Top-Down vs. Bottom-Up Revisited

Raghu Ramakrishnan and **S. Sudarshan**
University of Wisconsin-Madison
Madison, WI 53706, USA
{raghu,sudarsha}@cs.wisc.edu

Abstract

Ullman ([Ull89a, Ull89b]) has shown that for the evaluation of safe Datalog programs, bottom-up evaluation using Magic Sets optimization has time complexity less than or equal to a particular top-down strategy, Queue-based Rule Goal Tree (QRGT) evaluation. This result has sometimes been incorrectly interpreted to mean that bottom-up evaluation beats top-down evaluation for evaluating Datalog programs—top-down strategies such as Prolog (which does no memoing, and uses last call optimization) can beat both QRGT and bottom-up evaluation on some Datalog programs.

In this paper we compare a Prolog evaluation based on the WAM model (using last call optimization) with a bottom-up execution based on Magic Templates with Tail Recursion optimization ([Ros91]), and show the following: (1) Bottom-up evaluation makes no more inferences than Prolog for range-restricted programs. (2) For a restricted class of programs (which properly includes safe Datalog) the cost of bottom-up evaluation is never worse than a constant times the cost of Prolog evaluation (and can be much better than Prolog for many programs). Our other main contribution is to identify the factors that make the cost of an inference potentially more expensive in the bottom-up model in the general case; this leads to a clearer understanding of the potential implementation costs of memoing/set-oriented evaluations, and suggests an important direction for future research.

1 Introduction

The following example demonstrates that Prolog with last call optimization (see, eg. [MW88]) can beat bottom-up evaluation using Magic Sets/Magic Templates optimization ([RLK86, Sek89, BMSU86, BR87b, Ram88]) for some Datalog programs.

Example 1.1 This example is from [Ros91]. Let P be the program

$p(X, Z) \leftarrow e(X, Y), p(Y, Z).$
$p(n, X) \leftarrow t(X).$
$e(1, 2), \ldots, e(n-1, n).$
$t(1), \ldots, t(m).$
Query: ?-$p(1, X)$.

Given the subgoal $?p(1, X)$ Prolog sets up subgoal $?e(1, X)$ and gets an answer that binds X to 2. Using this binding Prolog sets up a subgoal $?p(2, X)$, which in turn sets up subgoal $?p(3, X)$ and so on till the subgoal $?p(n, X)$ is set up. However, Prolog can deduce that there are no more answers to $e(1, X)$, and when an answer for $?p(2, X)$ is found, it can directly return (with bindings for variable X) to the subgoal $?p(1, X)$, bypassing the subgoal $?p(2, X)$. This is known as last call optimization. But applying this optimization repeatedly, when Prolog find an answer for subgoal $?p(n, X)$, it returns directly (in unit time, with bindings for X) to the

subgoal $?p(1, X)$, bypassing all intermediate subgoals. Since there are m answers for $?p(n, X)$, Prolog backtracks to $?p(n, X)$ a total of m times, and evaluates the program in time $O(n + m)$. Bottom-up evaluation (using Magic Sets rewriting), on the other hand, generates each fact $p(i, j), 1 \leq i \leq n, 1 \leq j \leq n$, and takes $O(m * n)$ time. Notice that Prolog "generates" only facts $p(n, j), p(1, j), 1 \leq j \leq m$ (here "generating" a fact is interpreted as the act of Prolog's control returning, with appropriate variable bindings, to the point where a subgoal was set up). □

Prolog's use of last call optimization essentially allows it to avoid "generating" some facts that bottom-up evaluation generates. To achieve this effect for the case of bottom-up evaluation, Ross [Ros91] proposed a rewriting technique that extends the Magic Templates rewriting algorithm [Ram88]. For the convenience of the reader, we describe this technique in Section 4. Ross compares this technique with bottom-up evaluation, and proves that under some sufficient conditions it computes fewer facts than bottom-up evaluation. However Ross does not compare his technique with Prolog.

We present a model for Prolog evaluation in Section 3; we call this *Prolog** evaluation. In Section 5 we show that bottom-up evaluation of a program using Ross' technique makes no more inferences than a Prolog* evaluation of the program. This result does not imply that bottom-up evaluation dominates Prolog*, since the cost of an inference in bottom-up evaluation can in general be greater than the cost of an inference in Prolog*.

In Section 7 we define a restricted class of programs (which properly includes safe Datalog), and in Section 8 show that for this class of programs, each inference in bottom-up evaluation has unit cost. Hence, for this class of programs, the order-of-magnitude time complexity of bottom-up evaluation is never more than that of Prolog*; of course for some programs the order-of-magnitude time complexity of Prolog* could be much worse than that of bottom-up evaluation.

