Efficient Goal Directed Bottom-up Evaluation of Logic Programs*

Michael Codish
Department of Mathematics and Computer Science
Ben Gurion University of the Negev
84105 Beer Sheva, Israel
codish@cs.bgu.ac.il

Abstract

This paper introduces a new strategy for the efficient goal directed bottom-up evaluation of logic programs. Instead of combining a standard bottom-up evaluation strategy with a Magic-set transformation, the evaluation strategy is specialized for the application to Magic-set programs which are characterized by clause bodies with a high degree of overlapping. The approach is similar to other techniques which avoid re-computation by maintaining and reusing partial solutions to clause bodies. However, the overhead is considerably reduced as these are maintained implicitly by the underlying Prolog implementation. The technique is presented as a simple meta-interpreter for goal directed bottom-up evaluation. No Magic-set transformation is involved as the dependencies between calls and answers are expressed directly within the interpreter. The proposed technique has been implemented and shown to provide substantial speed-ups in applications of semantic based program analysis based on bottom-up evaluation.

1 Introduction

Bottom-up evaluation of logic programs has been proposed for a variety of application areas as an alternative to Prolog's top-down evaluation strategy. Bottom-up computing lies also at the heart of deductive databases. The basic bottom-up scheme involves a query-program transformation termed Magic-sets [2] (and by now a class of algorithms: Generalized Magic-sets [4], Magic Templates [21], Alexander Templates [25]). Using this technique, a program-query pair is transformed into a magic program whose bottom-up fixed point evaluation is devised to simulate top-down evaluation of the original program and query. The idea originates from retrieval languages for relational databases [27]

*This work was supported by a grant from the Israel Science Foundation

and has been shown advantageous also in the context of semantics based program analysis [6, 3, 14, 18] and as a means to perform tabulation [28] much the same as in XSB [23].

In this paper we present a new implementation strategy for bottom-up evaluation of magic transformed programs which we term *Induced magic*. The novel idea is to optimize the general fixed point evaluation algorithm for the special case when it is applied to programs generated by the Magic-set transformation. We take advantage of the specific structure of these programs to provide an efficient evaluation strategy. Once this is done, it becomes apparent and straightforward to bypass the Magic-set transformation stage altogether and to apply a goal directed evaluation induced by the Magic-set transformation to the original given logic program. The advantage of using the Induced magic approach is that the prefixes in magic clauses are not re-solved — as intermediate results are kept available locally in the runtime environment. Moreover, there is no additional overhead to maintain these results as they are not remembered globally as for example when using Supplementary magic predicates [4, 21]. On the other hand, because we do not remember intermediate results globally, we do have to recompute them between iterations of evaluation and we do not benefit from the reduction of complexity sometimes associated with the use of Supplementary magic (for certain types of programs). The proposed technique can be applied in combination with most of the known optimization techniques for bottom-up evaluation.

The rest of this paper is organized as follows: Section 2 surveys briefly the essence of bottom-up evaluation of logic programs. We illustrate the approach using a simple Prolog interpreter for naive bottom-up evaluation together with a simplified version of the Magic-sets transformation. Section 3 focuses on the overlapping of clause bodies introduced by the Magic-sets transformation — a potential source of inefficiency for bottom-up evaluation. We describe the use of Supplementary Magic-sets to avoid this and present Induced Magic-sets as an alternative which does not incur the overhead in maintaining supplementary predicates. Once again the approach is illustrated by a simple Prolog interpreter. Section 4 evaluates our proposed technique and finally Section 5 presents a conclusion. In the appendix we present simple Prolog interpreters for semi-naive and eager evaluation and their counterparts using Induced Magic-sets. Optimized versionf of these interpreters have been used in our experimental evaluation.

In the following we assume a familiarity with the standard definitions and notation for logic programs as described in [16, 1]. For a survey of results and techniques for deductive databases we refer the reader to [22, 19].

2 Bottom-up Evaluation with Magic-Sets

The fundamental operation in bottom-up approaches is the application of a rule to a set of facts to generate new facts. The essence of this approach originates in the evaluation of the least fixed point of the classic T_P operator for logic programs.

$$
\begin{array}{l}
iterate \leftarrow operator,\ fail.\\
iterate \leftarrow retract(flag),\ iterate.\\
iterate.\\[4pt]
cond\_assert(F) \leftarrow\\
\quad \backslash{+}in\_database(F),\ !,\\
\quad assert(F).\\[4pt]
in\_database(fact(G)) \leftarrow fact(B),\\
\quad subsumes(B, G),\ !.\\
in\_database(flag) \leftarrow flag,\ !.
\end{array}
$$

The "control"

$$
\begin{array}{l}
operator \leftarrow\\
\quad user\_clause(Head, Body),\\
\quad prove(Body),\\
\quad (\\
\quad\quad cond\_assert(fact(Head))\\
\quad\quad \rightarrow\ \ cond\_assert(flag)\\
\quad).\\[4pt]
prove([\,]).\\
prove([B|Bs]) \leftarrow\\
\quad fact(B),\ prove(Bs).
\end{array}
$$

The "logic"

Figure 1: A Prolog interpreter for bottom-up evaluation.

