New Generation Computing, 15 (1997) 141-162
OHMSHA, LTD. and Springer-Verlag

On the Completeness of Naive Memoing in Prolog

Suzanne Wagner DIETRICH* and Changguan FAN
Department of Computer Science and Engineering,
College of Engineering and Applied Sciences,
Arizona State University,
Box 875406, Tempe, AZ 85287-5406 USA.

Received 29 December 1994
Revised manuscript received 18 December 1995

Abstract Memoing is often used in logic programming to avoid redundant evaluation of similar goals, often on programs that are inherently recursive in nature. The interaction between memoing and recursion, however, is quite complex. There are several top-down evaluation strategies for logic programs that utilize memoing to achieve completeness in the presence of recursion. This paper's focus, however, is on the use of *naive* memoing in Prolog. Using memoing *naively* in conjunction with recursion in Prolog may not produce expected results. For example, adding naive memoing to Prolog's evaluation of a right-recursive transitive closure may be incomplete, whereas adding naive memoing to Prolog's evaluation of a left-recursive transitive closure may be terminating and complete. This paper examines the completeness of naive memoing in linear-recursive, function-free logic programs evaluated with Prolog's top-down evaluation strategy. In addition, we assume that the program is definite and safe, having finite base relations and exactly one recursive predicate. The goal of the paper is a theoretical study of the completeness of naive memoing and recursion in Prolog, illustrating the limitations imposed even for this simplified class of programs. The naive memoing approach utilized for this study is based on extension tables, which provide a memo mechanism with immediate update view semantics for Prolog programs, through a source transformation known as ET. We introduce the concept of *ET-complete*, which refers to the completeness of the evaluation of a query over a Prolog program that memos selected predicates through the ET transformation. We show that left-linear recursions defined by a single recursive rule are ET-complete. We generalize the class of left-linear recursions that are ET-complete by introducing pseudo-left-linear recursions, which are also defined by a single linear

* This work was partially supported by the National Science Foundation under Grant CCR-9008737.

recursive rule. To add left-linear recursions defined by *multiple* linear recursive rules to the class of ET-complete recursions, we present a left-factoring algorithm that converts left-linear recursions defined by multiple recusive rules into pseudo-left-linear recursions defined by a single recursive rule. Based on these results, the paper concludes by identifying research directions for expanding the class of Prolog programs to be examined in future work.

Keywords: Memoing, Logic Programming, Recursion.

§1 Introduction

Memoing is used to save intermediate results in a computation. Memoing in functional programming is used to eliminate repeated computations. Recursively defined functions may have redundant computations in that subsidiary function values may be evaluated more than once. With a memoing mechanism, the function values are computed once and then stored for later use, thus improving the efficiency of the program execution.[3,11] Memoing in logic programming achieves a similar effect of avoiding redundant evaluation of a similar goal,[8] often on programs that are inherently recursive in nature. The interaction between memoing and recursion in logic programming, however, is quite complex. One important issue is the effect of memoing on the completeness of the evaluation of recursive predicates.[25]

Complete evaluation strategies for logic programs (ET*,[6] OLDT,[20] QSQR,[23] XWAM[24]) use memoing as a tool to modify the top-down evaluation of a logic program to be complete. The evaluation strategies utilize a memo table so that subsequent calls, whose answers are in the table, use a table lookup rather than recomputing the results. The table lookup effectively places a bound on the search down an otherwise possibly infinite path, and thus, is a mechanism that provides for a fair search strategy. The complete evaluation strategies *manage* the memo table and *cleverly* direct the execution of the program to provide a complete evaluation. Note that this completeness mechanism has the advantageous side-effect of avoiding duplicate computation for similar subgoals. Unfortunately, using memoing *naively* in conjunction with recursion may not produce expected results. For example, given the following right-recursive specification of transitive closure, Prolog's evaluation of the query $tc(X, Y)$ is complete for any acyclic *edge/2* relation.

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

However, simply memoing the resulting tuples of *tc/2* (to improve efficiency) may make the evaluation incomplete (see detailed example in Section 2.2). The goal of this paper is to show when this *naive* memoing evaluation of linear

recursive Prolog programs is complete.

Complete and efficient recursive query evaluation has also been a focused research issue in deductive databases,[2] which use a function-free logic programming language, known as Datalog, to access the database declaratively. In this database context, where all answers are found as soon as possible, the evaluation of a logic program assumes an eager set-at-a-time paradigm using a primarily bottom-up strategy. This forward-chaining computation naturally memos the derived relations while eliminating duplicates but typically, due to its eager evaluation strategy, derives results that are irrelevant in answering the query. In the logic programming context, where the first answer is found as soon as possible, the evaluation of a logic program assumes a lazy tuple-at-a-time evaluation strategy. This backward-chaining computation naturally focuses the search on data relevant to finding an answer but duplicates computation in the process, which may cause an unfair search down an infinite path. Thus, much of the research on efficient and complete evaluation strategies combine top-down and bottom-up characteristics.[4] Forward-chainig strategies, such as magic sets,[1] add top-down filtering to bottom-up evaluation to filter out irrelevant computations. Backward-chaining strategies, such as the ET* algorithm,[6] add bottom-up memoing to top-down evaluation to avoid duplicate computation. It is now recognized that the evaluation of a program and query by a bottom-up strategy with filtering and a top-down strategy with memoing are quite similar.[25]

While there is a need for complete evaluation strategies that handle general recursions, there is a trade-off of efficiency for generality. The deductive database literature motivates the need for more efficient (and complete) evaluation strategies that optimize the evaluation of simpler, specialized recursions.[22] Linear recursions, which have at most one recursive subgoal in the rule body, are an example of a simple recursion that appears quite frequently in practice. Specialized recursive query evaluation strategies have been proposed for linear recursions[15,16] in the database context. Examples of linear recursions include the well-known left- and right-recursive versions of transitive closure.

The main contribution of this paper is a careful study on the completeness of *naive* memoing added to Prolog's fixed top-down evaluation strategy. We begin this ongoing investigation by focusing initially on linear recursions specified by pure, function-free logic programs. Since these specialized recursions occur frequently in the database context, we will make simplifying assumptions for these initial results consistent with deductive databases.[21,22] Specifically, we assume:

- The program is a definite, function-free logic program.
- The program is linear recursive and contains only one recursive predicate.
- All rules in the program are safe. A rule is safe if all of the variables appearing in the rule, either in the head or the body, appear in a positive literal in the body of the rule.

- All base relations are finite and consist of ground terms.
- The program is rectified, i.e., all literals, including the head of each rule, have distinct variables for each argument. This (merely syntactic) restriction disallows repeated variables and constants appearing in the literal and thus, does not permit implicit aliasing of variables.