Finally in Section 9, we identify the factors that make the cost of an inference in the bottom-up model in general more expensive than in the Prolog model. It is important to note that these factors are equally applicable to *any evaluation method that memos all goals and facts*, not just bottom-up evaluation. Thus, our results shed light on a broader issue than top-down versus bottom-up evaluation strategies; namely the important issue of memoing versus non-memoing strategies.

2 Background

We assume the standard definitions of Horn clause logic programs and assume that the programs that we are given consist of sets of Horn clause rules that do not contain negated literals in the body.[1] Note that Prolog programs that use non-declarative features are not logic programs under the standard definition of Horn clause logic programs, and we do not consider them in this paper. The programs that are generated as a result of the rewriting we use are not necessarily Horn clause programs—they use higher order syntactic features that are part of Hilog [CKW89]. We do not describe these features formally, but the following rules illustrate the extensions to Horn clauses that we use:

$R1: \ A \leftarrow query(p(X, Y), A), r(X, Y)$

$R2: \ query(p(X, Y), q(Y)) \leftarrow query(q(Y), q(Y)), r(X, Y)$

[1] We can extend this class to cover certain restricted forms of negation such as stratified negation.

Suppose we have facts $query(p(a,X), q(b,X))$, $r(a,c)$ and $r(a,d)$. Using rule $R1$ we can infer the facts $q(b,c)$ and $q(b,d)$. The use of this higher-order syntax is not essential for our discussion, but it makes the presentation concise.

We assume that Semi-Naive evaluation [BR87a, Ban85] is used. In a bottom-up evaluation, a derivation is made as follows: previously derived facts are unified with each predicate in the body of a rule; the head of the instantiated rule is the fact that is derived. Semi-naive bottom-up evaluation has the property that no derivation step is repeated in the evaluation—in other words, each time a given fact is derived it is derived using a different rule, or using different body facts. Although our programs use Hilog syntax and are not Horn programs, we can use Semi-Naive evaluation with very minor changes, since the programs do not have a variable occurring in place of a predicate occurrence in any rule body.

3 A Model for Prolog Evaluation

We now present our model of Prolog evaluation.

Definition 3.1 Prolog* : We define *Prolog** evaluation as Prolog evaluation using the standard Warren abstract machine model (see eg. [MW88]), with last call optimization, but without using any other optimizations that affect the number of subgoals set up, or the number of answers generated.[2] We assume that Prolog* evaluation proceeds till all answers are generated (i.e., Prolog* does not stop at the request of the user), and that Prolog* evaluation terminates and is sound. □

If a Prolog* evaluation is not complete, or does not terminate, bottom-up evaluation can certainly do no worse. Hence, we only consider Prolog* evaluations that terminate and are complete. This also has the benefit of simplifying our proofs considerably.

Suppose we had a subquery $?p(\overline{t})$, and a set of rules defining p. In order to answer the subquery, Prolog* tries solving it using each of the rules. Consider a rule of the following form:

$$R:\ p(\overline{t_0}) \leftarrow q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n}).$$

1. Prolog* first attempts to unify the subgoal with the head of R. If the unification fails, Prolog* proceeds to try other rules defining p, and when all of them are done, it has finished answering the subgoal.

2. If unification succeeds in the previous step, some of the variables in the rule and in the subgoal get instantiated. If the body of the rule is empty, the call returns successfully right away. Otherwise the first literal in the body of the instantiated rule constitutes the next subgoal that Prolog* attempts to solve.

3. Prolog* then attempts to compute an answer to the subgoal. Computation of the answer may result in binding some of the variables in the rule.

4. On successfully computing an answer to a subgoal, Prolog* normally returns to the point where the subgoal was generated. However, when last call optimization (see Section 3.1 below) is applicable, Prolog* can avoid returning to the point where the subgoal is invoked, but can instead return directly to the point of invocation of an earlier subgoal. We refer to this step as the "generation of an answer" to the subgoal to which control returns.

[2] For instance, we disallow Intelligent Backtracking (see eg. [CD85]).

5. On successfully getting an answer for a literal, if there are no more literals in the body of the rule Prolog* returns from the rule to the calling subgoal as described in the previous step. If there are more literals in the rule body Prolog* sets up a subgoal on the next literal in the body of the rule, and continues as in Step 3.

6. If there are no (more) answers to a subgoal, Prolog* backtracks to the subgoal generated just prior to this one. Also after successfully generating an answer to the original query on the program Prolog* backtracks to the last subgoal that was generated. In either case it attempts to generate a new answer for that subgoal. Depending on whether it succeeds or fails, computation proceeds according to Step 3 or Step 6.

