Semi-naive Evaluation in Linear Tabling

Neng-Fa Zhou
Department of Computer Science
CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

Yi-Dong Shen
Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing
ydshen@ios.ac.cn

Taisuke Sato
Department of Computer Science
Tokyo Institute of Technology & CREST JST
sato@mi.cs.titech.ac.jp

ABSTRACT

Semi-naive evaluation is an effective technique employed in bottom-up evaluation of logic programs to avoid redundant joins of answers. The impact of this technique on top-down evaluation had been unknown. In this paper, we introduce semi-naive evaluation into linear tabling, a top-down resolution mechanism for tabled logic programs. We give the conditions for the technique to be safe and propose an optimization technique called *early answer promotion* to enhance its effectiveness. While semi-naive evaluation is not as effective in linear tabling as in bottom-up evaluation, it is worthwhile to be adopted. Our benchmarking shows that this technique gives significant speed-ups to some programs.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Language Classifications—*Constraint and logic languages*

General Terms

Languages

Keywords

Prolog, Semi-naive evaluation, Recursion, Tabling, Memoization, Linear tabling

1. INTRODUCTION

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 [6, 9, 11, 14, 18, 20]. 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 the evaluation of functions [8]. OLDT [16] 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 [13]. 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 [5], B-Prolog [21, 19], Mercury, and YAP [12].

Linear tabling has emerged as an alternative effective tabling scheme [15, 21, 19]. 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. In our linear tabling scheme, 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. Nevertheless, naive re-evaluation of all looping subgoals may be computationally unacceptable. In [19], we have proposed several optimization techniques that make linear tabling computationally more competitive than before.

Semi-naive evaluation is an effective technique used in bottom-up evaluation of Datalog programs [2, 17]. 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 evaluation on top-down evaluation had been unknown. In this paper, we propose to introduce semi-naive evaluation into linear tabling. We have made efforts to properly tailor semi-naive evaluation to linear tabling. In our semi-naive evaluation, answers for each tabled subgoal are divided into three regions as in bottom-up evaluation, but answers are consumed sequentially not incrementally so answers produced in a round are consumed in the same round. We have found that incremental consumption of answers is not suited to linear tabling since it may require more rounds of iteration to reach fixpoints. Consuming answers incrementally, however, 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.

The remainder of the paper is structured as follows: In the next section we define the linear tabling method to be optimized. This tabling method is the same as the one described in [19]. In Section 3, we introduce semi-naive evaluation into the tabling method. We prove its completeness and also prove the safeness of the early promotion technique. In Section 4, we evaluate the effectiveness of the technique and in Section 5 we compare our semi-naive evaluation with the one employed in bottom-up evaluation. In Section 6, we conclude the paper.

2. LINEAR TABLING

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. We first give the framework and then describe the method we choose. The description is made as much self-contained as possible. The reader is referred to [7] 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 rule in a tabled predicate is called a *tabled rule* and a subgoal of a tabled predicate is called a *tabled subgoal.* Tabled predicates are transformed into a form that facilitates execution: each tabled 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 (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 an *answer* F in the table 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 "$H \leftarrow B_1, ..., B_m$" 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. 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 before it. Let $G_i = (A...)$ be a goal where A is the selected subgoal. A is called a *pioneer* in the derivation if no variant of A has been selected in goals before G_i in the derivation. 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 is an ancestor of A', and A and A' are variants. The subgoals A and A' are called *looping subgoals.* Moreover, A' is called a *follower* of A.

Notice that a derivation $G_i \Rightarrow \ldots \Rightarrow G_j$ does not form a loop if the selected subgoal of G_i is not an ancestor of that of G_j. Consider, for example, the goal "$p(X), p(Y)$" where p is defined by facts. The derivation "$p(X), p(Y)$" $\Rightarrow p(Y)$ does not form a loop even though the selected subgoal $p(Y)$ in the second goal is a variant of the selected subgoal $p(X)$ of the first goal since $p(X)$ is not an ancestor of $p(Y)$.