Figure 1 illustrates a simple Prolog interpreter for naive bottom-up evaluation of any (finitary) logic program P. Each clause $h \leftarrow b_1, \ldots, b_n$ in P is represented as a fact of the form $user\_clause(h,\ [b_1, \ldots, b_n])$. The interpreter can be divided conceptually into two components. On the right, the predicate `operator/0` provides the "logic" and the inner loop of the algorithm which for each $user\_clause(Head, Body)$ in P proves the *Body* using facts derived so far and calls the predicate `cond_assert/1` which asserts the *Head* if it is new. A fact is new if it is not subsumed[1] by any of the facts derived so far. When a new fact is asserted to the Prolog database, a *flag* is raised (unless the flag has already been raised). The control component, on the left, invokes iterations of the `operator` until no new facts are derived. Iteration terminates when $retract(flag)$ fails in the second clause indicating that no new facts were asserted in the previous iteration. Bottom-up evaluation is initiated by the query `?- iterate` which leaves the result of the evaluation in the Prolog database.

Simple interpreters, such as the one depicted in Figure 1, while not in the pure subset of Prolog, have proven extremely useful in the design of program analyses based on abstract interpretation. For example, the analyses described in [9, 10, 7, 8, 14] were all implemented based on straightforward optimizations

[1] A common variation of this program considers as a new fact one which is not a variant of any of the facts derived so far.

and enhancements of this simple working interpreter. Formally, the interpreter in Figure 1 computes a non-ground variation of the standard T_P semantics of the input program (assuming it is finite). Two common variations are the c-semantics and the s-semantics (when `cond_assert` performs a *subsumes* check or a *variant* check respectively) [13, 5]. Both of these semantic variations are widely applied in the context of deductive databases[2] as well as in the context of semantics based program analysis [6, 3].

The main drawback of bottom-up evaluation of the type described above is that all consequences of the program are generated, not just the facts relevant to processing a given query. The essential idea in most bottom-up methods is to combine a top-down generation of goals with a bottom-up generation of facts. In the "bottom-up" approach, this is achieved through a source-to-source program transformation. The Magic-sets transformation [2] and other related techniques such as generalized Magic-sets [4], Magic Templates [21], Alexander Templates [25] and others, originate as an optimization technique in the context of deductive databases. The common principle underlying these techniques is a transformational approach in which a "magic program" $P_G^{\mathcal{M}}$ is derived from a given program P and goal G. The minimal model of the derived program is more efficient to compute (bottom-up) and contains the information from the minimal model of P which is relevant for the goal G. This same approach has proven useful in the context of program analysis because the (non-ground) minimal model of a transformed program $P_G^{\mathcal{M}}$ exhibits also information about the set of *calls* which arise in the computations of G.

In this paper we illustrate our approach using a simplified version of the Magic Templates algorithm. We assume that the body of a rule is evaluated from left-to-right as with Prolog's execution strategy. We do not consider the "adornments" (which indicate the call patterns of "bound" and "free" argument positions in the head of a predicate) used in the Magic-sets transformation. Moreover to simplify presentation we will assume that the initial query G is atomic. Our results are orthogonal to these refinements of the basic algorithm and it is straightforward to adapt our technique to deal with the more general definition.

The Magic-sets transformation is defined as follows: let P be a logic program and G an initial goal. The corresponding magic program is $P_G^{\mathcal{M}}$. For each n-ary predicate symbol p/n in P and G, $P_G^{\mathcal{M}}$ will contain two new predicate symbols: p^c/n (read as "*p/n is a call*") and p^a/n (read as "*p/n is an answer*"). For each clause $h \leftarrow b_1, \ldots, b_n$ in P, $P_G^{\mathcal{M}}$ will contain corresponding clauses of the form $b_i^c \leftarrow h^c, b_1^a, \ldots, b_{i-1}^a$ $(i = 1..n)$ (read as "*b_i is a call if h is a call and $b_1, \ldots, b_{i-1}$ are answers*") and $h^a \leftarrow h^c, b_1^a, \ldots, b_n^a$ (read as "*h is an answer if h is a call and $b_1, \ldots, b_n$ are answers*"). In addition $P_G^{\mathcal{M}}$ will contain the "seed" fact p^c ("*p is a call*") corresponding to the initial query p. The $n+1$ magic

[2] The s-semantics and the c-semantics are sometimes referred to as the set of *generated consequences* and *irredundant generated consequences* in the deductive database community (cf. [17, 20]).

clauses derived from a clause $h \leftarrow b_1, \ldots, b_n$ are depicted in Figure 2.

$$
\begin{array}{l}
b_1^c :- \ h^c. \\
b_2^c :- \ h^c, b_1^a. \\
b_3^c :- \ h^c, b_1^a, b_2^a. \\
\\
b_n^c :- \ h^c, b_1^a, b_2^a, \ \ldots, \ b_{n-1}^a. \\
h^a :- \ h^c, b_1^a, b_2^a, \ \ldots, \ b_{n-1}^a, b_n^a.
\end{array}
$$

Figure 2: The $n+1$ magic clauses derived from $h :- \ b_1, \ldots, b_n$.