The above model is a simplified description of Prolog evaluation, and omits many details such as how control flow is directed. The important point to note is that each of Steps (1)-(6) takes at least constant (in our model 1 unit) time, and the computation can be viewed as a sequence of such steps. We call each of these steps in a Prolog* evaluation as an *action* performed in the evaluation.

3.1 Last Call Optimization

Consider a rule of the form: R : $p(\overline{t}) \leftarrow q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n})$. Suppose we had a subquery $p(\overline{a})$, and in answering this subquery we had invoked further subqueries $q_1(\overline{a_1}), \ldots, q_n(\overline{a_n})$. Suppose further that

1. R is the last clause defining p, and

2. For each of $q_1(\overline{a_1}), \ldots q_{n-1}(\overline{a_{n-1}})$ there are no more alternatives, i.e., each of them has returned its last answer, and

3. We are currently trying the last clause for q_n.

The subgoal $?q_n(\overline{a_n})$ will return zero or more successful answers. When each answer is returned, no more computation is done at rule R, but control merely passes back to the point where the subgoal $?p(\overline{a})$ was invoked. When no more answers can be generated for $?q_n(\overline{a_n})$, Prolog* backtracks to the point where the subgoal $?p(\overline{a})$ was generated. In each of these cases, there is no reason to return to rule R—Prolog* can therefore change the return address so that the call to $?q_n(\overline{a_n})$ returns directly, bypassing R. This optimization is called *last call optimization* (see for instance [MW88]). In particular, when q_n is the same as p, i.e., when R is tail recursive, we may return directly past a large number of invocations of R. By bypassing R, Prolog* in effect bypasses a step where a bottom-up evaluation using Magic Templates rewriting would have created facts for the head predicate p of rule R. Example 1.1 demonstrates the possible benefits of this optimization.

If bottom-up evaluation is to perform as well as Prolog*, it too must bypass the step of computing a fact for the head of rule R. This is precisely the optimization achieved by the program rewriting technique of Ross [Ros91], which we describe in Section 4.

4 Magic Templates With Tail Recursion

Ross ([Ros91]) proposed a modification to Magic Templates ([Ram88]). We describe Ross' technique, which we call *Magic Templates with Tail Recursion* (MT-TR) rewriting, in this

section, with a few minor modifications.[3]

Facts of the form $query(p(\overline{t_1}), q(\overline{t_2}))$ are generated in the bottom-up evaluation of the rewritten program. Such a fact intuitively says (1) $?p(\overline{t_1})$ is a goal (and it is generated in a Prolog* evaluation of P), (2) $?q(\overline{t_2})$ is an "ancestor" [4] of $?p(\overline{t_1})$, and (3) a solution to $?p(\overline{t_1})$ provides bindings for variables in $\overline{t_1}$; applying these bindings to $q(\overline{t_2})$ gives us answers for $q(\overline{t_2})$. These answers are the facts that must be generated explicitly; answers to the subgoal $?p(\overline{t_1})$ need not be generated as facts.

MT-TR Rewriting: Given program P and a query $?q(\overline{t})$ on P, we generate a program using the following rewrite rules. We call the resultant rewritten program P^T.

0. Generate the rule (actually a fact) $query(q(\overline{t}), q(\overline{t}))$. Call this a *Type 0* rule.

Consider each rule R_j in the program P. Let rule R_j be of the form

$R_j : \ h(\overline{t}) \leftarrow p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n})$

Let $\overline{V}$ denote a tuple of all variables that appear in R_j.

1. Generate the rule $sup_{j,0}(\overline{V}, A) \leftarrow query(h(\overline{t}), A)$. Call such rules *Type 1* rules.

2. If the body of R_j is non-empty, generate the following rules and call them *Type 2* rules:
$$sup_{j,1}(\overline{V}, A) \leftarrow sup_{j,0}(\overline{V}, A), p_1(\overline{t_1})$$
$$\vdots$$
$$sup_{j,n-1}(\overline{V}, A) \leftarrow sup_{j,n-2}(\overline{V}, A), p_{n-1}(\overline{t_{n-1}})$$

3. If the body of R_j is empty generate the rule $A \leftarrow sup_{j,0}(\overline{V}, A)$. Call such rules *Type 3* rules.

4. If the body of R_j is non-empty, for each predicate p_i, $i \neq n$ in the body of R_j generate a rule
$$query(p_i(\overline{t_i}), p_i(\overline{t_i})) \leftarrow sup_{i-1}(\overline{V}, A)$$
Call such rules *Type 4* rules.

