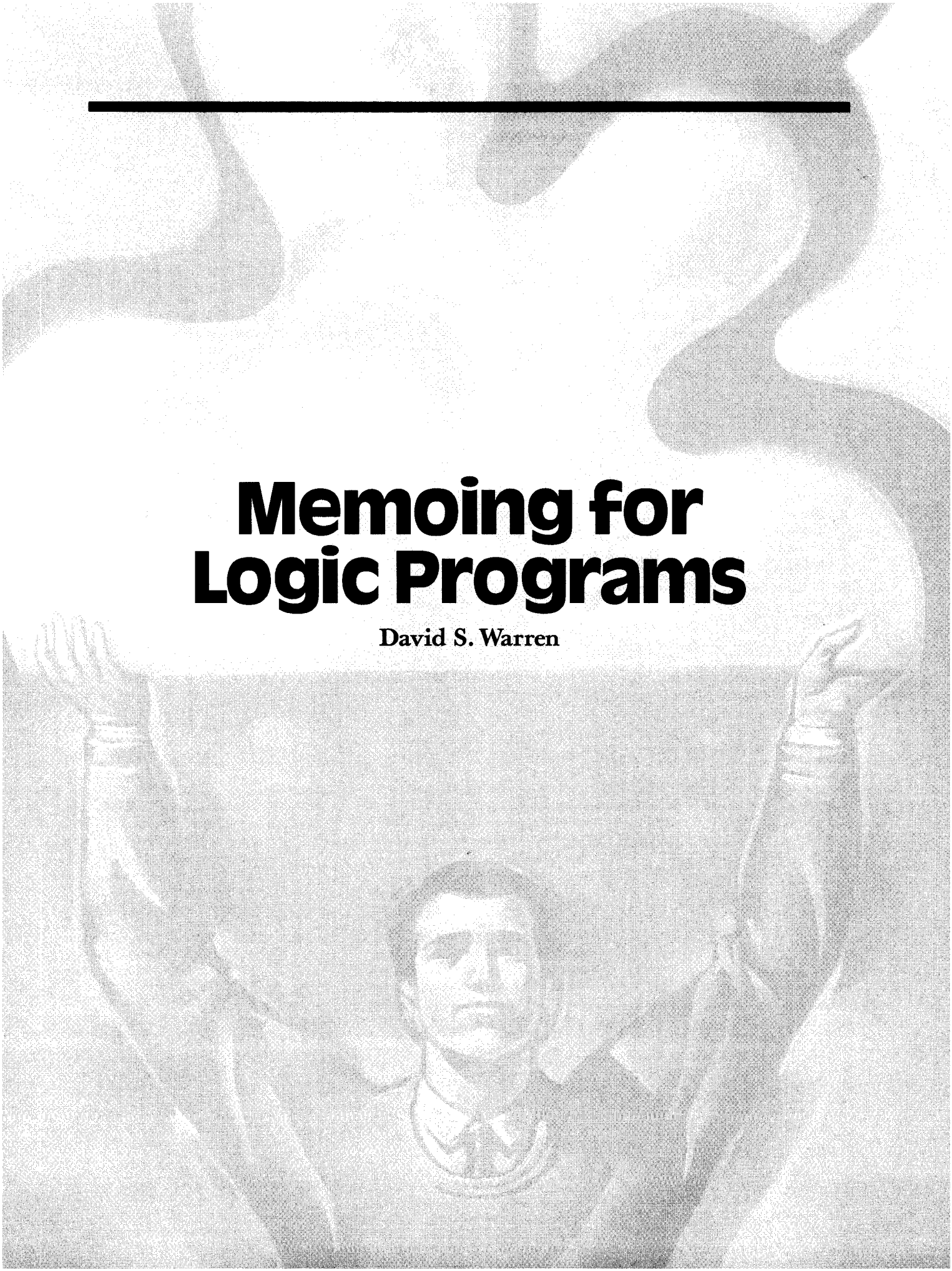
Memoing for Logic Programs

David S. Warren



The power of logic programming (LP) comes from the synergism between the logic (declarativeness) and the programming (procedurality [13, 37]. From the logic point of view, a program is a set of sentences in a given logic. In this declarative view, answers are logical consequences of the program as a set of statements in the logic. And logical consequence is defined in terms of a logical semantics, usually a theory of models. So one can understand a logic program in terms of a logical semantics and entailment in that semantics, that is, in terms of properties of models of the program.

Procedurality comes from the evaluation strategy (or proof method) applied to determine logical consequence. Although there are other languages and approaches, in this article we concentrate on logic programs based on Horn clauses, which include Prolog and all its variants. The proof method used in Prolog is SLD refutation applied to Horn clauses. Prolog's power as a programming language comes from the fact that when SLD resolution is used as the evaluation technique, a Horn clause can be understood as *defining* a procedure (in the ordinary sense of procedural languages, e.g., a Pascal procedure), and evaluation can be understood as *executing* such a procedural program. One complication is that, in Prolog programs, a procedure may have multiple definitions which are interpreted as alternate definitions of the procedure. Thus procedurally, Prolog is a nondeterministic language and execution carries out a depth-first search through the tree of possible alternative executions. A further complication is that parameters are passed by unification.

So what is the synergism that arises from the combination of logic and programming? One way to approach this is to ask what the logic contributes to programmers, and what the programming contributes to logicians. From the programmers' point of view, logic provides a semantics for their programs, a high-level, abstract way to understand what their programs do. It makes their programs more easily understandable and more easily analyzable. And this happens regardless of whether the programmer "knows" logic or not. From the logicians' point of view, the programming (or procedurality) shows them how to deal with computational aspects of their specifications. It gives them the ability to control how the theorem prover proceeds.

Thus the interplay of declarativeness and procedurality is fundamental to logic programming; loss of either means the loss of logic programming. Just as the interplay helps in programming and understanding logic programs, it helps in program processing and program analysis. In this article we discuss three areas of research in LP. The first is a new evaluation strategy, *OLDT*,[1] for logic programs. Since OLDT terminates more often than does Prolog's strategy, it makes more Horn clause programs computationally meaningful, and thus increases the declarativeness of the LP framework. The second topic discussed is *abstract interpretation*, which can be understood as transforming a given program and then directly evaluating the transformed program. We will see that the improved termination properties of OLDT are often necessary for the resulting computation to be meaningful. The third topic is *partial deduction* in which a logic program is evaluated partially, resulting in a new program. Again we will find that the avoidance of some infinite loops by use of OLDT has application here. The common thread, then, is the avoidance of infinite loops and the resulting increased declarativeness of the Horn clause logic programming paradigm.

The concept underlying the OLDT evaluation strategy is termed 'memoing'. The idea of memoing has been around for awhile, and variants have been rediscovered several times in various guises. Here we call it OLDT resolution [34]. (Even though the "T" stands for 'Tabling', I have used the term 'memoing' in the title since it seems to be more widely used for this concept in the general programming community.) There are many ways to understand this idea, and its presentation in a logic programming guise is particularly general and powerful. As briefly indicated, its importance is that, by avoiding some infinite loops, it increases the logic component, the declarativeness, of logic programming. The pleasing and elegant story sketched here about declarativeness and procedurality is (perhaps sadly) not the whole story. There is a gap between the declarative view of a set of Horn clauses and the answers given by Prolog's SLD evaluator. While it is the case that every answer returned to the SLD evaluator (assuming it includes the occur-check) is a logical consequence of the program, the evaluator may go into an infinite loop, even for extremely simple, and logically meaningful, programs. This is because the SLD tree evaluations that must be searched (due to the program's nondeterminism) may be infinite.

For a programming language such as Prolog, infinite loops are a part of the programmer's life, and the programmer is responsible for implementing algorithms that terminate. However, in the area of deductive databases, completeness is critical, and research in that area has been instrumental in the development of these techniques. We will explore the close relationship between the magic templates evaluation strategy and OLDT.

The other two topics discussed here have developed in the attempt to make logic programs more efficient. The first program improvement topic is known as *abstract interpretation*. It is a framework for extracting information from the

[1]The letters of the name come from *Ordered selection strategy with Linear resolution for Definite clauses with Tabling*

text of a program in a finite way. Such information concerns the possible states of the computation at run time, which can be used by an optimizing compiler to generate more efficient code. We will look at the problem in the context of logic programs, in which we can view that problem as transforming the concrete program into an abstract program, and then executing that program. We will see that the termination properties of OLDT evaluation are important for this evaluation.

The final topic is known as *partial deduction,* or *partial evaluation* in more general programming contexts. Partial deduction can be viewed as a program improvement strategy in which part of the execution of the program, that which would be needed by every execution (of a particular form), can be carried out at compile time, thus leaving less computation to be done at run time and thus improving the run time performance. Partial evaluation is normally carried out by developing a metainterpreter to evaluate parts of the object program and delay evaluation of other parts. We will see that by using OLDT to evaluate the metainterpreter, the rules of a partially evaluated program naturally appear in the tables.

OLDT

To review briefly, Prolog is a logic programming language based on Horn clauses. A Horn clause in first-order logic is a universally quantified implication in which a conjunction of atomic formulas implies an atomic formula. The Horn clause is:

$$\forall X \forall Y \forall Z(arc(X,Y) \wedge path(Y,Z) \Rightarrow path(X,Z))$$

could be a part of a description of paths through a graph. As a Prolog *rule,* this implication is written in the following (inverted) form:

```
path(X,Z) :- arc(X,Y), path(Y,Z).
```

Identifiers beginning with upper-case letters are variables and are assumed to be universally quantified. An implication with an empty antecedent is called a *fact* and written, for example, as: arc(a,b).

Figure 1 shows a simple Prolog program that defines a small directed graph (represented by the *arc* relation), and defines the binary property of reachability (the *path* relation).

The evaluation strategy that Prolog uses is the SLD proof strategy for Horn clauses. For example, the query:

```
:- path(a,A).
```

asks whether there is a path from given node *a* to some arbitrary node *A* in the graph. The evaluator is to find all such nodes. As stated in the introduction, SLD resolution can be understood procedurally as executing a nondeterministic program with procedure calls. To establish that there is a path from *a* to some node *A*, the evaluator invokes procedure *path*. There are two definitions (rules) for this procedure; let us (omnisciently) use the second one. It calls *arc*(*a*, *y*) to get the answer *b* back for *Y* and then calls procedure *path* (*b*, *A*). This time let us (omnisciently) use the first definition for *path,* which calls *arc*(*b*, *A*) which succeeds with a value of *d* matching *A*. So the second invocation of *path* returns to the first invocation which returns to the query, and the answer found by this sequence of choices is *d*. This SLD refutation is shown in Figure 2.

