Proceedings of the Symposium on Logic Programming

EXTENSION TABLES: MEMO RELATIONS IN LOGIC PROGRAMMING

Suzanne Wagner Dietrich

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract

Memo-ization is a useful optimization technique whose application to functional programming has been well explored [1],[2],[3]. The memo facility employs a dynamic programming approach to computation in which intermediate results are saved and later reused to avoid redundant work. This paper explores the application of this principle to Horn clause programming. The table in which the intermediate results of computation are saved is known as an *extension table* since the set of tuples that satisfy a predicate, its extension, is stored in the table. Because of the nondeterminism in logic programming, *an extension table facility improves the termination and completeness characteristics of depth-first evaluation methods in the presence of recursion*, and thus the consequences of memo-izing in a relational system are much more interesting and far-reaching than in functional environments.

1 Introduction

Logic programming systems use a declarative specification of programs which provide a separation of program logic from the control of the execution of that program. Unfortunately, this separation of logic from control may lead to programs whose execution may be very inefficient. In addition, logically correct programs may not terminate due to the underlying execution method.

One solution is to make the programmer responsible for efficient program execution using impure features of the logic programming language. This solution is undesirable since the declarative reading of the program is obscured by the specification of control. Some researchers have considered an alternative that separates the specification language from the control language [4]. Unfortunately, both of these approaches give the programmer the responsibility for cleverly directing the execution of the program.

Another solution is to make the system responsible for the efficient execution of a logically correct program. One alternative is to use optimizing compilers to improve the performance of programs on the underlying execution method. Program transformation is often used to mechanically transform programs into equivalent ones which are executed more efficiently [5]. Another alternative is to use improved execution methods for the evaluation of a program specification [6], [7]. Instead of altering the program specification, an evaluation method is modified to achieve efficient program execution.

The problem of terminating the execution of logically correct programs which are defined using recursive rules has received a lot of interest recently in the database community. Many strategies for the evaluation of recursive queries in deductive databases have been proposed [8],[9],[10],[11], [12],[13],[14], [15],[16]. These works pertain primarily to *Datalog* programs which are Horn clause programs without function symbols. These methods generally apply to a restricted set of Datalog programs. Our research, therefore, is also applicable to the evaluation of recursive rules in deductive databases. However, its applicability extends beyond the realm of databases to the evaluation of logic programs in general.

This paper explores the use of an efficiency mechanism which employs the dynamic programming principle [17] where the results of subproblems are saved in a table; if the same result is needed later, it is retrieved via a table lookup instead of being recomputed. One such application of this principle used to achieve efficiency for the evaluation of numerical functions is known as memo functions [1]. Our research applies this dynamic programming technique to logic programming. The table in which the results of computation are saved is called an *extension table* since it is the extension of a predicate that is stored in the table. Extension table evaluation is interestingly different from memo evaluation in functional programming because of the nondeterminism in logic programming. *Extension tables favorably affect the termination characteristics of depth-first evaluation methods in the presence of recursive definitions* and thus speaks to one of the glaring problems in efficient logic programming languages such as Prolog.

This paper assumes that the reader is familiar with logic programming and, in particular, the programming language Prolog. Section 2 discusses the use of memo functions in functional languages. Section 3 introduces extension tables as a caching mechanism. In Section 4, we propose the use of extension tables as an evaluation procedure for general logic programs with recursive rules. Section 5 comments on research related to extension tables. Conclusions and open research problems are included in Section 6.

2 Memo Functions

A simple but often effective mechanism for improving the efficiency of the execution of a program is to save intermediate results in a table and to use a table lookup to retrieve a result instead of recomputing it. As noted earlier, this is known as the dynamic programming principle. Michie [1] referred to this mechanism as a *memo* facility. He called a function which has this technique applied to its evaluation, a *memo* function.

The evaluation of a memo function first performs a table lookup via a search strategy for the current request. If an entry is found in the table which has its arguments equal to the calling arguments, then the value in that entry is returned. If an answer is not found in the table, then the computational procedure is used to compute the answer. The computed answer and its arguments are then saved in the table.

