Optimizations of Bottom-Up Evaluation with Non-Ground Terms
(Extended Abstract)

S. Sudarshan
AT&T Bell Labs,
600 Mountain Ave,
Murray Hill, NJ 07974, U.S.A.
sudarsha@research.att.com

Raghu Ramakrishnan
Computer Sciences Department,
Univ. of Wisconsin,
Madison, WI 53706, U.S.A.
raghu@cs.wisc.edu

Abstract

Memoization, such as that performed by bottom-up evaluation, helps detect loops, avoid repeated computation when subgoals are generated repeatedly, and in conjunction with a fair search strategy, ensures that evaluation is complete. Programs that generate non-ground facts (i.e., facts containing universally quantified variables) and also need memoization are important in several contexts. Current bottom-up evaluation techniques (and other memoization techniques), are very inefficient for programs that generate facts containing large non-ground terms.

We present an efficient bottom-up evaluation technique for programs that generate non-ground facts. We show that bottom-up evaluation can be implemented with a time cost that is within a $\log\log$ factor of (a model of) Prolog evaluation, for all queries on definite clause programs; conversely, there are programs where bottom-up evaluation is arbitrarily better than Prolog. These results significantly extend earlier results comparing bottom-up evaluation and top-down evaluation, An implementation of our technique enables us to run some programs faster (asymptotically and practically) than using either Prolog or unoptimized bottom-up evaluation.

1 Introduction

Programs that generate non-ground facts/data-structures (i.e., facts/terms containing universally quantified variables) are of considerable importance, and are widely used in Prolog. A common use is in difference-lists, which are used to append (representations of) lists in constant time. Non-ground data-structures are often used in programs, such programs as for parsing Definite

Clause Grammars (DCGs), that also benefit from memoization of facts. For instance, chart parsing of DCGs is naturally supported using memoization, and can be much more efficient than top-down parsing in some situations. Meta-interpreters, partial evaluators, abstract interpreters and other such programs often operate on data structures that contain variables. Memoing of subgoals and their answers is very important for some of these programs [23].

Bottom-up (i.e., forward chaining) evaluation using (variants of) Magic rewriting [2, 9] is a way of implementing memoization. It is shown in [12] that in the absence of non-ground facts, bottom-up evaluation using a variant of Magic Templates rewriting (MTTR rewriting) [14] is as fast as Prolog up to a data-independent constant factor (assuming that all answers have to be generated, and intelligent backtracking optimization is not used).

However, the above result does not apply to programs that generate non-ground facts. The major problem is that bottom-up evaluation performs some unifications, which we call 'answer-return' unifications, that Prolog does not perform. If not treated specially, these unifications can be costly in the presence of large non-ground terms in facts, and the overhead of bottom-up evaluation is no longer within a constant factor of Prolog. Even if a given program generates only ground answers (to all subgoals), it may generate non-ground subgoals; correspondingly, non-ground query facts would be generated if either Magic Templates or MTTR rewriting is used.[1]

The problems described above (or equivalent ones) also occur with other memoization techniques such as Extension Tables [4], or OLDT resolution [19].

In this extended abstract we present an efficient bottom-up query evaluation mechanism, *Opt-NGBU* query evaluation, for programs that generate non-ground facts. This technique is a combination of an extended rewriting technique, *MGU MTTR rewriting* (Section 3), and an optimized Semi-Naive evaluation technique, *Opt-NG-SN* evaluation, which stands for Optimized NonGround Semi-Naive evaluation (Section 5). The technique maintains extra information with query and answer facts in order to speed up 'answer-return' unifications, and incorporates several other optimizations as well.

We show that given a program and a query, if Prolog evaluation of the query on a database takes time t, then Opt-NGBU query evaluation on the given database, without subsumption-checking, takes time $O(t \cdot \log\log t)$. (The comparison does not take the size of the program into account, but does account for the size of the database. We also assume that all answers to the query are required, and some optimizations such as intelligent backtracking are not used.)

The above result provides an upper bound on how much worse bottom-up evaluation can be compared to Prolog. Conversely, Opt-NGBU evaluation can be much faster than Prolog evaluation if redundant computation

[1] The adornment prepass of Magic Templates rewriting can be used to avoid the generation of non-ground query facts, but carries the cost of computing irrelevant facts.

is avoided through subsumption-checks; the time complexity of evaluation may be significantly better, and infinite loops may be avoided. Checking whether a goal or an answer is already memoed can be expensive, and the cost must be balanced against the cost of recomputation. However, even *without* subsumption-checking, Opt-NGBU evaluation is complete for definite clause programs, while avoiding the repeated computation in iterative deepening (a technique used to make Prolog evaluation complete; see, e.g. [7]).

Our evaluation optimizations have been implemented in the CORAL deductive database system, and we present performance numbers that show the benefits of our optimizations. Our optimizations enable the use of Magic Templates with Right Recursion [14], which creates programs that generate non-ground facts (Section 7).

1.1 A Motivating Example

We now consider an example that illustrates the main issues involved in our optimization techniques.

Example 1.1 A difference list (see e.g. [7]) is a non-ground data-structure that permits the *append* operation to be done in constant time in Prolog.[2] An example of a difference list is $dlist([1,2,3|X],X)$ (the second occurrence of the variable X (logically) gives constant time access to the end of the list). Let us consider what happens if we use difference lists with bottom-up evaluation. Suppose we have the following rules, as part of a larger program:

$$\begin{array}{l} R1: paths(L1,L2,L):-\ dappend(L1,L2,L). \\ R2: dappend(dlist(X,Y),dlist(Y,V),dlist(X,V)). \end{array}$$

Rule $R2$ above specifies how to append two difference lists.