Each row in Figure 2 is a list of atoms, the first is the query. We have added a pseudoatom, *ans*[*A*], to the end of the query to collect the answer, which will be the value of the variable *A* at the end of execution. We might think of it as a pseudoprocedure to write out the computed answer. Each row follows from the previous row by matching the first atom with the head of a rule (or a fact), replacing the atom by the body of the matching rule, and applying the match to all the atoms. The rows are divided into columns to emphasize the procedure-calling nature of the computation. Each column represents a stack frame when viewing the computation as execution of a procedural program. Each row represents a state of the run-time stack, growing to the left. The contents of a frame are the calls of the subprocedures that remain to be made in that level. One might think of these frame contents as continuations. Thus the first row has the initial stack frame, containing *path*(*a*, *A*), the initial procedure call to make. The second row has the call from the first stack frame removed and a new stack frame pushed onto the stack, containing *arc*(*a*, *Y*), *path*(*Y*, *A*), the procedures to be called from this level. The third row has called and returned from the procedure for *arc*(*a*, *Y*), so only the *path*(*b*, *A*) call remains to be done from this level. The fourth row represents the stack after this call is made. Notice that the second frame of the fourth row is now empty. This is because there are no subprocedures left to call at this level; when return is made to it, it need only return to the previous level. The fifth row represents the

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

path(X,Y) :- arc(X,Y).
path(X,Z) :- arc(X,Y), path(Y,Z).
```

Figure 1. Simple Prolog program

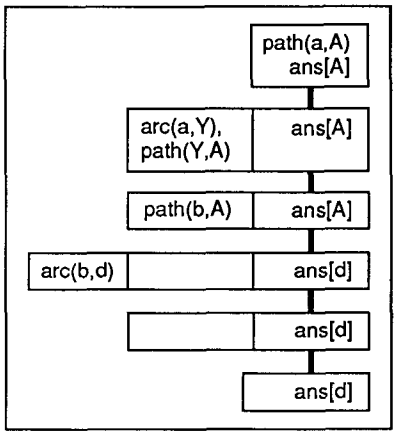


Figure 2. SLD refutation

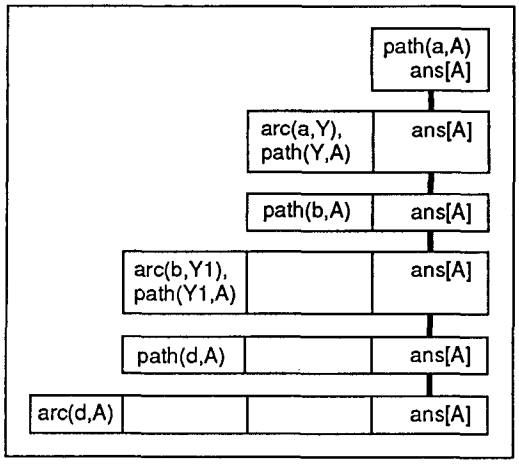


Figure 3. Failed SLD resolution path

```
arc(a, b).
arc(b, a).
arc(b, d).