Memo-ization has been argued to be a useful optimization technique for functional programs [2]. He uses as an example the following clear and natural definition of the fibonacci function:

```
fib 1 = 1.
fib 2 = 1.
fib n = fib(n-1) + fib(n-2), n>2.
```

This definition results in an inefficient exponential execution since it performs a large amount of redundant work. The memo function for this definition of fibonacci achieves a linear run-time, assuming an efficient search strategy on the look-up table, instead of an exponential one as in the original definition. This is because all calls to fib will first perform a table lookup before using any rules to compute the answer.

Memo functions preserve total correctness. The memo-ized program and the original program are equivalent in the sense that they compute the same answers but memo functions may compute them more (or in some cases less) efficiently. Also, if the original program terminates on a given set of input so does its memo-ized version. The only difference in the memo-ized evaluation is that when the function call successfully returns, the argument-value pair for that call is stored in the look-up table. If that same call is made subsequently, the memo-ized version will use the value in the look-up table whereas the original program will recompute it. In both evaluations, the computation terminates. In general, memo evaluation will not change the termination characteristics of the original program. If the original program diverges on a given set of inputs, then so does its memo-ized version. A nonterminating computation for a function call results from a recursive call to that function with the same arguments. The evaluation of the original program results in an infinite loop due to these recursive calls with the same arguments. The evaluation of the memo-ized version of the function would also result in a nonterminating computation. Since the first function call has not successfully returned, there is no entry in the look-up table that can be used for the recursive call. The memo evaluation would proceed by again trying to find the answer via the computational procedure and so enters an infinite loop. Thus, due to the deterministic nature of functional programs, the memo facility will not affect termination.

3 Caching Mechanism

Extension tables, an application of Warren's work on recall tables [18] to logic programming, have been proposed (but not developed) as a caching mechanism for an integrated Database-Prolog system [19]. The database aspects of Prolog allow many uses of extension tables on simple predicates. For example, an extension table can be used for the storage of intermediate relations.

```
pq(X,Y) :- p(X),q(Y).
```

If a relation *pq* is defined to be the join of relations *p* and *q*, then each time *pq* is called *p* and *q* will be executed. If *pq* were kept in the extension table, the join would be performed only once and subsequent requests for *pq* would retrieve the joined tuples from the table.

Another more traditional use of caching would be to reduce the number of disk accesses. There may be very large relations in the system that are only sparsely accessed and always by a key, for example, a large file directory. A cache on such a predicate would store tuples retrieved from disk in the extension table and so future references to these tuples would find them in the extension table and so would not have to access the disk. An *open file table*, which is a cache for the file directory, can be easily implemented with extension tables.

Extension tables can also provide a flexible caching mechanism. The programmer can indicate to a preprocessor that an extension table is to be kept on a particular predicate by annotating the predicate in the source program. The caching behavior of the system can be reconfigured simply by changing the annotations and running the preprocessor on the program.

An extension table can be viewed as a straightforward implementation of the dynamic programming principle to logic programming. The goals and answers must be saved so that results can be retrieved for subsequent requests of a goal. In the procedural meaning of a Horn clause program, the evaluation of a goal is a procedure call and a derived answer is a return. Recall that a memo function saved its calling arguments and value at the time of return from the function call. Since memo functions are deterministic, there is exactly one return for each call in the look-up table. However, in Horn clause programs which are nondeterministic, a single call may have multiple returns, or it may have no returns. Thus the information management of the look-up table is more complicated.

The extension table facility saves a call at the time it is first made. Since extension tables memo-ize the relations defined by a call, the table lookup can be employed if the current request for a subgoal is subsumed by a previous request. For example, if a call is made for *p(X,b)* and there has been a previous call to *p(X,Y)*, then a table lookup can be used because *p(X,b)* is a selection of the relation for *p(X,Y)*. So if the current call can use a table lookup, then the answers stored in the extension table are returned one at a time. Any returns added to the table during the processing of the call are also used as returns. Otherwise, the current call is the first such one and so it is saved and the predicate's definition is used to determine an answer. The answer that is computed is then conditionally added to the extension table. Since the table stores the extension of a predicate, a computed answer which is an instance of an answer already stored in the extension table is not added and that current computation is made to fail.