The paper assumes that the reader is familiar with Prolog and its top-down, tuple-at-a-time evaluation strategy. For the non-specialist in the field of memoing, Section 2 discusses memoing, extension tables and recursion. Specifically, Section 2.1 reviews the ET strategy for managing memo tables, known as *extension tables*, through a source transformation in Prolog. An important feature of the memo table management is the support of immediate update view semantics so that the effects of any changes to the extension tables are immediately visible. The original implementation of the ET source transformation[6] uses assert and retract to maintain the extension tables as part of Prolog's database. A more efficient implementation,[9] which is described in Section 2.1, maintains extension tables as a global data structure and provides built-in predicates for table manipulation. Section 3 defines left-linear recursions and the concept of *ET-complete*, which refers to the completeness of the evaluation of a query over a Prolog program that memos selected predicates through the ET transformation where immediate update view semantics are supported for the memoed results. We show that left-linear recursions defined by a single recursive rule are ET-complete. In Section 4, we extend the completeness results to a more general class of recursions that we call *pseudo-left-linear recursions*, which are also defined by a single linear recursive rule. To add left-linear recursions defined by *multiple* (linear) recursive rules to the class of ET-complete recursions, we present, in Section 5, a left-factoring algorithm that converts left-linear recursions defined by multiple recursive rules into pseudo-left-linear recursions defined by a single recursive rule. In Section 6, we conclude the paper with a discussion of future research directions.

§2 Memoing

We clarify our use of the term *memoing* by describing a memoing meta-interpreter for pure function-free Prolog. We then review the ET strategy for managing the memo tables, known as extension tables, through a source transformation in Prolog. We also provide a motivational example that illustrates how memoing affects the completeness of the top-down evaluation of a recursive goal.

Figure 1 presents a memoing meta-interpreter that saves all calls and solutions. The interpreter maintains two relations, called/1 and solution/1, to save calls and solutions, respectively. When evaluating a call, the meta-interpreter checks to see if the call is subsumed by any previous calls. $Call_1$ is said to be subsumed by $call_2$ if there is a substitution for variables of $call_2$ such that the

```
/*r1*/ solve(true) :- !.
/*r2*/ solve((A, B)) :- !, solve(A), solve(B).
/*r3*/ solve(A) :- subsumed(called(A)), !,
                   solution(A).
/*r4*/ solve(A) :- save(called(A)),
                   (solution(A)
                   ;
                   rule(A, B),
                   solve(B),
                   not(subsumed(solution(A))),
                   save(solution(A))).
```

Fig. 1 A memoing meta-interpreter

resulting literal is the same as $call_1$. If the current call is subsumed by a previous call, then answers that satisfy the call are retrieved from the memo relation solution/1 instead of recomputing them. Otherwise, the new call is saved in called/1. Note that a new call may be a generalization of a previous one. In this case, solutions to the previous call are also solutions to the new call. For example, if p(X, Y) is the current call and p(a, X) is a previous call, then all solutions to p(a, X) are also solutions to p(X, Y). The subgoal solution/1 in rule r4 is used to retrieve those answers before calculating new answers using the given rules. An answer is saved in solution/1 if it is not subsumed by any existing solutions.

The meta-interpreter introduces an extra level of interpretation. For a given program, this extra level of interpretation can be removed through partial evaluation that unfolds[19] the meta-interpreter using the given program. Another disadvantage of using the meta-interpreter is that it saves all calls and answers, of which only some of them are used in a later computation. Due to the cost associated with maintaining the memo relations, it is often desirable to use memoing selectively.

2.1 Extension Tables

Extension tables[6] are a memo mechanism that can be attached to predicates selectively. An extension table consists of two subtables: a call table and an answer table. The call table stores calls to the predicate; the answer table stores answers to the predicate. Through a source-to-source program transformation, extension tables can be selectively attached to predicates so that they can be evaluated with memoing. The original implementation of extension tables uses assert and retract to maintain extension tables as part of Prolog's database.[6] A more efficient implementation maintains extension tables as a global data structure and provides built-in predicates for table manipulation.[9] For a given program $\mathcal{P}$ and a list $\mathcal{L}$ of predicates to be evaluated with memoing, Algorithm 2.1, denoted by ET($\mathcal{P}$, $\mathcal{L}$), translates the program $\mathcal{P}$ into one with memoing on the predicates given in $\mathcal{L}$. We use the notation $\mathcal{P}_{et(\mathcal{L})}$ to denote the resultant program.

Algorithm 2.1 uses the following built-in predicates for table manipula-

tion:

- et_subsume(CallAnswer, p/n, p(X_1, ..., X_n))
  check extension table p/n to see if the term p(X_1, ..., X_n) is subsumed by any terms in the call or answer subtable specified by argument CallAnswer, denoted by either call or answer.
- et_insert(CallAnswer, p/n, p(X_1, ..., X_n))
  et_insert_unique(CallAnswer, p/n, p(X_1, ..., X_n))
  insert term p(X_1, ..., X_n) into the call or answer subtable of p/n specified by CallAnswer. The difference between et_insert/3 and et_insert_unique/3 is that the latter checks for subsumption before insertion. With et_insert_unique/3, a tuple is inserted only if it is not subsumed by any existing tuples in the extension table. Note that an inserted tuple is added to the *end* of the list of calls or answers in the appropriate subtable.
- et_retrieve(CallAnswer, p/n, p(X_1, ..., X_n))
  retrieve calls or answers from extension table p/n specified by CallAnswer. The bindings in term p(X_1, ..., X_n) specify the selection condition for retrieval. This built-in predicate is nondeterministic, so all calls or answers satisfying the selection condition can be retrieved through backtracking.

Example 1

Let $\mathcal{P}$ denote the (left-recursive) transitive closure program given in Fig. 2. Figure 3 gives the result of Algorithm 2.1, ET($\mathcal{P}$, [tc/2]), with memoing added to the recursive predicate tc/2.

```
/*r1*/ tc(X, Y) :- edge(X, Y).
/*r2*/ tc(X, Y) :- tc(X, Z), edge(Z, Y).
```

Fig. 2 A left-recursive transitive closure specification

```
/*r1'*/ code_tc(X, Y) :- edge(X, Y).
/*r2'*/ code_tc(X, Y) :- tc(X, Z), edge(Z, Y).

/*r3*/ tc(X, Y) :- et_subsume(call, tc/2, tc(X, Y)), !,
                   et_retrieve(answer, tc/2, tc(X, Y)).

/*r4*/ tc(X, Y) :- et_insert(call, tc/2, tc(X, Y)),
                   (et_retrieve(answer, tc/2, tc(X, Y))
                   ;
                   code_tc(X, Y),
                   et_insert_unique(answer, tc/2, tc(X, Y))).
```

Fig. 3 $\mathcal{P}_{et([tc/2])}$

2.2 Recursion

Memoing is often used in logic programming to avoid redundant evaluation of similar goals, often on programs that are inherently recursive in nature.