path(X,Y) :- arc(X,Y).
path(X,Z) :- arc(X,Y), path(Y,Z).
```

Figure 4. Looping Prolog program

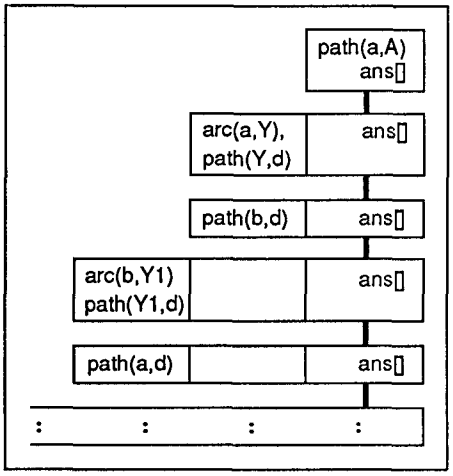


Figure 5. Infinite SLD resolution path

state of the stack when return is made to the second frame, and the sixth row when return is made to the top level. Since only the *ans* pseudoatom remains, a successful SLD refutation has been found.

Now Figure 2 represents only a single execution path through the nondeterministic Prolog program, the one we chose omnisciently. There is another successful path through the program, one which corresponds to the answer *b*. Prolog performs a depth-first search through the tree of SLD refutations (or procedure executions), backtracking when a computation cannot be continued (i.e., it *fails*). So the rows of Figure 2 correspond to nodes in the SLD tree; the entire Figure 2 represents one path through the SLD tree. Another path, one representing a failed computation, is shown in Figure 3. This path shares the first three nodes with the previous figure, but it then is extended by using the second definition for *path*, which is represented by the fourth node (or row, or frame-stack state.) This execution soon fails, since no rule head (or fact) matches *arc(d, A)*.

Formally, SLD resolution is refutation complete for Horn clauses [15]. This means that if a ground answer is a logical consequence of a program, then there is an SLD refutation that generates that answer (or a more general one). There may, however, be some paths that are infinite in length. Since Prolog must search this tree for the answers, its depth-first search may get caught on an infinite path, before it gets to the answer. Consider, for example, the program of Figure 4. The figure shows an infinite sequence of SLD resolution steps for this program and the query:

:– path(a,d).

as shown in Figure 5. The last row shown has the identical list of atoms as the first, so this cycle can be repeated forever. This, of course, corresponds to cycling continually around the loop in the directed graph represented by the *arc* relation of the program. Thought of as a procedural program, this path occurs because the call to the procedure *path(a, d)* results in an identical recursive call to itself.

We see that we cannot depend on our evaluation strategy to give us all the correct answers; the evaluator may get caught in an infinite loop before it finds a correct path. This phenomenon does not surprise Prolog programmers. They must understand this problem very early in their programming careers and learn to program around it. But to logicians (and database theoreticians and users), it is very unpleasant that such a simple and obviously correct definition of transitive closure has such disastrous behavior. The OLDT strategy is able to avoid such loops as this.

The first thought is to blame the infinite looping behavior on the unfair depth-first tree search strategy that Prolog uses. A fair breadth-first search strategy would guarantee that all complete paths (and thus all answers) are found. However, with regard to termination, breadth-first search has exactly the characteristics of depth-first search. A breadth-first search of the tree containing the path indicated in Figure 5 will still run infinitely. Thus even a breadth-first method will not tell us when all the answers have been produced. For generating all answers, both depth-first and breadth-first searches terminate if and only if the search tree is finite.

Of course, no general strategy is able to generate all and only the answers for arbitrary Horn clause programs and terminate for all programs that have only finitely many answers. If some strategy did accomplish this, since Horn clauses are Turing complete, it would have solved the halting problem. However, there are some very simple and important programs (such as our transitive closure example) for which the SLD tree is infinite. Even for propositional rules, rules with no variables, the SLD tree may be infinite. Perhaps more important, or as our example shows, for the so-called Datalog programs, which contain no structure symbols (a.k.a. function symbols or constructors), the SLD tree may be infinite. And for such a simple class, it is possible, and highly desirable, to have an evaluation strategy that gives all answers and terminates. In particular, research into deductive databases (much of which has concentrated on the Datalog language)

brought these points into focus.

Another LP application area in which this problem arises is in grammatical systems such as natural language processing systems. Grammars have a particularly elegant form as Horn clauses (see the Definite Clause Grammar formalism [25]). However, as a consequence of the SLD refutation strategy, processing strings with left-recursive grammars generate identical recursive subprocedure calls and thus infinite loops. Therefore, users of DCG's must avoid perfectly reasonable left-recursive grammars. What is needed is not just another way to search the tree of SLD derivations, but a whole new strategy that generates a different tree. We will see how the OLDT strategy fulfills this need.

Memoing

The OLDT strategy has been discovered in different guises from a number of different starting points and given a number of different names [7, 26, 34, 38, 39]. For this presentation, we will describe it as a variant of an evaluation procedure for nondeterministic procedural programs. This means that we will be depending on our procedural understanding of SLD refutation as an execution procedure for procedural programs. For a deterministic program (or viewed along a single predetermined path through the search tree), we have seen how SLD simply calls and returns from procedures, just as any interpreter of a procedural language would. The modification is simply to add memoing to this procedural interpreter.

Recall the notion of memoing in a deterministic language with procedure calls [20]. We will assume that procedures have no side effects, so the output of a procedure is completely determined by the input. The concept of memoing is simple: during execution, maintain a table of procedure calls and the values they return. If the same call is made later in the computation, do not reexecute the procedure, but use the saved answer to update the current state as though the procedure had been called and had returned that value. It is well known that such a strategy can turn an exponential algorithm into a polynomial one, the usual example being the naive definition of the Fibonacci function.

For a nondeterministic program, the concept of memoing is the same, but the situation is slightly more complex. However, we can conceptually transform a nondeterministic computation into a set of deterministic ones in the following natural way. Intuitively, we think of a machine that is carrying out a nondeterministic procedure as duplicating itself at a point of choice, and as disappearing when it encounters failure. Thus at any time, we have a set of deterministic machines computing away. The set gets larger when any one has to make a nondeterministic choice, and it gets smaller when any one fails. To add memoing, we imagine a single global table containing every procedure call that has been made by any machine, and for each such call, the answers that have been returned for it. Since the situation is nondeterministic, there may be none, one, or many answers for any single call. Now each machine, before it makes a procedure call, looks in the global table to see if the call has already been made. If not, it adds the call to the table and continues computing. During its computation, whenever a machine returns from a procedure, it finds the associated call in the global table, adds the answer it has just computed, and continues computing. (If the answer is already in the table, then this answer is a duplicate, and the machine fails.) When a procedure is about to be called, if the call is found to be already in the table, then for each associated answer in the table, the machine must fork off a new copy of itself to continue the computation with that answer. It is possible that not all the answers are in the table at this time; some may still be in the process of being computed by other machines and will show up later. Thus when a machine encounters a call already in the table, it forks off copies of itself to continue with the answers that are there, and it remains suspended on that table entry. Then whenever a new answer gets added to the table, the suspended machine makes a duplicate of itself to continue computing with that new answer. When a machine finishes its computation successfully, it disappears. The entire computation is complete when (and if) no machines are computing.

Intuitively, it is not difficult to convince oneself that this algorithm is a correct evaluation strategy: 1) Consider any path through the program taken by the memoing evaluator. There will be a corresponding path for the nonmemoing evaluator constructed by expanding all the subpaths for answers that were taken from the table. 2) For any nonmemoed path there will correspond a memoed path. This can be shown by induction on the length of memoed subpaths.

Clearly, this is a very abstract description of memoing in a multiprogramming context. Its implementation on a sequential machine requires maintaining the states of the various machines and scheduling their execution.

OLDT Proper

In our description of SLD resolution we concentrated on the procedural control, and handled parameter passing by matching an atom with the head of a rule and applying the match to every atom in the entire stack. Therefore, in preparation for memoing we modify our SLD representation to handle the passing of parameters into and out of procedures *explicitly*. The reason for this slightly different representation is to make the procedure calls and returns explicit, so they can be used for memoing.

Consider again the program of Figure 1, and the SLD refutation of Figure 2. Figure 6 represents ex-

actly the same computation, but here the calling, returning, and parameter passing is handled slightly differently. In the SLD representation of Figure 2, the atom that generates a call is deleted from the call frame. In this new representation of Figure 6, we retain the atom representing the call in the calling frame, indicating such a call-atom C by *call*[C]. For example, the second row of the figure follows from the first row due to a call of atom *path*(a, A), and that atom becomes *call*[*path*(a, A)] in the first (right-most) frame of the second row. We can think of this *call-atom* as representing the work to be done when the called subprocedure eventually returns, that is, the work of updating the local variables using the values returned.

Now consider how we create a new frame when a call is made. In the old representation the new frame contained the atoms of the body of the instance of the rule that was called. In the new representation, we also add the instance of the head of the called rule at the end of new frame. A head H of a called rule is represented as *ret*[H]. This can be seen in the second (left-most) frame of the second row. The atoms in a frame can be thought of as representing work to be done by the corresponding routine: the regular atoms are calls to be made; the *return-atom* represents the work of returning the values of the result parameters from the subprocedure.

With this implementation we now clearly see the procedure calling and returning and the parameter passing, in and out: Passing parameters into a procedure is done by matching and taking the corresponding instance of the rule (or subprocedure definition). Passing parameters out of a procedure is accomplished by matching the return atom (which has been further instantiated during execution of the subprocedure) with the corresponding call atom and applying that match to the entire frame containing the call atom. Notice that, in this example, the value of the variable A in the answer pseudoatom (shown in the right-most column of Figure 6) is not supplied until the subprocedures have returned back to the top level. The last three rows of the figure show the returning of the value d for A back from the subroutines.

Thus the revised representation, exemplified in Figure 6, is simply a slightly more incremental version of the SLD resolution strategy, one in which variable values are passed explicitly into and out of procedures. We will call this the OLD strategy (no "T" yet).

Now, adding memoing to this representation is relatively straightforward. Recall that this refutation we have been looking at is *one* path through a tree of OLD resolution steps; each row in our tables is one node in that tree. At each step of adding a new row to the table, there could be a branch in the tree; another child of that node could be obtained by using another rule that matches the call atom. In the example of Figure 6, we would have a branch immediately under the first row (i.e., the root of the OLD tree). The second row of the figure came from the second rule of the program; we could obtain a sibling to the second row by using the first rule of the program, since its head also matches the call atom.

Now consider constructing the entire OLD tree. Each leaf of the tree corresponds to the state of one of our deterministic computing machines. A branch point in the tree corresponds to a nondeterministic choice and the replication of a machine into several copies of itself, one for each child. It is in this context now that we add memoing.

The point at which the initial atom in the top (left-most) frame of a stack is expanded by a rule and

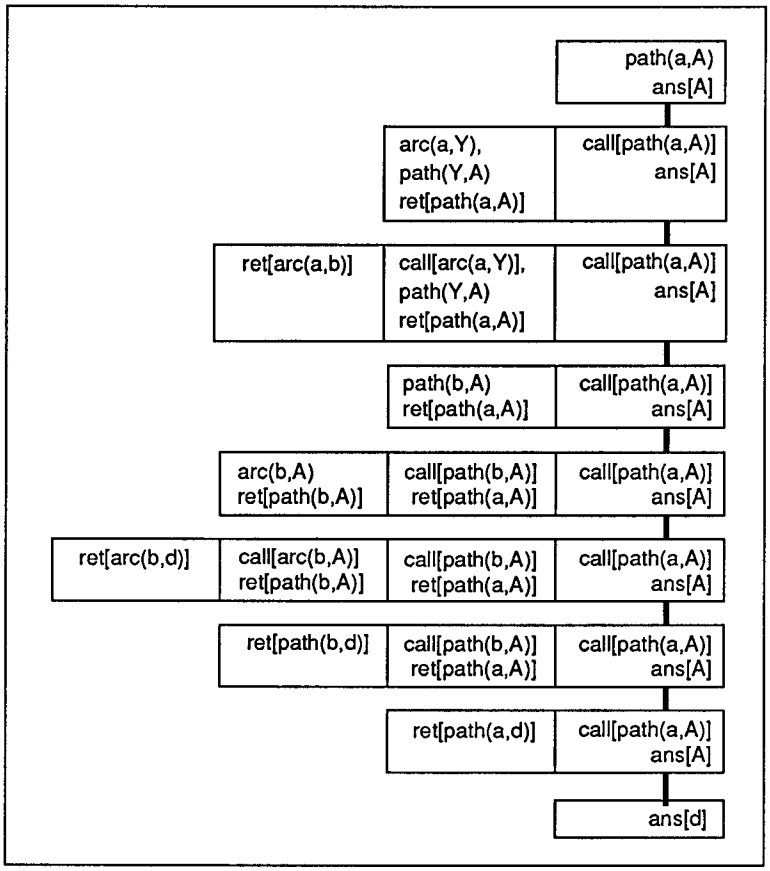


Figure 6. OLD refutation

becomes a call atom is the point where the call is made, and that is the point at which the call is added to the global memo table. We stipulate that for each atom, only one such node (row) may exist in the (tabling) OLDT tree, and it has a child for every matching rule. Such a node is called a *solution* node. Any left-most frame whose only atom is a return atom represents an answer to the call atom of the preceding frame. Nodes whose initial atoms are duplicates (up to variable renaming) of initial atoms of solution nodes are called *look-up* nodes. A look-up node gets a child for each answer node that shows up in the tree for its corresponding solution node. The solution nodes of the tree make up the calls of the memo table, and the answer nodes make up the returns. Such a tree is called an OLDT tree, and may be finite even when the corresponding SLD tree is infinite.

As these OLDT trees can get rather large, we give only small examples. As a simple example of a finite OLDT tree, consider the definition of transitive closure and the trivial single-arc graph in Figure 7. Notice that the definition is left-recursive, so the SLD tree would be infinite and the Prolog evaluator would loop infinitely. Figure 8 shows the OLDT tree for this program on the query *path(a, X)*.

Assume that the left branch is expanded first and is fully completed. The root is the *solution* node for the atom *path(a, X)*, and its right child is generated from the second rule for *path*. Now the initial atom of that right child is *path(a, Z)*, which is the same as the initial atom of the root node, up to change of variables. The right-child node is a *look-up* node and therefore does not get expanded by the program rules. Instead, it waits for answers to be generated elsewhere in the tree. The node just above the left leaf is such an answer node, and its answer generates a child of the look-up node, which is the node shown. However, the initial atom, *arc(b, Y)*, of this child matches no rule, and thus the node is a failure node. And the OLDT tree is complete, showing how an OLDT may be finite even though the SLD tree is infinite.

```
arc(a, b).