This extension table facility can be implemented easily in Prolog by simply modifying the clauses that define a predicate. We will call this approach the *ET* algorithm. The code to implement an extension tabe facility is shown in Figure 1. To use an extension table for a predicate *pred*, the predicate *code_pred* must be defined using the original clauses that defined *pred*. The calls for *pred* are saved by asserting them into the predicate *call_pred*. The answers for *pred* are saved in the extension table using the predicate *et_pred*, i.e. the extension of *pred* is given by *et_pred*.

Note that if *pred* is recursive, then the recursive calls in the body of the clauses for *code_pred* are to *pred*, which checks the table, and not directly to *code_pred*.

The extension table facility can be requested for any predicate on which an extension table is desired. This allows for not introducing the overhead of saving the calls and answers on predicates which are already extensionally defined or on predicates which are very fast to compute. The translation of a predicate to its associated extension table code can be performed at run-time given the original definition for a predicate. The translation can also be performed at preprocessing-time given a declaration indicating that an extension table facility is to be used on that predicate.

```
/* if called before use et */
pred(X1,...,Xn) :-
    call_pred(Y1,...,Yn),
    subsumes(pred(Y1,...,Yn),
             pred(X1,...,Xn)),!,
    et_pred(X1,...,Xn).
/* else save call and compute answer */
pred(X1,...,Xn) :-
    assert(call_pred(X1,...,Xn)),
    code_pred(X1,...,Xn),
    not(et_pred(Y1,...,Yn),
        subsumes(pred(Y1,...,Yn),
                 pred(X1,...,Xn))),
    assert(et_pred(X1,...,Xn)).
```

Figure 1: *ET* Algorithm

The *ET* algorithm has been implemented in Stony Brook Prolog [20] which is a Prolog compiler based on the Warren Prolog engine [21] and in Cprolog. For Stony Brook Prolog, the goal *et(pred/arity)* automatically generates an extension table version of the code for *pred* which replaces the original definition of the predicate. In Cprolog, a simple Prolog program is used to generate the straightforward program transformation needed to implement the extension table algorithm. The output of this program is the set of clauses which gives the extension table code for the predicate. This code must be consulted before querying the predicate. (Note that the extension table implementation requires that the predicate *assert* has been modified so that a new clause added to the chain after the current last one has been retrieved will be found on backtracking.) Extension tables can be a useful caching mechanism. Some queries, such as fib(20,F) on the naive fibonacci program, which run out of space in Prolog can be answered quickly by the extension table evaluation.

4 Evaluation Strategy

Extension tables can be used as an evaluation procedure for general Prolog programs. Assume, for simplicity of presentation, that the logic programs which will be considered in this section are Datalog programs. The use of extension table evaluation of logic programs with function symbols will be addressed later.

4.1 The *ET* Algorithm

Recall the classic Farmer-Wolf-Goat-Cabbage (alias Missionaries and Cannibals) puzzle. The farmer, wolf, goat and cabbage are all on the north bank of a river and the problem is to transfer them to the south bank. The farmer has a boat which he can row taking at most one

```
/*     Initial state      */
  state(n,n,n,n).
/*    Farmer takes Wolf   */
  state(X,X,U,V):-
       safe(X,X,U,V),
       opp(X,X1),
       state(X1,X1,U,V).
/*   Farmer takes Goat    */
  state(X,Y,X,V):-
       safe(X,Y,X,V),
       opp(X,X1),
       state(X1,Y,X1,V).
/*  Farmer takes cabbage  */
  state(X,Y,U,X):-
       safe(X,Y,U,X),
       opp(X,X1),
       state(X1,Y,U,X1).
/* Farmer goes by himself */
  state(X,Y,U,V):-
       safe(X,Y,U,V),
       opp(X,X1),
       state(X1,Y,U,V).

/* Opposite shores (n/s)  */
  opp(n,s).    opp(s,n).

/*   Farmer is with Goat  */
  safe(X,Y,X,V).
/*  Farmer not with Goat  */
  safe(X,X,X1,X) :-  opp(X,X1).
```

Figure 2: Farmer-Wolf-Goat-Cabbage Puzzle

passenger at a time. The goat cannot be left with the wolf unless the farmer is present. The cabbage, which counts as a passenger, cannot be left with the goat unless the farmer is present. Consider the Prolog clauses in Figure 2 which declaratively defines a solution to the puzzle (origin of this solution is unknown).