5. If the body of R_j is non-empty generate the following rule:
$query(p_n(\overline{t_n}), A) \leftarrow sup_{n-1}(\overline{V}, A)$. Call such rules *Type 5* rules.

We say that P^T generates a subgoal $?p(\overline{t})$ if it derives a fact $query(p(\overline{t}), \ldots)$. Type 0 and Type 4 rules generate subgoals that must be explicitly solved; however, Type 5 rules provide last call optimization—in effect they say "solve the last subgoal in rule R_j, but instead of generating answers for it, use the bindings to directly generate answers for the query that invoked the rule". Type 1, 2 and 3 rules collectively perform the same function as rules in the original program, except that they are restricted to generate facts only if there is a corresponding subgoal; thus they avoid generating many irrelevant facts. The Hilog notation is not critical—we can rewrite the program in normal Horn clause syntax, but the Hilog notation is more concise.

Bottom-up evaluation of P^T may in some cases generate many more facts that bottom-up evaluation of the Magic Templates rewritten form of P. However, as we show in later sections

[3] Essentially, we treat all predicates as right-recursive, whereas this is a parameter to Ross' rewriting.

[4] But not necessarily a *proper* ancestor.

it has the advantage (for our purposes) of being consistently better than or comparable with Prolog* on the class of programs we consider.

5 A High Level Comparison of Bottom-up Evaluation and Prolog*

We show that the derivations performed by the semi-naive evaluation of P^T can be mapped onto the actions performed during an evaluation of the query on P using Prolog*. While this comparison is interesting in its own right, we also use it in Section 8 to show that for a particular class of programs bottom-up evaluation dominates Prolog* evaluation. The following theorem summarizes our results.

Theorem 5.1 *Let P be a range-restricted program*[5] *and Q a query on P. Let P^T be the MT-TR rewriting of P with query Q. Then there is a mapping M of derivations in the semi-naive evaluation of P^T to actions of the Prolog* evaluation of Q on P, with the following properties.*

1. *M maps every derivation of a fact query$(p(\overline{t}), \ldots)$ by the bottom-up evaluation to a generation of the subquery $?p(\overline{t})$ by Prolog*.*
2. *M maps every derivation of a fact $p(\overline{a})$ by bottom-up evaluation, where p is a predicate from P, to a "generation" of the fact by Prolog*.*
3. *M is one-to-one on the above.*
4. *M maps derivations of facts $sup_{i,j}(\ldots)$ to an action of Prolog* (such as a unification of a subgoal with the head of a rule, or generation of a subquery), and no more than 2 different derivations are mapped on to the same action of Prolog*.* □

By taking last call optimization into account, this theorem extends earlier results of Ramakrishnan [Ram88] and Seki [Sek89] which compare the number of inferences made by bottom-up evaluation with the number of inferences made by specific top-down strategies. We observe again that the number of inferences is only one aspect of the cost of an evaluation strategy. To obtain a more accurate comparison, the cost of each inference has to be taken into account.

6 A Model for Bottom-up Evaluation

Our model for bottom-up evaluation is as follows. The given program is first rewritten using Magic Templates with Tail Recursion (Section 4), and then by Semi-Naive rewriting [BR87a, Ban85]. Next the rewritten program is evaluated using Semi-Naive evaluation. We model the entire set of actions in a Semi-Naive bottom-up evaluation as a sequence of uses of facts to derive other facts. In an iteration, a fact $r(\overline{a})$ is said to be newly derived if it was derived for the first time in the previous iteration. Each fact is labeled as newly derived for precisely one iteration. Each newly derived fact $r(\overline{a})$ is used to make inferences as described below:

1. Choose (semi-naive rewritten) rules to use $r(\overline{a})$ in.

[5] An earlier version of this paper, which appeared in the International Logic Programming Symposium, 1991 erroreously omitted the restriction of this theorem to range-restricted programs.

2. *Standardize apart* $r(\overline{a})$ from the rule body, i.e., make variable names in $r(\overline{a})$ distinct from those in the rule by renaming variables if required.

3. Unify $r(\overline{a})$ with a body literal in the rule. This unification provides bindings for variables in the rule body.

4. If the rule has two body literals perform the following actions:

   (a) Index the other predicate in the rule body to fetch facts that can unify with the (partially) instantiated literal.

   (b) When each fact is fetched, standardize it apart from the instantiated rule body.

   (c) Unify the renamed facts with the rule body.