path(X,Y) :- arc(X,Y).
path(X,Y) :- path(X,Z), arc(Z,Y).
```

Figure 7. Left recursive transitive closure

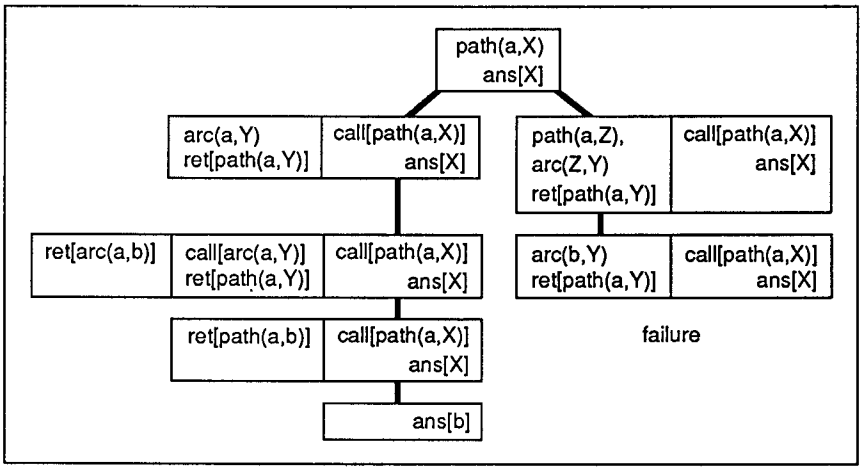


Figure 8. OLDT refutation

Magic

Another way to understand the memoing evaluation strategy was developed in the deductive database community, and goes by the name of *Magic* [1, 2, 27, 28, 29, 36]. Datalog, the subset of Horn clauses that do not use structure symbols, was noticed to be an elegant database language [17]. (For simplicity, we will not consider negation in bodies of rules, which, under some definitions, are part of the "Datalog" language.) Datalog can use recursion to express queries not expressible in the relational algebra of standard relational database theory. Our definition of transitive closure is an example.

When Datalog is used as a database language, the nontermination of SLD evaluation makes it unsuitable as an evaluation strategy. The standard way to evaluate database queries is to convert them into the relational algebra, (optimize them), and then evaluate them as expressions, using the relational operations to combine component relations. This is a bottom-up strategy that calculates new relations by combining old ones. At each step, it is possible to eliminate duplicate tuples, since the relations are explicitly created. From this starting point, the obvious way to evaluate recursive definitions is to begin with all the defined relations empty, and then iteratively compute new values for the relations, using the old relation values as input, until the relations produced as output are the same as the ones that are input, which indicates that a fixed point has been reached. This is the standard way to compute a least fixed point by iteration.

For example, consider the relation defined by the Datalog program of Figure 9 (which is the same program as in Figure 1). We start with the relation corresponding to *path*/2 being empty, and the relation for *arc* being {*arc(a, b)*, *arc(c, b)*, *arc(b, d)*}. The first iteration uses the first rule for *path* to add {*path(a, b)*, *path(c, b)*, *path(b, d)*}; the second rule does not add any tuples, since the relation for *path* is empty at the first iteration. The second iteration now uses the new value of path containing the three indicated tuples that

was computed on the first interaction. Therefore, on the second iteration, the first rule again computes {*path*(*a, b*), *path*(*c, b*), *path*(*b, d*)}; the second rule now computes the join of the *arc* and *path* relations of the previous iteration and adds {*path*(*a, d*), *path*(*c, d*)}. So the relation for *path* at the end of the second iteration is {*path*(*a, b*), *path*(*c, b*), *path*(*b, d*), *path*(*a, d*), *path*(*c, d*)}. On the third iteration, the same value for *path* is computed again, indicating that the least fixed point of the relational operator has been reached.

We have just described the bottom-up evaluation procedure. Clearly, this process terminates for any Datalog program over a finite domain: the size of the relations is nondecreasing at each iteration and the maximum relation size is finite. The fact that each intermediate relation is created at one time and the tuples accumulated together means that duplicates can be eliminated. This bottom-up strategy is correct, but it can be inefficient. One source of inefficiency is the redundant computation done at each level that completely recomputes the previous level, as seen in our example. This recomputation is relatively easy to avoid. Within each relation, we can distinguish the new tuples generated on the current iteration. When generating the tuples for the next iteration, we do not generate any tuples using only *old* tuples computed on previous iterations. This approach is called seminaive bottom-up evaluation [35].

Another serious source of inefficiency is that this strategy may compute many tuples that are completely irrelevant to the question at hand. The bottom-up nature of the strategy means that given a Datalog program, the same computation will be carried out regardless of the query. The query is used only at the final step to determine which of the computed tuples provide an answer. In our example, asking for points reachable from node *c* requires computation of the entire transitive closure, from which a selection is done to extract the desired answers. Computation of irrelevant tuples can be avoided in the (nonrecursive) relational algebra by a compile-time optimization known as 'pushing selects in'. Thus, even though a query may be expressed as first joining two large relations and then selecting a few tuples from that relation, the optimizer can determine that it is equivalent, but more efficient, to first select a few tuples from one of the relations and then join that much smaller relation with the other larger one to obtain the answer. In effect, the select operation has been pushed inside the join operation. However, the compile-time optimization of 'pushing selects in' does not apply in the presence of recursion. In a top-down (goal-directed) evaluation as done in SLD refutation, the selection is automatically achieved. Only the selections of relations demanded by *calls* are ever computed.

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

path(X,Y) :- arc(X,Y).
path(X,Z) :- arc(X,Y), path(Y,Z).
```

Figure 9. Simple Datalog program

The solution to this problem in the bottom-up framework can loosely be described as pushing the selects in at run time. In Datalog a kind of dynamic selection can be achieved by adding another literal to the body of a rule. The new filtering relation must be defined to be true of the 'selecting values'. These 'selecting values' are those that would occur in some call executed in a top-down evaluation of the query [2, 5].

For example, adding filtering literals to the path rules of Figure 9 gives the rules shown in Figure 10. In general, each rule for a predicate *p* gets a new literal in its body consisting of *calls_to_p* applied to the arguments of the head. Predicates defined only by ground facts (called extensional predicates) are not changed.

The new relations, such as *calls_to_p*, must be defined. Therefore, we add clauses to define the tuples for which there could be a call in a top-down evaluation. This can be done from the clauses in the original program. For example, schematically for a predicate $p(\_, \_)$, and a rule:

$$q(\_, \_) :- r(\_, \_), s(\_, \_), p(\_, \_), t(\_, \_)$$

we know that in a top-down evaluation, if this rule is used, then *p* will be called if *q* is called and computation succeeds through *r* and *s*. The effect can be captured with the following rule:

$$calls\_to\_p(\_, \_) :- calls\_to\_q(\_, \_), r(\_, \_), s(\_, \_)$$

And of course, since the query itself defines the first call, it contributes a fact to the transformed program.

Applying the magic transformation to the transitive closure program of Figure 9 for the query *path*(*c, X*) results in the program of Figure 11. The two rules defining *path* are as in Figure 10. The first rule (i.e., the fact) for *calls_to_path* is generated from the query, since the query itself will generate a call to *path*(*c, X*). The second rule for *calls_to_path* is generated from the second rule for *path* of Figure 9, since a call to *path* that succeeds through *arc* will cause another call to *path*. To evaluate the query *path*(*c, X*), the magically transformed program, shown in Figure 11 is evaluated by means of the seminaive bottom-up procedure. The bottom-up procedure must be extended (in the straightforward way) to handle variables.

Now consider the bottom-up computation of the transformed program of Figure 11. On the first iteration, neither of the rules for *path* applies (since *calls_to_path* and *path* are both empty). So, from the *calls_to_path* fact, we get only *calls_to_path*(*c, X*). In the second iteration, the first rule for *path* applies and generates *path*(*c, b*) by joining the *calls_to_path* of iteration

one and *arc*. We also get *calls_to_path(c, Z)* from the *calls_to_path* rule. Now, on the third iteration we get *path(b, d)* and *calls_to_path(d, Z)*, and on the fourth iteration we get *path(c, d)*. The fifth iteration produces nothing new, and the fixed point is reached.

The correspondence between the filtered bottom-up evaluation and the OLDT evaluation of the same query should be reasonably clear. A table entry of an OLDT computation essentially corresponds to a *calls_to_path* tuple and the *path* tuples that are instances of it. The algorithms are essentially the same [31]. There are some differences depending on whether and how subsumption checking is used to eliminate duplicates [23].

The algorithm presented here is essentially the Magic Template algorithm ([27]), which is a generalization of the Magic Sets algorithm that was developed for evaluating a subset of Datalog programs. Our development assumes that the bottom-up evaluation will happily process tuples containing variables. But handling variables requires a more complex (and perhaps less efficient) mechanism than relational-algebra processors normally contain. By allowing the transformation to apply only to *range-restricted* Datalog programs (those in which each variable in the head of a rule appears in the body), it can be modified to generate programs that are guaranteed not to generate tuples with variables during bottom-up evaluation. In this situation the *magic* relations turn out to be projections of our *calls_to_p* relations (i.e., to contain only some of their columns). For example, in our *path* example, *calls_to_path* would be a unary predicate and the fact defining it in Figure 11 would be *calls_to_path(c)*, which is range-restricted, instead of the *calls_to_path(c, X)* that we have, which is *not* range-restricted.

This *magic* transformation can be understood as a compilation technique for Datalog (and more generally Horn clause) programs. It allows a bottom-up evaluation engine to evaluate a program in an efficient top-down manner. The primary difference between direct implementations of *Magic* and OLDT relate to the data structures maintained. In the OLDT algorithm a tree of the run-stacks of the computations is maintained, so that answers are returned directly to those computations desiring them. In the magic algorithm, no tree is maintained, and answers are distributed to those needing them by (re-)performing a join.

There is some confusion of terminology in the community. The magic templates evaluation strategy has been referred to as a bottom-up evaluation strategy, and OLDT as a top-down evaluation strategy. As we have seen, the computations these strategies carry out for any program and query are essentially the same. Thus these differing characterizations must be understood carefully. Adding memoing introduces a bottom-up component to a top-down strategy, and making the magic transformation introduces a top-down component to a bottom-up strategy. The interesting point is that the resulting algorithm is essentially the same in either case.

Properties of OLDT

As discussed, perhaps the most important property of the OLDT algorithm is that it terminates for all Datalog programs. Termination is required for it to be of use in applications to deductive databases.

The algorithm also has good efficiency properties. In order to analyze the complexity of the algorithm, we must first identify a subclass of programs and queries. Since the general algorithm on arbitrary programs, not just Datalog, and queries is undecidable, complexity analysis does not apply. For propositional Horn clause programs (those containing no variables), and arbitrary propositional queries, the OLDT algorithm is $O(N \log N)$ for programs (+ queries) of size N, which is optimal. (The *log* factor comes from the table look-up operations. If the proposition symbols are prenumbered, making direct array look-up possible, the log factor disappears and the complexity is linear.)

Another class of programs, for which we can compare the OLDT complexity with other known complexity results, are those obtained from context-free grammars. Given a logic program corresponding to a context-free grammar, the evaluation of a query in which a string is provided is the recognition problem. In this case, the OLDT algorithm reduces to a variant of Earley's context-free recognition algorithm [8]. If the grammar is in Chomsky form (at most two nonterminals on the right-hand-side of any rule) and if care is taken in the representation of the string, then OLDT has the same complexity as Earley's algorithm. In particular, it is $O(N^3)$ for arbitrary grammars, where N is the length of the input string. If the grammar is unambiguous, it is $O(N^2)$.

These results suggest that OLDT's complexity properties are quite good.