Bottom-up evaluation (without Magic rewriting) essentially performs forward chaining on the set of rules and facts given, and generates all facts that can be inferred using the given facts and the rules in the program. Given a query, it enumerates all answers, but computes facts irrelevant to the query. The Magic Templates transformation [9] rewrites the rules in the program by adding 'filters' corresponding to subqueries, and adds rules to the program that specify how (further) subqueries are generated from a given (sub) query; only facts 'relevant' to the query are generated by the rewritten program. The Magic Templates rewriting of the above program, is shown below:

$$\begin{array}{ll} R1': & paths(L1,L2,L):-\ query(paths(L1,L2,L)), \\ & \qquad\qquad\qquad\qquad dappend(L1,L2,L). \\ QR1.1: & query(dappend(L1,L2,L)):-query(paths(L1,L2,L)) \\ R2': & dappend(dlist(X,Y),dlist(Y,V),dlist(X,V)):- \\ & \quad query(dappend(dlist(X,Y),dlist(Y,V),dlist(X,V))). \end{array}$$

[2] Difference list append has a side effect of modifying the first of the lists to be appended, but the modification is undone on backtracking.

Rule $R1'$ is the same as the original rule, except that an extra literal has been added so that a $paths(\ldots)$ fact is generated only if there is a corresponding $query(paths(\ldots))$ fact. Rule $R2'$ is similarly generated from rule $R2$. Rule $QR1.1$ is generated from the first (and only) literal in the body of rule $R1$, and specifies how a subquery is generated on *dappend* given a query on *paths*.

Given a query $?paths(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), A)$, a fact

$$query(paths(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), A))$$

is added to the program. Bottom-up evaluation of the above rewritten program, with the above *query* fact generates the following facts (the name of the rule used is specified in brackets)

$$query(dappend(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), A))\ [\mathbf{QR1.1}]$$

$$dappend(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), dlist([1,2,3,4,5,6,7,8|Z],Z)))\ [\mathbf{R2'}]$$

$$paths(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), dlist([1,2,3,4,5,6,7,8|Z],Z)))\ [\mathbf{R1'}]$$

We note that when using rule $R2'$, unification binds variable X from the fact $query(dappend(\ldots))$ above to $dlist([6,7,8|Y],Y)$.

The first cause for inefficiency in bottom-up evaluation in the presence of non-ground facts is that we cannot directly use the difference lists stored in the fact

$$query(dappend(dlist([1,2,3,4,5|X],X), dlist([6,7,8|Y],Y), A))$$

when making an inference using rule $R2'$. If we do so, the variable X will get bound to $dlist([6,7,8|Y],Y)$, and the fact cannot then be used to make further derivations. Prolog evaluation performs destructive updates of variable bindings, which it undoes on backtracking; such an approach does not work when facts are memoed since subgoals and their answers could be used in multiple places, and there is no backtracking. One could copy facts before using them, but the copying could increase the time complexity significantly. We use a term-representation based on 'persistent-versioning' to greatly reduce the cost of copying (Section 4).

The second cause for inefficiency is less obvious, and is present with Magic Templates rewriting as well as with its variants (MTTR rewriting [14], and Alexander Templates [16]). The unifications in the first two derivations in the evaluation shown above have corresponding unifications in a Prolog evaluation. But there are no unifications in a Prolog evaluation corresponding to the unifications in the third derivation — the derivation corresponds to a step in the Prolog evaluation where a rule invocation succeeds and returns to the point where the subgoal was generated; no unification occurs at this stage in a Prolog evaluation. We call a unification step where an answer to a subgoal is unified with a rule body as an "*answer-return unification*" step.

Consider the answer-return unification in the evaluation above. In the answer-return unification performed using rule $R1'$ above, there are two facts that contain (potentially) large difference lists. The two lists corresponding to $L1$ (as well as the two lists corresponding to $L2$) are unified in the course of the derivation. The unifications (as well as the renaming done prior to unification) have a time cost linear in the size of the lists. On the other hand, if we use Prolog, this unification step is absent, and assuming that occur checks are not performed, the entire cost of answering a query on *dappend* using Prolog is a just a constant. As another example, if we used the well-known list append program with non-ground lists, the cost of appending two non-ground lists of length n using unoptimized bottom-up evaluation would be $O(n^2)$ as opposed to $O(n)$ using Prolog.

In the case of ground facts, there are effective term representation techniques that can reduce the cost of the extra unifications to $O(1)$ (see [12]). In Section 5 we discuss how to tackle the above problem in the case where facts may be non-ground. □

2 Basics

We assume some familiarity with bottom-up evaluation, and refer the reader to [1] for a survey of the area. We also assume some familiarity with Semi-Naive evaluation and Supplementary Magic Templates rewriting [2, 9]. To make our discussion and analysis simpler, we assume that all non-equality literals in rules of the program have as arguments only distinct free variables. This can be achieved by a straightforward preprocessing transformation, without any increase in the time complexity of either Prolog evaluation or bottom-up evaluation.[3] We assume that equality is a base predicate with a single fact "$=(X, X)$".

3 MGU Magic and MGU MTTR Rewriting

A full description of MGU Magic and MGU MTTR rewriting may be found in [17], but we provide an intuitive description here.

We described the intuition behind Magic Templates rewriting in Example 1.1. As another example, the Magic Templates rewriting of the rule $R1: \ p(X) :- q(X), r(X)$ is

$$\begin{array}{ll} query(q(X)) & :-\ query(p(X)). \\ query(r(X)) & :-\ query(p(X)), q(X). \\ p(X) & :-\ query(p(X)), q(X), r(X). \end{array}$$

The first rule defines what query is generated on q given a query on p. The second rule defines what query is generated on r (the second literal in the rule body) given a query on p and an answer fact for q.