The formal relation between the top-down execution of a goal G with the program P and the bottom-up evaluation of the magic program $P_G^{\mathcal{M}}$ is a well studied problem. Basically, the calls that arise in a computation of G with P correspond to the atoms of the form p^c in the bottom-up semantics of $P_G^{\mathcal{M}}$ and the answers to these calls correspond to the atoms of the form p^a in the bottom-up semantics of $P_G^{\mathcal{M}}$. For a more thorough discussion on this relation see for example [6, 11].

3 Induced Magic-sets

The literature on efficient bottom-up evaluation is extensive and we do not propose to cover it in detail here. We focus here on a specific optimization technique, termed *Induced Magic-sets* which can be combined with other techniques described in the literature such as for example a semi-naive evaluation algorithm and rule ordering techniques based on the strongly connected components in a program's call graph.

Let us come back to the $n+1$ magic clauses derived from a program clause $h \leftarrow b_1, \ldots, b_n$ depicted in Figure 2. These clauses are characterized by their "triangular" structure in which the prefixes of each clause occur as the bodies of the clauses above it. This is a potential source of inefficiency for bottom-up evaluation where the basic step considers the solutions of a clause body using the facts derived so far to derive new instances of its head. When solving the body $h^c, b_1^a, \ldots, b_i^a$ of a clause in Figure 2 we are re-solving the bodies of each of the i clauses above it.

Supplementary Magic transformations (also called Supplementary Magic Templates or Generalized Supplementary Magic-sets [4, 21]) address precisely this issue. In this approach a program transformation is applied to introduce continuation passing rules which eliminate the common subexpressions in the $n+1$ rules depicted in Figure 2. The technique is similar to the transformation applied in the context of Earley parsing for context free grammars [12]. Conceptually, we simply replace the first $i-1$ atoms in the body of the i^{th} magic clause $b_i^c \leftarrow h^c, b_1^a, \ldots, b_{i-2}^a, b_{i-1}^a$ by the head of the $i^{th}-1$ clause obtaining, $b_i^c \leftarrow b_{i-1}^c, b_{i-1}^a$ and hence avoiding the re-computation of these atoms. However, care must be taken because the head of the $i^{th}-1$ clause b_{i-1}^c may not contain all of the variables in its body $h^c, b_1^a, \ldots, b_{i-2}^a$. This requires the introduction of new *supplementary* predicates which extend the heads of the clauses to be "reused" to include additional variables from the bodies. The basic scheme of the Supplementary Magic transformation is illustrated in Figure 3. Each clause contains at most two calls in its body and re-computation during bottom-up evaluation is avoided because supplementary predicates (s_i in the figure) are employed to remember results which can then be reused.

$$
\boxed{\begin{array}{l}
b_1^c :- \ h^c. \\
b_2^c :- \ s_1. \\
b_3^c :- \ s_2. \\
\quad\vdots \\
b_n^c :- \ s_{n-1}. \\
h^a :- \ s_n.
\end{array}}
\qquad
\boxed{\begin{array}{l}
s_1 :- \ h^c, b_1^a. \\
s_2 :- \ s_1, b_2^a. \\
\quad\vdots \\
s_n :- \ s_{n-1}, b_n^a.
\end{array}}
$$

Figure 3: Supplementary Magic clauses for $h :- \ b_1, \ldots, b_n$.

The application of Supplementary magic to avoid re-computation due to the overlapping of clause bodies in magic programs is not always profitable. In many cases, adding variables in the heads of clauses (so that they may be reused) can be very costly — especially when the number of facts inferred during bottom-up evaluation for a given predicate is exponential in the number of its arguments. In such cases, re-computing the clause bodies is often less time consuming than the overhead in maintaining supplementary predicates during bottom-up evaluation. In other cases, the use of Supplementary magic reduces the complexity of the computation. This is related to the way *"joins"* are performed and to the elimination of existentially quantified variables in the Supplementary Magic clauses. A classic and well-known example is the following:

Example 1 *Consider the following program which contains n^2 facts of the form.*

$$\begin{array}{llll} edge(1,1). & edge(1,2). & \cdots & edge(1,n). \\ edge(2,1). & edge(2,2). & \cdots & edge(2,n). \\ & \vdots & & \\ edge(n,1). & edge(n,2). & \cdots & edge(n,n). \end{array}$$