The evaluation of the goal *state(s,s,s,s)* under Prolog's evaluation strategy will enter an infinite loop. A trace of the execution shows that, due to the fixed order of the clauses, the farmer takes the goat back and forth across the river. But consider the evaluation of this program if we put an extension table facility on the recursive predicate *state*. Using the extension table as an evaluation procedure, the computation terminates and concludes that the four can cross the river satisfying the constraints.

The Prolog code to implement the extension table for *state* is given in Figure 3. As discussed above, this requires (1) the definition of the predicate *code_state* is that of the original clauses for *state* and (2) the generation of the following new clauses for *state*.

The divergence of the query *state(s,s,s,s)* on the original code for *state* resulted from the repeated use of the recursive rules. This resulted in infinitely many uses of the second clause for *state*. The extension table facility eliminates the non-termination because the recursive calls use a table lookup. If it tried to enter a state that it was in before, there was no solution found in the extension table, so the evaluation failed and was forced to consider another alternative.

```
/*     Initial state      */
  code_state(n,n,n,n).
/*    Farmer takes Wolf   */
  code_state(X,X,U,V):-
       safe(X,X,U,V),
       opp(X,X1),
       state(X1,X1,U,V).
/*   Farmer takes Goat    */
  code_state(X,Y,X,V):-
       safe(X,Y,X,V),
       opp(X,X1),
       state(X1,Y,X1,V).
/*  Farmer takes cabbage  */
  code_state(X,Y,U,X):-
       safe(X,Y,U,X),
       opp(X,X1),
       state(X1,Y,U,X1).
/* Farmer goes by himself */
  code_state(X,Y,U,V):-
       safe(X,Y,U,V),
       opp(X,X1),
       state(X1,Y,U,V).

  state(X1,X2,X3,X4) :-
       call_state(Y1,Y2,Y3,Y4),
       subsumes(state(Y1,Y2,Y3,Y4),
              state(X1,X2,X3,X4)),!,
       et_state(X1,X2,X3,X4).
  state(X1,X2,X3,X4) :-
       assert(call_state(X1,X2,X3,X4)),
       code_state(X1,X2,X3,X4),
       not(et_state(Y1,Y2,Y3,Y4),
           subsumes(state(Y1,Y2,Y3,Y4),
                    state(X1,X2,X3,X4))),
       assert(et_state(X1,X2,X3,X4)).
```

Figure 3: *ET* algorithm on *state*

In the previous examples, the extension table evaluation resulted in a terminating computation which found all the answers for a query. It would be nice if extension table evaluation were convergent and complete for all Datalog programs. It is unreasonable to expect that this evaluation strategy is complete for general Prolog programs with structure symbols, since it is easy to write a program which loops by building larger and larger data structures. Unfortunately, the *ET* algorithm is not complete even for Datalog. There are logic programs without recursive data structures for which this evaluation strategy fails to find all answers.

Consider the following program which computes the transitive closure of the relations *p* and *q*.

```
        p(a,b). p(c,d). q(b,c). q(d,e).
R1:     pqs(X,Y) :- p(X,Y).
R2:     pqs(X,Y) :- q(X,Y).
R3:     pqs(X,Y) :- pqs(X,Z),p(Z,Y).
R4:     pqs(X,Y) :- pqs(X,Z),q(Z,Y).
```

The set of answers for the query *pqs(X, Y)* is (a,b), (a,c), (a,d), (a,e), (b,c), (b,d), (b,e), (c,d), (c,e), (d,e). Consider the execution of this program with an extension table facility on the predicate *pqs*. The answers (a,b), (c,d), (b,c) and (d,e) are derived via the rules R1 and R2. Using these answers that have been stored in the extension table for *pqs*, R3 computes the answer (b,d) which is saved in the extension table. (The answer (b,d) is available for use by R3 but this tuple does not lead to another answer.) Similarly, R4 computes the answers (a,c), (c,e) and (b,e). The evaluation has terminated but the answers (a,d) and (a,e) were not derived. The answer (a,d) was not derived because R3 needed the tuple (a,c) from R4's computation which occurred after R3 had failed. Similarly, R4 could have derived (a,e) if (a,d) had been derived.

