the *previous* region once being consumed by Q.

Consider again the conjunction (P, Q) where Q is the first follower that exhausts its answers. The answers in the current region Q_c are promoted to the *previous* region after Q has consumed all its answers in round i. By doing so, the join $P_a \bowtie Q_c$ will not be recomputed in round $i+1$ since Q_c must have been promoted to the *old* region in round $i+1$.

Consider, for example, the following program:

[3]No interesting necessary condition has been found to make incremental consumption safe in linear tabling. During the evaluation of a top-most looping subgoal T another subgoal T' may join the current cluster and become a new top-most looping subgoal. The difficulty of identifying a necessary condition arise from the fact that an answer old to a subgoal in the cluster of T may be new to another subgoal in the cluster of T'.

[4]If the lazy strategy is used, then Q must be the first follower encountered in this round.

```
?- p(X,Y).

p(a,b).                    (C1)
p(b,c) :- p(X,Y).          (C2)
```

Before C2 is executed in the first round, p(a,b) is in the *current* region. Executing C2 produces the second answer p(b,c). Since the subgoal p(X,Y) in C2 is the first follower that exhausts its answers in the current round, it is qualified to promote its answers. So the answers p(a,b) and p(b,c) are moved from the *current* region to the *previous* region immediately after being consumed by p(X,Y). As a result, the potential redundant consumption of these answers by p(X,Y) are avoided in the second round of iteration since they will all be transferred to the *old* region before the second round starts.

THEOREM 2. Early promotion does not lose any answers.

Proof: First note that although answers are tabled in three disjoint regions, all tabled answers will be consumed except for some last depending subgoals that would skip the answers in their *old* regions (see Theorem 1). Assume, on the contrary, that applying early promotion loses answers. Then there must be a last depending subgoal A_k in a rule "$H{:}{-}A_1, ..., A_k, ..., A_n$" and a tabled answer A for A_k such that A has been moved to the *old* region before being consumed by A_k so that A will never be consumed by A_k. Assume A is produced in round i. We distinguish between the following two cases:

(1) The last depending subgoal A_k is not selected in round i. In round $j(j > i)$, A_k is selected either because H is new or some $A_i(i < k)$ consumes a new answer. By Theorem 1, A_k will consume all answers in the three regions, including the answer A.

(2) Otherwise, A must be produced by A_k itself or a variant subgoal of A_k that is selected either *before* or *after* A_k in round i. If A is produced by A_k itself or *before* A_k is selected, then the answer will be consumed by A_k since promoted answers will remain new by the end of the round. If A is produced by a variant *after* A_k is selected, then the answer cannot be promoted because A_k exhausts its answers *before* the variant. In this case, the answer A will remain new in the next round and will thus be consumed by A_k.

Both of the above two cases contradict our assumption. The proof then concludes. □

5. IMPLEMENTATION

Changes to the Prolog machine ATOAM [Zhou 1996] are needed to implement linear tabling. In this section we describe the changes to the data structures and the instruction set. We also show how to implement the cut operator.

5.1 Data structures

A new data area, called *table area*, is introduced for memorizing tabled subgoals and their answers. The *subgoal table* is a hash table that stores all the tabled subgoals encountered in execution. For each tabled subgoal and its variants, there is an entry in the table that contains the following information:

`Copy`
`PionnerAR`
`State`
`TopMostLoopingSubgoal`
`DependentSubgoals`
`AnswerTable`

The field `Copy` points to the copy of the subgoal in the table area. In the copy, all variables are numbered. Therefore all variants of the subgoal are identical.

The field `PionnerAR` points to the frame of the pioneer, which is needed for implementing cuts (described below). When the choice point of a tabled subgoal is cut off before the subgoal reaches completion, the field `PionnerAR` will be set to `NULL`. When a variant of the subgoal is encountered again after, the subgoal will be treated as a pioneer.