Algorithm 2.1 ET translation algorithm ET($\mathcal{P}$, $\mathcal{L}$).
Input: A program $\mathcal{P}$ and a list $\mathcal{L}$ of predicates to be evaluated with memoing.
Output: A program with memoing for selected predicates given by $\mathcal{L}$.

(1) For each predicate p/n in $\mathcal{L}$, do the following:

(a) Rewrite each rule defining p/n by changing the rule head from:
$p(X_1, ..., X_n)$:- .....
to
code_p(X_1, ..., X_n) :- .....

(b) Reorder all rules of code_p/n such that the nonrecursive rules appear before the recursive rules.

(c) Define two new rules for p/n:

```
p(X1, ..., Xn) :- et_subsume(call, p/n, p(X1, ..., Xn)), !,
                  et_retrieve(answer, p/n, p(X1, ..., Xn)).
p(X1, ..., Xn) :- et_insert(call, p/n, p(X1, ..., Xn)),
                  (et_retrieve(answer, p/n, p(X1, ..., Xn))
                   ;
                   code_p(X1, ..., Xn),
                   et_insert_unique(answer, p/n, p(X1, ..., Xn))).
```

(2) All other rules are unchanged.

Although memoing is straightforward for nonrecursive programs, the interaction of memoing in a recursive program is quite complex. One important issue is the effect of memoing on the completeness of the evaluation of recursive predicates. We use the transitive closure of a simple directed acyclic graph, as shown in Fig. 4, to illustrate the effects of memoing and recursion.

```
edge(c, d).
edge(b, c).
edge(a, b).
```

Fig. 4 Edge relation

Consider a right-recursive specification of transitive closure expressed in Prolog as shown in Fig. 5. Prolog's evaluation of the query tc(S, D) on this program finds all answers, which are {(c, d), (b, c), (a, b), (b, d), (a, d), (a, c)}. A memoing evaluation in Prolog, however, will fail to find all the answers. The invocation of the query remembers that the most general call has been made to the predicate tc. The nonrecursive rule computes the answers that are edges: (c, d), (b, c) and (a, b). The recursive rule first uses edge(c, d) to generate the

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

Fig. 5 A right-recursive transitive closure specification

subgoal tc(X, c). The subgoal tc(X, c) is a selection of the answers computed for the more general query call tc(S, D). Therefore, the subgoal tc(X, c) uses the extension table to retrieve (b, c), which generates the answer (b, d) to the query. Backtracking into edge, the evaluation then uses edge(b, c) to generate the subgoal tc(X, b), which retrieves (a, b) from the extension table to compute the query answer (a, c). Note that this new answer (a, c) is an answer for the subgoal tc(X, c), which the evaluation already failed. Continuing to backtrack into edge, the evaluation uses edge(a, b) to generate the subgoal tc(X, a), which does not result in the computation of any answers. Thus, the memoing evaluation of this right-recursive specification of transitive closure is incomplete, as indicated by the missing solution (a, d).

Consider a left-recursive specification of transitive closure expressed in Prolog as shown in Fig. 2. Prolog's evaluation of the query tc(S, D) on this program *also* finds all answers *but does not terminate*. The left-recursive specification enters an infinite loop caused by an infinite unfolding of the recursive rule. So a Prolog programmer, who has to be cognizant of both the program logic *and* control, will not choose a left-recursive specification of transitive closure. But how does memoing affect Prolog's evaluation of left-recursion?

The first question that must be addressed is: how is memoing implemented? We have already reviewed the extension table implementation of memoing using extension table built-ins. Other implementations of memoing in Prolog suggest the use of asserts to save the computed answer. One problem with assert is the semantics for dynamic predicates.[13] Most Prolog implementations choose the *logical view* semantics for dynamic predicates where answers asserted or retracted during the call to a dynamic predicate are not visible to that call. The extension table built-in implementation, however, provides *immediate update view* semantics where the effects of any insertions or deletions to the extension table are immediately visible.

Let's now consider Prolog's evaluation of the query tc(S, D) on $\mathcal{P}_{et(\{tc/2\})}$, the left-recursive transitive closure specification of Fig. 2 in conjunction with the extension tables memoing strategy. The invocation of the query remembers that the most general call has been made to the predicate tc. The nonrecursive rule computes the answers that are edges, {(c, d), (b, c), (a, b)}, and saves the answers in the memo table for tc. The recursive rule then generates the subgoal tc(X, Z) and uses a table look-up for this repeated call. Logical view semantics limits the answers for the recursive subgoal to the answers that have been computed thus far, which are the edges. Thus, the recursive rule in conjunction with logical view semantics will only compute paths in the graph of length 2, consisting of edges joined with edges, and fail to be complete. Immediate update view semantics, however, will add the paths of length 2 to the end of the memoed answers and continued backtracking into the recursive subgoal allows for *all* answers to be used. Thus, Prolog's evaluation of a left-recursive transitive

closure with a memoing strategy having immediate update view semantics is complete.

Up to this point, we were considering the transitive closure of a directed *acyclic* graph because having cycles in the underlying graph would cause Prolog's evaluation of the right-recursive transitive closure to be incomplete. The transitive closure of a directed graph having cycles is handled naturally by Prolog's memoing evaluation, with immediate update view semantics, of the left-recursive transitive closure program of Fig. 2.

The varying results on these simple (yet important) transitive closure examples demonstrates that naive memoing, although an apparantly straightforward approach for memoing in Prolog, does not produce expected results in the presence of recursion. These results, however, are not surprising for those familiar with the top-down complete evaluation strategies, such as ET*, QSQR and OLDT, which use sophisticated memoing approaches to provide a complete evaluation of a recursive logic program. This work examines the complex interaction of recursion and naive memoing, where the ET strategy represents a practical approach to providing database memoing in Prolog. In the following sections, we identify classes of linear recursions that can be completely evaluated with naive memoing using Prolog's top-down evaluation strategy in conjunction with immediate update view semantics for memoed results.

§3 Left-Linear Recursions

This section focuses on specialized recursions, known as *left-linear* recursions. We formally define left-linear recursions and introduce an optimized memoing strategy for these specialized recursions. This section also introduces the concept of *ET-complete*, which refers to the completeness of the evaluation of a query over a Prolog program that memos slected predicates through the ET source transformation. We show that left-linear recursions defined by a single recursive rule are ET-complete.

The complete evaluation of a query is dependent both on the program and the query. Given a query, the arguments of the query may be bound or free. Typically, a **binding pattern** or **adornment** for the query is used to indicate how the program will be invoked by that query. If we use b to denote a bound argument and use f to denote a free argument, then the adornment of a query is a string of b's and f's. For example, the adornment for tc(a, X) is bf. A query on predicate p with adornment α is denoted by the adorned query p^{α}.

Definition 3.1
If α and β are two adornments for the same predicate, then $\beta \leq \alpha$ if α has b in every position where β has b.