5. Create head facts.

6. Check for subsumption (by previously generated facts) of the derived head fact. If it is not subsumed, discard all facts that are subsumed by it, insert it into the relation, and mark it as a newly derived fact.

7 A Restricted Class of Programs

From parts (1)-(3) of Theorem 5.1 we see that each program fact and goal is generated by Prolog* at least as often as by bottom-up evaluation. We now present a class of programs for which each derivation in bottom-up evaluation takes unit cost, and in Section 8 we show that for this class of programs bottom-up evaluation "dominates" Prolog* evaluation.

Let p be a predicate in program P. A fact $p(\overline{t})$ is said to be *non-ground structure free* (NGSF) iff each argument of the fact is either a ground term or a variable. A meta-level fact of the form $query(p(\overline{t}), q(\overline{s}))$ (resp. $sup_{i,j}(\overline{u}, q(\overline{s}))$) is said to be NGSF iff $p(\overline{t})$ and $q(\overline{s})$ (resp. $\overline{u}$ and $q(\overline{s})$) are NGSF.

For example, facts $p(f(a, g(b)), X), query(p(X, Y), q(X, Y))$, $sup_{i,j}(f(a), p(f(a), X))$ and $p(X, g(c, g(c, g(c))), X)$ are non-ground structure free, but the facts $p(f(X))$ and $query(p(f(X)), q(X))$ are not. We say that a program is *non-ground structure free* iff every fact derived in every bottom-up evaluation of the program is non-ground structure free.

Definition 7.1 NGSF Evaluable : A program P with query Q is said to be *NGSF Evaluable* if in every evaluation of P^T, (1) all facts produced are NGSF, and (2) all facts produced for predicates from P are ground. □

The class of NGSF Evaluable programs subsumes the class of safe (i.e., range-restricted) Datalog programs. The following is a sufficient condition for a program P to be NGSF Evaluable.[6]

Condition Strongly NGSF Evaluable: We say that a program P with query Q is *strongly NGSF evaluable* if P satisfies the following condition:

1. P is range restricted (i.e., for each rule R in P, every variable in the head of R also appears in the body of R).

[6] We can derive other sufficient conditions, for instance by using the adorned program, and requiring that the query be NGSF, and ground on all bound arguments.

2. For every rule in P, for every literal $p(\overline{t})$ in the body of the rule, any variable that appears with an enclosing function symbol in $p(\overline{t})$ also appears in a literal to the left of $p(\overline{t})$ in the rule.

3. Those variables in the head of the rule that appear only in the last literal in the body of the rule do not appear with enclosing function symbols in the head of the rule.

4. The query on the program does not have any variables that are enclosed in function symbols. □

The intuition behind this condition can be understood as follows: Firstly, all facts produced by the program are ground. Secondly, a Prolog* evaluation of the query on the program would not create any subgoal containing non-ground structures. Thirdly, when Prolog* uses tail recursion optimization on these programs, structures in answers to queries on tail-recursive predicates will not be used to "build" larger structures in the head of the rule.

Proposition 7.1 *If P with query Q is Strongly NGSF Evaluable, then it is NGSF Evaluable.* □

While we would like the class of programs we consider to subsume the class of programs whose Magic Sets rewriting computes only ground facts, without the extra restriction provided by Part 3 of Condition Strongly NGSF Evaluable, P^T may compute facts with large non-ground structures that are hard to handle in bottom-up evaluation.

Next we show how to implement the operations of unification, indexing and checking of subsumption efficiently for the rewritten program P^T, given that P with the query is NGSF Evaluable.

7.1 Unification

Efficient unification of large terms that are ground can be achieved by a term representation called *hash consing* [Got74, SG76]. The idea behind this representation is to assign each term a unique identifier (ID)—the identifiers of two terms match if and only if they are identical. Thus unification of ground terms takes only constant time. The ID for a term $f(t_1, \ldots, t_n)$ is constructed from the IDs $(i_1, \ldots, i_n)$ of its subterms: we construct a hash value using $(i_1, \ldots, i_n)$ and the functor f. We then look up a hash table to see if $f(i_1, \ldots, i_n)$ is present, and if it is not, we insert it into the hash table. In either case, we use the address of this hash table entry as the ID of $f(t_1, \ldots, t_n)$. The ID is thus computed in constant time.

Proposition 7.2 *Unification of two NGSF facts can be done in constant time during the evaluation of a NGSF program.* □

After performing unification for the rule body, one must create the head fact. This takes only constant time if the program is NGSF: with a hash consed representation of ground terms, the head fact shares structured ground terms with body facts by using the hash consed IDs.