Supplementary Magic Templates [2, 9] can be viewed as a way of eliminating the common subexpressions in the above rules. The Supplementary Magic Templates rewriting of rule $R1$ is as follows:

$$\begin{array}{ll} sup_{1,0}(X) & :-\ query(p(X)). \\ query(q(X)) & :-\ sup_{1,0}(X). \end{array}$$

[3] We assume that the size of the program is fixed. Thus, although the above transformation can defeat rule indexing techniques used by Prolog, the loss of speed is by at most a constant factor.

$$
\begin{array}{ll}
sup_{1,1}(X) & :-\ sup_{1,0}(X), q(X). \\
query(r(X)) & :-\ sup_{1,1}(X). \\
p(X) & :-\ sup_{1,1}(X), r(X).
\end{array}
$$

But Supplementary Magic Templates rewriting actually has a deeper significance than just common subexpression elimination. Consider a Prolog evaluation of the original rule. At any stage in the evaluation when control is at the rule, the rule variables have some bindings. The variable bindings stored in tuples for $sup_{1,0}$ correspond to rule variable bindings at points in a Prolog evaluation the rule is initially invoked and the query has been unified with the rule head. The variable bindings in tuples for $sup_{1,1}$ correspond to rule variable bindings at points where an answer has just been returned for the first literal ($q(X)$) in the rule body. In general the bindings in tuples for $sup_{i,j}$ correspond to rule variable bindings at points where an answer has just been returned for the jth literal in the rule.

Our rewriting technique, MGU Magic rewriting, may be viewed as an extension of Supplementary Magic Templates rewriting to introduce a *goal-identifier* field into query, answer and supplementary facts. A goal-identifier is an integer identifier that uniquely identifies a subgoal. The goal-identifiers help avoid certain unnecessary inferences made using subsumed facts (that, in particular, are not made by Prolog evaluation) (see [17]), and they provide an efficient mechanism for indexing facts similar to that used in QSQR [21].

Goal-identifiers are generated using a 'meta-predicate' $\textsf{goal\_id}(goal, id)$. If $\textsf{goal\_id}$ is used with subsumption checking, subgoals are mapped one-to-one to identifiers (modulo renaming) by calls on $\textsf{goal\_id}(goal, id)$. If subsumption checking is not used, calls on $\textsf{goal\_id}(goal, id)$ return a new identifier on each call.

The evaluation of an MGU Magic rewritten program generates facts of the form $query(p(\vec{t}), id)$. Such a fact denotes that there is a subgoal $?p(\vec{t})$, and id is a goal-identifier of $?p(\vec{t})$ (the symbol p appears both as a predicate and a functor, but the notation is not higher order). The MGU Magic rewritten program also generates facts of the form $answer(id, p(\vec{a}))$. Such a fact says that $p(\vec{a})$ is an answer to a subgoal on p that has a goal-identifier id. (There is an extra rule in the rewritten program to generate answers of the form $q(\vec{a})$ for the initial query $?q(\vec{t})$ on the program.)

Example 3.1 We use the program from Example 1.1 as our running example, and illustrate MGU Magic rewriting using it.

The preprocessed form of the program is as follows:

$$
\begin{array}{ll}
paths(L1, L2, L) & :-\ dappend(L1, L2, L).. \\
dappend(V1, V2, V3) & :-\ V1 = dlist(X, Y), V2 = dlist(Y, V), \\
 & V3 = dlist(X, V).
\end{array}
$$

The MGU Magic rewriting of the preprocessed program is shown below (we have left out the rules generating $query$ facts from the initial query on the program, and we have unfolded several rules in the rewritten program to get rule $R4$, in order to keep the presentation concise). In the rewritten program below, rules $R1$, $R2$, and $R3$ are generated from the first rule of the preprocessed program, and rule $R4$ is generated from the second rule of the preprocessed program.

$$
\begin{aligned}
R1 : & \ sup_{1,0}(HId, L1, L2, L, ID) :- query(paths(L1, L2, L), HId), \\
& \qquad \mathsf{goal\_id}(dappend(L1, L2, L), ID). \\
R2 : & \ query(dappend(L1, L2, L), ID) :- sup_{1,0}(HId, L1, L2, L, ID). \\
R3 : & \ answer(HId, paths(L1, L2, L)) :- sup_{1,0}(HId, L1, L2, L, ID), \\
& \qquad answer(ID, dappend(L1, L2, L)). \\
R4 : & \ answer(HId, dappend(V1, V2, V3)) :- \\
& \qquad query(dappend(V1, V2, V3), HId), V1 = dlist(X, Y), \\
& \qquad V2 = dlist(Y, V), V3 = dlist(X, V).
\end{aligned}
$$

We can also add rules to the program to select out answers to the initial query on the program. The rules are straightforward, and to keep the discussion simple, we omit them. □

A fact of the form $sup_{i,j}(hid, \vec{v}, id1)$ represents an instance of R_i such that R_i is being used to solve a query with identifier hid, $\vec{v}$ stores the bindings of rule variables after a successful solution up to the jth literal, and $id1$ is the identifier of the query generated on the $j+1$th literal. MGU Magic rewriting generates programs where each rule has at most two literals in the body.

MGU MTTR rewriting extends the Magic Templates with Right Recursion rewriting of Ross [14], in exactly the same fashion as MGU Magic rewriting extends Supplementary Magic Templates rewriting. Details may be found in [17].