The *ET* algorithm will produce correct answers and terminate evaluation for some Prolog programs. Simple syntactic checks which will guarantee that the *ET* algorithm is complete for a given Datalog program are being investigated. If the clauses of a program are reordered so that the recursive rules occur last, then more programs can be evaluated with this strategy. However, as shown above, there are programs for which the reordering of clauses does not yield a version of the program which can be evaluated with the simple extension table facility. Thus, there is a need for an extension table facility that will work for a broader class of programs.

4.2 The ET_{interp} Algorithm

The *ET* algorithm fails to be complete on programs that require a breadth-first component. Such programs have the property that, during evaluation via the *ET* algorithm, new answers for an entry are added after a call that used that entry has been backtracked over. A straightforward solution to this problem is to re-activate the computation that could use these new answers to produce additional results. That is, any time an answer is derived for a subgoal, every clause which contains that subgoal could be re-activated by the return of the new answer for that subgoal. However, to accomplish this, the state of the computation, which is kept in the run-time stack of environments, must be saved over backtracking. Thus, a complete general extension table facility must save, in addition to the subgoals and their extensions, a representation of the run-time stack.

The explicit saving of a representation of the run-time stack requires that the program clauses be represented as data as well. Thus, the implementation of this general extension table facility is that of an interpreter written in Prolog. This interpretive approach is called ET_{interp}. The interpreter is essentially a top-down left-to-right depth-first evaluator with a breadth-first component. The interpreter is described in full detail in [22].

This extension table implementation is related to Earley deduction which is a proof procedure for definite clauses [23]. Earley deduction is a demand-driven strategy which uses dynamic programming to yield an execution method with both top-down and bottom-up evaluation characteristics. It is a generalization of Earley's [24] context-free parsing algorithm applied to definite clause grammars. A comparison of Earley deduction and extension tables is given in [22].

4.3 The ET^* Algorithm

The implementation of a general extension table facility as in ET_{interp} requires an extra level of interpretation. Due to this fact, and also because it explicitly maintains the continuation goal list, this implementation can be quite inefficient. However, the conventional iterative algorithm for computing a least fixed point can be used to take advantage of the efficient depth-first search and the compilation techniques that have been developed for Prolog implementations. The conventional approach to computing a least fixed point is to initialize the relations to be empty; then perform an iteration using these relations to determine what new tuples to add to them; and repeat this process until no new tuples are added. The extension table implementation described in this section uses this iterative approach, which allows it to take advantage of Prolog's efficient inference mechanism. In this implementation, Prolog regenerates the run-time stack at each iteration instead of saving it explicitly as the continuation goal list which it must do in the interpreted implementation.

The iterative implementation performs iterations of the *ET* algorithm. Thus, this iterative algorithm is called ET^*. The *ET* algorithm began with completely empty extension tables, but consider how it would execute if the extension tables for some predicates already had some answers in them. In this case, the first call of a predicate on an iteration must first return all answers in the extension table before calling the original code for the predicate to possibly compute additional answers. These answers must be returned to guarantee that an answer is returned to a call at least once. Any subsequent call on an iteration would use all the answers in the extension table: those initially in the table as well as those added by the first call. So the *ET* algorithm computes a monotonic function on predicate extensions. The set of extensions we want is the least fixed point of this function.

The implementation of ET^* requires a slight modification of the *ET* algorithm. Since the stopping condition for the ET^* algorithm depends on the addition of new tuples to the extension table, a flag must be set when a new answer is stored in the extension table. This will be accomplished by the use of the predicate *et_changed*.

In addition, the code must be modified so that the extension table answers are returned to the first call on an iteration. This modification results in a noop when the *ET* algorithm is used just as a caching mechanism since there are no answers stored in the extension table on the first call. The modified version of the *ET* algorithm is given in Figure 4.

```
/* if called before use et */
  pred(X1,...,Xn) :-
       call_pred(Y1,...,Yn),
       subsumes(pred(Y1,...,Yn),
                pred(X1,...,Xn)),!,
       et_pred(X1,...,Xn).
/* else save call and compute answers */
  pred(X1,...,Xn) :-
       assert(call_pred(X1,...,Xn)),
      (et_pred(X1,...,Xn);
       code_pred(X1,...,Xn),
       not(et_pred(Y1,...,Yn),
           subsumes(pred(Y1,...,Yn),
                    pred(X1,...,Xn))),
       et_changed,
       assert(et_pred(X1,...,Xn))).
```