The *bound-is-easier assumption*[22] states that if we can evaluate a query with fewer arguments bound, then a query with more arguments bound is easier to evaluate since the solutions to the latter is a selection of the solutions to the

former. In other words, the bound is easier assumption indicates that if $\beta \leq \alpha$, then p^{α} would not be harder to evaluate than p^{β}.

We now define the left-linear[16] property of a linear recursive program with respect to the binding pattern of a given query on the recursive predicate.

Definition 3.2
Given a linear recursive program with a single recursive predicate p, p is **left-linear** with respect to an adornment α if for each recursive rule, when we list the p-subgoal first, the adornment on that subgoal is also α and the corresponding bound arguments in the head and the recursive subgoal share the same variable.

Intuitively, a left-linear recursion guarantees that the recursive call has the same binding pattern as the original call and that the bound arguments have the same bindings. This property guarantees that the recursive call is subsumed by the initial call. For example, the left-recursive program tc in Fig. 2 is left-linear with respect to the adorned queries tc^{bf} and tc^{ff}.

When a rule is evaluated with a top-down evaluation strategy, the subgoals in the rule are evaluated from left to right and the bindings are propagated through the unification of variables. We assume this left-to-right sideways information passing for binding pattern propagation. So after evaluating a subgoal, the unbound variables in that subgoal become bound. We use the **rule/goal graph**[22] to describe the propagation of bindings and interactions between goals and rules. The rule/goal graphs for the program tc with respect to the adorned queries tc^{bf} and tc^{ff} are given in Fig. 6 (a) and (b), respectively. A rule/goal graph consists of goal nodes and rule nodes with the given adorned query as the starting goal node. A goal node p^{α} has one rule node successor for each rule defining the predicate p. A rule node has one goal node successor for the next subgoal to be evaluated and a rule node successor with binding indications for the variables in the rule at that point in the computation. In Fig 6 (a), tc^{bf} and $edge^{bf}$ are goal nodes; while $r_{1.0}^{[X|Y]}$, $r_{2.0}^{[X|Y,Z]}$ and $r_{2.1}^{[X,Z|Y]}$ are rule nodes. Rule node $r_{i.j}^{[X_1,\ldots,X_m|Y_1,\ldots,Y_n]}$ represents that, for a given adornment on the rule

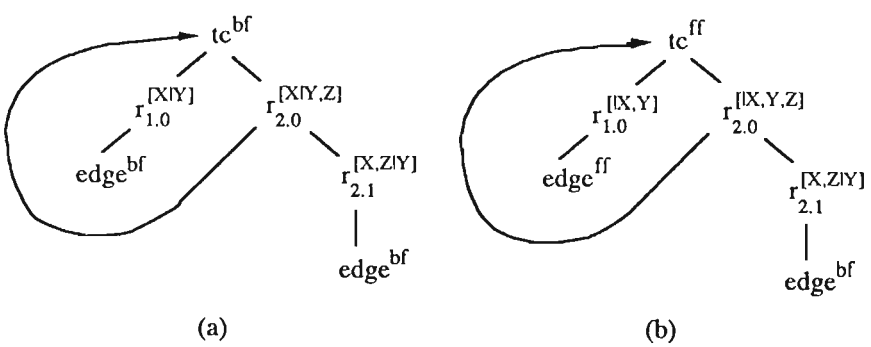

Fig. 6 Rule/goal graphs for adorned queries tc^{bf} and tc^{ff}

head, after evaluating the first j subgoals of rule i, variables X_1, ..., X_m are bound, while variables Y_1, ..., Y_n are still free. The rule node $r_{2,1}^{[X,Z|Y]}$ represents that both X and Z are bound and Y is still free after solving the first subgoal in rule r2. Figure 6 shows that for both adorned queries, the recursive subgoals have the same adornments as the adorned queries. Due to the definition of left-linear this is always the case, and furthermore, the recursive call is always subsumed by the initial query.

3.1 Left-Linear Optimization

Due to the left-linear property, the ET algorithm for left-linear recursions can be further optimized.[7] Namely, we can remove the subsumption checking for recursive calls since the calls are guaranteed to subsume each other for left-linear recursions. Figure 7 gives the optimized version of the ET algorithm for program tc. The original version of the ET algorithm for program tc appears in Fig. 3. Since the recursive call tc/2 in rule r2′ is subsumed by the first call, it is always a table lookup and hence rule r3 is deleted and tc/2 in rule r2′ is substituted with the et_retrieve/3 built-in predicate. In this optimized ET algorithm, rule r1″ returns the edge relation and the returned answers are stored in the extension table by rule r4″. Rule r2″ retrieves the answers from the extension table and joins them with the edge relation to generate new answers. The et_insert_unique/3 in r4″ inserts the new answer into the extension table if the answer is not subsumed by any existing answers. Since the extension table built-in predicates provide immediate update view semantics, when the answers in the extension table are used up, all possible solutions are calculated. The evaluation of the optimized ET algorithm is more efficient since unnecessary subsumption checking is removed.


```
/*r1"*/ code_tc(X, Y) :- edge(X, Y).
/*r2"*/ code_tc(X, Y) :- et_retrieve(answer, tc/2, tc(X, Z)),
                         edge(Z, Y).

/*r4"*/ tc(X, Y) :- code_tc(X, Y),
                    et_insert_unique(answer, tc/2, tc(X, Y)).
```

Fig. 7 Optimized ET algorithm for left-linear tc queries

3.2 Completeness

We have just seen how left-linear recursions allow for optimizations to the ET source transformation. We now introduce the concept of *ET-complete*, which refers to the completeness of the evaluation of a query over a Prolog program that memos selected predicates through the ET source transformation, and show that left-linear recursions defined by a single recursive rule are ET-complete. It is important to note that immediate update view semantics is required to guarantee ET-completeness.

We assume the terminology of[14] to define answers computed by Prolog. We assume Prolog computes only correct answers, i.e., those that are a logical consequence of the program. Furthermore, since we assume that all facts are ground and all rules in the program are safe, all computed answers are ground substitutions.

Definition 3.3
Let $\mathcal{P}$ be a (definite, function-free) logic program, $\mathcal{L}$ be a list of predicates defined in $\mathcal{P}$ and $\mathcal{P}_{et(\mathcal{L})}$ be the result of ET($\mathcal{P}$, $\mathcal{L}$) that translates $\mathcal{P}$ into a program with extension tables for the predicates given in $\mathcal{L}$. Given a query Q, if θ is a ground substitution for variables in Q as computed by Prolog with immediate update view semantics during the evaluation of $\mathcal{P}_{et(\mathcal{L})} \cup \{Q\}$, then θ is an **ET-computed answer**.

Definition 3.4
Let $\mathcal{P}$ be a (definite, function-free) logic program, $\mathcal{L}$ be a list of predicates defined in $\mathcal{P}$ and $\mathcal{P}_{et(\mathcal{L})}$ be the result of ET($\mathcal{P}$, $\mathcal{L}$) that translates $\mathcal{P}$ into a program with extension tables for the predicates given in $\mathcal{L}$. Given a query Q, Q is **ET-complete** with respect to the program $\mathcal{P}_{et(\mathcal{L})}$ if for any θ that is a correct answer for $\mathcal{P} \cup \{Q\}$, θ is an ET-computed answer for $\mathcal{P}_{et(\mathcal{L})} \cup \{Q\}$.