The field `State` indicates whether the subgoal is a looping subgoal, whether the answer table has been revised, and whether the subgoal is *complete* or *visited*. When execution backtracks to a top-most looping subgoal, if the *revised* bit is set, then another round of evaluation will be started for the subgoal. A top-most looping subgoal becomes complete if this *revised* bit is unset after a round of evaluation. At that time, the subgoal and all of its dependent subgoals will be set to *complete*.

The `TopMostLoopingSubgoal` field points to the entry for the top-most looping subgoal, and the field `DependentSubgoals` stores the list of subgoals on which this subgoal depends. When a top-most looping subgoal becomes complete, all of its dependent subgoals turn to complete too.

The field `AnswerTable` points to the answer table for this subgoal, which is also a hash table. Hash tables expand dynamically.

In ATOAM different structures of frames are used for different types of predicates [Zhou 1996]. A new frame structure is introduced for tabled predicates. The frame for a tabled predicate contains the following two slots in addition to those slots stored in a choice point frame[5]:

`SubgoalTable`
`CurrentAnswer`

The `SubgoalTable` points to the subgoal table entry, and the `CurrentAnswer` points to the last answer that was consumed. The next answer can be reached from this reference on backtracking.

5.2 Instructions

Three new instructions, namely `table_start`, `memo`, and `check_completion`, are introduced into the ATOAM for encoding the three table primitives. The following shows the compiled code of the transitive closure program:

```
% p(X,Y):-p(X,Z),e(Z,Y).
% p(X,Y):-e(X,Y).
```

[5]A choice point frame has the following slots: AR (parent frame), CPS (continuation program pointer on success), CPF (continuation program pointer on failure), TOP (top of the control stack), B (parent choice point), H (top of the heap), and T (top of the trail stack).

```
p/2: table_start 2,1
     fork r2
     para_value y(2)
     para_var y(-13)
     call p/2          % p(X,Z)
     para_value y(-13)
     para_value y(1)
     call e/2          % e(Z,Y)
     memo
r2:  fork r3
     para_value y(2)
     para_value y(1)
     call e/2          % e(X,Y)
     memo
r3:  check_completion p/2
```

The `table_start` instruction takes two operands: the arity (2) and the number of local variables (1). The entrance of the ending rule is taken as an operand so that a pioneer can know where to go after its choice point is stolen. The `check_completion` instruction takes the entrance as an operand so that the predicate can be re-entered when it needs re-evaluation.

5.3 Handling cuts

As discussed in Subsection 3.1.5, the lazy strategy is not suited for tabled programs with cuts. The goal "$p(X), !, q(X)$" produces all the answers for $p(X)$ even though only one is needed. For tabled predicates that contain cuts and tabled subgoals that are in the scopes of cuts, the eager strategy should be used.[6]

The cut operator is handled easily. Consider the following rule `H:-L,!,R`. The cut discards the choice points created for `L`. If any tabled subgoals are encountered during the execution of `L`, then we need to roll back the choice points of these subgoals and set the `PionnerAR` slot of each of the incomplete tabled subgoals to `NULL`. When a tabled subgoal that has occurred during the execution of `L` is encountered again after the cut, it will be treated as a pioneer not a follower. If `H` is a tabled predicate, then the cut sets the backtracking pointer to the dummy ending clause so all the alternative rules below this one will be skipped.

Consider the following tabled program:

```
:-eager_consume p/1.
p(1).
p(2).
```

and the goal

```
?-p(X),!,p(Y).
```

The first subgoal `p(X)` produces the answer `p(1)` before the cut is encountered. The cut discards the choice point of `p(X)` and cut off the relation between `p(X)`

[6]In B-Prolog version 6.7, the lazy strategy is adopted by default, but the user can use the directive `:-eager_consume` to change the strategy.