In this case, bottom-up evaluation of the clause

$$path(X_1, X_k) \leftarrow edge(X_1, X_2), edge(X_2, X_3), edge(X_{k-1}, X_k)$$

(say with $k = 4$) involves n^4 resolution steps to prove the body of the clause. In contrast, if the clause is transformed to the form

$$\begin{array}{l} p(X_1, X_4) \leftarrow supp(X_1, X_3), edge(X_3, X_4). \\ supp(X_1, X_3) \leftarrow edge(X_1, X_2), edge(X_2, X_3). \end{array}$$

Then, solving the body of the second clause involves n^3 resolution steps but infers only n^2 instances of $supp(X_1, X_3)$ and so, evaluation for the first clause also involves n^3 steps giving a total cost of $2n^3$.

In view of this situation, in many deductive database systems, the decision whether to apply Supplementary magic or not is left to the user and often not applied by default. For example in XSB, the directive `:- suppl_table(2)` [23] implements a version of supplementary magic but is not applied by default. In most cases it requires more space for supplementary predicates and experience shows that its often slows down the evaluation — despite the theoretically better complexity characteristics [24]. Similar experiences have been reported in the area of program analysis where attempts to introduce Supplementary magic have led to slow downs in most cases. The technique is not applied in any of the current implementations of bottom-up semantic based analyzers for logic programs (e.g. [7, 8, 14]).

This paper addresses the situation when Supplementary magic is not applied. We show how the results of previously solved clause bodies in a magic program can be partially re-used without introducing an additional overhead. In fact, goal directed bottom-up evaluation is easily obtained even without performing the Magic-sets transformation. Instead, a bottom-up control strategy is induced and can be applied directly to the original program. We show, that for a typical set of program analysis benchmarks, Supplementary magic indeed does not pay off and that our simple approach provides substantial speed-ups over standard bottom-up evaluation using Magic sets.

Let us take yet another look at the $n + 1$ magic clauses derived from a program clause $h \leftarrow b_1, \ldots, b_n$ depicted in Figure 2. Consider an iteration of a bottom-up evaluation (naive or even semi-naive) in which (some of [3]) these clauses are considered. This involves solving each of the $n + 1$ clauses using the facts already derived and adding new facts corresponding to the clause heads. To avoid re-computation we propose a simple alternative. Consider the last clause body. Its prefixes constitute the bodies of the other clauses in Figure 2. Hence when solving this body (from left-to-right) all of the other (magic) clause bodies are solved along the way.

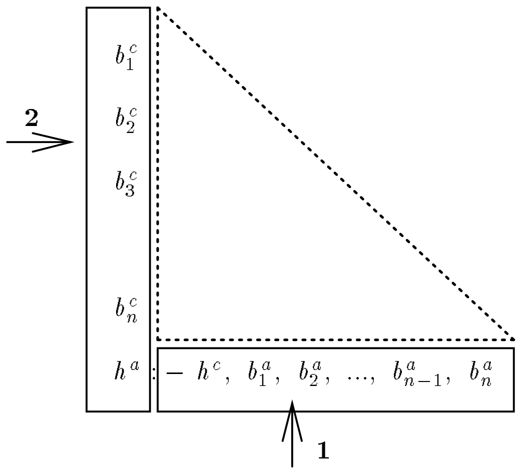

Figure 4: Induced Magic-sets for $h :- \ b_1, \ldots, b_n$.

Our approach is illustrated in Figure 4. The bottom-up evaluator processes, from left-to-right, the *Body* of the last magic clause in Figure 2. When the execution solves the "next" atom (e.g. b_1^a at point **1** in the figure) then the (corresponding instance of the) "next" clause head (e.g. b_2^c at point **2** in the figure) is added to the set of facts derived so far.

Figure 5 presents an operator for goal directed bottom-up evaluation which does not require any form of program transformation. The control is the same as that specified in Figure 1. The interpreter maintains the annotations for *calls* and *answers* as tags associated with the facts derived during bottom-up evaluation. The operator specifies that to process a clause $Head \leftarrow Body$, we

[3] In semi-naive evaluation if one of the clauses in Figure 2 is used then so will all of the clauses below it.

should first consider a call to (the) *Head*, then prove (the) *Body* and then assert that there is an answer for (the) *Head*. When proving a clause body $b_1, \ldots, b_n$, we should assert that there is a call to b_1, then solve b_1 with a previously inferred answer and continue iteratively for the rest of the body. One might

$$operator \leftarrow user\_clause(H, Body),\ fact(call(H)),$$
$$\quad prove(Body), cond\_assert(fact(ans(H))).$$

$$prove([\ ]).$$
$$prove([B|Bs]) \leftarrow cond\_assert(fact(call(B))),\ fact(ans(B)), prove(Bs).$$

Figure 5: Operator for goal directed bottom-up evaluation.

argue that the operator depicted in Figure 5 is not bottom-up in nature and this is a correct observation. However, For a given program clause $h \leftarrow b_1, \ldots, b_n$, this operator expresses precisely (step-by-step) the actions performed by the bottom-up interpreter of Figure 1 when processing the corresponding $n+1$ Magic clauses. One should not forget the original motivation for the Magic set transformation: to force the bottom-up evaluation of a Magic program to behave like the top-down evaluation of the original program.