We now extend the definition of ET-completeness with respect to an adorned query, which represents a set of queries with the same adornment.

Definition 3.5
Let $\mathcal{P}$ be a (definite, function-free) logic program, $\mathcal{L}$ be a list of predicates defined in $\mathcal{P}$ and $\mathcal{P}_{et(\mathcal{L})}$ be the result of ET($\mathcal{P}$, $\mathcal{L}$) that translates $\mathcal{P}$ into a program with extension tables for the predicates given in $\mathcal{L}$. Given a predicate p and an adorned query p^{α}, p^{α} is **ET-complete** with respect to the program $\mathcal{P}_{et(\mathcal{L})}$ if for any θ that is a correct answer for $\mathcal{P} \cup \{Q\}$ for any query Q on p with adornment α, θ is an ET-computed answer for $\mathcal{P}_{et(\mathcal{L})} \cup \{Q\}$.

Theorem 3.1
Let $\mathcal{P}$ be a linear recursive (definite, function-free) logic program with a single recursive predicate p of arity n. Assume p has only one linear recursive rule and is left-linear with respect to the query adornment α. Then p^{α} is ET-complete with respect to $\mathcal{P}_{et([p/n])}$.

Proof
Let Q be a query on p with adormnent α. Based on the definition of ET-completeness, we must prove that every correct answer for $\mathcal{P} \cup \{Q\}$ is an ET-computed answer for $\mathcal{P}_{et([p/n])} \cup \{Q\}$.

Since the answers for $\mathcal{P} \cup \{Q\}$ are the answers calculated using the nonrecursive rules, unioned with the answers calculated using the recursive rule unfolded once, unioned with the answers calculated using the recursive rule unfolded twice, ..., and so on, we prove the theorem by induction on the number

of unfoldings of the recursive rule.

Basis: $i=0$, we need to show that all correct answers for $\mathcal{P} \cup \{Q\}$ calculated with the nonrecursive rules are ET-computed answers. This is obviously true since the ET transformation uses the bodies of the nonrecursive rules as given in the original program and guarantees that the nonrecursive rules are evaluated before the recursive rule.

Hypothesis: Let $i=k$ and assume that all correct answers for $\mathcal{P} \cup \{Q\}$ calculated with at most k steps of unfolding the recursion are ET-computed answers and therefore are saved in the extension table.

Induction: Let $i=k+1$ and assume that θ is a correct answer for $\mathcal{P} \cup \{Q\}$ calculated with $k+1$ steps of unfolding the recursion. We need to show that θ is an ET-computed answer.

Let p have m rules r_1, ..., r_m, of which only one is recursive:

r_1: p(X) :- $q_1(Y_1)$.

.....

r_m: p(X) :- p(Z), $q_m(Y_m)$.

where $q_j (j = 1, ..., m)$ represent the conjunction of nonrecursive subgoals in rules r_1, ..., r_m, respectively and X, $Y_j (j = 1, ..., m)$, and Z are tuples of arguments. Since p is left-linear with respect to the adornment α, the recursive subgoal is guaranteed to be subsumed by the original call and therefore, is a table lookup. By assumption, correct answers for $\mathcal{P} \cup \{Q\}$ calculated with at most k steps of unfolding the recursion are ET-computed answers and are saved in the extension table. Since the extension table memo mechanism guarantees immediate update view semantics while inserting a new answer at the end of the list of answers, backtracking into the table lookup will retrieve all the answers calculated with at most k steps of unfolding the recursion. These answers are joined with the nonrecursive subgoals q_m to generate correct answers corresponding to at most $k+1$ steps of unfolding the recursion. Thus, any correct answer for $\mathcal{P} \cup \{Q\}$ calculated with at most $k+1$ steps of unfolding the recursion is guaranteed to be an ET-computed answer. □

This section defined left-linear recursions and illustrated an optimization of the ET source transformation for left-linear recursions. Theorem 3.1 proved that left-linear recursions are ET-complete. Thus, for the motivational left-linear transitive closure example, Prolog's evaluation of the adorned queries tc^{bf} and tc^{ff} with memoing is complete. The next section examines the rule/goal graphs of the non-left-linear tc adornments, bb and fb, to motivate the generalization of the class of left-linear recursions that are ET-complete.

§4 Pseudo-Left-Linear Recursions

In this section, we extend the result of the last section to a more general class of linear recursions that we call *pseudo-left-linear* recursions. Intuitively,

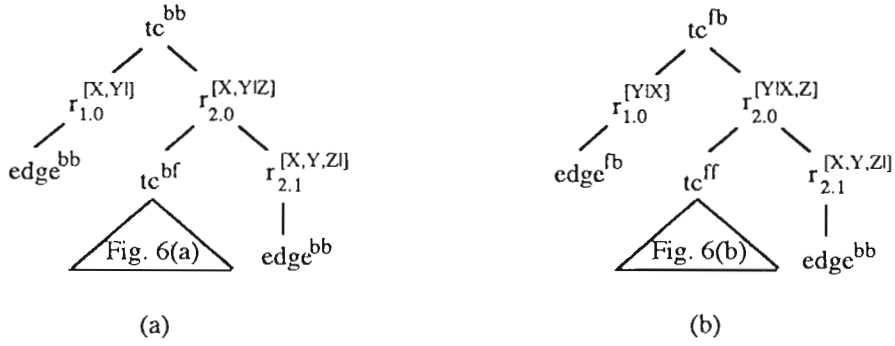

Fig. 8 Rule/goal graphs for adorned queries tc^{bb} and tc^{fb}

the generalization includes recursions that are left-linear after *one* unfolding of the recursive rule.

We will motivate this generalization using the left-recursive tc program in Fig. 2. Recall that the predicate tc is left-linear with respect to the adornments bf and ff and thus, according to Theorem 3.1, tc^{bf} and tc^{ff} are ET-complete. However, tc is *not* left-linear for the adorned queries tc^{bb} and tc^{fb}. Consider the rule/goal graphs for program tc with respect to the adorned queries tc^{bb} and tc^{fb} in Fig. 8 (a) and (b), respectively. For the adornment bb, after unfolding the recursive rule once, the recursive subgoal has the adornment bf. We observe that $bf \leq bb$ and that tc is left-linear with respect to the adornment bf. Similarly, for the adornment fb, after unfolding the recursive rule once, the recursive subgoal has the adornment ff. We observe that $ff \leq fb$ and that tc is left-linear with respect to the adornment ff. Thus, the adorned queries tc^{bb} and tc^{fb} are a selection of the answers for the unfolded adorned queries tc^{bf} and tc^{ff}, respectively, which are known to be ET-complete (by Theorem 3.1). We formalize this concept by the following definition of pseudo-left-linear recursions and prove that pseudo-left-linear recursions are ET-complete.