tcl:
```
tcl(X,Y):-edge(X,Y).
tcl(X,Y):-tcl(X,Z),edge(Z,Y).
```

tcr:
```
tcr(X,Y):-edge(X,Y).
tcr(X,Y):-edge(X,Z),tcr(Z,Y).
```

tcn:
```
tcn(X,Y):-edge(X,Y).
tcn(X,Y):-tcn(X,Z),tcn(Z,Y).
```

sg:
```
sg(X,X).
sg(X,Y):-edge(X,XX),sg(XX,YY),edge(Y,YY).
```

Fig. 2. Datalog programs.

and its entry in the table. So when `p(Y)` is executed, it is treated as a pioneer. The solutions returned to the top goal are `(1,1)` and `(1,2)`.

Consider the following tabled predicate with a cut in it:

```
:-eager_consume p/1.
p(X):-!,p(X).
p(a).
```

Since there is no subgoal appearing to the left of the cut, the cut just sets the backtracking pointer to the dummy ending clause. When the subgoal `p(X)` after the cut, which is a follower, is encountered, it fails because no answer is available. When execution backtracks to the pioneer `p(X)`, it completes its evaluation and fails.

6. PERFORMANCE EVALUATION

In this section, we empirically compare the two answer consumption strategies and evaluate the effectiveness of semi-naive optimization. We also compare the performance of B-Prolog (version 6.7) with XSB (version 2.7.1). We use a Linux machine in the evaluation.

6.1 Benchmarks

We use benchmarks from three different sources: Datalog programs shown in Figure 2, the CHAT benchmark suite [Demoen and Sagonas 1999], and a parser, called *atr*, for the Japanese language defined by a grammar of over 860 rules [Uratani et al. 1994]. The graph used in the transitive-closure programs consists of 100 vertices and 4000 edges, the graph used in the same-generation program consists of 50 vertices and 1561 edges, and the corpus used in the parser contains 10 sentences. All the benchmarks are available from *probp.com/bench.tar.gz*.

6.2 Comparison of the two answer-consumption strategies

Table I compares the two answer-consumption strategies on speed and stack space[7] efficiencies. The numbers in the parentheses indicate the actual amounts of consumption of CPU time (in milliseconds) and stack space (in bytes). The difference

[7]The total usage of the local, global and trail stacks.

Table I. Comparison of the lazy and eager strategies.

program	CPU time		Stack space	
	Lazy	Eager	Lazy	Eager
tcl	1 (120.0)	1.00	1 (232)	1.00
tcr	1 (211.0)	1.01	1 (5800)	1.00
tcn	1 (266.0)	0.94	1 (6916)	1.00
sg	1 (3184.0)	0.87	1 (6992)	1.02
cs_o	1 (135.1)	1.30	1 (9400)	1.36
cs_r	1 (275.7)	1.17	1 (9400)	1.36
disj	1 (67.8)	1.05	1 (5688)	1.41
gabriel	1 (113.0)	1.01	1 (8764)	1.18
kalah	1 (71.0)	0.96	1 (12260)	2.03
pg	1 (53.1)	2.49	1 (8960)	3.59
peep	1 (329.8)	0.98	1 (6816)	2.88
read	1 (327.1)	0.88	1 (9676)	2.22
atr	1 (730.0)	1.00	1 (21084)	1.06
average	1	1.13	1	1.62

of these two strategies in terms of CPU time is small on average. This result implies that for programs with cuts declaring the use of the eager strategy would not cause significant slow-down. The difference in the usage of stack space is more significant than in CPU time. This is because, as discussed before, the lazy strategy has better locality than the eager strategy.

6.3 Effectiveness of semi-naive optimization

Table II shows the effectiveness of semi-naive optimization in gaining speed-ups under both strategies. Without this optimization, the system would consume about 40% more CPU time on average under either strategy. Our experiment also indicates that on average over 95% of the gains in speed are attributed to the *early promotion* technique.