4 An Evaluation of Induced Magic-sets

Induced Magic-sets provide a simple and efficient implementation technique which, much the same as Supplementary Magic, avoids the re-computation of clause bodies due to common subexpressions typical to magic programs. However, in contrast to Supplementary Magic, intermediate solutions (of common subexpressions) are maintained by the underlying (Prolog) implementation. As a consequence, the overhead in maintaining supplementary predicates to remember solutions which later have to be looked up is reduced. Indeed, Sudarshan and Ramakrishnan [26] observe that supplementary predicates in a bottom-up evaluation capture the variable bindings at the corresponding program points in a top-down Prolog evaluation. This is true also of our technique, only we make direct use of these program points to maintain intermediate solutions to clause bodies.

There is an important difference between the two techniques: With Supplementary magic, intermediate solutions are maintained globally by materializing supplementary predicates, while with Induced magic they are found locally on the runtime stack. This means that during a single application of the bottom-up operator, while iterating over the magic clauses, intermediate results are reused with no additional overhead. However, with each new iteration of the bottom-up operator the computation begins from scratch (unless a semi-naive strategy detects that the corresponding clauses will not contribute to the evaluation).

Table 1 summarizes how a single clause $C \equiv \; a_1 :- a_2, \ldots, a_n$ can influence several factors in one iteration of naive bottom-up evaluation. In the standard Magic-sets program, C contributes n clauses; in each iteration of the bottom-up evaluation the $(n^2+n)/2$ atoms in these clause bodies are solved with the facts derived so far and for each selection of facts, n potentially new clause heads are derived and conditionally asserted. In the Supplementary Magic approach, C contributes $2n$ clauses; each iteration solves $3n$ body atoms and for each selection of facts derives $2n$ potentially new clause heads. In the Induced Magic-sets approach, only n atoms are solved in each iteration and for each selection of facts only n new atoms are conditionally asserted.

Method	Size heads	Size bodies	Atoms Solved	Conditional Asserts
Magic	n	$\frac{n^2+n}{2}$	$\frac{n^2+n}{2}$	n
Supplementary	$2n$	$3n$	$3n$	$2n$
Induced	1	$n-1$	n	n

Table 1: A clause of size n participating in one iteration of bottom-up evaluation.

Experimental Results: We have compared the performance of the three methods for Magic sets (standard, Supplementary and Induced) applying several different evaluation strategies and optimization techniques. These include the use of a semi-naive evaluation strategy, the use of an eager (almost semi-naive) evaluation strategy as described in [28] and a modular evaluation following the strongly connected components in the program's call graph. Interpreters for semi-naive and eager evaluation (standard and Induced magic versions) are given in the Appendix. In brief, with the eager strategy, iteration is replaced by recursion and new facts are used as soon as they are inferred. One advantage is that it is not necessary to distinguish between new and old facts. Another advantage is that the resulting depth first strategy considers first lower components in the call graph but without the overhead in actually computing its strongly connected components. The disadvantage is that the evaluation strategy is only "almost" semi-naive — there is some degree of recomputation and hence redundant unifications are performed. In the experiments, semi-naive and eager strategies were applied to Magic and Supplementary Magic transformed (abstract) programs. The Induced Magic strategy was applied directly to the given (abstract) programs.

The experiments have been carried out in the context of semantic based program analysis where abstract interpretations are applied to provide information about type-dependencies (types) and groundness dependencies (modes) as described in [8] and [7] respectively. In addition we have compared the

techniques for several (concrete) Prolog programs. A standard set of program analysis benchmark programs is considered. The programs range in size from 2 clauses to 271 clauses. The concrete programs chosen include: $qsort(n)$ — quicksort on n elements in reverse order, $queens(n)$ — placing n queens on a chess board, $path(n,k)$ the program from Example 1, *zebra* — the zebra puzzle, and $inssort(n)$ — a program for insertion sort on n elements. The complete set of benchmark programs can be obtained from
`ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/fastmagic.tgz`.

For each class of programs the results for the evaluation strategy which gave the best results are presented in the corresponding tables: For mode analysis and for the concrete programs — eager evaluation. For type analysis: semi-naive evaluation for Induced magic and semi-naive evaluation with strongly connected components for the Magic and Supplementary Magic techniques. The explanation for this is that for programs involving standard (concrete) unification, eager evaluation may perform redundant unifications but is faster as it does not have to distinguish between old and new facts. For type analysis, which involves a more complex unification algorithm, minimizing the number of unifications pays off even at the extra expense in maintaining time stamps on the facts (to distinguish new from old).