Definition 4.1

Let $\mathcal{P}$ be a linear recursive (definite, function-free) logic program with a single recursive predicate p and a single recursive rule for p with the recursive subgoal appearing left-most in the rule body. Given an adorned query p^{α}, let the adornment on the recursive subgoal be β. p is **pseudo-left-linear** with respect to the adornment α if (a) $\beta \leq \alpha$, (b) p is left-linear with respect to the adornment β, and (c) for every bound position i in β, the corresponding arguments in position i of both the head and the recursive subgoal share the same variable.

Theorem 4.1

Let $\mathcal{P}$ be a linear recursive (definite, function-free) logic program with a single recursive predicate p of arity n and a single recursive rule for p. Let p be pseudo-left-linear with respect to the query adornment α. Then p^{α} is ET-

complete with respect to $\mathcal{P}_{et([p/n])}$.

Proof

Let p have m rules r_1, ..., r_m, of which only one is recusive:

r_1: p(X) :- $q_1(Y_1)$.

.....

r_m: p(X) :- p(Z), $q_m(Y_m)$.

Let β be the adornment for the recursive p-subgoal with respect to the query adornment α after unfolding the recursive rule r_m once. By definition of pseudo-left-linear, p is left-linear with respect to the adornment β. According to Theorem 3.1, p^β is ET-complete with respect to $\mathcal{P}_{et([p/n])}$. Since $\beta \leq \alpha$, i.e., α has more bindings than β, and the bound arguments in the recursive subgoal have the same bindings as that in the original goal, the answers to p^α are a selection of the answers to p^β. So if the answers to the adorned query p^α were computed as a selection of the answers to p^β, the proof would be trivial. However, this is not the case. We must justify that the interaction of the two subgoals using the same extension table for p does not invalidate the ET-completeness of p^β.

The initial adorned query p^α represents a new call for p, which is stored in the call subtable. The et_retrieve built-in retrieves known answers for p; there are none. The code_p rules are then executed; the first m-1 rules are nonrecursive and unique answers are saved in the extension table. The recursive call to p^β represents a new call, since α can not subsume β because β is more general than α, and is saved in the call subtable. The et_retrieve built-in retrieves known answers for p; the extension table contains the answers computed thus far for p^α, which are also answers for p^β. After all the known answers are returned, the code_p rules are executed, computing all the answers for p^β. Since β is more general than α, the call to p^β will recompute the answers already computed for p^α. These answers will not be returned due to the failure of et_insert_unique. However, these (duplicate) answers were already returned through the et_retrieve built-in.

The correct answers for $\mathcal{P} \cup \{Q\}$ consist of two sets: those generated with the nonrecursive rules and those generated with the recursive rule. All correct answers of $\mathcal{P} \cup \{Q\}$ calculated with the nonrecursive rules are ET-computed answers. This is obviously true since the ET transformation uses the bodies of the nonrecursive rules as given in the original program and guarantees that the nonrecursive rules are evaluated before the recursive rule. Any correct answer of $\mathcal{P} \cup \{Q\}$ calculated by the recursive rule must also be an ET-computed answer since the recursive call to p^β is ET-complete with respect to $\mathcal{P}_{et([p/n])}$ by Theorem 3.1, and these answers for p^β contain all possible answers for p^α. All the answers for p^β are joined with the nonrecursive subgoals q_m. Thus, any correct answer for $\mathcal{P} \cup \{Q\}$ is guaranteed to be an ET-computed answer. □

Example 2

By adding an extension table to predicate tc, given in Fig. 2, the left-recursive transitive closure can be completely evaluated for any query adornment with Prolog's top-down evaluation strategy.. This result follows from Theorem 3.1 and Theorem 4.1. Consider the four possible adornments for tc: *bb*, *bf*, *fb* and *ff*. According to Theorem 3.1, the adorned queries tc^{bf} and tc^{ff} are ET-complete. According to Theorem 4.1, tc^{bb} and tc^{fb} are ET-complete. Therefore, Prolog with memoing through the use of extension tables can completely evaluate any query over the left-recursive transitive closure specification given in Fig. 2.

Example 3

With an extension table on the recursive predicate tc/2, the right-recursive transitive closure given in Fig. 9 can be completely evaluated with Prolog's top-down evaluation strategy. First, reorder the subgoals in the recursive rule so that the recursive subgoal appears left-most. The resulting program is left-linear with respect to the query adornments fb and ff. Thus, by Theorem 3.1, the adorned queries tc^{fb} and tc^{ff} are ET-complete. The resulting program is pseudo-left-linear with respect to both query adornments bf and bb. Thus, by Theorem 4.1, tc^{bf} and tc^{bb} are ET-complete. Therefore, Prolog with memoing through the use of extension tables can completely evaluate any query over the right-recursive transitive closure specification given in Fig. 9.

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

Fig. 9 Program tc with right-recursion

Although we have just proved that pseudo-left-linear recursions are completely evaluated with memoing through the ET source transformation in Prolog, the efficiency of the evaluation may be a concern. The results found for the query p^{α} before the recursive call are recomputed in the more general call to p^{β}. An obvious optimization is to first evaluate p^{β}, and then select the answers for p^{α} from the answers to p^{β}.

§5 Multiple Linear Recursions

We now consider linear recursive programs that are defined by multiple recursive rules having a single recursive predicate. We prove that any linear recursive program can be translated into a pseudo-left-linear recursive program with a single recursive rule. The resulting program is pseudo-left-linear with respect to *any* adorned query on the recursive predicate.

Consider the example of Fig. 10, which represents a transitive closure defined on two relations p and q.[6] The definition of the predicate pqs/2 includes two recursive rules. Given the query pqs(X, Y), the set of answers is {(a, b), (a, c),

```
p(a, b).    q(b, c).
p(c, d).    q(d, e).
pqs(X, Y) :- p(X, Y).
pqs(X, Y) :- q(X, Y).
pqs(X, Y) :- pqs(X, Z), p(Z, Y).
pqs(X, Y) :- pqs(X, Z), q(Z, Y).
```

Fig. 10 Transitive closure on two relations

(a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e), (d, e)}. However, Prolog does not return the complete set of answers even if we memo the answers to the predicate pqs/2, due to the multiple recursive rules. The answers to the second recursive rule may be used by the first recursive rule to generate more answers and vice versa. Therefore, the evaluation is incomplete.