Example 3.2 We now show (a high level view of) the bottom-up evaluation of the rewritten program generated in Example 3.1 from our running example. Suppose we are given a query

$?paths(dlist([a, b|X], X), dlist([c|Y], Y), P)$

and suppose that its goal-id is 0. A 'seed' query fact

$query(paths(dlist([a, b|X], X), dlist([c|Y], Y), P), 0)$

is added to the rewritten program before it is evaluated bottom-up. Suppose also that the goal-id of

$?dappend(dlist([a, b|X], X), dlist([c|Y], Y), P)$

is 1. Then the bottom-up evaluation of the MGU Magic rewritten program generates the following sequence of facts:

$sup_{1,0}(0, dlist([a, b|X], X), dlist([c|Y], Y), P, 1)$, using rule $R1$,
$query(dappend(dlist([a, b|X], X), dlist([c|Y], Y), P), 1)$, using rule $R2$,
$answer(1, dappend(dlist([a, b, c|Y], [c|Y]), dlist([c|Y], Y),$
$\qquad dlist([a, b, c|Y], Y)))$, using rule $R4$, and
$answer(0, paths(dappend(dlist([a, b, c|Y], [c|Y]), dlist([c|Y], Y),$
$\qquad dlist([a, b, c|Y], Y))))$, using rule $R3$. □

4 Representation of Terms and Facts

A *binding environment (bindenv)* stores bindings for variables. A variable in a bindenv b may be free, or may be bound to a structure s' (which is possibly an atomic value). Variables within s' are also interpreted in b.[4] We represent a fact as a pair $\langle structure, bindenv\rangle$. The following is a representation of the fact $g(f(4,4),X)$:

$$\langle g(W,Y), \{Y \rightarrow X, Z \rightarrow 4, W \rightarrow f(Z,Z)\}\rangle$$

Given a fact f, $f.structure$ denotes the structure of f, and $f.bindenv$ denotes the bindenv of f. Thus $f = \langle f.structure, f.bindenv\rangle$. We use the notation $\langle s, e\rangle$, where s is a term, to denote s interpreted in bindenv e. We say that terms $\langle s1, e1\rangle \equiv \langle s2, e2\rangle$ if both represent exactly the same term. During evaluation, the same fact may be generated with different representations. We call each such representation of a fact an *occurrence* of the fact.

Binding environments are implemented using "fully persistent versions of data structures" [6, 5]. When applied to bindenvs represented as arrays, a fully persistent versioning scheme permits us to carry out the following operations efficiently:

1. Create a new child version of an existing bindenv (which itself may have been created as a child version of another bindenv, and so on). The child version has the same bindings as the parent version when it is created, but any changes made to the child version will not affect the parent version.
2. Add a new variable to a version of a bindenv.
3. Lookup and/or change the binding of a variable in a version of a bindenv.

Operation (1) can be done in constant time, and operations (2) and (3) can be done in time $O(min(\log\log m, \log n))$, where m is the total number of versions of bindenvs that have been created and n is the number of versions of the variable that have been modified [5]. For brevity, we use the notation $\mathcal{V}$ (defined below).

Definition 4.1 ($\mathcal{V}$) Consider an evaluation of a program. Let $\{V_1, V_2, \ldots\}$ be the variables used in the evaluation. Let n_i denote the number of versions of V_i that are modified in the evaluation, and let m denote the total number of versions of bindenvs that are created in the evaluation. Then $\mathcal{V}$ denotes $\max_i(\min(\log\log m, \log n_i))$. $\square$

We have implemented a simpler scheme due to D.H.D. Warren ([22], cited in [8]) which has an access cost of $\log n$ (where n is the number of variables in the bindenv) in the CORAL deductive database system [11]. The scheme was proposed for implementing OR-parallel Prolog.

[4] We do not allow bindings in bindenvs of stored facts to refer to other bindenvs, since we do not know how to create versions of facts efficiently using such a representation.

5 How to Apply a Rule

The basic operation in bottom-up evaluation is the application of a rule to produce new facts. In this section we present an algorithm to apply a rule, with several optimizations to handle non-ground facts more efficiently. The rule application procedure described here is used with Semi-Naive evaluation. We assume that the rules to be evaluated are those generated by MGU Magic or MGU MTTR rewriting.

Procedure Apply_Rule is shown below. It essentially performs a left to right nested loops join.[5] Note that due to our rewriting, rules have either one or two body literals. We describe informally some of the procedures that it uses; details are presented in [17]. An important point to note in Apply_Rule is the creation of versions of bindenvs to ensure that unification operations do not affect any stored facts.

Procedure ApplyRule(R).
Let R be: $p(\vec{t})\colon -q1(\vec{t_1})[, q2(\vec{t_2})]$. /* [ ] denotes an optional argument */
1. Fetch facts for $q1$.
   For each fetched fact $\langle str1, env1\rangle$ do the following:
   1.1 Create a new version $env1'$ of $env1$. Unify $\langle str1, env1'\rangle$ with $q1(\vec{t_1})$.
   1.2. If the unification succeeds, Then
      1.2.1. If $q2$ is not the goal_id predicate, Then
         a. Fetch $q2$ facts that unify with the instantiated $q2(\vec{t_2})$.
         b. For each fetched fact $\langle str2, env2\rangle$
               Execute Smart_Unify $(R, \langle str1, env1\rangle, \langle str2, env2\rangle, \langle R', r\_env'\rangle)$.
               If Smart_Unify succeeds, Insert_Head_Fact($\langle R', r\_env'\rangle$)
      1.2.2. Else
            Rename_and_Reunify($R, \langle str1, env1\rangle, \langle R', r\_env'\rangle$)
            If $q2$ is the goal_id predicate, evaluate it.
            /* Else the rule has only one literal */
            Insert_Head_Fact($\langle R', r\_env'\rangle$).
end Apply_Rule.