Figure 4: Modified *ET* Algorithm

So to compute the least fixed point, we begin with the extension tables initially empty. Each iteration of the fixed point computation corresponds to calling the query using the modified *ET* algorithm, but the extension tables contain the answers left from the previous iteration. Thus, at the beginning of each iteration the calls are removed from the extension table but the answers remain untouched. The answers to the query are returned to the user tuple at a time as they are computed in Prolog. Since each iteration recomputes the answers from the previous iteration plus perhaps some more, the answers that are returned to the user are saved and duplicates are not returned. This process is iterated until there is a complete iteration in which no new answers are generated. The Prolog code to implement ET^* is given in Figure 5.

```
et_star(Query) :-
  repeat,
      (remove_calls,
       call(Query),
       not(duplicate(Query));
       nochange,!,fail).
```

Figure 5: ET^* Algorithm

ET^* takes advantage of compilation techniques developed for Prolog compilers. It does not have to explicitly save continuations as the ET_{interp} algorithm does. Instead, it can be understood as recomputing the continuations as it needs them. These continuations are represented in the run-stack of environment frames that is maintained by the compiled system. Thus, not only is one level of interpretation avoided, but the efficient stack-based memory allocation scheme of the depth-first search of Prolog is used.

As discussed earlier, a pragma (or declaration) can be used in the program source to direct a preprocessor to generate the desired code. In addition, the run-time transformation to the extension table algorithm can also take advantage of compiled code. If the original definition of the predicate *pred* has been compiled and loaded, then this compiled code can be used to define *code_pred*.

Hands-on experience indicates that ET^* is, in general, more efficient than ET_{interp}. Both the interpretive and iterative implementations of the general extension table facility were coded in Stony Brook Prolog and in Cprolog. In both implementations, ET^* was more efficient than ET_{interp} on the examples tested. These test programs included fibonacci, transitive closure, the farmer-wolf-goat-cabbage problem and the parsing of a left-recursive grammar. As already noted, this speedup results from the ability of the iterative algorithm to use Prolog's depth-first evaluation strategy and compiled code.

Both ET_{interp} and ET^* are not complete for logic programs with function symbols. These general evaluation strategies may fail to find an answer when recursive data structures are involved. This is due to the depth-first search rule used by the interpreter. Consider the evaluation of the query *p(a)* on the following Prolog program:

```
p(X) :- p(f(X)).
p(a).
```

Prolog will enter an infinite loop via successive applications of the first rule for *p* generating the calls: p(a), p(f(a)), p(f(f(a))), ... Since each call is different, the extension table lookup is not used and the recursion is not terminated.

ET^* can easily be modified to be complete for general logic programs by introducing a depth-bound. A depth-bound on a call can be used to fail a deduction path if the depth (in the OR-tree) of the call exceeds the indicated limit. When a path fails due to the depth-bound being reached, the *nochange* predicate is set to fail, in order to trigger another iteration. This failure gives other rules a chance to be used. After each iteration the depth-bound is increased so that the search can continue further. This strategy requires that a counter is maintained for checking the depth bound. While such a strategy is also available for use with the regular depth-first evaluation as used by Prolog to obtain a complete algorithm (Stickel suggested such a strategy in his Prolog Technology Theorem Prover [25]), it is more reasonable to consider it in the context of the extension table evaluation method. Although in both standard Prolog and ET^* each iteration completely recomputes the work of the previous iteration, there is no duplication of work within an iteration of the *ET* algorithm.

The last iteration of the ET^* algorithm does not produce any new answers. Thus, this iteration is completely redundant. There is a check that can be made on an iteration to determine whether another iteration may be necessary. This requires that the calls that have failed on the current iteration have been flagged. If a new answer is computed which any one of the failed calls could have used, then another iteration is required. This check may reduce the number of iterations by one but not necessarily. If the answer that forced another iteration does not produce a new answer on the next iteration, then the same number of iterations is performed.