This program can be translated into a program with a single recursive rule by left-factoring the recursive subgoal pqs/2. The translated program is given in Fig. 11, where s/2 is a new predicate. It is left-linear with respect to the query pqs(X, Y). By Theorem 4.1, with memoing on the recursive predicate pqs/2, query pqs(X, Y) can be completely evaluated with Prolog's top-down evaluation strategy.

```
pqs(X, Y) :- p(X, Y).
pqs(X, Y) :- q(X, Y).
pqs(X, Y) :- pqs(X, Z), s(Z, Y).
s(Z, Y) :- p(Z, Y).
s(Z, Y) :- q(Z, Y).
```

Fig. 11 Result of left-factoring pqs/2 in Fig. 10

Generally, it may not be possible to apply a left-factoring operation to a program directly. For example, Figure 12 defines the transitive and symmetric closure of a given base relation. In this program, the recursive subgoals in the recursive rules are not exactly the same, so we can not left-factor the subgoals for p/2 directly. However, Algorithm 5.1 provides a left-factoring approach that handles such a case, converting any program with multiple linear recursive rules into one that has a single linear recursive rule. The resulting program is logically equivalent to the original program and is pseudo-left-linear with respect to any query adornment.

The translated program is logically equivalent to the original program. The translation is derived by introducing Z, a tuple of new variables, as the

```
base(1, 2).
base(2, 3).
p(X, Y) :- base(X, Y).
p(X, Y) :- p(X, Z), base(Z, Y).
p(X, Y) :- p(Z, Y), base(X, Z).
p(X, Y) :- p(Y, X).
```

Fig. 12 Transitive and symmetric closure

Algorithm 5.1 Left-factor recursive subgoals.
Input: A linear recursive program with a single recursive predicate but multiple linear recursive rules.
Output: A linear recursive program with a single recursive rule. The recursive predicate is pseudo-left-linear with respect to any query adornment on the resulting program.

(1) Reorder the subgoals of each recursive rule so that the recursive subgoal appears left-most in each rule.
(2) Assume the program has m recursive rules for predicate p with the following forms after step (1):

r_1: $p(X) :- p(Y_{11}), q_{12}(Y_{12}), ..., q_{1k_1}(Y_{1k_1})$.
.....
r_m: $p(X) :- p(Y_{m1}), q_{m2}(Y_{m2}), ..., q_{mk_m}(Y_{mk_m})$.

where, X and Y_{ij} ($1 \leq i \leq m$, $j = 1, 2, ...$) are tuples of arguments. Introduce a new rule r_0' that is linear recursive having as its left-most literal p(Z), where Z has the same number of variables as X but are distinct, and having as its right-most literal s(XZ), where XZ represents the concatenation of the variables given by X and Z. Replace rules $r_1, ..., r_m$ with rules $r_0', r_1', ..., r_m'$:

r_0': $p(X) :- p(Z), s(XZ)$.
r_1': $s(XY_{11}) :- q_{12}(Y_{12}), ..., q_{1k_1}(Y_{1k_1})$.
.....
r_m': $s(XY_{m1}) :- q_{m2}(Y_{m2}), ..., q_{mk_m}(Y_{mk_m})$.

$XY_{11}, ...,$ and YY_{m1} are the concatenations of X with $Y_{11}, ...,$ and X with Y_{m1}, respectively. Note that exit rules, if any, are unchanged.

argument to the recursive subgoals and left-factoring. The new predicate s is introduced to represent the nonrecursive component of the bodies of the m recursive rules. This intermediate step in the transformation is shown below, where, in each rule for s, Z is explicitly unified with Y_{i1} ($1 \leq i \leq m$):

r_1'': $s(XZ) :- Z = Y_{11}, q_{12}(Y_{12}), ..., q_{1k_1}(Y_{1k_1})$.
.....
r_m'': $s(XZ) :- Z = Y_{m1}, q_{m2}(Y_{m2}), ..., q_{mk_m}(Y_{mk_m})$.

By substituting Y_{i1} ($1 \leq i \leq m$) into Z, Z and Y_{i1} are implicitly unified and we get the resulting set of rules for $r_1', ...,$ and r_m' as in the algorithm.

Since Z is a tuple of variables distinct from X, any query on p results in a recursive subgoal representing the most general call on p. Thus, p is left-linear with respect to the most general adornment on the recursive subgoal. Therefore, p is pseudo-left-linear with respect to any query adornment. According to

Theorem 4.1, with memoing on the recursive predicate, the resulting program with any query adornment can be completely evaluated with Prolog's top-down evaluation strategy. One disadvantage with this algorithm is that the resulting program computes the whole relation for the recursive predicate. Although this generalization may be costly with respect to the original query, reuse of memoed results in subsequent computation may justify the additional cost. Note that the generalization may actually be an optimization if the most general call would have been made at some point during the computation anyway. By analyzing the binding pattern propagation, this situation can be detected at compile time.

Theorem 5.1
Let $\mathcal{P}$ be a linear recursive (definite, function-free) logic program with a single recursive predicate p of arity n and multiple linear recursive rules for p. Let $\mathcal{F}$ be the output of the source-to-source program transformation given by Algorithm 5.1 using $\mathcal{P}$ as its input program. Let γ represent a valid query adornment for p. Then p^{γ} is ET-complete with respect to the program $\mathcal{F}_{et([p/n])}$.

Proof
Due to the generality of the input for Algorithm 5.1, any linear recursive program with a single recursive predicate can be translated into a pseudo-left-linear recursive program with a single recursive rule. The source-to-source program transformation of Algorithm 2.1 adds memoing to the recursive predicate p. By Theorem 4.1, any query on the resulting program can be completely evaluated with Prolog's top-down evaluation strategy. □

Example 4
Consider the transitive and symmetric closure program given in Fig. 12, and the result of applying Algorithm 5.1 to that program is given in Fig 13. This is a pseudo-left-linear recursive program with respect to any query adornment and therefore, by adding memoing on p/2, it can be completely evaluated with Prolog's top-down evaluation strategy.

```
base(1, 2).
base(2, 3).
p(X, Y) :- base(X, Y).
p(X, Y) :- p(U, V), s(X, Y, U, V).
s(X, Y, X, Z) :- base(Z, Y).
s(X, Y, Z, Y) :- base(X, Z).
s(X, Y, Y, X).
```

Fig. 13 Result of Algorithm 5.1 on program of Fig. 12

§6 Summary and Future Work

This paper identified a simplified class of Prolog programs for which naive memoing in the presence of recursion was complete, illustrating the limitations imposed even for this restricted class of programs. This paper did not

include an empirical study of the efficiency of such a naive memoing approach over using one of the general, complete strategies. We expect the "rule of thumb" of trading generality for efficiency to hold in this respect as well. At the very least, this theoretical study identifies a class of programs for which the management of the memo tables may be simplified and represents the foundation of an optimization strategy for top-down memoing techniques, such as the ET* evaluation strategy.[6] For example, ET* utilizes the ET source transformation in combination with an iterative construct to guarantee completeness by computing the least fixed point. Thus, for the simplified recursions identified in this paper, only the memoing provided by the ET source transformation is required for completeness, resulting in an obvious performance improvement over ET*.