Consider the case of rules with two body literals (with neither having **goal_id** as predicate). Due to our rewriting, such rules must have a supplementary literal and an answer literal in the body. For such rules, Apply_Rule fetches facts for the two literals, and calls Smart_Unify with the two facts. Smart_Unify unifies versions of the two fetched facts with their respective body literals, and returns as its last argument a (possibly renamed) instantiated version R' of the rule R, with a bindenv r_env'. We omit details of Smart_Unify for lack of space, but give some intuition below.

[5] Our Semi-Naive rewriting ensures that the left-most literal in the join is either a δ literal or a base literal.

Smart_Unify calls a procedure Return_Unify to handle rules that involve answer-return unification; we discuss this procedure in more detail later. For rule applications that do not involve answer-return unification, or if the optimizations used in Return_Unify do not apply, Smart_Unify ensures that variables in the query/supplementary fact are not renamed; variables in the rule and in the answer fact are renamed instead, and the bindenv of the head fact is a version child of the bindenv of the query/supplementary fact. Thereby, bindenvs are inherited in such a manner that the bindenv of an answer fact is a descendant of the bindenv of the query fact that resulted in its generation.

For all other rules, procedure Rename_and_Reunify is called. It renames the rule, unifies it with a version of the fact for the first body literal, and returns the result as its last argument. In either case, Insert_Head_Fact inserts the derived fact into the appropriate relation, after performing subsumption-checking if required; again, we omit details.

5.1 Context Identifiers and Return-Unification

We now describe our optimization of answer-return unification. The goal is to reduce the cost of answer-return unifications by keeping extra information with facts. We start by describing the intuition behind the optimization, then present some details, and finally present an example.

Our evaluation algorithm propagates (versions of) bindenvs of facts, as well as parts of their structure, through derivations. The idea is as follows. Consider a subgoal $\langle q.structure, q.bindenv \rangle$. The subgoal is used in rules, to generate further subgoals and answers. As descendant versions of its bindenv are created, and used in rules, variables corresponding to those in the subgoal get bound. Finally if an answer fact to the subgoal q gets generated using a descendant bindenv env' of $q.bindenv$, we can show that the variables that occur in q have been bound in env' in much the same way that variables in a subgoal are bound during Prolog evaluation (our rule evaluation mechanism ensures it by the way in which bindenvs are inherited, and fact structures created). This knowledge can then be used to optimize answer-return unification. We discuss the optimized unification step in more detail shortly.

The difference from Prolog is that while Prolog solves each occurrence of a subgoal separately, bottom-up evaluation may eliminate duplicate/subsumed facts and hence some answers that were generated for one subgoal occurrence may have to be used where other occurrences of the same subgoal were generated. (If a given query is generated from more than one supplementary fact, all but one occurrence of the query may be deleted by duplicate elimination.) But in such a case the structure and bindenv of an answer may not have been inherited from a subgoal for which the answer is used, and one cannot optimize answer return unification using knowledge about the bindenv.

The discussion above was in terms of answers to subgoals. However, in bottom-up evaluation using MGU Magic rewriting, answer return unification actually occurs when a supplementary fact and an answer fact are unified with a rule. The supplementary fact stores rule variable bindings from the point when the query was generated, and the query itself is generated using the supplementary fact.

To find if the optimized answer-return unification step is applicable, we need to keep track of what fact occurrences are generated from what other fact occurrences, and how the bindenvs are inherited. In order to do so, with each supplementary fact we store a field *cont_id* ("context identifier"). With all facts, we store a field *par_id* ("parent context identifier"). These two fields hold identifiers that help keep track of how bindenvs were inherited from a supplementary fact; the idea is that if the *cont_id* field of a supplementary fact matches the *par_id* field of an answer fact, answer-return unification optimization is applicable. The *cont_id* and the *par_id* values do not affect subsumption checking.

We now present an intuitive description of how the *cont_id* and *par_id* values are generated and propagated. We generate a new *cont_id* for each supplementary fact. The query fact generated from the supplementary fact inherits a version of the bindenv of the supplementary fact, as well the *cont_id* value of the supplementary fact in its *par_id* field. (In case last-call optimization is used on a literal, the *par_id* field of the query fact is set to the *par_id* field of the supplementary fact since answers will not be generated for the query fact.) A supplementary fact is generated either from a query fact occurrence or from a supplementary fact for the previous body literal. In either case, it inherits both a version of the bindenv and the value of the *par_id* field from the fact it is generated from. An answer fact is generated from a supplementary fact, and inherits a version of its bindenv, and its *par_id* value. Thus, versions of bindenvs are propagated through derivations, and the *cont_id* and *par_id* fields are used to keep track of links between the bindenvs of supplementary facts and the bindenvs of answer facts. We say that an answer fact is *generated from* a supplementary fact if the *cont_id* value of the supplementary fact is the same as the *par_id* value of the answer fact.

Now we get back to how answer-return unification is actually achieved. The details are shown in Procedure Return_Unify, which first tests for the applicability of the optimization, and returns failure if it is not applicable. The intuition is as follows. If (a descendant of) a query fact bindenv has been inherited by the answer fact and variables in it bound appropriately at each step in the chain of inheritance, replacing the bindenv of the query fact by the bindenv of the answer fact has the same effect as applying the (most general) unifier of the query and the answer facts to the query. In the actual derivation we use the supplementary fact instead of the query fact (the query fact bindenv is a version of the supplementary fact bindenv). Return_Unify need only replace the supplementary fact bindenv by the answer fact bindenv to carry out unification. However, we show a more general version that works even when the adornment optimization of Magic rewriting [2] has been used.