Proposition 7.3 *Renaming of variables in NGSF facts can be done in constant time. In the evaluation of an NGSF program, given a rule with (NGSF) facts for each body literal, the head fact can be created in constant time.* □

7.2 Indexing

We now describe a technique for indexing relations in P^T that works in time proportional to $O(1)$ + the number of facts that are retrieved. We use the following terminology. During bottom-up evaluation of P^T we may instantiate a literal $q(\overline{s})$ in a rule, and thereby get bindings for some of the variables in another literal, say $p(\overline{t})$ in the rule body. The partially instantiated $p(\overline{t})$ literal is referred to as the *pattern*. Indexing relation p refers to the operation of fetching from relation p those facts that unify with the pattern. The only rules that require indexing are the Type 3 rules; the others have just one literal in the rule body.

Definition 7.2 Pattern Forms : Consider a predicate p of arity n from program P. We define a *pattern form* for p as a string of length n from the alphabet $G_1, \ldots, G_n, V_1, \ldots, V_n$, with the following restriction — going left to right the first occurrence of G_{i+1} (resp. V_{i+1}) in the pattern form must be after the first occurrence of G_i (resp. V_i). Given a NGSF fact, we associate with it a pattern form as follows: going from the left argument to the right, we rename each new ground argument value we encounter by G_{i+1}, (resp. new variable by V_{i+1}) where $G_1, \ldots G_i$ (resp. $V_1, \ldots, V_i$) have been used so far. Clearly each fact has associated with it a unique pattern form.

We extend the definition to predicates $sup_{i,j}$ (resp $query$) as follows. Let p be any predicate in program P, and F_p a pattern form for predicate p. For the last argument of each $sup_{i,j}$ (resp. both arguments of $query$) the pattern form can have a corresponding argument of the form $p(F_p)$ (instead of G_k or V_k). □

The intuition behind pattern forms is to provide a canonical representation of arguments of facts, that can be used for indexing. An example of a pattern is $p(a, X, X, b, a)$, and the corresponding pattern form is $p(G_1, V_1, V_1, G_2, G_1)$. The pattern/fact $sup_{1,2}(a, X, q(X))$ has the associated pattern form $(G_1, V_1, q(V_1)$. The pattern/fact $sup_{1,2}(a, a, A)$ has the associated pattern form (G_1, G_1, V_1). Given a program with a finite number of predicates, there are only a finite number of such pattern forms for each predicate.

We say that a fact/pattern *unifies with a pattern form* if the following variant of normal unification of the fact/pattern and the pattern form succeeds: each V_i is treated as a variable during unification, and the unification of G_i is defined as follows. The most general unificand (mgu) of G_i and X is $[X/G_i]$, and G_i does not unify with G_j for $i \neq j$. The mgu of G_i with any ground term t is $[G_i/t]$ provided there is no substitution G_j/t for $i \neq j$; otherwise G_i does not unify with t.

Indexing Technique: For each pattern form F_i for a predicate we construct a separate ground hash index on each subset of the G_is in the pattern form. We call each index *an index of pattern form* F_i, and use the generic name *pattern form indices* to refer to these indices. When inserting a fact into the relation we insert it into all the indices of its associated pattern form. To retrieve facts that unify with a pattern, we check each pattern form F_j that unifies with the pattern. For each such pattern form we find those G_is that are bound to ground terms on unification with the pattern, and use the F_j index on these G_is to retrieve facts.

By using the hash consed representation of ground terms we can insert facts into a hash index or retrieve a single fact from a hash index in constant time. By the construction of rules in P^T and since all facts for predicates in the original program P are ground, the only patterns that arise for predicates $sup_{i,j}$ are NGSF and have no repeated variables; we call such

patterns *simple patterns*. We only need handle NGSF patterns for predicates p that occur in the original program, and we do not perform indexing for the *query* predicate, except to perform subsumption checking (which we discuss later). For the rewritten program P^T, indexing as described above retrieves all and only those facts that unify with the pattern, given that the original program P with query Q is NGSF Evaluable. For lack of space we have omitted the proof of correctness.

Inserting a fact into a single index takes constant time. The number of indices inserted into is a constant (it is independent of the number of facts generated, and is dependent only on the arity of the predicate and the number of predicates in P). Hence insertion of facts into relations can be done in constant time. There are only a constant number of pattern forms, and each can be indexed in time $O(1)$ + the number of facts retrieved from that index. No fact is retrieved more than once. Hence we have the following proposition.