Another possible optimization can be introduced to the ET algorithm when adding an answer to the extension table. When a new answer is added to the extension table, any answers already there that are instances of the new one can be deleted. Obviously, some work has already been duplicated in deriving the more general answer but removing the instances of this answer will stop any further duplication of effort by rules that use these answers later on. This optimization requires a minor modification to the ET algorithm.

5 Related Research

In the area of logic programming, there has been research by Tamaki and Sato paralleling our work of extension tables. Their work is related to ours in that they propose using tabulation with a demand-driven strategy to evaluate recursive queries. The multistage depth-first strategy proposed in [26] iterates to compute the least fixed point of the program being evaluated. The main difference between the two strategies is that the multistage depth-first strategy restricts the table lookup to answers computed on the previous iteration. This limitation guarantees the completeness of multistage depth-first without having to introduce a depth-bound. However, it implies that multistage depth-first will have to perform more iterations, in general, than the ET^* algorithm. Since an iteration recomputes everything from the previous iteration plus perhaps more, the number of iterations a strategy uses to completely evaluate a query is an important measure of efficiency. The ET algorithm, one-pass of the ET^* algorithm, can completely evaluate certain logic programs. Thus, extension tables has the potential for efficient and complete execution of logic programs through optimization techniques. In general, there are certain rules to follow that will lead to a more efficient evaluation of a Datalog program with extension tables:

- Put recursive rules last
- Convert right recursion to left recursion
- Left-factor recursive rules to get just one

These rules, in general, reduce the number of iterations that the ET^* algorithm needs to completely evaluate a program. An area for future research is to be able to identify or transform logic programs into logically equivalent programs such that the one pass extension table algorithm can be used to completely evaluate that program.

6 Future Research

In this paper we explored the application of the dynamic programming principle to Horn clause programs. Memo-ization, an optimization technique for functional programming, applied the dynamic programming principle of computation to the evaluation of the memo function. In a nondeterministic language such as logic programming there are multiple returns for a single call. This makes the use of memo *relations*, or extension tables, much more complicated than the straightforward use of memo functions.

The execution method using extension tables for caching is a simple modification to Prolog's top-down left-to-right depth-first evaluation strategy. The extension table evaluation may terminate the execution of goals that involve recursive rules in cases in which Prolog's strategy diverges. A simple, efficient implementation of extension tables in Prolog, the ET algorithm, was described. It correctly and efficiently evaluates recursive queries for many Datalog programs. Unfortunately, the ET algorithm is not complete for some logic programs. Therefore, we described the ET^* algorithm, which iterates over the ET algorithm, to find all answers to a query. This algorithm that employs a depth-first search rule can be complete by adding a depth-bound.

There are a variety of directions in which extension tables could be developed.

One area for further research is the question of a more efficient implementation of the primitives used in the ET algorithms. The algorithms require much table lookup and subsumption checking. Hashing techniques can improve the table search time. The compilation of the subsumption check, similar to the compilation of unification done by Prolog compilers, can reduce that cost.

It may also be useful to be able to determine dynamically when all answers for a given call have been developed. Such a call need not be deleted for subsequent iterations in ET^* and thus later iterations require less computation.

For some programs the extension tables may get very large. There are two possibilities for dealing with such situations. One is to explore how these tables might efficently be stored on, and accessed from, disk. Another alternative would be to fix the amount of memory that the extension tables are permitted to use, and then develop a replacement strategy when this space overflows. In the functional framework this is relatively straightforward [1] but in our relational framework, issues of termination again are involved.

Extension tables provide a new execution method that is appropriate for certain kinds of Horn clause programs. Different, but logically equivalent, programs will be evaluated with different efficiency using this evaluation technique. For example, equivalent left-recursive and right-recursive grammars have different orders of complexity when processed with ET^*. So the ET^* evaluation method induces a theory of optimization (as does any evaluation strategy): given a Horn clause program, find an equivalent one that is more efficiently executed by the ET algorithm. Developing the details of this theory is an interesting area for further research.

One particular optimization direction is to recall that for some programs the simple ET algorithm is complete. If this can be determined statically, then the second pass of ET^*, which would simply verify that all answers had already been obtained, need not be performed. Also, this suggests that an important goal of optimization might be to transform an arbitrary program into one for which the simple ET algorithm is complete, if possible.