6.4 Comparison with XSB

Table III compares B-Prolog (BP) with XSB on speed and stack space[8] efficiencies. For XSB the default setting[9] is used. BP is faster than XSB for most of the programs and on average as well, and BP consumes an order-of-magnitude less stack space than XSB for six of the programs.

The empirical data on the usage of table space are not shown. In BP, both subgoal and answer tables are maintained as dynamic hashtables and the amount of table space consumed can be dominated by the initial sizes of the hashtables. In XSB, in contrast, tables are maintained as tries [Ramakrishnan et al. 1998]. The usage of the table space should be the same if the same data structure is used.

The necessity of re-evaluating looping subgoals has been blamed for the low speed of iterative-computation-based tabling systems [Zhou et al. 2000; Guo and Gupta 2001]. Our new findings indicate that re-evaluation is not a dominant factor. Table

[8]The total usage of the local, global, choice point, and trail stacks for XSB.

[9]The SLG-WAM and the local scheduling strategy are used in the default setting.

Table II. Effectiveness of semi-naive optimization.

program	CPU time (with semi)	CPU time (without semi)	
		Lazy	Eager
tcl	1	1.81	1.83
tcr	1	1.29	1.30
tcn	1	1.80	1.93
sg	1	1.15	1.50
cs_o	1	1.15	1.11
cs_r	1	1.15	1.13
disj	1	1.56	1.56
gabriel	1	1.34	1.33
kalah	1	1.36	1.50
pg	1	1.33	1.07
peep	1	1.07	1.07
read	1	2.09	1.30
atr	1	1.03	1.03
average	1	1.42	1.38

Table III. Comparison of B-Prolog and XSB.

program	BP (Lazy)	XSB	
		CPU time	Stack space
tcl	1	1.41	8.62
tcr	1	1.37	33.58
tcn	1	1.19	32.99
sg	1	1.22	109.27
cs_o	1	0.90	0.77
cs_r	1	0.86	0.93
disj	1	1.03	1.15
gabriel	1	0.76	2.36
kalah	1	1.11	0.75
pg	1	1.11	2.09
peep	1	0.76	3.23
read	1	0.96	11.30
atr	1	1.77	21.87
average	1	1.11	17.61

IV gives the statistics about the maximum (max num. its.) and average (ave. num. its.) numbers of iterations for tabled subgoals to reach their fixpoints. While for some programs, the maximum number of iterations performed is high (e.g., the maximum number for *atr* is 6), the average numbers are quite low. These statistics give an insight into the reason why an implementation of linear tabling could achieve comparable speed performance with XSB.

7. RELATED WORK

There are three different tabling schemes, namely OLDT [Tamaki and Sato 1986; Sagonas and Swift 1998], CAT [Demoen and Sagonas 1998], and linear tabling

Table IV. Statistics about re-evaluation.

program	num. subgoals	max num. its.	ave. num. its.
tcl	1	2	2.00
tcr	51	2	1.96
tcn	51	2	1.98
sg	153	2	1.32
cs_o	76	2	1.14
cs_r	76	2	1.16
disj	74	2	1.20
gabriel	59	2	1.20
kalah	102	3	1.24
pg	48	2	1.13
peep	49	3	1.29
read	131	5	1.34
atr	7139	6	1.81

[Shen et al. 1999; 2001; Zhou et al. 2000; Zhou and Sato 2003; Zhou et al. 2004]. SLG [Chen and Warren 1996] is a formalization based on OLDT for computing well-founded semantics for general programs with negation. These three schemes differ in the handling of consumers.

In OLDT, a consumer fails after it exhausts all the existing answers and its state is preserved by freezing the stack so that it can be reactivated after new answers are generated. The CAT approach does not freeze the stack but instead copies the stack segments between the consumer and its producer into a separate area so that backtracking can be done normally. The saved state is reinstalled after a new answer is generated. CHAT [Demoen and Sagonas 1999] is a hybrid approach that combines OLDT and CAT.

