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 less declarative and less readable 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; Sagonnas et al. 1994], and theorem proving [Nielsen 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. 2001; Zhou and Sato 2003], Mercury, and YAP [Rocha et al. 2001].

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. 2001; Zhou and Sato 2003]. 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. 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 how to resolve a looping subgoal that is a descendent of a variant ancestor. One strategy is to fail the subgoal after it consumes all the available answers, and another strategy is to let the descendent produce answers by using the alternative clauses of the ancestor. Another decision concerns when answers are consumed. The lazy answer consumption strategy (lazy strategy in short) gives priority to answer production and the eager answer consumption strategy (eager strategy in short) prefers answer consumption to production. This paper gives a comprehensive analysis of these 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 evaluation 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 evaluation on top-down evaluation had been unknown. In this paper, we also 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 redun-
dant 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, which employs the lazy strategy by default but allows the eager strategy to be used for subgoals that are in the scope of a cut 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 needed in this paper. In Section 3 we give the linear tabling framework and possible tabling strategies. In Section 4 we introduce semi-naive evaluation into the tabling method and prove its completeness. In Section 5 we describe the implementation of our tabling system and also show how to implement cuts. In Section 6 we compare the tabling strategies experimentally, evaluate the effectiveness of semi-naive evaluation, 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

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. This section gives the framework and possible strategies. The description is made 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 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 $\text{memo}(H)$ where $H$ is the head, and each tabled predicate contains a dummy ending rule whose body contains only one subgoal named $\text{check\_completion}(H)$. For example, given the definition of the transitive closure of a relation,

\begin{align*}
p(X,Y) & : \neg p(X,Z), e(Z,Y). \\
p(X,Y) & : \neg e(X,Y).
\end{align*}

The transformed predicate is as follows:

\begin{align*}
p(X,Y) & : \neg p(X,Z), e(Z,Y), \text{memo}(p(X,Y)). \\
p(X,Y) & : \neg e(X,Y), \text{memo}(p(X,Y)). \\
p(X,Y) & : \text{check\_completion}(p(X,Y)).
\end{align*}

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** Derivation. 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 $\theta$ such that $A_1\theta = F$ and $G' = \ldots$
\((A_2, \ldots, A_k)\theta\). \(G'\) is derived from \(G\) by using a rule “\(H \leftarrow B_1, \ldots, B_m\)” if \(A_1\theta = H\theta\) and \(G' = (B_1, \ldots, B_m, A_2, \ldots, A_k)\theta\). \(A_1\) is said to be the parent of \(B_1, \ldots,\) and \(B_m\). The relation ancestor is defined recursively from the parent relation.

**Definition Looping subgoals.** 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 \(G_i = (A\ldots)\) 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 \(G_i = (A\ldots)\) and \(G_j = (A'\ldots)\). 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 \(p(Y)\).

**Definition Dependent subgoals.** 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.

**Definition SLD tree.** 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.

**Definition Top-most looping subgoals.** A node in an SLD tree is called a top-most node if the selected subgoal of the node is not dependent on any of its ancestors. In particular, A top-most node is called a top-most looping node if the selected subgoal is the pioneer of a loop in a path. The selected subgoal of a top-most node is called a top-most subgoal and the selected subgoal of a top-most looping node is called a top-most looping subgoal. The inter-dependent subgoals under a top-most looping subgoal form a cluster.

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. 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

Linear Tabling Strategies and Optimization Techniques,
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.

3. LINEAR TABLING STRATEGIES

Linear tabling relies on iterative evaluation of top-most looping subgoals to compute fixpoints. A linear tabling method can be defined in terms of the strategies used in the three primitives: `table_start(A)`, `memo(A)`, and `check_completion(A)`. 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 devised to produce and return answers. The lazy answer consumption strategy (lazy strategy in short) gives priority to answer production and the eager answer consumption strategy (eager strategy in short) prefers answer consumption to production. In the following we define the three primitives in the lazy and eager strategies, respectively.

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.

`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 `temporary complete` 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, it is resolved by using rules. During each round, a pioneer subgoal needs to be evaluated only once.\(^1\)

If \(A\) is a follower of some ancestor \(A_0\), meaning that a loop has been encountered, then either of following tactics can be used:

— **Resolving by using answers.** \(A\) 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 \(A\) needs be iterated until no new answer can be produced.