Tables 2, 3 and 4 illustrate the results for the mode and type analyses and for concrete evaluations (for each of the three methods). The tables are ordered by the number of clauses in the programs. All of the benchmarks were performed running Sicstus Prolog version 3 release 5 on a Sparc 4000 with 4 cpu's (167 mHz) and 384 megabytes memory.

In the comparisons we consider: (a) the cost of the analysis, in seconds (**time**), (b) the number of (abstract) unifications performed when solving clause body atoms in the evaluation (**steps**), and (c) the number of (abstract) atoms in the resulting (abstract) "minimal model" (**atoms**). The analysis times include the time to read the programs and to perform the Magic and Supplementary Magic set transformations where applicable. The cost of the computation of strongly connected components (evaluated for type analyses using Magic and Supplementary Magic) is excluded. This is because our implementation applies the Sicstus Prolog libraries for this (which is not the fastest) and yet to give the maximum benifit to the Magic and Supplimentary Magic techniques.

Some Observations:

atoms: The number of atoms created during the bottom-up evaluations are more or less the same for Induced and standard magic. The slight differences are due to different (non-ground) representations for equivalent results. The main point to note is the extra space requirments in the database for the Supplementary magic evaluation. For large benchmark programs there is the danger that Supplementary magic will run out of space much fatser than Induced magic

Program	Induced Magic			Standard Magic			Supplem. Magic		
	time	steps	atoms	time	steps	atoms	time	steps	atoms
append.pl	0.01	12	3	0.01	13	3	0.05	21	13
rev.pl	0.02	91	15	0.02	108	15	0.13	95	51
inssort.pl	0.00	14	4	0.02	19	4	0.14	34	23
qsort.pl	0.00	66	8	0.02	107	8	0.27	82	45
queens.pl	0.01	43	12	0.02	63	12	0.31	90	55
mapcol.pl	0.02	98	28	0.02	164	28	0.31	147	82
serialize.pl	0.06	529	33	0.12	808	33	0.64	546	325
treeorder.pl	0.20	1578	98	0.23	2854	98	0.51	1003	369
pg.pl	0.02	101	26	0.09	171	26	0.83	221	146
zebra.pl	0.10	1473	16	0.27	6294	16	1.22	1025	564
nandc.pl	0.11	1293	36	0.17	1882	36	1.15	793	362
life.pl	0.05	163	25	0.12	218	25	1.20	308	192
plan.pl	0.04	209	44	0.12	404	44	0.96	297	153
browse.pl	0.07	600	64	0.16	896	63	1.17	650	365
meta.pl	0.07	689	46	0.13	841	45	0.69	610	242
dnf.pl	0.04	150	8	0.09	318	8	0.98	316	136
scc1.pl	0.08	524	41	0.27	1483	41	2.51	755	411
ronp.pl	0.46	3797	118	0.59	4751	118	1.33	1369	827
tsp.pl	0.17	1227	152	0.44	2096	152	1.94	978	535
gabriel.pl	0.11	853	78	0.23	1311	77	1.49	839	434
mastermind.pl	0.13	861	76	0.28	1648	74	1.50	1025	478
disj_o.pl	0.07	612	57	0.25	2960	57	1.91	440	237
cs_o.pl	0.08	349	72	0.28	660	72	2.73	579	357
tictactoe.pl	0.10	176	39	0.27	377	39	2.43	397	237
neural.pl	0.21	1303	130	0.60	2047	128	2.60	1112	601
kalah_r.pl	0.13	438	100	0.40	763	100	3.48	796	478
nbody.pl	1.10	9246	323	2.21	19581	323	6.09	4543	2415
cs_r.pl	0.14	465	76	0.63	2280	76	5.24	1437	781
boyer.pl	0.37	1763	163	0.79	2150	162	3.21	1863	1125
read_o.pl	2.91	20300	292	4.51	28234	291	6.10	7559	2416
ann.pl	3.12	36202	372	4.25	55256	371	5.58	5216	2114
peep.pl	0.24	320	51	0.70	733	51	7.56	1251	693
asm.pl	0.42	2128	191	2.60	6420	190	10.9	5201	2376

Table 2: Analysis times (mode analysis, eager evaluation).