Procedure Return_Unify $(R, s, a, \langle R', r\_env' \rangle)$

/* R is a rule, s is a supplementary fact, and a an answer fact.*/

1. If $s.cont\_id \neq a.par\_id$ Then return failure.
2. Set r_env' = new version of $a.bindenv$.
3. Let R' be a renamed version of R with variable names starting from after the highest numbered variable in r_env'. Add all variables in R' to r_env'.
4. Bind each variable in the supplementary literal of $\langle R', r\_env' \rangle$ to the corresponding argument of $s.structure$.[6]
5. Bind each variable in the answer literal of $\langle R', r\_env' \rangle$ to the corresponding argument of $a.structure$.[7]
6. Update_Context_Ids(R', r_env', s).
7. Return success.

end Return_Unify

We show [17] that the unifier computed by Return_Unify is a most general unifier of the supplementary and answer facts with the rule body. Most importantly, it executes in $O(\mathcal{V})$ time, which is quite small.

5.2 An Example

Example 5.1 We use the bottom-up evaluation shown in Example 3.2 to illustrate the effect of our optimization technique. Figure 1 shows physical details of the evaluation of the rewritten program, described at a high level in Example 3.2. We use the following notation. The *par_id* of each fact is shown following the fact, and for supplementary facts, the *cont_id* is shown following the *par_id*.

The figure should be read as a sequence of derivations, from top to bottom. The bindenvs are shown as tables, and under each variable we either have a blank (the variable is not bound), a value for its binding, or a pointer to its binding. We use the pointers to emphasize that structures are shared between facts used in a derivation and the derived fact, and are not copied unnecessarily. In the figure, several facts point to one bindenv — this notation should be interpreted as each fact having its own version of the bindenv (but with the same bindings), and is done only to keep the figure concise.

The main points to note are the following. The initial query fact is assumed to have a *par_id* value of 0. When using rules $R1$, variables $L1$, $L2$ and L are added to the bindenv, with $L1$ bound to the first dlist, $L2$ to the second, and L to P. The derived fact is given a new *cont_id* value 1,

[6] Each argument of the supplementary literal is a distinct variable. Hence the concept of having for each variable in the supplementary literal a "corresponding argument" in the fact (Step 4 of Return_Unify) is well-defined.

[7] The concept of "corresponding argument" is well-defined for Step 5. Such a literal is of the form $answer(ID, q(\vec{X}))$, where $\vec{X}$ is a tuple of distinct variables, due to the preprocessing. All facts used with the literal are of the form $answer(id, q(\vec{a}))$. The arguments "corresponding" to the variables in $\vec{X}$ are the arguments of $q(\vec{a})$ in the above fact.

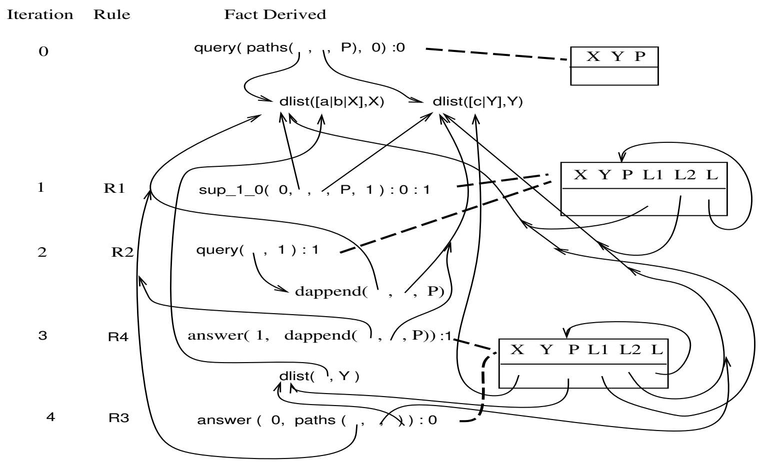

Figure 1: Evaluation of Program That Uses *dappend*

and the bindenv of the *query* fact is inherited by the derived fact. When using rule $R2$ the variables in the query are not bound further, the bindenv is inherited by the derived fact, and the derived fact has a *par_id* of 1. As an optimization, we project out variables from the bindenv that do not appear (directly or indirectly) in the derived fact, and hence the bindenv is not changed further. Next, when using rule $R4$, the variables L and P in the bindenv get bound to the result of *dappend*ing the two lists. It is important to note that this corresponds to the variable P in the initial query getting bound in Prolog evaluation of the query. (The variable X also gets bound in this step, as it does in Prolog evaluation.) The derived answer fact inherits the *par_id* of 1 from the supplementary fact used in its derivation, and it inherits the bindenv from the *query* fact used in the derivation.

Due to the above bindings, the bindenv of the answer fact generated using $R4$ is such that if the supplementary fact generated by $R1$ is interpreted in the bindenv, the variable P is bound to the result of *dappend*ing the two given lists (and X is bound correctly as well). Hence, when using the derived fact in rule $R3$, return unification should be applicable, and this is indeed the case – the *par_id* of the answer fact is equal to the *cont_id* of the supplementary fact (both are 1). Since Return_Unify succeeds, no renaming is required, and unification takes $O(\mathcal{V})$ time.