OLDT Conclusion

I believe this OLDT algorithm (and its variants and optimizations) will be increasingly recognized as important in logic programming. In deductive databases, the magic set implementation is already having a

```
path(X,Y) :- calls_to_path(X,Y), arc(X,Y).
path(X,Z) :- calls_to_path(X,Z), arc(X,Y), path(Y,Z).
```

Figure 10. Magic-filtered Datalog program

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

path(X,Y) :- calls_to_path(X,Y), arc(X,Y).
path(X,Z) :- calls_to_path(X,Z), arc(X,Y), path(Y,Z).

calls_to_path(c,X).
calls_to_path(Y,Z) :- calls_to_path(X,Z), arc(X,Y).
```

Figure 11. Magic-ally transformed Datalog program

great impact. In fact, it has been the deductive database community, through its strong commitment to declarativeness, that has focused attention on these techniques, and contributed to their optimization and application.

The OLDT algorithm described here applies only to pure Horn clause programs. It can, however, be directly extended to apply to general logic programs that include negations in the bodies of rules. Just as SLD can be extended to SLDNF by a recursive invocation of the procedure for ground negative subgoals, so OLDT can be extended to OLDTNF [32]. Indeed, OLDT can be extended to compute the well-founded semantics of general logic programs [41].

There have been several implementations of this algorithm. The LDL deductive database system developed at MCC [22] includes an implementation of the magic set algorithm. The deductive database system developed at ECRC includes an implementation of a version of the SLD-AL algorithm [38] which is essentially OLDT. Work by R. Ramakrishnan is under way at Wisconsin to explore implementation of the Magic Templates strategy. Here at Stony Brook, we are in the process of extending the WAM implementation of Stony Brook Prolog to include tabulation. We call this new virtual machine the XWAM [40].

Abstract Interpretation

Next we turn to another general area of recent active research in logic programming: abstract interpretation.

In the area of declarative languages, the issue of optimization is of prime importance. The most successful example of the implementation of a declarative language is the relational database query language, Structured Query Language (SQL). This is the standard language used to interact with relational databases. It is critically important for the SQL compiler to produce a near optimal way to evaluate whatever query it is given. It is only because such optimization is possible that users are willing to use SQL. For general Horn clause programs, which include recursive definitions and recursive data structures, a comparable global optimization cannot yet (and may never) be achieved. Prolog, with its depth-first backtracking search through the tree of SLD refutations, is a procedural implementation of the declarative Horn clause language. And, as a procedural programming language, it can benefit from the kinds of optimization techniques developed for traditional procedural languages.

To perform certain optimizations, the compiler must know certain properties of any execution of the program being processed. For example, any Prolog predicate might be called with any arbitrary set of its parameters given ground values. This multidirectionality is one source of Prolog's power. However, many Prolog predicates are called in only one direction (i.e., in one *mode*). If the Prolog compiler knows the mode of a call, it can often use that information to generate more efficient code.

Such information depends on global properties of the program and query. One way to gain such information would be to execute, at compile time, the program for all inputs, and then use the information obtained. This direct approach is, of course, infeasible in practice. Any interesting program will have infinitely many computations. Therefore, rather than execute the program at compile time, we "generalize" the program to make a new approximate program. The approximate program preserves the structure of the original program but operates on abstract objects (perhaps representing sets of objects) rather than the concrete objects. Then the abstract program is executed to provide approximate information about all the executions of the original program. This process is called *abstract interpretation.*

One simple example of abstract interpretation that is familiar to everyone is the *rule of signs.* Consider a program in a very simple programming language that allows a sequence of assignments, each of which assigns to a variable an expression of plus, minus and times over integers and variables. Consider, for example, the program of Figure 12.

Given a set of integer values for the input variables, X and Y, executing this program will generate integer values for the output variables: U, V, W and Z. For example, initializing $X = 5$ and $Y = -6$, the program computes $Z = -30$, $W = 900$, $U = -930$, and $V = 870$. We can extract some information that relates the inputs and outputs of the program by using the rule of signs. We abstract the integers into three sets: 1) negative integers (−), 2) zero (0), and 3) positive integers (+). We also denote the whole set of integers by (*all*). We know how the operators of plus, minus and times operate on these sets of integers. For example, the product of two positive integers is a positive integer. The abstracted table for times is:

*	−	0	+	*all*
−	+	0	−	*all*
0	0	0	0	0
+	−	0	+	*all*
all	*all*	0	*all*	*all*

and the table for addition is:

+	−	0	+	*all*
−	−	−	*all*	*all*
0	−	0	+	*all*
+	*all*	+	+	*all*
all	*all*	*all*	*all*	*all*

and the table for minus would be similar. Now by simulating the program, using these four sets as the objects and the abstracted operations as the operations, we may be able to determine, for example, that whenever certain inputs are positive, certain outputs are positive. In our example of Figure 12, if we initialize $X = +$ and $Y = -$, we compute $Z = -$, $W = +$, $U = -$

and $V = all$.

This extremely simple example shows how information can be extracted from a program by approximate computation. The information extracted is approximate, and it may be the case that little information is gained. In the example, adding a positive and a negative integer may result in *any* integer, and so we obtain no information at all about the run-time value of V. On the other hand, we may gain precise and useful information. In our example, with a very simple finite computation, we have found that, under our input assumptions, the value of the variable U is always negative. Such information might allow the compiler to choose a more efficient representation for the value of that variable.

Inferring Modes for Datalog Programs

The concept of abstract interpretation is general and can be applied to any programming language, and many compilers use it to gain information for use in optimization. It has also been extensively explored in the context of logic programming. It might be said both that LP is in greater need of such information extraction and that LP provides a more elegant basis for abstract interpretation.

One kind of information that is important to a Prolog compiler is *mode* information: which variables will be bound to ground terms and which to variables at execution time. As a simple example, consider a program without function symbols (a Datalog program), such as shown in Figure 13. We use, for example, the notation, $p(+,-)$, to represent all calls to p in which the first argument is a ground term and the second is a variable. We may wish to know the modes of all calls that might possibly be made during the evaluation of a query with mode $p(+,-)$. One way to determine the modes of all calls is to modify the program by changing all constants in the program to the same constant, say +. For the example of Figure 13, the transformation would give the program shown in Figure 14. All the tuples in the q relation have collapsed to the single tuple $q(+,+)$. This is an *abstracted* program in which every constant has been mapped to the same abstract value, +.

Now consider executing the mode query $p(+,-)$ using the abstracted program. We find that the first rule generates the call $q(+,-)$. The second rule first generates the call $q(+,-)$, and then since that call binds the returned variable to +, the second call is $q(+,-)$ also. Thus we have discovered that for any query of mode $p(+,-)$, every call will be of the same mode, $q(+,-)$. We could alternatively pose the query $p(-,+)$. In this case the first rule would generate a similar call $q(-,+)$. The second rule would generate the call $q(-,-)$, and then the call $q(+,+)$. Thus for such an initial query, we have three different ways in which q might be called.

From this example, we see that by executing the program and remembering the forms of the calls, we can determine all the calling modes that might occur during execution of the concrete Prolog program, given an initial query of the given mode. In our development, we have assumed Prolog's evaluation strategy for the (abstract) interpretation, but we could also use the tabling strategy, OLDT, to evaluate the abstracted query and program. It should not take too much thought to convince oneself that OLDT evaluation will generate exactly the same calls as the Prolog evaluation will (and also exactly the same returns). And OLDT has the additional advantage of accumulating the calls in the table automatically. For example, executing the query $q(-,+)$ on the transformed program with OLDT generates a table with the three call patterns (each with the same return: $q(+,+)$).

The Datalog program we have considered here has no recursion and is very simple. Consider the slightly more complex example of Figure 15.

The program pictured there is the abstraction of a program that computes the transitive closure of the relation q. The abstraction transformation, as before, leaves the rules for p unchanged, and collapses all the q-tuples into the one fact: $q(+,+)$. Prolog execution of the abstracted program enters an infinite loop. The reason is that the abstraction operation loses the information the concrete Prolog program requires in order to terminate. In this case, that information is that the directed graph defined by the q predicate in the concrete program is acyclic. The abstraction operation turns it into the trivial cyclic graph. However, as before, we can apply OLDT evaluation, which will terminate on the abstract Datalog program and generate a set of call patterns. The question is

```
Z := X*Y
W := Z*Z
U := Z-W
V := Z+W
```

Figure 12. Simple linear assignment program

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

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

Figure 13. Simple Datalog program

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

q(+,+).
```