Linear tabling relies on iterative computation of looping subgoals to compute fixpoints. Linear tabling is arguably the easiest scheme to implement since no effort is needed to preserve states of consumers and the garbage collector can be kept untouched for tabling. Linear tabling is also the most space-efficient scheme since no extra space is needed to save states of consumers. Nevertheless, linear tabling without optimization could be computationally more expensive than the other two schemes.

Guo and Gupta's DRA method [Guo and Gupta 2001] is based on the same idea of linear tabling but identifies looping clauses dynamically and iterates the execution of looping clauses to compute fixpoints. While in our linear tabling iteration is performed on only top-most looping subgoals, in DRA iteration is performed on every looping subgoal. Besides the difference in answer consumption strategies and optimizations, the linear tabling scheme described in this paper differs from the original version [Zhou et al. 2000; Shen et al. 2001] in that followers fail after they exhaust their answers rather than steal their pioneers' choice points. This strategy is originally adopted in the DRA method.

The two consumption strategies have been compared in XSB [Freire et al. 1998] as two scheduling strategies. The lazy strategy is called *local scheduling* and the eager strategy is called *single-stack scheduling*. Another strategy, called *batched schedul-*

ing, is similar to local scheduling but top-most looping subgoals do not have to wait until their clusters become complete to return answers. Their experimental results indicate that local scheduling constantly outperforms the other two strategies on stack space and can perform asymptotically better than the other two strategies on speed. The superior performance of local scheduling is attributed to the saving of freezing stack segments. Although our experiment confirms the good space performance of the lazy strategy, it gives a counterintuitive result that the eager strategy is as fast as the lazy strategy. This result implies that the cost of iterative evaluation is considerably smaller than that of freezing stack segments, and for predicates with cuts the eager strategy can be used without significant slow-down. In our tabling system, different answer consumption strategies can be used for different predicates. The tabling system described in [Rocha et al. 2003] also supports mixed strategies.

semi-naive optimization is a fundamental idea for reducing redundancy in bottom-up evaluation of logic database queries [Bancilhon and Ramakrishnan 1986; Ullman 1988]. As far as we know, its impact on top-down evaluation had been unknown before [Zhou et al. 2004]. OLDT [Tamaki and Sato 1986] as implemented in SLG-WAM [Sagonas and Swift 1998] does not need this technique since it is not iterative and the underlying delaying mechanism successfully avoids the repetition of any derivation step. An attempt has been made by Guo and Gupta [Guo and Gupta 2001] to make incremental consumption of tabled answers possible in DRA. In their scheme, answers are also divided into three regions but answers are consumed incrementally as in bottom-up evaluation. Since no condition is given for the completeness and no experimental result is reported on the impact of the technique, we are unable to give a detailed comparison.

Our semi-naive optimization differs from the bottom-up version in two major aspects: Firstly, no differentiated rules are used. In the bottom-up version differentiated rules are used to ensure that at least one new answer is involved in the join of answers for each rule. Consider, for example, the clause:

$H : -P, Q.$

The following two differentiated rules are used in the evaluation instead of the original one:

$H : -\Delta P, Q.$
$H : -P, \Delta Q.$

Where ΔP denotes the new answers produced in the previous round for P. Using differentiated rules in top-down evaluation can cause considerable redundancy, especially when the body of a clause contains non-tabled subgoals.

The second major difference between our semi-naive optimization and the bottom-up version is that answers in our method are consumed sequentially not incrementally. A tabled subgoal consumes either all available answers or only new answers including answers produced in the current round. Neither incremental consumption nor sequential consumption seems satisfactory. Incremental consumption avoids redundant joins but may require more rounds to reach fixpoints. In contrast, sequential consumption never need more rounds to reach fixpoints but may cause redundant joins of answers. The early promotion technique alleviates the problem of sequential consumption. By promoting answers early from the *current* region to the *previous* region, we can considerably reduce the redundancy in joins.