— **Stealing alternative rules.** \(A\) is resolved by using the alternative rules of \(A_0\). After \(A_0\) is taken over by \(A\), the dummy ending rule becomes the only alternative rule for \(A_0\). This means that \(\text{check completion}(A_0)\) will be executed when control backtracks to \(A_0\). Since the rule that leads to the loop from \(A_0\) to \(A\) has not been completely evaluated, stealing the alternative rules may lose answers. For this reason, the top-most looping subgoal of \(A_0\) needs be iterated until the fixpoint is reached.

\(\text{memo}(A)\). 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 \(\text{memo}\) postpones the consumption of answers until all rules have been tried.

\(\text{check completion}(A)\). 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 \text{complete} and \(A\) is failed after all the answers are consumed.

If \(A\) is a follower that is being resolved by its ancestor’s alternative rules, then \(A\) turns to return answers and fails after all the existing 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 \(A\). If so, \(A\) is re-evaluated by calling \(\text{table start}(A)\) after all the dependent subgoals’ states are initialized. Otherwise, if no new answer is produced, \(A\) is resolved by answers after its state being set to \text{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 \text{temporary complete}. Notice that \(A\)'s state cannot be set to \text{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. \text{Temporarily complete} subgoals are evaluated only once during each round.

**Example.** Consider the following example program and the query \(p(a,Y0)\). We assume that the tactic \(\text{stealing alternative rules}\) is used.

\[
p(X,Y):-p(X,Z),e(Z,Y),\text{memo}(p(X,Y)). \quad (\text{ri})
\]

\(^1\)If evaluating a subgoal produces some new answers then the top-most looping subgoal will be re-evaluated and so will the subgoal. If evaluating a subgoal does not produce any new answer, then evaluating it again would not produce any new answers either. Therefore, it is safe to evaluate a subgoal only once in each round.

Linear Tabling Strategies and Optimization Techniques,
\[ p(X,Y) : - e(X,Y), \text{memo}(p(X,Y)). \text{ (r2)} \]
\[ p(X,Y) : - \text{check} \_\text{completion}(p(X,Y)). \text{ (r3)} \]
\[ e(a,b). \]
\[ e(b,c). \]

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

1: \( p(a,Y0) \)
   \( \downarrow \) apply r1
2: \( p(a,Z1), e(Z1,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) loop found, use r2 of \( p(a,Y0) \)
3: \( e(a,Z1), \text{memo}(p(a,Z1)), e(Z1,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) apply \( c(a,b) \)
4: \( \text{memo}(p(a,b)), e(b,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) add answer \( p(a,b) \)

In goal 2, the follower \( p(a,Z1) \) steals the alternative rules from its pioneer \( p(a,Y0) \). After the answer \( p(a,b) \) is added into the table, \( \text{memo}(p(a,b)) \) fails. The failure forces execution to backtrack to goal 2.

2: \( p(a,Z1), e(Z1,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) use r3
5: \( \text{check} \_\text{completion}(p(a,Z1)), e(Z1,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) use answer \( p(a,b) \)
6: \( e(b,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) use \( e(b,c) \)
7: \( \text{memo}(p(a,c)) \)

Since \( p(a,Z1) \) is a follower being resolved by using its parent’s alternative rules, \( \text{check} \_\text{completion}(p(a,Z1)) \) just returns answers in the table. Consuming the answer \( p(a,b) \) leads to the production of the second answer \( p(a,c) \). After that, execution backtracks to goal 5.

5: \( \text{check} \_\text{completion}(p(a,Z1)), e(Z1,Y0), \text{memo}(p(a,Y0)) \)
   \( \downarrow \) use answer \( p(a,c) \)
6: \( e(c,Y0), \text{memo}(p(a,Y0)) \)

\( e(c,Y0) \) fails and execution backtracks to the top subgoal goal \( p(a,Y0) \) on which the clause r3 is tried.