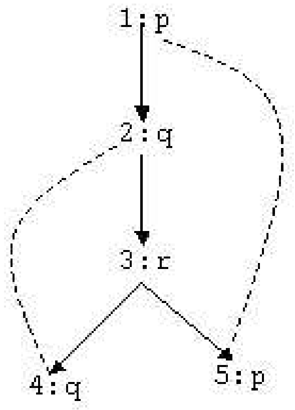

Figure 1: A top-most looping subgoal.

Definition 3. The *SLD tree* for a given goal G and a given program is a tree that satisfies the following: (1) G is the root of the tree; and (2) for each node, the derived goals are the children of the node. The children are ordered based on the textual order of the applied rules in the program. For each node in an SLD tree, the path from it to a leaf corresponds to a derivation of the node. A node in an SLD tree is called a *top-most looping node* if the selected subgoal of the node is the pioneer of a loop in a path and the pioneer is not contained in any other loops. The selected subgoal of a top-most looping node is called a *top-most looping subgoal*

For example, there are two loops in the SLD tree in Figure 1. Node `1:p` is a top-most looping node while `2:q` is not since `q` is contained in `p`'s loop.

Definition 4. A subgoal A is said to be *dependent* on another subgoal A' if A' occurs in the derivation of A. Two subgoals are said to be *inter-dependent* if they occur in each other's loop. The inter-dependent subgoals under a top-most looping subgoal form a *cluster.*

For example, in Figure 1 the subgoals `p`, `q`, and `r` are all inter-dependent and the three subgoals form a cluster.

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. To simplify the presentation, we assume that 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 in the SLD tree, the subgoal may have to be re-evaluated to ensure that no answer is lost. The evaluation of a top-most looping subgoal must be re-evaluated until the fixpoint is reached.

A linear tabling method is defined by the strategies used in the three primitives: *table_start(A)*, *memo(A)*, and *check_completion(A)*. The following defines a method.

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 of the current path, meaning that it is encountered for the first time, then it is resolved by using rules.

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

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 consumption of answers until all rules have been tried.

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 whether any new answers were produced during the last round of evaluation of A. If so, A is resolved again by using rules. Otherwise, if no new answers were produced, A is resolved by answers after being 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 *temporary complete*. 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 evaluated. For example, in Figure 1 `q` reaches its fixpoint only after the top-most loop subgoal `p` reaches its fixpoint.

In [19] an optimization technique, called *subgoal optimization*, is used to avoid evaluating the same subgoal more than once in each round of evaluation of a cluster. When a subgoal is encountered which is *temporary complete*, if the subgoal has been resolved by using rules in this round before, then this subgoal can be treated like a follower and be resolved by using answers only.

Example

Consider the following example program 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 that 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)` was produced in the last round of evaluation, the top-most looping subgoal `p(a,Y0)` needs to be re-evaluated. 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 is resolved by using the answers `p(a,b)` and `p(a,c)`.

3. SEMI-NAIVE EVALUATION

The basic linear tabling method 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. The semi-naive algorithm [17] 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 evaluation into linear tabling and identify the necessary conditions for the evaluation to be complete. We also propose a technique called *early answer promotion* to further avoid redundant consumption of answers.

3.1 Preparation

To make semi-naive evaluation 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*.

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 is not suited to 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 5. 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 $m(A) = m(A')$ but not vice versa.

Definition 6. 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 7. 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.*

3.2 Semi-naive evaluation

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.[1] Therefore only new answers in $A_{k_{new}}$ should be used. □

THEOREM 2 (COROLLARY). Base rules need not be considered in the re-evaluation of any subgoals.

Semi-naive evaluation would be unsafe if it were applied to new subgoals. 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, and in the second round the rule `C2` creates 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). If semi-naive evaluation 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.