Figure 14. Abstracted program

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

q(+,+).
```

Figure 15. Abstracted recursive program

whether these OLDT results are the ones we want. The answer is yes, but it would take more machinery to prove it than we want to develop here. So here we see that the use of OLDT for evaluating the abstracted program is more than just a convenience for collecting the table of results; without it many abstract programs would not terminate.

Let us summarize our simple examples of using abstract interpretation to determine the calling modes for Datalog programs. In the concrete domain, the program executes with variables and a number of constants. Variables are abstracted to themselves, and all constants are abstracted to a single constant, +. The abstraction is analogous to the abstraction of numbers to one of +, −, or 0, in the simple numeric example that began this section. In the case of Datalog, the abstract *operation* is the same as the concrete operation—they are both unification. That is, unification of the abstracted values gives the same answer as the abstraction of the unification of the concrete values.

Note that the information extracted from the Datalog program by the abstract interpretation we have described is approximate. That is, it is possible for a particular mode to be inferred, and yet no execution ever to generate a call of that mode. Conjunctions of subgoals might fail in the concrete execution due to mismatching values, but succeed in the abstracted program when all constants are mapped to the single value +.

Inferring Modes for Prolog Programs

As a more complex example, consider the problem of mode inference for general Prolog programs, including those with function symbols. In the Datalog case we were able to convert all program objects from the concrete domain to the abstract domain at compile time. However, with more complex abstractions, this may not be possible. Next we develop a simple mode inference abstraction for programs with function symbols, which will require some object abstraction during execution.

The abstract domain will consist of three values, represented by Prolog terms: X, $p(Y)$, and $p(+)$, where X and Y are variables. Abstract value X will represent a concrete value of a variable; the abstract value $p(Y)$ will represent all terms; and $p(+)$ will represent all ground terms. This representation was chosen because its increasing instantiation order, $X < p(Y) < p(+)$, reflects the order in which mode information will be obtained during abstract program execution. That is, we may initially know that a program variable is unbound (X), then learn that it is bound to some arbitrary term ($p(Y)$), and then learn it is ground ($p(+)$).

As an example, consider a clause for *append/3*:

append([X|$L1$], $L2$, [X|$L3$]) :−
  append($L1$, $L2$, $L3$).

Consider how we might translate it to an abstract clause. The first argument, [X|$L1$], is not in the abstract domain, so we must allow for it to be abstracted during execution. As a first step we move these complex arguments into the body of the clause and do the unification explicitly:

append($A1$, $L2$, $A3$) :−
  [X|$L1$] = $A1$, [X|$L3$] = $A3$,
  append($L1$, $L2$, $L3$).

Next, in the abstract evaluation, the variables will have one of the three values in the abstract domain, so the two explicit unification operations must be modified to perform the abstraction operation on their left, list-term operands. We use the infix operand => /2 to represent this abstraction operation and obtain the clause:

append($A1$, $L2$, $A3$) :−
  [X|$L1$] => $A1$, [X|$L3$] => $A3$,
  append($L1$, $L2$, $L3$).

The abstraction operation ensures that its two operands are consistently instantiated. This operation ought to be viewed as a *constraint*; that is for example, that [X|$L1$] => $A1$ represents the constraint that *at all times*, $A1$ should represent the instantiation state of a simple constructor with two subterms of instantiation states X and $L1$. However, in simple Prolog, there is no way to ensure that this constraint is enforced automatically throughout the entire computation. For example, in the *append* example, the execution of the recursive *append* call might change the instantiation state of its arguments, $L1$, $L2$ and $L3$. Thus, in the absence of a constraint LP system, we must ensure that after the return of the recursive invocation to *append*, the instantiation states of $A1$ and $A3$ are updated accordingly. Therefore, our final abstract clause is:

append($A1$, $L2$, $A3$) :−
  [X|$L1$] => $A1$, [X|$L3$] => $A3$,
  append($L1$, $L2$, $L3$),
  [X|$L1$] => $A1$, [X|$L3$] => $A3$.

We can think of the => /2 operator as bringing the states of instantiation of the arguments into synchrony. The first pair of => subgoals models the propagation of instantiations from the head of the clause to the subgoal; the second pair models the propagation of the instantiations from the body back to the head on return from the recursive call.

The entire abstract mode-inferencing program for *append/3* and *nrev/2* (naive reverse), written in Prolog, is shown in Figure 16. The definition for => /2 is shown in Figure 17 in Prolog using metapredicates *var/1* and the control primitive, cut (!). It is defined only for the list structure symbol as necessary for application to this program.

Evaluated by Prolog's strategy, this abstract program for *append/3* could easily loop infinitely: a call of *append*($p(X)$,$p(Y)$,$p(Z)$) generates the identical recursive call. However, the OLDT algorithm guarantees termination. The calls and an-

swers in the resulting OLDT table safely approximate the instantiation patterns of possible executions of *append/3* and *nrev/2*. For example, we can pose the query *append(X,Y,p(+))* to this mode inference program and evaluate it using OLDT. In effect, this query asks what mode values would be returned for an initial call to *append/3* with two variables and a ground term. The resulting answer is *append(p(+),p(+),p(+))*, indicating that the two variable arguments will become ground. We can also look at the tables constructed by the OLDT algorithm and see what subcalls would need to be made, and their modes. In this case, the only call to *append* in the table is *append(X,Y,p(+))*.

As another example, consider the query *nrev(p(+),X)*. This evaluation results in the call table of:

```
call: append(p(+),p(+),X)
  ans: [append(p(+),p(+),p(+))]
call: nrev(p(+),X)
  ans: [nrev(p(+),p(+))]
```

which is what is expected.

As a more interesting example, consider *nrev(X,p(+))*. The table generated for this call is:

```
call: nrev(X,p(+))
  ans: nrev(p(A),p(+));
  ans: nrev(p(+),p(+))
```

There are several interesting things to note. We know from the semantics of list reversal that the logically correct answer should be *nrev(p(+),p(+))*, i.e., that *nrev* could only succeed with the first argument ground. But the answer given is *nrev(p(X),p(+))*, which means that, as far as can be inferred by this method for this program, *nrev* could return with its first argument bound to any term. This shows that this inference may be weaker than we would wish. On the other hand, Prolog, when given a query of mode *nrev(X,p(+))* would loop infinitely. We might note in passing that even under the OLDT evaluation strategy applied to the concrete program, this goal would result in an infinite loop, since it would generate infinitely many different calls to *nrev* and *append*.

Work in Abstract Interpretation

This section on abstract interpretation has shown by examples how abstract interpretation works and how it can be used to infer interesting run-time properties of logic programs. We have also demonstrated that by using the OLDT evaluation strategy, we can simply transform the object program and then evaluate it directly with OLDT to get certain kinds of abstract interpretations very simply. A similar approach based on OLDT is suggested and developed in [11] and extended and used in [9]. A correspondence between earlier work on abstract interpretation by [19] and the magic templates evaluation strategy, which is clearly in this same spirit, was pointed out by [23].

Here we have barely scratched the surface of the work on abstract interpretation and its applications to logic programming. A general framework for abstract interpretation of logic programs has been developed by [4]. As a few examples of applications, abstract interpretation has been applied to mode inferencing (as exemplified in this article) in [6], type checking/inferencing in [9], nonfloundering inference in [18], and analysis of variable aliasing in [21]. These authors and many others have done interesting work applying abstract interpretation to a wide variety of

```
% Mode inference using: X for var, p(X) for partial (or unknown),
%   and p(+) for ground.

% append([],X,X).
% append([X |L1],L2,[X |L3]) :- append(L1,L2,L3).
append(p(+),L,L).
append(X1,L2,X3) :-
    [X |L1] => X1,
    [X |L3] => X3,
    append(L1,L2,L3),
    [X |L1] => X1,
    [X |L3] => X3.

% nrev([],[]).
% nrev([X |L],R) :- nrev(L,M), append(M,[X],R).
nrev(p(+),p(+)).
nrev(X1,R) :-
    [X |L] => X1,
    nrev(L,M),
    [X |p(+)] => X2,
    append(M,X2,R),
    [X |L] => X1.
```

Figure 16. Abstract program for mode inference for *nrev/2* and *append/3*

```
% Define "consistent abstraction"; note use of var and cut.
[Y |Z] = > p(X) :- var(X),var(Y),var(Z),!.
[p(Y) |Z] => p(X) :- var(X),var(Y),var(Z),!.
[p(+) |Z] => p(X) :- var(X),var(Z),!.
[Y |p(Z)] => p(X) :- var(X),var(Y),var(Z),!.
[Y |p(+)] => p(X) :- var(X),var(Y),!.
[p(Y) |p(Z)] => p(X) :- var(X),var(Y),var(Z),!.
[p(+) |p(Z)] => p(X) :- var(X),var(Z),!.
[p(Y) |p(+)] => p(X) :- var(X),var(Y),!.
[p(+) |p(+)] => p(+).
```

Figure 17. Definition of the abstract operator: => /2

problems in the analysis of logic programs.

Partial Deduction

The process of executing the instructions of a program written in some source language is normally divided into two steps: compilation and execution. A standard assumption made in this context is that programs are usually executed many times for each time they are compiled. Under this assumption, it is advantageous to do as much work as possible during compilation time, leaving as little as possible to do at run time. The process of carrying out part of the computation at compile time that would otherwise normally be done at run time is called *partial evaluation.* In the context of logic programming, where computation can be viewed as logical deduction, this process might more logically be called *partial deduction.*

One very simple example of partial evaluation is the processing of macros. Macros are expanded at compile time so that less computation needs to be done at run time. Many programming languages support a macro language. If the macro language is a subset of the programming language itself, then macro expansion becomes partial program execution carried out at compile time. One could easily introduce a macro language in Prolog, simply by allowing a predicate to be declared to be a macro. Then the compiler evaluates any call to such a predicate at compile time.

Consider the simple example shown in Figure 18. In this program fragment, we use a configuration declaration in *system_type*/1 to indicate which external system call is to be made. Then depending on the *system_type,* we either call *old_syscall* or *new_syscall.* The predicate *process*/2 performs some further processing of the result of the system call. This program can be executed exactly as it is; at every execution it decides which syscall to make. However, if we declare the three predicates: *syscall, system_type* and *p_syscall* as macro predicates, then the compiler can evaluate them once. This compile-time evaluation can be understood as the sequence of resolution steps shown in Figure 19. All the resolution steps involving macro predicates are carried out at compile time, and in this case rule 4 of Figure 19 is compiled instead of rule 1. An advantage of this approach to macros in which the macro language is a subset of the programming language is that the semantics of a program is the same, regardless of what predicates are declared to be macros. A program can be written and debugged without any macros. Then some predicates can be declared as macros to improve the run-time performance, and the semantics of the program does not change.

In the general case of partial (or compile-time) evaluation, there are two basic problems to be solved: 1) determining what calls to evaluate at compile time, and 2) when to stop evaluating. In the simple case of macros, these problems have particularly easy solutions: evaluate every call to each predicate that is declared to be a macro. Termination of partial evaluation is guaranteed by forbidding recursive definitions in macros.

There are, however, more interesting and powerful cases of partial evaluation in which some part of the input is known at compile time and one tries to execute the part of the program that depends *only* on that input. An elegant example is that of an interpreter. An interpreter takes as input a program and data for the program, and it simulates the behavior of the given program on that data. If we consider the input program fixed and evaluate the interpreter with respect to it, we get a version of the interpreter specialized for that input program. In a sense, this allows us to turn an interpreter into a compiler by means of partial evaluation. Such uses of partial evaluation are much more dynamic, and many partial evaluation systems require human intervention to provide input about which calls are to be evaluated and which are not. In such systems, a programmer controls the partial evaluation process.

Partial evaluation can be defined as a transformation technique that, given a Program P and a goal G, produces a new program P' which is more efficient than P on G or its instances. Note that the concept of 'instances' is natural in the logic programming framework. Thus, to be more precise in the context of Horn clause programming, at compile time we find a partial SLD tree for goal G of the form depicted in Figure 20.

G is a goal and this represents a partial SLD tree, with leaves: L_1, *, L_2, *, and L_3. At each nonleaf node *all* rules that match the selected literal are applied to generate children. Thus the leaves are either failing clauses that cannot be expanded (indicated in the diagram by *'s) or are conjunctions of literals (indicated by the L_i's of the diagram), which are left unexpanded. Continuing the schematic example, given such a partial SLD tree, we can modify the program by adding the rules:

$G :- L_1.$
$G :- L_2.$
$G :- L_3.$

That is, for any partial SLD tree, we create new rules by pairing the root goal with each nonempty leaf node. Since these rules are guaranteed to follow logically from the program rules, adding them does not change the logical semantics of the program. The idea is that by doing these resolutions at compile time and adding the rules to the program, the work of deriving the L_i from G is done just once (at compile time) and not redone every time G (or one of its instances) is encountered during execution.

However, simply adding these new rules does not decrease execution time but instead increases it, since now there are more rules than in the original program, and all the old rules will still apply. The point

is to delete some other rules, in particular the ones that get from G to the L_i, so they will not be used again at execution time. But this must be done carefully, since those rules may be used elsewhere in the program. For example, one of the predicates with an atom in L_1 might be defined in terms of a generalization of G. In this case deleting the old rules for G could make the new partially evaluated program incorrect.

A Simple Partial Evaluator

To explore these issues in partial evaluation further, we develop a simple partial evaluator. The purpose of this example is simply to develop our intuitions and show how easily some kinds of partial evaluation can be carried out. It is by no means appropriate for all applications of partial evaluation in Logic Programming. This partial evaluator, not surprisingly, has the structure of a Prolog metainterpreter. The idea is that it will be given a goal G and a program P and it will partially evaluate G with respect to P. We begin with the well-known 3-line metainterpreter shown in Figure 20 [33], and we modify it to delay some of the evaluation. We will assume that each atom is declared to have exactly one of the following 'partial evaluation' statuses: 1) *eval,* meaning that such calls are to be completely evaluated at compile time, 2) *partial,* meaning that such calls are to be delayed but should be recursively partially evaluated at compile time, and 3) *uneval,* meaning that such calls are to be completely delayed to run time, for example, for built-in system operations. These statuses determine which calls are to be evaluated at compile time.

Consider the extension of the vanilla metainterpreter shown in Figure 21, which can be used to carry out such a partial evaluation. The predicate *part_eval*(G, L) computes a partial SLD tree for G returning nonempty leaf nodes in L. The evaluation is with respect to the program stored in *rule*/2. For example, a call to *part_eval* passing the G of the schematic partial SLD tree of the previous subsection would return the list $[L_3, L_2, L_1]$. (In this program the delayed list of unevaluated goals is constructed in reverse order.) The call to *new_copy* makes a copy of the goal, disconnecting the variables from the call, so the goal passed into this procedure remains unchanged.

The predicate *part_ev*/3 does most of the work and mirrors the structure of the vanilla metainterpreter of Figure 20. The two extra arguments contain lists of goals that are not evaluated but delayed (i.e., subgoals that are *not* selected for expansion). The third evaluation rule of the vanilla metainterpreter has been specialized to three rules to handle the three different partial evaluation statuses: 1) If the goal is to be evaluated, the rules are applied, just as in the metainterpreter; 2) If the goal is not to be evaluated, it is simply added to the list of delayed, unevaluated goals; 3) If the goal is to be partially evaluated, the goal itself is delayed, and a recursive call is made to partially evaluate it. Note, however, that no attempt is made here to accumulate the new rules generated by the recursive partial evaluation (and so *Li* is an anonymous variable).

Macro expansion, as described here, can be carried out using this partial evaluator. Each macro predicate is declared such that all its calls

#	Rule
1.	p(X,Y) :- sys_call(X,Z), process(Z,Y).
2.	sys_call(X,Y) :- system_type(S), p_syscall(S,X,Y).
3.	system_type(os3r5).
4.	p_syscall(os3r2,X,Y) :- old_syscall(X,Y).
5.	p_syscall(os3r5,X,Y) :- new_syscall(X,Y).

Figure 18. Program with Macros

#	From	Derived Rule
1.	(rule 1)	p(X,Y) :- sys_call(X,Z), process(Z,Y).
2.	(1, rule 2)	p(X,Y) :- system_type(S), p_syscall(S,X,Z), process(Z,Y).
3.	(2, rule 3)	p(X,Y) :- p_syscall(os3r5,X,Z), process(Z,Y).
4.	(3, rule 5)	p(X,Y) :- new_syscall(X,Z), process(Z,Y).

Figure 19. Partial evaluation

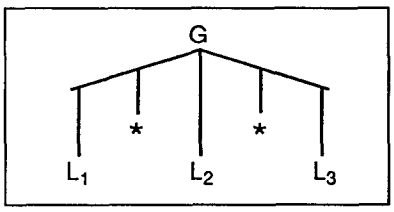


Figure 20. A partial SLD tree