1: \( p(a,Y0) \)
   \( \downarrow \) apply r3
8: \( \text{check} \_\text{completion}(p(a,Y0)) \)

Since the new answers \( p(a,b) \) and \( p(a,c) \) are 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 \( p(a, c) \).

### 3.2 The eager strategy

The eager strategy uses answers first to resolve tabled subgoals if there are answers available. A pioneer turns to produce answers only when no answer is available and more answers are demanded. Since answers are returned eagerly, a pioneer and a follower may not have the ancestor-decedent relationship. Consider, for example, the goal \( \text{“} p(X), p(Y) \text{”} \) where \( p/1 \) is defined by facts. The subgoal \( p(Y) \) is encountered before \( p(X) \) is completely evaluated. It is in general unsafe to let a follower steal its pioneer’s choice point if the pioneer is not an ancestor [Shen et al. 2001] and it is expensive to test the ancestor-decedent relationship at runtime. For this reason, we do not consider the stealing-alternative tactic for the eager strategy. The following shows how the three primitives behave differently in the eager strategy.

**table_start(\( A \)).** This primitive is the same as that for the lazy strategy except when \( A \) is a pioneer that has never occurred in the current round. If \( A \) is such a pioneer, then it is resolved by using answers first. After all the existing answers are exhausted, it is resolved by using rules. As seen in the definition of the primitive memo(\( A \)), an answer is returned immediately after it is created.

**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.

**check_completion(\( A \)).** If \( A \) is a top-most looping subgoal, just as the definition 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 state is 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 temporary complete. Notice that unlike in the lazy strategy, the primitive check_completion(\( A \)) does not return any answers in the eager strategy. As described above, all available answers must have been returned by table_start(\( A \)) and memo(\( A \)) by the time check_completion(\( A \)) is executed.

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 \( \text{“} p(X), p(Y) \text{”} \).

\[
\begin{align*}
p(1):= & \text{memo}(p(1)). & (r1) \\
p(2):= & \text{memo}(p(2)). & (r2) \\
p(X):= & \text{check_completion}(p(X)). & (r3)
\end{align*}
\]

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

\[
\begin{align*}
1: & p(X), p(Y) \\
\downarrow & \text{use } r1 \\
2: & \underline{\text{memo}(p(1))}, p(Y) \\
\downarrow & \text{add answer } p(1)
\end{align*}
\]

Linear Tabling Strategies and Optimization Techniques,
When the subgoal \( p(Y) \) is encountered, the system treats it as a follower and uses the tabled answer \( p(1) \) to resolve it. After that the first solution \((1,1)\) is returned. When execution backtracks to \( p(X) \), the second answer \( p(2) \) is added into the table.

\[
\begin{align*}
3: & \quad p(Y) \\
\quad \downarrow \text{use answer } p(1)
\end{align*}
\]

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 used.

\[
\begin{align*}
1: & \quad p(X), p(Y) \\
\quad \downarrow \text{use } r2 \\
4: & \quad \text{memo}(p(2)), p(Y) \\
\quad \downarrow \text{add answer } p(2) \\
5: & \quad p(Y) \\
\quad \downarrow \text{use answer } p(1)
\end{align*}
\]

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.

4. SEMI-NAIVE EVALUATION

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. The semi-naive algorithm [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 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.

4.1 Preparation

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

\[
\begin{array}{|c|c|c|}
\hline
\text{old} & \text{previous} & \text{current} \\
\hline
\end{array}
\]

The names of the regions indicate the rounds during which the answers in the regions are produced: \textit{old} means that the answers were produced before the previous round, \textit{previous} the answers produced during the previous round, and \textit{current} the answers produced in the current round. The answers stored in \textit{previous} and \textit{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 \( m(p_1, \ldots) = m(p) \) for any subgoal \( p(\ldots) \).

**Definition 6.** For a program, a level mapping \( m \) represents the call graph if for each rule \( H : \text{A}_1, \ldots, \text{A}_n \) in the program, \( m(H) > m(\text{A}_i) \) iff the predicate of \( H \) does not call either directly or indirectly the predicate of \( \text{A}_i \) and \( m(H) = m(\text{A}_i) \) iff the predicates of \( H \) and \( \text{A}_i \) 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 \( \text{A}_i \) and \( \text{A}_j \) occur in a loop, then \( m(\text{A}_i) = m(\text{A}_j) \) but not vice versa.