Overall, the time cost of the evaluation shown is $O(\mathcal{V})$, regardless of the sizes of the difference lists, ignoring the cost of setting up the initial query and printing the answer. (As in Prolog, no occur checks are performed.) The answer-return unification step would take time proportional to the sizes of the difference lists, if our optimizations were not used. $\Box$

5.3 Correctness and Cost of Apply_Rule

We call a version of Semi-Naive evaluation that uses procedure Apply_Rule to perform rule application as *Opt-NG-SN evaluation.* We call the query evaluation technique that first rewrites the program and query using MGU MTTR rewriting, and then evaluates it using Opt-NG-SN evaluation as *Opt-NGBU evaluation.*

Theorem 5.1 *Let P be a program and Q a query on the program. Let P^{MGU_T} be the program generated from P and Q by either MGU Magic or MGU MTTR rewriting. Then Opt-NG-SN evaluation of P^{MGU_T} is such that (1) Every fact generated as an answer for Q is an answer to Q, and (2) Every answer to Q is subsumed by the set of answers generated.* □

We index supplementary and answer facts using hash-indices on the *goal-id* fields. This technique is essentially the same as the one used in QSQR [21], and provides constant time insertion, and constant time lookup per retrieved fact. Occur checks are not necessary for soundness in Return_Unify, since the rule literals have distinct variables that are not present in the facts. It is straightforward to show that Return_Unify runs in $O(\mathcal{V})$ time. Further, we show (in [17]) that in the absence of subsumption checking, every call to Return_Unify succeeds. Hence answer-return unification can always be done efficiently.

In the general case, subsumption-checking is a costly operation, and we are not aware of efficient subsumption-checking techniques for the case of arbitrary non-ground facts. For ground facts, subsumption is the same as equality, and hash-consing [15] can be used to perform equality checking in constant time in many cases. Approximate forms of subsumption checking can often be done efficiently, and often suffice in practise.

Checking for subsumption avoids recomputation, and can prevent the computation from entering into an infinite loop. But if subsumption checking is done on goals, answer-return unification optimization will apply to only one use of each answer fact. The cost of checking for subsumption, and of renaming and unifying answer facts when answer-return unification optimization fails, has to be carefully balanced against the benefits of avoided recomputation.

6 A Comparison With Prolog*

For our comparison of Prolog with bottom-up evaluation, we use a model of Prolog evaluation that incorporates last-call optimization. The model is quite straightforward, and corresponds closely to the intuitive 'procedural' model of Prolog evaluation, augmented with last-call optimization. We call the model of evaluation as *Prolog** evaluation, The detailed model may be found in [17].

We make the following simplifying assumption: Given term occurrences a, $a1$ and b, if $a \equiv a1$, (i.e., they represent the same term) then the time taken to unify a and b is the same as the time taken to unify $a1$ and b. We also assume that bottom-up evaluation as well as Prolog* evaluation use the same indexing technique for base relations.

Operation	Bot. Up (No Opt.)	Prolog	Opt-NGBU
Unification			
Answer-return	O(size of terms)	$O(1)$	$O(\mathcal{V})$
Other	O(size of terms)	O(size of terms)	$O(\mathcal{V}\cdot$ size of terms$)$
Indexing			
Answer-return	$O(\sum_{f_i\in\mathcal{F}} size(f_i))$	$O(1)$	$O(1)$
Other	$O(\sum_{f_i\in\mathcal{F}} size(f_i))$	$O(\sum_{f_i\in\mathcal{F}} size(f_i))$	$O(\sum_{f_i\in\mathcal{F}} size(f_i))$

Table 1: Bottom-Up Evaluation using MGU MTTR rewriting vs. Prolog

Table 1 summarizes a comparison between various costs in bottom-up evaluation of an MGU MTTR rewritten program and Prolog evaluation. In the table, $size(f_i)$ denotes the size of f_i, and $\mathcal{F}$ denotes the set of all facts that are derived.

Theorem 6.1 *Let P be a program, and Q a query. Given any database, suppose the cost of Prolog* evaluation of Q is t units of time.*[8] *Opt-NGBU evaluation without subsumption-checking evaluates the query on the given database in time $O(t \cdot \log\log t)$. (The size of the program is not taken into account in this time complexity measure.)* $\Box$

The proof of this theorem is presented in [17], where we also discuss how we can relax the assumption that the size of the program is a constant. We remind the reader that our analysis ignores constant costs, and the effect of factors such as virtual memory, and assumes that all answers are generated, and no intelligent backtracking is used.

Although bottom-up evaluation may be a bit slower if no subsumption-checking is done, it still has the benefit of being sound and complete unlike Prolog, and does not repeat computation in the manner of iterative deepening. The biggest benefit of our optimizations is for programs that generate non-ground facts, but where some of the facts generated are ground, and subsumption checking is both necessary and cheap for these facts. We present one such program in Example 7.2. For such programs Opt-NGBU combines the best features of Prolog evaluation and bottom-up evaluation.

The question of how bottom-up and top-down methods compare is considered important, and has been under investigation by several researchers [20, 3, 9, 16]. Our result carries the comparison of top-down and bottom-up methods farther than earlier results in three important ways: (a) it extends the class of programs considered from safe Datalog to full logic programs, (b) it compares bottom-up evaluation with a sophisticated model of Prolog evaluation, which incorporates last-call optimization, and (c) it takes all time costs into account (earlier results with the exception of [20] ignored the cost of unification, and only compared the number of operations such as inferences performed). Since we remove all these restrictions, we believe our work represents a major advance on earlier work.

[8] Where each action of Prolog* evaluation takes at least unit time.

7 Discussion

The evaluation technique we described can be extended and optimized in several different ways. In the case of range-restricted programs, where no non-ground facts are generated, bindenvs need not be stored explicitly, and $\mathcal{V}$ reduces to $O(1)$. We have implemented our optimization techniques (except for MGU MTTR rewriting) on the CORAL deductive database system [11], and we present some preliminary performance figures.