3.3 Analysis

In the semi-naive evaluation technique described above, answers produced in the current round are consumed immediately rather than postponed to the next round as in the

[1]A looping subgoal that occurs in a round of evaluation must also occur in the subsequent rounds unless it is complete. Therefore, the subgoal C must have occurred in the previous round.

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,[2] 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.

3.4 Early promotion of answers

[2]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'.

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 8. Let Q be a subgoal that is selected the first time in the current round (i.e. no variant of Q has been selected before in this round). 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). The answers in Q_c are promoted to the *previous* region if Q is the first variant subgoal encountered 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:

```
?- 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 first encountered in the current round (the top-most looping subgoal is not counted), 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. As a result, the potential redundant consumption of these answers are avoided in the second round of iteration since they will all be transferred to the *old* region before the second round starts.

THEOREM 3. 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 3.1). 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 3.1, A_k will consume all answers in the three regions, including the answer A.
2. Otherwise, A must be produced by a variant subgoal of A_k that is selected either *before* or *after* A_k in round i. If A is produced *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 *after* A_k is selected, then the answer cannot be promoted because the subgoal that produces it is not encountered the first time in the round. 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. □

```
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).
```

Figure 2: Datalog programs.

4. PERFORMANCE EVALUATION

The semi-naive evaluation technique described in this paper has been implemented in B-Prolog version 6.6.[3] In this section, we evaluate the effectiveness of the described technique using benchmarks from two different sources: Datalog programs shown in Figure 2 and the CHAT suite [4]. All the benchmarks are available from *probp.com/bench.tar.gz*.

Table 1 shows the effectiveness of the semi-naive technique in reducing the number of consumed answers (#Ans) and the CPU time for each of the program. The CPU times were measured on a Windows XP machine with 1.7GHz CPU and 760mega RAM. The table shows only the effectiveness of semi-naive evaluation in avoiding redundant joins in recursive rules. Base rules are never considered in re-evaluation of subgoals. This case is covered by an optimization technique called *clause optimization* [19], and is thus not taken into account here.

The technique is more effective for the Datalog programs than for the Chat programs. The speed of *tcl* is almost doubled thanks to this technique.

Table 2 shows that the gains in speed are attributed almost entirely to the *early promotion* technique.

Semi-naive evaluation is not overhead-free. Three extra words are needed for each tabled subgoal: two for representing regions of answers and the third for keeping track of which subgoals in a rule have consumed new answers. Nevertheless, these extra words can be reclaimed once the subgoal becomes complete.

Table 3 compares the CPU times taken by B-Prolog (BP) and XSB (version 2.6) to run the programs. For BP, the auto-tabling optimization[4], which could speed-up the Chat programs by several times, is disenabled. For XSB the default setting is used. BP is faster than XSB for the Datalog programs but not the Chat programs.

The implementations of linear tabling have been considerably slower than XSB due to re-evaluation of looping subgoals [21, 5]. Our lastest implementation, while maintaining good space efficiency,[5] offers comparable speed performance with XSB even without auto-tabling optimization.

[3]Available from www.probp.com.
[4]*Auto-tabling* is an optimization technique that tables extra predicates to avoid re-evaluation of their subgoals.
[5]Semi-naive evaluation does not affect the stack space performance.

Table 1: Effectiveness of semi-naive evaluation.

source	program	$\frac{Nosemi}{Semi}$	
		#Ans	CPU time
Datalog	tcl	1.50	1.96
	tcr	1.29	1.21
	tcn	1.79	1.65
	sg	1.21	1.12
Chat	cs_o	1.01	1.01
	cs_r	1.02	1.01
	disj	1.02	1.02
	gabriel	1.07	1.06
	kalah	1.09	1.13
	pg	1.29	1.26
	peep	1.02	1.00