**Definition 7.** Let \( H : \text{A}_1, \ldots, \text{A}_n \) be a rule in a program and \( m \) be the level mapping that represents the call graph of the program. \( \text{A}_k \) is called the last depending subgoal of the rule if \( m(\text{A}_k) = m(H) \) and \( m(H) > m(\text{A}_i) \) for \( i > k \).

The last depending subgoal \( \text{A}_k \) 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 \( \text{A}_k \) that have occurred before must already be complete.

**Definition 8.** Let \( H : \text{A}_1, \ldots, \text{A}_n \) 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., \( m(H) > m(\text{A}_i) \) for \( i = 1, \ldots, n \), then the rule is called a base rule.

### 4.2 Semi-naive evaluation

**Theorem 1.** Let \( H : \text{A}_1, \ldots, \text{A}_k, \ldots, \text{A}_n \) be a rule where \( \text{A}_k \) 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 \( \text{A}_i, \ldots, \text{A}_k, \ldots, \text{A}_{k-1} \), if \( C \) has occurred in an early round and the combination does not contain any new answers, then it is safe to let \( \text{A}_k \) consume new answers only.

**Proof:** Let \( \text{A}_k^{old} \) and \( \text{A}_k^{new} \) be the old and new answers of the subgoal \( \text{A}_k \), respectively. For a combination of answers of \( \text{A}_1, \ldots, \text{A}_{k-1} \), if the combination does not contain new answers then the join of the combination and \( \text{A}_k^{old} \) must have been done and all possible answers for \( C \) that can result from the join must have been consumed.
produced during the previous round because the subgoal \( C \) has been encountered before. Therefore only new answers in \( A_{\text{new}} \) should be used. \( \square \)

**Theorem 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:

\[ ?- p(X, Y). \]

\[
p(X, Y) :- p(X, Z), q(Z, Y). \quad (C1)
p(b, c) :- p(X, Y). \quad (C2)
p(a, b). \quad (C3)
q(c, d) :- p(X, Y), t(X, Y). \quad (C4)
t(a, b). \quad (C5)
\]

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.

### 4.3 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 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 \( P_a \Join (Q_p \cup Q_c) \) 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:

\[
Q'_o = Q_o \cup Q'_p \\
Q'_p = Q_p \cup Q_c \\
\]

where \( \alpha \) 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.

\[
P_a \Join (Q'_p \cup Q'_c) = P_a \Join (Q_c \cup \alpha \cup Q_c')
\]

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.
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, 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 9.** 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:

---

3No 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 $T'$.

Linear Tabling Strategies and Optimization Techniques,
?- 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 j. We distinguish between the following two cases:

1. The last depending subgoal A_k is not selected in round j. 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 j. 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.

☐

5. IMPLEMENTATION

Changes to the Prolog machine ATOAM [Zhou 1996] are needed to implement linear tabling. This section describes the changes to the data structures and the instruction set.

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:
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 temporary complete. 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 field 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:

- SubgoalTable
- CurrentAnswer

The SubgoalTable points to the subgoal table entry. 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:

\[
\% p(X,Y):-p(X,Z),e(Z,Y).
\]
tcl: \[\text{tcl}(X,Y) :- \text{edge}(X,Y). \]
\[\text{tcl}(X,Y) :- \text{tcl}(X,Z), \text{edge}(Z,Y). \]

tcr: \[\text{tcr}(X,Y) :- \text{edge}(X,Y). \]
\[\text{tcr}(X,Y) :- \text{edge}(X,Z), \text{tcr}(Z,Y). \]

tcn: \[\text{tcn}(X,Y) :- \text{edge}(X,Y). \]
\[\text{tcn}(X,Y) :- \text{tcn}(X,Z), \text{tcn}(Z,Y). \]

sg: \[\text{sg}(X,X). \]
\[\text{sg}(X,Y) :- \text{edge}(X,XX), \text{sg}(XX,YY), \text{edge}(Y,YY). \]

Fig. 2. Datalog programs.