```
eval(true).
eval((A,B)) :- eval(A), eval(B).
eval(G) :- rule(G,B), eval(B).
```

Figure 21. Vanilla Meta-Interpreter

have status *eval*, and all calls to regular predicates have status *uneval*. Then each occurrence M of a call to a macro predicate in the body of a rule of the original program is replaced by (the reverse of) L where *part_eval*(M, L).

There are several problems with this partial evaluator as it stands. For example, it may partially evaluate the same predicate (infinitely) many times. Also, the generated rules need somehow to be saved. Both of these issues can be solved by using the OLDT algorithm as the basis for the partial evaluation extension, instead of SLD as was done here. In fact, we can evaluate this partial evaluating metainterpreter (of Figure 21) using the OLDT resolution strategy. Then the table for *part_eval*/2 eliminates such duplicate evaluations and contains exactly the newly generated rules. Again, use of the OLDT strategy in place of the SLD strategy eliminates loops.

Two problems remain: 1) guaranteeing termination and 2) determining what rules to delete after the new ones have been added. While the OLDT strategy eliminates some simple kinds of infinite loops, it cannot be guaranteed to terminate always. If all predicates were declared with status of *eval*, then partial evaluation would reduce to complete evaluation, in which case termination is clearly undecidable. Thus, in the simple partial evaluation framework we have developed here, a termination proof would have to be developed independently for each application, specifically depending on the declared evaluation statuses and the actual programs.

The second remaining issue involves what rules can be deleted from the partially evaluated program. A simple approach is to enforce syntactic constraints on programs to ensure that no unevaluated predicate can depend on a partially evaluated (or completely evaluated) one. In this case, the old rules can be safely deleted, as long as only instances of the original goal G are evaluated with the partially evaluated program.

Another approach involves the generation of new predicates symbols and new subgoals. In this case all the rules of the original program are retained in the final program, but the partially evaluated goals make calls to new predicates, defined by the newly generated rules. This can be done by generating a new predicate symbol for each partially evaluated goal, and replacing the delayed goal with a new goal obtained from the new predicate symbol applied to the variables of the original goal. For example, given a call such as *interpret*(*sentence* (*Parse*), *S*0, *S*) to be partially evaluated, a new predicate symbol, perhaps *interpret_sent*/3, is generated, and the new goal, *interpret_sent* (*Parse*, *S*0, *S*), is delayed. Then the new rules define these new predicates, and none of the original program rules are deleted. Looking carefully, we can see that the new predicates correspond to particular instances of the predicates as defined in the original program. Since we add new predicates, the old ones are unchanged, and unevaluated calls to old predicates can still use the old definitions. Another advantage of this approach is that it eliminates some unnecessary record structures in some calls. In the example, the call *interpret*(*sentence* (*Parse*), *S*0, *S*) becomes the call *interpret_sent*(*Parse*, *S*0, *S*), eliminating in the process the structure symbol *sentence*. In a Prolog system that indexes only on the main functor symbol of the first argument, this can greatly improve the indexing characteristics of the partially evaluated program.

As a final example of partial evaluation, we show how the partial evaluator of Figure 21 can be used to eliminate a level of interpretation of a metainterpreter. As noted, a partial evaluation of an interpreter can behave like a compiler.

Consider the program shown in Figure 22. It shows a set of rules that can be partially evaluated using the evaluator of Figure 21 (evaluated with the OLDT strategy). The first three rules are the vanilla metainterpreter, defining a predicate *demo*. The next two rules define clauses for *append* and the last two define clauses for naive reverse, *nrev*. Figure 22 defines rules for a metainterpreter and for the *nrev* program.

We illustrate how the partial evaluator of Figure 21 can be used to derive a more efficient set of rules to evaluate queries such as *demo*(*nrev*([*a*, *b*, *c*], *Rev*)). The partial evaluator can automatically remove the level of interpretation represented by the *demo* predicate and the three rules that define the vanilla metainterpreter.

We define the *status*/2 predicate for the partial evaluator as shown in Figure 23. Then we use the OLDT evaluation strategy to execute the partial evaluator for the query *demo*(*nrev*(*X*,*Y*)). Then, looking in the resulting tables for *part_eval*, we find the rules shown in Figure 24. (In the figure, we write them in Prolog notation.) The rules are grouped to define *append* and *nrev*. Note that the rules for *demo* (*nrev*(*X*,*Y*)) mirror exactly the rules for *nrev*, and the interpretation overhead of calling *demo*/1 and *clause*/2 is entirely eliminated. If we replace *demo*(*nrev*(*X*,*Y*)) by *demo_nrev*(*X*,*Y*) using a new predicate symbol as suggested above and make a similar replacement for *append*, we obtain the 'compiled' rules for *nrev*. As a matter of fact, note that since *nrev* only calls *append* with the second argument as a singleton list (and the recursive call within *append* therefore does the same), the clauses generated for *demo*(*append* (_, _, _)) are specialized so they assume that the second argument is a singleton list.

Other Issues in Partial Deduction

Partial evaluation, as the evaluator illustrates, is particularly easily understood and implemented in logic programming. This transparency is due to the simple syntax and semantics of pure logic programs. In particular, the logic variable and

the uniform treatment of nondeterminism makes partial deduction elegant. Initial work in partial evaluation in logic programming was done in [12], and work on the formal foundations is presented in [16].

However, real Prolog programs are not always so pure. Partial evaluation in the context of full Prolog is a more difficult process. One difficulty arises because application of a partial deduction step may cause a variable to get its value at run time earlier than it would without the partial deduction. This happens when a variable that becomes bound to a value in a partial deduction step also appears in the clause to the left of the atom resolved on. As a simple example, by partially evaluating the atom $q(X)$, in the clause:

$p(X) :- var(X), q(X).$

using a fact $q(a)$, we would generate the clause:

$p(a) :- var(a).$

In Prolog, *var*/1 is a metapredicate that, when it is executed, tests whether or not its argument is bound to a variable. Called with $p(Z)$, the original clause would succeed, binding Z to a. However, the second clause would fail, since a is not a variable. Simple partial deduction in the presence of metalogical constructs such as *var*/1 may not preserve meanings.

In this particular case, we can limit this backward propagation of variable values by introducing a new variable and an explicit equality (which is not evaluated) as follows. We first rewrite the original clause as:

$p(X) :- var(X), X = Y, q(Y).$

Then when we partially evaluate the atom $a(Y)$ as before, we obtain:

$p(X) :- var(X), X = a.$

This clause does have behavior equivalent to that of the original clause.

Partial evaluation of full Prolog has been discussed in [30]. In [3], it is argued that rather than extending the techniques into the realm of the impure, the energy would be better spent designing logic languages that are better than Prolog, in which impurities do not arise.