In theory semi-naive optimization can be an order of magnitude faster than naive-evaluation in bottom-up evaluation [Bancilhon and Ramakrishnan 1986]. Our experimental results show that semi-naive optimization gives an average speed-up of 40% to linear tabling if answers are promoted early, and almost no gain in speed if no answer is promoted early. In linear tabling, only looping subgoals need to be iteratively evaluated. For non-looping subgoals, no re-evaluation is necessary and thus semi-naive optimization has no effect at all on the performance. Our experiment shows that for most looping subgoals the fixpoints can be reached in 2-3 rounds of iteration. In contrast more rounds of iteration are needed to reach fixpoints in bottom-up evaluation. In addition, in bottom-up evaluation, the order of the joins can be optimized and no further joins are necessary once a participating set is known to be empty. In contrast, in linear tabling joins are done in the strictly chronological order. For a conjunction (P, Q, R) the join $P \bowtie Q$ is computed even if no answer is available for R. Because of all these factors, semi-naive optimization is not as effective in linear tabling as in bottom-up evaluation.

Our semi-naive optimization requires the identification of last depending subgoals. For this purpose, a level mapping is used to represent the call graph of a given program. The use of a level mapping to identify optimizable subgoals is analogous to the idea used in the stratification-based methods for evaluating logic programs [Apt et al. 1988; Chen and Warren 1996; Przymusinski 1989]. In our level mapping, only predicate symbols are considered. It is expected that more accurate approximations can be achieved if arguments are considered as well.

8. CONCLUSION

In this paper we have described the linear tabling framework, two answer consumption strategies (namely, *lazy* and *eager* strategies), and the semi-naive optimization. We have compared the two strategies both qualitatively and quantitatively. Our results indicate that, while the lazy strategy has better space efficiency than the eager strategy, the eager strategy is comparable in speed with the lazy strategy. This result implies that for all-solution search programs the lazy strategy should be adopted and for partial-solution search programs including programs with cuts the eager strategy should be used.

We have tailored semi-naive optimization to linear tabling and have given the necessary conditions for it to be complete. Moreover, we have proposed a technique called *early answer promotion* to reduce redundant consumption of answers. Our experimental result indicates that semi-naive optimization gives significant speedups to some programs.

Linear tabling has several attractive advantages including its simplicity, ease of implementation, and good space efficiency. Early implementations of linear were several times slower than XSB. This paper has demonstrated for the first time that linear tabling with optimization is as competitive as OLDT on time efficiency as well.

Tabling has been found useful in a number of problem domains and it is expected that more and more Prolog systems will support tabling in the future. The encouraging results of this paper will undoubtedly attract more attention to linear tabling.

REFERENCES

Apt, K., Blair, H. A., and Walker, A. 1988. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann, 89–142.

Bancilhon, F. and Ramakrishnan, R. 1986. An amateur's introduction to recursive query processing strategies. *Proc. of ACM SIGMOD '86*, 16–52.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Dawson, S., Ramakrishnan, C. R., and Warren, W. 1996. Practical program analysis using general purpose logic programming systems — A case study. *ACM SIGPLAN Notices 31,* 5, 117–126.

Demoen, B. and Sagonas, K. 1998. CAT: The copying approach to tabling. In *Proceedings of Programming Language Implementation and Logic Programming (PLILP).* LNCS 1490, Springer-Verlag, 21–35.

Demoen, B. and Sagonas, K. 1999. CHAT: The copy-hybrid approach to tabling. In *Proceedings of Practical Aspects of Declarative Programming (PADL).* LNCS 1551, Springer-Verlag, 106–121.

Eisner, J., Goldlust, E., and Smith, N. A. 2004. Dyna: A declarative language for implementing dynamic programs. In *Proc. of the 42nd Annual Meeting of ACL.*

Freire, J., Swift, T., and Warren, D. S. 1998. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*.

Guo, H.-F. and Gupta, G. 2001. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings International Conference on Logic Programming (ICLP).* LNCS 2237, Springer-Verlag, 181–195.