\%
\[\text{p}(X,Y) :- e(X,Y). \]
\[\text{p/2}: \text{table} \_\text{start} 2,1,r3 \]
\[\text{fork} \text{r2} \]
\[\text{para} \_\text{value} y(2) \]
\[\text{para} \_\text{var} y(-13) \]
\[\text{call} \text{p/2} \]
\[\text{para} \_\text{value} y(-13) \]
\[\text{para} \_\text{value} y(1) \]
\[\text{call} \text{e/2} \]
\[\text{memo} \]
\[\text{r2}: \text{fork} \text{r3} \]
\[\text{para} \_\text{value} y(2) \]
\[\text{para} \_\text{value} y(1) \]
\[\text{call} \text{e/2} \]
\[\text{memo} \]
\[\text{r3}: \text{check} \_\text{completion} \text{p/2} \]

The table\_start instruction takes three operands. In the example, they are, respectively, the arity of the predicate (2), the number of local variables (1), and the entrance of the ending rule (r3). 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

6. PERFORMANCE EVALUATION

The semi-naive evaluation technique described in this paper has been implemented in B-Prolog version 6.6. 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 [Demoen and Sagonas 1999]. All the benchmarks are available from probp.com/bench.tar.gz.

Table I 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

\footnote{Available from www.probp.com.}
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 [Zhou and Sato 2003], 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 II 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 III. BP vs. XSB (CPU time).

<table>
<thead>
<tr>
<th>program</th>
<th>BP XP</th>
<th>XSB XP</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcl</td>
<td>1</td>
<td>1.80</td>
<td>1.36</td>
</tr>
<tr>
<td>tcr</td>
<td>1</td>
<td>1.40</td>
<td>1.06</td>
</tr>
<tr>
<td>tcn</td>
<td>1</td>
<td>1.25</td>
<td>1.11</td>
</tr>
<tr>
<td>sg</td>
<td>1</td>
<td>1.14</td>
<td>1.10</td>
</tr>
<tr>
<td>cs</td>
<td>1</td>
<td>0.68</td>
<td>0.53</td>
</tr>
<tr>
<td>cs</td>
<td>1</td>
<td>0.82</td>
<td>0.64</td>
</tr>
<tr>
<td>disj</td>
<td>1</td>
<td>0.59</td>
<td>0.48</td>
</tr>
<tr>
<td>gabriel</td>
<td>1</td>
<td>0.61</td>
<td>0.50</td>
</tr>
<tr>
<td>kalah</td>
<td>1</td>
<td>1.00</td>
<td>0.77</td>
</tr>
<tr>
<td>pg</td>
<td>1</td>
<td>1.05</td>
<td>0.88</td>
</tr>
<tr>
<td>peep</td>
<td>1</td>
<td>0.37</td>
<td>0.37</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>0.45</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Table III compares the CPU times taken by B-Prolog (BP) and XSB (version 2.6) to run the programs. For BP, the auto-tabling optimization, which could speed-up the Chat programs by several times, is disabled. 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 [Zhou et al. 2001; Guo and Gupta 2001]. Our lastest implementation, while maintaining good space efficiency, offers comparable speed performance with XSB even without auto-tabling optimization.

---

6 *Auto-tabling* is an optimization technique that tables extra predicates to avoid re-evaluation of their subgoals.

7 Semi-naive evaluation does not affect the stack space performance.
7. RELATED WORK

A history of tabling and linear tabling, tabling strategies, related work by [Freire et al. 1998]. related work on semi-naive evaluation.

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 [Zhou and Sato 2003].

Semi-naive evaluation is a fundamental idea for reducing redundancy in bottom-up evaluation of logic database queries [Bancilhon and Ramakrishman 1986; Ullman 1988]. As far as we know, its impact on top-down evaluation has been unknown before this work. 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, 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:

\[ H : \neg P, Q. \]

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

\[ H : \neg \Delta P, Q. \]
\[ H : \neg P, \Delta Q. \]

Where \( \Delta P \) 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 [Bancilhon and Ramakrishnan 1986]. 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 [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

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 ODLT.

9. ACKNOWLEDGEMENT

Neng-Fu 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.

REFERENCES


Linear Tabling Strategies and Optimization Techniques,