	read	1.08	1.10

Table 2: Effectiveness of early promotion.

program	$\frac{NoEP}{EP}$	
	#Ans	CPU time
tcl	1.50	1.96
tcr	1.29	1.32
tcn	1.79	1.81
sg	1.21	1.24
cs_o	1.01	1.02
cs_r	1.02	1.01
disj	1.02	1.04
gabriel	1.05	1.04
kalah	1.07	1.09
pg	1.29	1.31
peep	1.01	1.01
read	1.02	1.03

Table 3: BP vs. XSB (CPU time).

program	BP	XSB	
		XP	Linux
tcl	1	1.80	1.36
tcr	1	1.40	1.06
tcn	1	1.25	1.11
sg	1	1.14	1.10
cs_o	1	0.68	0.53
cs_r	1	0.82	0.64
disj	1	0.59	0.48
gabriel	1	0.61	0.50
kalah	1	1.00	0.77
pg	1	1.05	0.88
peep	1	0.37	0.37
read	1	0.45	0.39

5. DISCUSSION

This section compares our semi-naive evaluation with the bottom-up version of the technique. A comparison of linear tabling with related tabling models and methods can be found in [19].

Semi-naive evaluation is a fundamental idea for reducing redundancy in bottom-up evaluation of logic database queries [2, 17]. As far as we know, its impact on top-down evaluation had been unknown before this work. OLDT [16] as implemented in SLG-WAM [13] 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 [5] to make incremental consumption of tabled answers possible in DRA, a tabling scheme similar to linear tabling. 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 evaluation 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 evaluation 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 evaluation can be an order of magnitude faster than naive-evaluation in bottom-up evaluation [2]. Our experimental results show that semi-naive evaluation gives an average speed-up of 28% 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 evaluation 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 evaluation is not as effective in linear tabling as in bottom-up evaluation.

Our semi-naive evaluation 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 [1, 3, 10]. 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.

6. CONCLUSION

We have described how to incorporate semi-naive evaluation into linear tabling. Our contributions are as follows:

- We have tailored semi-naive evaluation to linear tabling and have given the necessary conditions for it to be complete.
- We have proposed a technique called *early answer promotion* to reduce redundant consumption of answers.
- We have implemented semi-naive evaluation in B-Prolog and evaluated its effectiveness.

Semi-naive evaluation is not as effective in linear tabling as in bottom-up evaluation. Nevertheless, semi-naive evaluation is worthwhile in linear tabling because (1) the space overhead is minor if not negligible compared with gains in speed, and (2) the implementation effort needed remains small compared with that of OLDT.

7. ACKNOWLEDGEMENT

Neng-Fa Zhou is supported in part by RF-CUNY, Yi-Dong Shen is supported in part by the National Natural Science Foundation of China, and Taisuke Sato is supported in part by CREST.

8. REFERENCES

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

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

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

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

[5] Guo, H.-F., and Gupta, G. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings*

International Conference on Logic Programming (ICLP) (2001), LNCS 2237, Springer-Verlag, pp. 181–195.

[6] Johnson, M. Memoization of top down parsing. *Computational Linguistics 21*, 3 (1995).

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

[8] Michie, D. "memo" functions and machine learning. *Nature* (1968), 19–22.

[9] Pientka, B. *Tabled higher-order logic programming.* PhD thesis, Technical Report CMU-CS-03-185, December 2003.

[10] Przymusinski, T. C. 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* (1989), ACM Press, pp. 11–21.

[11] Ramakrishnan, C. Model checking with tabled logic programming. In *ALP News Letter* (2002), ALP.

[12] Rocha, R., Silva, F., and Costa, V. S. On a tabling engine that can exploit or-parallelism. In *Proceedings International Conference on Logic Programming (ICLP)* (2001), LNCS 2237, Springer-Verlag, pp. 43–58.

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

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

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

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

[17] Ullman, J. D. *Database and Knowledge-Base Systems*, vol. 1 & 2. Computer Science Press, 1988.

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

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

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

[21] Zhou, N., Shen, Y., Yuan, L., and You, J. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming 2001(1)* (2001), 1–15.