Johnson, M. 1995. Memoization of top down parsing. *Computational Linguistics 21,* 3.

Liu, M. 1999. Deductive database languages: Problems and solutions. *ACM Computing Surveys 31,* 1, 27–62.

Lloyd, J. W. 1988. *Foundation of Logic Programming*, 2 ed. Springer-Verlag.

Michie, D. 1968. "memo" functions and machine learning. *Nature*, 19–22.

Nielson, F., Nielson, H. R., Sun, H., Buchholtz, M., Hansen, R. R., Pilegaard, H., and Seidl, H. 2004. The succinct solver suite. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference (TACAS), LNCS 2988*. 251–265.

Pientka, B. December 2003. Tabled higher-order logic programming. Ph.D. thesis, Technical Report CMU-CS-03-185.

Przymusinski, T. C. 1989. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS '89. Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 11–21.

Ramakrishnan, C. 2002. Model checking with tabled logic programming. In *ALP News Letter*. ALP.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., and Warren, D. 1998. Efficient access mechanisms for tabled logic programs. *J. Logic Programming 38*, 31–54.

Ramakrishnan, R. and Ullman, J. D. 1995. A survey of deductive database systems. *Journal of Logic Programming 23,* 2, 125–149.

Rocha, R., Silva, F., and Costa, V. S. 2003. A tabling engine designed to support mixed-strategy evaluation. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS).* 33–44.

Rocha, R., Silva, F., and Costa, V. S. 2005. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming (TPLP) 5,* 1 & 2, 161–205.

Sagonas, K. and Swift, T. 1998. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems 20,* 3, 586–634.

Sagonas, K., Swift, T., and Warren, W. 1994. XSB as a deductive database. *SIGMOD Record (ACM Special Interest Group on Management of Data) 23,* 2, 512–512.

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 391–454.

Shen, Y.-D., Yuan, L., You, J., and Zhou, N.-F. 2001. Linear tabulated resolution based on Prolog control strategy. *Theory and Practice of Logic Programming (TPLP) 1,* 1, 71–103.

Shen, Y.-D., Yuan, L.-Y., You, J.-H., and Zhou, N.-F. 1999. Linear tabulated resolutions for the well-founded semantics. In *Proceedings of Logic Programming and Nonmonotonic Reasoning.* 192–205.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, E. Shapiro, Ed. LNCS, Springer-Verlag, 84–98.

Ullman, J. D. 1988. *Database and Knowledge-Base Systems.* Vol. 1 & 2. Computer Science Press.

Uratani, N., Takezawa, T., Matsuo, H., and Morita, C. 1994. ATR integrated speech and language database. Technical Report TR-IT-0056, ATR Interpreting Telecommunications Research Laboratories. In Japanese.

Warren, D. S. 1992. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming 35,* 3, 93.

Warren, D. S. 1999. *Programming in Tabled Prolog.* DRAFT 1 (http://www.cs.sunysb.edu/ warren/xsbbook/book.html).

Zhou, N.-F. 1996. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems 18,* 6, 752–779.

Zhou, N.-F. and Sato, T. 2003. Efficient fixpoint computation in linear tabling. In *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming.* 275–283.

Zhou, N.-F., Sato, T., and Hasida, K. 2003. Toward a high-performance system for symbolic and statistical modeling. In *IJCAI Workshop on Learning Statistical Models from Relational Data.* 153–159.

Zhou, N.-F., Shen, Y.-D., and Sato, T. 2004. Semi-naive evaluation in linear tabling. In *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming.* 90–97.

Zhou, N.-F., Shen, Y.-D., Yuan, L.-Y., and You, J.-H. 2000. Implementation of a linear tabling mechanism. In *Proceedings of Practical Aspects of Declarative Programming (PADL).* LNCS 1753, Springer-Verlag, 109–123.