Program	Induced Magic			Standard Magic			Supplem. Magic		
	time	steps	atoms	time	steps	atoms	time	steps	atoms
append.pl	0.01	11	2	0.02	7	2	0.06	8	3
reverse.pl	0.02	27	4	0.07	26	4	0.13	25	7
inssort.pl	0.02	31	4	0.09	36	4	0.13	30	9
qsort.pl	0.09	279	9	0.29	326	9	0.29	154	28
queens.pl	0.02	65	10	0.19	77	10	0.28	65	22
mapcol.pl	0.02	61	14	0.26	63	14	0.31	62	24
serialize.pl	0.37	1015	15	0.64	894	14	0.77	456	52
treeorder.pl	0.11	258	20	0.38	290	20	0.55	248	48
pg.pl	0.19	374	22	0.62	508	22	0.76	260	50
zebra.pl	0.06	192	12	0.62	314	12	1.06	113	35
nandc.pl	0.25	892	24	1.12	1009	24	1.8	330	108
life.pl	0.11	159	25	0.94	197	25	1.54	206	78
plan.pl	0.08	193	29	0.68	308	29	0.88	214	58
browse1.pl	0.22	459	34	1.11	533	34	1.78	368	109
meta1.pl	0.11	491	29	0.48	498	30	1.24	938	168
dnf.pl	0.14	720	13	0.76	891	13	1.29	867	119
scc1.pl	0.31	539	36	1.35	657	36	2.14	596	88
ronp.pl	0.14	263	30	0.75	225	30	1.27	220	76
tsp.pl	0.98	1456	71	2.30	1716	71	2.92	790	184
gabriel.pl	0.13	231	36	1.02	297	36	1.38	258	86
mastermind.pl	0.23	663	57	1.54	1004	57	2.18	727	212
disj_r.pl	0.05	74	17	1.52	88	17	1.87	73	30
cs_o.pl	0.07	45	14	1.42	71	14	2.18	50	21
tictactoe.pl	0.15	294	31	1.54	348	30	3.87	403	146
neural.pl	0.41	814	75	2.72	1126	75	4.85	847	320
kalah_r.pl	0.69	945	91	2.68	1212	91	4.20	728	227
nbody.pl	0.28	408	70	2.81	718	69	5.29	485	213
cs_r.pl	1.85	2826	76	4.84	3764	76	5.31	1027	171
boyer.pl	0.10	48	7	1.37	71	7	1.65	50	21
read_o.pl	0.78	1885	54	3.58	1265	48	5.16	795	154
ann.pl	0.14	67	18	3.18	75	18	4.14	81	39
peep.pl	0.63	949	45	3.62	968	46	6.21	1042	222
asm.pl	1.02	1978	94	6.49	2948	94	10.8	2398	475

Table 3: Analysis times (types analysis, sn evaluation).

Program	Induced Magic			Standard Magic			Supplem. Magic		
	time	steps	atoms	time	steps	atoms	time	steps	atoms
qsort(10)	0.40	1178	242	0.38	2051	242	0.21	908	417
qsort(20)	7.16	4253	882	6.34	7196	882	3.35	3313	1532
queens(4)	0.38	1592	292	0.31	2032	292	0.28	1186	672
queens(5)	8.71	11653	1351	7.23	13559	1351	7.05	5778	3488
path(10,4)	0.38	3261	121	0.32	3915	121	0.07	665	151
path(10,5)	2.58	25929	121	2.36	29674	121	0.09	888	161
path(10,6)	24.88	246563	121	21.66	276077	121	0.10	1112	171
zebra	6.68	9739	889	6.27	32411	889	9.51	6673	2900
inssort(15)	0.77	902	272	0.59	1083	272	0.53	1007	512

Table 4: Analysis times (concrete eager evaluation).

steps: For all of the experiments, Induced magic performs less unification steps than standard magic. For the analysis benchmarks, there is no clear advanatge of Induced magic over Supplementary magic in the number of steps and on several programs Supplementary magic performs considerably less unifications. For the concrete programs Supplementary magic is a clear win. Of course we specifically chose the programs in Table 4 with longer clause bodies which benifit from the fact that Supplementary magic reduces the size of joins.

time: It is on time, that Induced magic shows its benifit. Mainly because of the time saved in reading the input programs where no Magic or Supplementary transformation is required. However, the information on times should be taken with care. The evaluation techniques used are implemented as meta-interpreters and using the Prolog database with dynamic code (the assert predicate).

5 Conclusion

We have described a new strategy for the efficient bottom-up evaluation of logic programs. We address the specific problem of re-computation of clause bodies due to common subexpressions typical to clause bodies of magic programs. Our approach provides an alternative for Supplementary Magic-sets which provides local caching of intermediate results without incurring any additional overhead.

For simplicity, we have illustrated the approach using a simple Prolog interpreter for naive bottom-up evaluation. It is straightforward to combine our approach with a semi-naive evaluation strategy and other classic optimizations. We include Prolog interpreters for semi-naive and eager evaluation with Induced Magic-sets as an appendix to this paper.

Besides the obvious relevance to bottom-up evaluation in deductive databases, our approach is of immediate benefit in the context of semantics based analysis of logic programs. Many of the implementations in the bottom-up approach are based on the combination of simple interpreters similar to those depicted in this paper and Magic-sets transformations. In addition, the technique proposed may be of relevance for the implementers of systems like XSB or those based on Supplementary magic.