Acknowledgements

I would like to thank David Scott Warren for our early work together on extension tables and for the many discussions that followed. I gratefully appreciate support that I received in the form of an Office of Naval Research fellowship through the American Society for Engineering Education.

References

[1] D. Michie, ""Memo" functions and machine learning," *Nature*, vol. 218, pp. 19–22, Apr. 1968.

[2] D. A. Turner, "The semantic elegance of applicative languages," in *Symposium on Functional Programming and Computer Architecture*, pp. 85–92, Association for Computing Machinery, 1981.

[3] R. M. Keller and M. R. Sleep, "Applicative caching," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 88–108, Jan. 1986.

[4] L. Naish, "Automating control for logic programs," Tech. Rep. 83/6, Department of Computer Science, University of Melbourne, Sep. 1984.

[5] J. Darlington, "Program transformation," in *Functional Programming and its Applications*, (J. Darlington, P. Henderson, and D. Turner, eds.), pp. 193–215, Cambridge, 1982.

[6] D. R. Brough and A. Walker, "Some practical properties of logic programming interpreters," in *International Conference on Fifth Generation Computer Systems*, pp. 149–156, 1984.

[7] M. Bruynooghe and L. M. Pereira, "Deduction revision by intelligent backtracking," in *Implementations of Prolog*, pp. 194–215, Ellis Horwood Limited, 1984.

[8] R. Reiter, "On structuring a first order database," in *2-nd National Conference Canadian Society for Computational Studies of Intelligence,Toronto*, pp. 90–99, July 1978.

[9] C. Chang, "On evaluation of queries containing derived relations in a relational database," in *Advances in Databases, Vol 1*, (H. Gallaire, J. Minker, and J. M. Nicolas, eds.), pp. 235–260, Plenum Press, 1981.

[10] D. McKay and S. Shapiro, "Using active connection graphs for reasoning with recursive rules," in *IJCAI*, pp. 368–374, 1981.

[11] L. J. Henschen and S. A. Naqvi, "Compiling queries in relational first-order databases," *Journal ACM*, vol. 31, no. 1, pp. 47–85, 1984.

[12] F. Bancilhon, "Naive evaluation of recursively defined relations," Tech. Rep. DB-004-85, MCC, 1985.

[13] M. Kifer and E. L. Lozinskii, "Query optimization in logical databases," Tech. Rep. 85/16, Department of Computer Science, SUNY at Stony Brook, 1985.

[14] J. D. Ullman, "Implementation of logical query languages for databases," *ACM Transactions on Database Systems*, vol. 10, pp. 289–321, Sep. 1985.

[15] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, "Magic sets and other strange ways to implement logic programs," in *Symposium on Principles of Database Systems*, pp. 1–15, ACM SIGACT-SIGMOD, 1986.

[16] L. Vieille, "Recursive axioms in deductive databases: the query-subquery approach," in *First International Conference on Expert Database Systems*, (Charleston), 1986.

[17] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[18] D. S. Warren, "Efficient parsing with general grammars," Tech. Rep. 81-028, Department of Computer Science, SUNY at Stony Brook, July 1981.

[19] E. Sciore and D. S. Warren, "Towards an integrated database prolog system," in *First International Workshop on Expert Database Systems*, pp. 801–815, 1984.

[20] "Stony Brook Prolog documentation," ~warren/cer/documentation on sbcs.csnet.

[21] D. H. D. Warren, "An abstract prolog instruction set," Tech. Rep. 309, SRI International, Oct. 1983.

[22] S. W. Dietrich and D. S. Warren, "Dynamic programming strategies for the evaluation of recursive queries," Tech. Rep. 85-31, Department of Computer Science, SUNY at Stony Brook, 1985.

[23] F. C. N. Pereira and D. H. D. Warren, "Parsing as deduction," in *21st Annual Meeting of the Association for Computational Linguistics*, (Cambridge, MA), pp. 137–144, June 1983.

[24] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, pp. 94–102, Feb. 1970.

[25] M. E. Stickel, "A prolog technology theorem prover," in *International Symposium on Logic Programming*, pp. 212–219, 1984.

[26] H. Tamaki and T. Sato, "Old resolution with tabulation," in *Third International Conference on Logic Programming*, pp. 84–98, 1986.