```
part_eval(G,Li) :- rule(G,B), part_ev(B,[],Li).

part_ev(true,L,L).
part_ev((A,B),L0,L) :- part_ev(A,L0,L1), part_ev(B,L1,L).
part_ev(G,L0,L) :- status(G,eval), rule(G,B), part_ev(B,L0,L).
part_ev(G,L,[G |L]) :- status(G,uneval).
part_ev(G,L,[G |L]) :- status(G,partial), new_copy (G,NG), part_eval(NG,_Li).
```

Figure 22. Partial evaluator

```
rule(demo(true), []).
rule(demo((A,B)), [demo(A),demo(B)]).
rule(demo(G), [clause(G,B),demo(B)]).

rule(clause(append([],L,L),true), []).
rule(clause(append([X |L1],L2,[X |L3]),append(L1,L2,L3)), []).

rule(clause(nrev([],[]),true), []).
rule(clause(nrev([X |L],R,(nrev(L,L1),append(L1,[X],R))), []).
```

Figure 23. Meta-interpreter with naive reverse

```
status(demo(X),T) :- var(X), !, T=partial.
status(demo(true),T) :- !, T=eval.
status(demo(( _,_ )),T) :- !, T=eval.
status(demo(nrev( _,_ )),T) :- !, T=partial.
status(demo(append( _,_,_ )),T) :- !, T=partial.
status(demo( _ ),T) :- !, T=partial.
status(clause( _,_ ),T) :- !, T=eval.
```

Figure 24. Definition of evaluation status

```
demo(append([],[A],[A])).
demo(append([A |B],[C],[A |D])) :- demo(append(B,[C],D)).

demo(nrev([],[])).
demo(nrev([A |B],C)) :- demo(nrev(B,D)), demo(append(D,[A],C)).
```

Figure 25. Result of partially evaluating interpreter on *nrev*

Partial evaluation has also been applied in logic programming to the derivation of virtual machines. Partial evaluation was used by [14] to derive the unification instructions of the WAM, the Prolog virtual machine. Similar techniques were used by [24] to derive the WAM instructions that support control flow. Recently, partial evaluation techniques have been applied to constraint logic programming languages [10].

Conclusion

The goal of this article has been to give readers concrete insight into three related areas of research in logic programming: 1) tabulated (or bottom-up) evaluation, 2) abstract interpretation, and 3) partial deduction.

Tabulated evaluation is a computation strategy under which more Horn clause programs terminate than under Prolog's SLD strategy. This means more Horn clause specifications can be used as programs. Applications of this algorithm have been made in the area of deductive databases, and in this article we have seen natural applications to abstract interpretation and partial

deduction. An application not explored in this article is the subarea of AI known as nonmonotonic reasoning. In such reasoning, new knowledge is inferred when certain information fails to be derivable. The new knowledge is said to be inferred by default. Because of OLDT's good termination characteristics, a system using it can more often finitely show that certain goals cannot be derived, enabling it to make more inferences by default.

Abstract interpretation in logic programming is a burgeoning area of research. We are learning how to use it to exact greater amounts of global information from logic programs. The many ways in which it is being used improve logic program performance.

Partial deduction is also a growing area of research in logic programming. Partial evaluation techniques have numerous applications. In this article we hinted at their use for creating compilers from interpreters. In general, partial deduction is good for transforming high-level interpretive specifications into lower-level efficient programs. Therefore, it too, contributes to supporting declarative programming, making it increasingly easy to implement special-purpose higher-level logic languages. ◻

References

1. Bancilhon, F., et al. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth PODS Symposium,* pp. 1–15, 1986.
2. Beeri, C. and Ramakrishnan, R. On the power of magic. In *Proceedings of the Sixth PODS Symposium,* pp. 269–283, 1987.
3. Benkerimi, K. and Lloyd, J.W. A partial evaluation procedure for logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming.* MIT Press, pp. 343–358, 1990.
4. Bruynooghe, M. A practical framework for the abstract interpretation of logic programs. *J. Logic Program. 10,* 2, 1991, 91–124.
5. Bry, F. Query evaluation in recursive databases: Bottom-up and top-down reconciled. To be published in *Data and Knowl. Eng.,* 1990.
6. Debray, S.K. and Warren, D.S. Automatic mode inference for logic programs. *J. Logic Program. 5,* 3, 1988, 207–230.
7. Dietrich, S.W. and Warren, D.S. Extension tables: Memo relations in logic programming. Tech. Rep., Department of Computer Science, SUNY at Stony Brook, Mar., 1986.
8. Earley, J. An efficient context-free parsing algorithm. *Commun. ACM, 13,* (1970) 94–102.
9. Filè, G. and Sottero, P. Abstract interpretation for type checking. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming,* Springer-Verlas, N.Y., 1991, pp. 311–322.
10. Hickey, T.J. and Smith, D.A. Toward the partial evaluation of CLP languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* ACM Press, June 1990, pp. 43–51.
11. Kanamori, T. and Kawamura, T. Abstract interpretation Based on OLDT-Resolution. ICOT Tech. Rep., 1990.
12. Komorowski, J. A specification of an Abstract Prolog Machine and its application to partial evaluation. Ph.D. thesis, Linkoping Univ., 1981.
13. Kowalski, R. *Logic for Problem Solving.* Elsevier North-Holland, N.Y., 1979.
14. Kursawe, P. How to invent a Prolog machine. *New Gen. Comput. 5,* 1 (1987).
15. Lloyd, J.W. *Foundations of Logic Programming.* Springer-Verlag, N.Y., 1984.
16. Lloyd, J.W. and Shepherdson, J.C. Partial evaluation in logic programming. *J. Logic Program. 11,* 3–4, 1991, pp. 217–244.
17. Maier, D. and Warren, D.S. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings, Menlo Park, Calif., ISBN 0-8053-6681-4, 535 pp, 1988.
18. Marriott, K., Sondergaard, H., and Dart, P. A characterization of non-floundering logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming,* MIT Press, 1990, pp. 661–680.
19. Mellish, C. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, Eds. *Abstract Interpretation of Declarative Languages,* Ellis Horwood, (1987) pp. 181–198.
20. Michie, D. Memo functions and machine learning. *Nature 218,* (1968) pp. 19–22.
21. Muthukumar, K. and Hermenegildo, M. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming,* MIT Press, (1991) pp. 49–63.
22. Naqvi, S. and Tsur, S. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, N.Y., 1989.
23. Nilsson, U. Abstract interpretation: A kind of magic. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming,* Springer-Verlag, N.Y., 1991, pp. 299–310.
24. Nilsson, U. Towards a methodology for the design of abstract machines of logic programming languages. To be published in *J. Logic Program.*
25. Pereira, F.C.N. and Warren, D.H.D. Definite clause grammars for language analysis. *Artif. Intell. 13,* (1980) 231–278.
26. Pereira, F.C.N. and Warren, D.H.D. Parsing as deduction. In *Proceedings of ACL Conference* (1983).
27. Ramakrishnan, R. Magic Templates: A spellbinding approach to logic programs. *J. Logic Program. 11,* 3–4 (1991) 189–216.
28. Rohmer, J., Lescouer, R., and Kerisit, J.-M. The Alexander method—A technique for the processing of recursive axioms in deductive databases. *New Gen. Comput. 4,* 3 (1986) 273–285.
29. Sagiv, Y. Is there anything better than magic? In *Proceedings of the 1990 North American Conference on Logic Programming,* MIT Press, 1990, pp. 235–254.
30. Sahlin, D. The Mixtus approach to automatic partial evaluation of full Prolog. In *Proceedings of the 1990 North American Conference on Logic Programming,* MIT Press, 1990, pp. 377–390.
31. Seki, H. On the power of Alexander templates. In *Proceedings of the Eight PODS Symposium,* Philadelphia, (1989) pp. 150–159.
32. Seki, H. and Itoh, H. A query evaluation method for stratified programs under the extended CWA. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* MIT Press, 1988, pp.

195–211.
33. Sterling, L. and Shapiro, E. *The Art of Prolog.* MIT Press, Cambridge, Mass., ISBN 0-262-19250-0, 1986, p. 427.
34. Tamaki, H. and Sato, T. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming* (London), *Lecture Notes in Computer Science 225,* Springer-Verlag, 1986, pp. 84–98.
35. Ullman, J.D. *Principles of Database and Knowledge-Base Systems, Vol 1.* Computer Science Press, N.Y., 1988.
36. Ullman, J.D. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth PODS Symposium,* Philadelphia, 1989, pp. 140–149.
37. vanEmden, M.H. and Kowalski, R. The semantics of predicate logic as a programming language. *J. ACM 23,* 4, 1976, 722–742.
38. Vieille, J. Recursive query processing: The power of logic. *Theoretical Comput. Sci. 69,* 1 (1989) pp. 1–53.
39. Warren, D.S. Syntax and semantics in parsing: An application to Mongague grammar. Ph.D. thesis, Univ. of Michigan, 1979.
40. Warren, D.S. The XWAM: A machine that integrates Prolog and Deductive, database query evaluation. Tech. Rep. 89/25, Department of Computer Science, SUNY at Stony Brook, Oct. 1989.
41. Warren, D.S. Computing the well-founded semantics of logic programs. Tech. Rep. 91/12, Department of Computer Science, SUNY at Stony Brook, June, 1991.

CR Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*nondeterministic languages, nonprocedural languages.* D.3.4 [**Programming Languages**]: Processors—*interpreters, optimization;* I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming, resolution.*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: OLDT.

About the Author:

DAVID S. WARREN is a professor of computer science at the State University of New York/Stony Brook. His research interests include logic programming, deductive databases, knowledge representation, and nonmonotonic reasoning. **Author's Present Address:** Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.