Proposition 7.4 *If P with query Q is NGSF Evaluable, the cost of each indexing operation in the evaluation of P^T is $O(1)$ + the number of matching facts found.* □

While the number of indices we need may seem to be large, firstly it is constant, and secondly we can avoid constructing many indices by constructing them in a lazy fashion as and when an indexing operation requires the index. We conjecture that for each predicate p we need construct no more indices than the number of adorned/rectified predicates generated from p by Ullman's techniques [Ull89a]. Our indexing technique allows us to get rid of the adornment step normally used with Magic sets rewriting, as well as the rectification step used by Ullman [Ull89a, Ull89b], without losing efficiency of unification or indexing of relations. This simplifies some of our proofs, and can help to avoid repeated computation in some cases.

7.3 Subsumption Checking

Subsumption checking is easy for relations with ground facts. However it is harder for relations with NGSF facts, and we show how the above indexing scheme can also be used for this purpose. Suppose we want to check for subsumption for a given fact $r(\overline{a})$. Unlike the patterns on $sup_{i,j}$ discussed earlier, $r(\overline{a})$ can have repeated variables but on the other hand we need retrieve only facts that either subsume $r(\overline{a})$ or are subsumed by it.

Given pattern forms F_1 and F_2, we say that $F_1 \geq F_2$ iff there is a renaming σ of G_is in F_1 and a substitution γ on V_is in F_1 such that $(F_1[\sigma])[\gamma] = F_2$. Let $pf(r(\overline{a}))$ denote the pattern form of $r(\overline{a})$.

1. To find facts that are subsumed by $r(\overline{a})$ we use all pattern forms F_i such that $pf(r(\overline{a})) \geq F_i$. We use all the ground arguments of $r(\overline{a})$ to index the facts in the corresponding pattern form indices. All retrieved facts are either equal to $r(\overline{a})$, or are subsumed by $r(\overline{a})$ and may be discarded.

2. To find facts that subsume $r(\overline{a})$, we find all pattern forms F_i such that $F_i \geq pf(r(\overline{a}))$. We use all argument positions of $r(\overline{a})$ that are ground in F_i to index facts in the corresponding pattern form indices. If any such fact is found it subsumes $r(\overline{a})$, and $r(\overline{a})$ is discarded.

Discarding facts involves removing them from the indices, but this is the exact converse of insertion. The cost of performing a particular subsumption check is $O(1)$ + the number of

facts discarded. The total cost of all subsumption checks is $O(n) + O(d)$ where there are n subsumption checks performed, and d facts discarded. But $d < n$, so we have the following result.

Proposition 7.5 *If P with query Q is NGSF Evaluable, the amortized cost per subsumption check is $O(1)$.* □

8 Bottom-Up Beats Top Down For NGSF Evaluable Programs

In this section we look at each step of the model of bottom-up evaluation presented in Section 6, and study the costs involved in evaluating an NGSF program. We look at each cost in detail, and show how to "allocate" it to different facts. Allocating costs to facts helps make the comparison with Prolog* easier, and helps us prove the dominance of bottom-up evaluation for NGSF Evaluable programs.

Consider the model for bottom-up evaluation presented in Section 6. Steps 1, 2 and 3 are charged to the given fact. Steps 2 and 3 take $O(1)$ time for the class of programs we consider (Propositions 7.3, 7.2), so we have only a unit cost charged to each fact on account of these three steps. Step 4a performs an indexing on a relation to fetch tuples that match a partially instantiated literal. For the class of programs we consider, this cost is $O(1) +$ $O($ number of facts fetched $)$ (Proposition 7.4). We assign the cost $O(1)$ to the given (newly derived) fact $p(\overline{a})$. The rest of the cost is assigned as below. For each fact fetched successfully, Steps 4b, and 4c are performed. Also for each of these steps, a head fact is created. We assign a unit cost to each head fact (Propositions 7.3, 7.2). The creation of each head fact in Step 5 takes $O(1)$ (Proposition 7.3). Hence we again assign a cost of $O(1)$ to each head fact created. The amortized cost of checking for subsumption of a fact is $O(1)$ and the cost of insertion into a relation if it is not subsumed is also $O(1)$ under our model; both these costs are assigned to the head fact.

We assigned each fact a cost of $O(1)$ for each attempt to use it in a rule body when it is newly derived. The number of times this can happen is bounded by the number of occurrences of the predicate in the program, and is hence bounded by $O(1)$. We have also assigned a total cost of $O(1)$ to a fact for each time that it is derived. Hence we have the following proposition

Proposition 8.1 *Suppose P with query Q is NGSF Evaluable. Then the time complexity for Semi-Naive bottom-up evaluation of P^T under our model is $O($* number of derivations of facts *$)$.* □

Our main result, which is stated in the following theorem, follows from Theorem 5.1 and Proposition 8.1.