Example 7.1 Consider the well-known program to append lists, with a query involving non-ground lists. The following table presents the relative time costs of three evaluation techniques, on lists of the specified lengths. The number of distinct variables in the list is shown in parentheses.

Dataset	Unopt.	MGU Magic + Opt. NGBU	MGU MTTR + Opt. NGBU
Length 25 (3 vars)	.31	.19	.08
Length 50 (3 vars)	0.98	.35	.15
Length 100 (3 vars)	3.85	.67	.30
Length 100 (25 vars)	3.87	.69	.30
Length 100 (ground)	.44	.55	.30

Even for ground lists, optimized evaluation with MGU Magic rewriting is not much slower than unoptimized evaluation, while optimized evaluation using MGU MTTR rewriting (which generates non-ground facts) is actually the fastest of the three. For non-ground lists, the cost of optimized evaluation grows linearly with the size of the lists, while for evaluation without our optimizations the cost grows roughly quadratically. □

Example 7.2 We ran two variants of a shortest path program [18] on the CORAL system. Prolog evaluation is inapplicable since it loops if there are cyclic paths. The shortest path program is best evaluated bottom-up (see [18]), and subsumption checking on subgoals is, in general, necessary for termination. Since subgoals for this program are ground, subsumption checking can be performed efficiently. For lack of space, we omit details of the program — details may be found in [17]. Both variants of the program used the query $?shortest\_path(X, Y)$. The first used a difference list representation, and the second used an ordinary list representation, but used *cons* rather than *append*. The second variation generated only ground facts, but generated path lists in reverse order (generating them in the correct order would be costly since *append* takes time linear in the length of its first argument). The ground program ran in 0.6 seconds on a sample dataset, while the non-ground program ran in 0.8 seconds. Thus the loss of speed due to the non-ground data-structure is reasonably small (33%), while providing the benefit of printing out paths in the correct order. □

7.1 Related Work

There have been several studies ([9, 16, 20, 3, 12]) that have compared bottom-up and top-down evaluation in terms of the number of facts computed, number of inferences made, and time taken. Section 6 discussed how our results subsumes the earlier ones. Pereira [8] describes an implementation of parsers for unification based grammar formalisms, using "virtual copy memory" (i.e., versioned memory). There seems to be no equivalent to answer-return unification in the context of [8].

The optimizations described in this paper work at the level of rule application, and are essentially independent of the control strategy used during evaluation such as those described in [10, 18]).[9] They can be applied to other memoing evaluation schemes such as QSQR [21] and Alexander [13, 16]. Persistent versioning can be used with Extension Tables [4], or OLDT resolution [19]. Our optimization of answer-return unification is not useful in the context of Extension Tables. However, with Extension Tables variables in rules would have to be versioned. In Opt-NGBU evaluation, we can avoid versioning rule variables in most cases, and for programs that (with adornment) generate only ground facts and queries, all bindenvs are empty, and have no versioning costs.

OR-parallel Prolog implementations share some of the problems bottom-up evaluation faces. However, the problems of subsumption checking and answer-return unification are not present in OR-parallel Prolog. Further, optimizations that try to avoid using bindenvs in facts are important for bottom-up evaluation although not for OR-parallel Prolog.

8 Conclusion

The results in this paper are significant in two ways. First, they provide an efficient memoization technique for definite clause programs that generate non-ground facts. We believe the techniques can be extended to other interesting domains such as bottom-up evaluation of constraint programs. Second, they extend our understanding of the similarities between top-down and bottom-up further than previous results, which considered only programs that generated only ground facts.

Acknowledgements

This work was done while the first author was at the University of Wisconsin, Madison. This work was supported by a David and Lucile Packard Foundation Fellowship, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

References

[1] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 16–52, May 1986.

[2] C. Beeri and R. Ramakrishnan. On the power of Magic. In *Procs. of the ACM Symp. on Principles of Database Systems*, pages 269–283, Mar. 1987.

[9] Some of these techniques modify Magic rewriting in minor ways. Corresponding changes may need to be made in our optimization technique.

[3] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.

[4] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *Procs. of the Symposium on Logic Programming*, pages 264–272, 1987.

[5] P. F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, pages 67–74, 1989. (Appeared as LNCS 382).

[6] J. R. Driscoll, N. Sarnak, D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Eighteenth Annual ACM Symp. on Theory of Computing*, 1986.

[7] R. A. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[8] F. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *Procs. of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 137–143, 1985.

[9] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Procs. of the International Conference on Logic Programming*, pages 140–159, August 1988.

[10] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Joint Int'l Conf. and Symp. on Logic Programming*, 1992.

[11] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Procs. of the Int'l Conf. on Very Large Databases*, 1992.

[12] R. Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Procs. of the International Logic Programming Symposium*, 1991.

[13] 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.

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

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

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

[17] S. Sudarshan. *Optimizing Bottom-Up Evaluation for Deductive Databases.* PhD thesis, University of Wisconsin, Madison, Aug. 1992.

[18] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Procs. of the Int'l Conf. on Very Large Databases*, Sept. 1991.

[19] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Procs. of the Third International Conference on Logic Programming*, pages 84–98, 1986. (Lecture Notes in Computer Science 225, Springer-Verlag).

[20] J. D. Ullman. Bottom-up beats top-down for Datalog. In *Procs. of the Eighth ACM Symp. on Principles of Database Systems*, pages 140–149, March 1989.

[21] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, pages 1–53, 1989.

[22] D. H. D. Warren. Logarithmic access arrays for prolog. Unpublished program, 1983.

[23] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3), Mar. 1992.