We plan to continue this theoretical investigation by extending the study of completeness of naive memoing to include: linear recursions with multiple recursive predicates, such as nested linear recursions; non-linear recursions, such as multi-linear recursions; and function symbols. Extending this theoretical study to include recursions with more than one recursive predicate is important since typical memoing examples in logic programming consist of non-linear recursions, such as the naive Fibonacci program and the double-recursive version of transitive closure. Although some non-linear recursions can be translated into linear ones,[26] there is no guaranteed method to convert arbitrary non-linear programs to equivalent linear ones.[22] Identifying the class of logic programs *with* functions for which naive memoing is complete is another important extension of this work. Functions often imply infinite relations. By transforming function symbols into uninterpreted relations, researchers[10,12,17] found that if the infinite relations defined by the functions have certain properties, such as finiteness constraints and monotonicity constraints holding over the attributes of the relation, the evaluation may be complete and terminating.

After completing the above theoretical extensions, our focus will shift to implementation level details. Recall that we are interested in the interaction of naive memoing and recursion in *Prolog*, rather than logic programming in general. Thus, an obvious next step is to look at naive memoing within the recognized WAM (Warren's Abstract Machine) implementation of Prolog. Although the XWAM[24] integrated memoing evaluation into the WAM, the XSB system[18] supercedes the XWAM and extends the standard functionality of Prolog by providing an implementation of SLG resolution.[5] A careful look at the SLG-WAM will be warranted, although XSB is clearly more powerful than naive memoing in Prolog.

Acknowledgements

We appreciate discussions with Saumya K. Debray that led to the identification of the generalization of this work to memo-completeness, where the memo strategy and selection strategy of the underlying execution method would not be fixed as in this study. The study of completeness of naive memoing

in Prolog, however, represents a significant first step due to the importance of Prolog as a logic programming language.

References

1) Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic Sets and Other Strange Ways to Implement Logic Programs," *Symposium on Principles of Database Systems*, ACM, pp. 1-15, 1986.
2) Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," *SIGMOD*, ACM, pp. 1-15, 1986.
3) Bird, R. S., "Tabulation Techniques for Recursive Programs," *Computing Surveys*, *12*, *4*, pp. 403-417, 1980.
4) Bry, F., "Query Evaluation in Deductive Databases: Bottom-up and Top-down Reconciled," *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pp. 25-44, December 1989.
5) Chen, W. and Warren, D. S., "Query Evaluation under the Well-Founded Semantics," *Symposium on Principles of Database Systems*, ACM, pp. 168-179, 1993.
6) Dietrich, S. W., "Extension Tables: Memo Relations in Logic Programming," *Symposium on Logic Programming*, IEEE, pp. 264-272, 1987.
7) Dietrich, S. W., "Shortest Path by Approximation in Logic Programming," *ACM Letters on Programming Languages and Systems*, *1*, *2*, pp. 119-137, June 1992.
8) Fagin, B. S. and Despain, A. M., "Goal Caching in Prolog," *Hawaiian International Conference on System Sciences*, IEEE, pp. 277-281, 1986.
9) Fan, C. and Dietrich, S. W., "Extension Table Built-ins for Prolog," *Software Practice and Experience*, pp. 573-597, July 1992.
10) Han, J., "Compilation-Based List Processing in Deductive Databases," *the International Conference on Extending Database Technology*, pp. 104-119, March 1992.
11) Keller, R. M., "Applicative Caching," *ACM Transactions on Programming Languages and Systems*, *8*, *1*, pp. 88-108, January 1986.
12) Kifer, M., "On Safety, Domain Independence and Capturability of Database Queries," *Proc. of the 3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, IPA and ACM, pp. 405-415, June, 1988.
13) Lindholm, T. G. and O'Keefe, R. A., "Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code," *Proc. of the International Conference on Logic Programming*, ACM, pp. 21-39, 1987.
14) Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, New York, NY, 1987.
15) Naughton, J. F., "One-Sided Recursion," *Symposium on Principles of Database Systems*, ACM, pp. 340-348, 1987.
16) Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D., "Efficient Evaluation of Right-, Left-, and Multi-Linear Rules," *SIGMOD*, ACM, pp. 235-242, 1989.
17) Ramakrishnan R., Bancilhon, F., and Silberschatz, A., "Safety of Recursive Horn Clauses with Infinite Relations," *Symposium on Principles of Database Systems*, ACM, pp. 328-339, 1987.
18) Sagonas, K., Swift, T., and Warren, D. S., "XSB as an Efficient Deductive Database Engine," *Proc. of the 1994 ACM SIGMOD International Conference on the Management of Data*, ACM, pp. 442-453, 1994.
19) Tamaki, H. and Sato, T., "Unfold/Fold Transformation of Logic Programs," *Proc. of the International Conference on Logic Programming*, ACM, pp. 127-138, July 1984.
20) Tamaki, H. and Sato, T., "OLD Resolution with Tabulation," *Proc. of the Interna-*

tional Conference on Logic Programming, ACM, pp. 84-98, July 1986.
21) Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Vol. 1.* Computer Science Press, Rockville, Md, 1988.
22) Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Vol. 2: The New Technologies*, Computer Science Press, Rockville, Md, 1989.
23) Vieille, L., "A Database-Complete Proof Procedure Based on SLD-Resolution," *Proc. of the International Conference on Logic Programming*, ACM, pp. 74-103, 1987.
24) Warren, D. S., "The XWAM: A Machine that Integrates Prolog and Database Query Evaluation," *Architecture Workshop of the North American Conference on Logic Programming*, ALP, 1990.
25) Warren, D. S., "Memoing for Logic Programs," *Communications of the ACM, 35, 3*, ACM, pp. 94-111, March 1992.
26) Zhang, W. and Yu, C. T., "A Necessary Condition for a Double Recursive Rule to Be Equivalent to a Linear Recursive Rule," *SIGMOD*, ACM, pp. 345-356, 1987.

Suzanne Wagner Dietrich, Ph.D.: She is an Associate Professor in the Department of Computer Science and Engineering at Arizona State University. Her research emphasis is on the evaluation of declarative logic programs especially in the context of deductive databases, including materialized view maintenance and condition monitoring in active deductive databases. More recently, her research interests include the integration of active, object-oriented and deductive databases as well as the application of this emerging database technology to various disciplines such as software engineering. She received the B. S. degree in computer science in 1983 from the State University of New York at Stony Brook, and as the recipient of an Office of Naval Research Graduate Fellowship, earned her Ph.D. degree in computer science at Stony Brook in 1987.

Changguan Fan, M.S.: He is a Ph.D. candidate in the Department of Computer Science and Engineering at Arizona State University and a software engineer at the Regenisys Corporation in Scottsdale, AZ. His research interests include the evaluation of logic programs, deductive database systems and database management systems. He received his B.S. in Computer Science from the Shanghai Institute of Railway Technology, Shanghai, China in 1982 and his M.S. in the Department of Computer Science and Engineering at Arizona State University in 1989.