Theorem 8.1 *Suppose we are given a program P that is NGSF Evaluable. Let t_P be the running time of a Prolog* evaluation of P, and let t_B be the running time of the bottom-up evaluation of P^T. Then there is some constant c, that is independent of t_P and t_B (but may be dependent on the arity of predicates in P, and the textual size of P) such that $t_B \leq c * t_P$.*

This means that bottom-up evaluation, in the worst case, can be only a constant factor worse than Prolog*. On the other hand, it is easy to find examples where the behavior of Prolog* is much worse than that of bottom-up evaluation (for some programs Prolog* does not

even terminate, although bottom-up evaluation does).

9 The Cost of An Inference for General Logic Programs

For a restricted class of programs (NGSF Evaluable programs), we showed that bottom-up evaluation performs inferences at unit cost. In general, the cost of an inference in the bottom-up approach may be significantly more than the cost of an inference in a top-down approach without memoing. We discuss three major reasons for this below: [7] Note that these problems also occur with top-down evaluations that perform memoing.

Instantiating Shared Variables: When facts contain large terms, it is essential for facts derived by a rule to share subterms with facts used to derive them. Otherwise extensive copying would be required, and the cost of an inference would be linear in the size of the fact generated. Even if subterms are shared, extensive copying may be required if a variable in a shared non-ground structure is instantiated. Boyer and Moore's structure sharing scheme [BM72] can avoid this copying, but it has other overheads such as the time required to traverse a term. More importantly, it does not help with the next problem that we describe.

On the other hand, Prolog uses a tuple at a time backtracking strategy, and hence it can destructively modify variable bindings, and on backtracking it can undo the modifications in order to perform further derivations.

Extra Unifications: There are several unification operations in bottom-up evaluation, for instance unification in the course of indexing, that have no counterpart in a Prolog* evaluation. In particular, answer facts have to be explicitly unified with a rule in order to generate new facts; this is done implicitly by Prolog* while generating the answer fact. In general, the cost of unification is at least linear in the size of the arguments, and the cost of these extra unifications can be significant. As a result, on the append program Prolog* takes $O(n)$ time whereas bottom-up evaluation takes $O(n^2)$ time, if lists of length $O(n)$ are appended. While we can reduce the time required for these unifications in some special cases, we do not know of any general technique for achieving time reductions.

Indexing: Indexing is needed for two purposes. One is to retrieve matching facts for a literal in a rule body. In the full version of this paper, we show how MT-TR can be extended in such a way that we only need to index on fields containing integers for this purpose. The second reason is to find all facts that either subsume or are subsumed by a newly generated fact. There appears to be no easy way to do this in general for non-ground facts. (Recall that hash consing cannot be used with non-ground terms.)

10 Conclusion

Bottom-up methods are comparable to or better than Prolog over a wide range of programs, but they can do considerably worse for some programs that manipulate large non-ground terms. However, bottom-up methods have the virtue of being complete (for positive Horn clause programs), and permit the use of a wide range of additional optimizations that we have not discussed (see, e.g., [NR91]). While some set-oriented/memoing top-down evaluation techniques may share many of the advantages of bottom-up evaluation, they also share the problems described in this paper.

[7] A detailed discussion with examples is presented in the full version of the paper.

Important directions for future work are to develop optimization techniques that extend the range of programs for which bottom-up techniques can be made competitive with Prolog, and to perform more detailed studies that lead to a clearer understanding of the pros and cons of the two approaches.

Acknowledgements

We would like to thank Divesh Srivastava for helpful discussions. The work of both authors was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319.

References

[Ban85] Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems*. Springer-Verlag, 1985.

[BM72] R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Computational Logic*, pages 101–116, 1972.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[BR87a] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[BR87b] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[CD85] J.H. Chang and A. M. Despain. Semi-intelligent backtracking of Prolog based on static data-dependency analysis. In *Proc. Symposium on Logic Programming*, pages 10–21, 1985.

[CKW89] Weidong Chen, Michael Kifer, and Davis S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, 1989.

[Got74] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan, May 1974.

[MW88] David Maier and David S. Warren. *Computing With Logic*. The Benjamin Cummings Publishing Company Inc., 1988.

[NR91] Jeffrey F. Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J-L. Lassez, editor, *Computational Logic: Essays in Honor of Alan Robinson*. 1991.

[Ram88] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[RLK86] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

[Ros91] Kenneth Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[Sek89] H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[SG76] M. Sassa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(4):31–34, June 1976.

[Ull89a] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.