The literature is rich in (bottom-up) evaluation strategies for deductive databases and logic programs. Each of the various techniques has its pros and cons. Induced Magic-sets, as proposed in this paper is no exception. The main benefit of our approach is for those cases when standard Magic sets are applied and Supplementary magic is not used. Then, we reduce considerably the overhead involved in maintaining solutions to clause bodies, and eliminate completely the need to apply the transformation associated with Magic-sets. For the future, we propose that bottom-up evaluation combine a mixture of the two techniques (Supplementary and Induced magic sets).

Acknowledgments

The many discussions with and comments from Bart Demoen, John Gallagher, Kostis Sagonas and Peter Stuckey are very much appreciated. Muhamed Abu-Zaid's help with the benchmarks seemed endless.

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, 1986.

[3] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[4] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3 and 4):255–300, 1991.

[5] Annalisa Bossi, Maurizio Gabrielli, Giorgio Levi, and Maurizio Martelli. The s-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.

[6] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[7] M. Codish and B. Demoen. Analysing logic programs using "`prop`"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, December 1995.

[8] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, IEEE Press, pp 136-145, June, 1996.

[9] M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In J. Hanus, M. Heering and K. Meinke, editors, *6th International Joint Conference on Algebraic and Logic Programming*, pages 31–45, Southhampton UK, 1997. LNCS 1290 Springer-Verlag.

[10] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In *Proceedings of the Fourth International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, 1997.

[11] S. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *The Journal of Logic Programming*, 18(2):149–176, February 1994.

[12] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[13] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[14] J. P. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Massachusetts Institute of Technology, 1994. The MIT Press.

[15] B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: implementation and evaluation of the domain Prop. In *Proceedings Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1993.

[16] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2^{nd} edition, 1987.

[17] M. J. Maher and R. Ramakrishnan. Deja vu in fixpoints of logic programs. Technical Report TR 893, Computer Sciences Department, University of Wisconsin-Madison, November 1989.

[18] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):501–517, August 1994. A shorter version appeared in VLDB, 1990.

[19] R. Ramakrishnan and J. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.

[20] R. Ramakrishnan. Parallelism in logic programs. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 246–260, January 1990.

[21] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *The Journal of Logic Programming*, 11(3 & 4):189–216, October/November 1991.

[22] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering.* Kluwer Academic Publishers, 1992.

[23] K. Sagonas, T. Swift, and D.S. Warren. The xsb programmer's manual version 1.5.0. http://www.cs.sunysb.edu/ sbprolog/manual/manual.html, February 1996.

[24] K. Sagonas and D. S. Warren, 1997. Personal communication.

[25] H. Seki. On the power of Alexander templates. In *Proceedings of the 8^{th} ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Philadelphia, Pennsylvania, 1989.

[26] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 557–574, Massachusetts Institute of Technology, Cambridge, Massachusetts 021-42, 1993. The MIT Press.

[27] J. Ullman. *Principles of Database and Knowledge – Base Systems*, volume 1. Computer Science press, 1988.

[28] J. Wunderwald. Memoing evaluation by source-to-source transformation. In Maurizio Proietti, editor, *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *LNCS*, pages 17–32. Springer, 1995.

Appendix: More Interpreters

This appendix presents the core idea of the interpreters used in the experimental evaluation of this paper (the actual interpreters used are based on simple optimizations of the ones described here).

Semi-naive Induced Magic-sets

Figure 6 illustrates a simple Prolog interpreter for semi-naive bottom-up evaluation. The interpreter is essentially the same as that presented in Figure 1 except that it ensures that at least one of the facts used when solving a clause body is a new fact, derived in the previous iteration. This is facilitated by associating each fact with the number of the iteration in which it was derived. As in the case of naive evaluation, each clause $Head \leftarrow Body$ in the input program is represented as a fact of the form $user\_clause(Head, Body)$. However, for semi-naive evaluation it is important to represent a fact $Head$ in the original program as $user\_clause(Head,[true])$ and to include $fact(true,0)$ in the program.

Figure 7 presents the intuition for the interpreter of Figure 8 which enhances the basic semi-naive interpreter for the evaluation of Induced Magic-sets, much the same as the interpreter shown in Figure 5. The control component is identical to that in Figure 6. For semi-naive Induced Magic we distinguish the atoms to the left and to the right of a new atom b selected in a clause body. If b is found in the body of a magic clause $h \leftarrow body$, then it is found in all of the magic clauses whose bodies extend $body$ (point **1** in Figure 7). While proving the atoms to the left of b we do not need to assert corresponding clause heads (those above point **2** in Figure 7). In the first clause

```
iterate(N) ←
    operator(N), fail.
iterate(N) ← fact(N,_), !,
    N1 is N + 1, iterate(N1).
iterate(_).

cond_assert(F) ←
    \+in_database(F), !,
    assert(F).

in_database(fact(N, G)) ←
    fact(_, B), subsumes(B, G).


fact(0, true).
```

The "control"

```
operator(N) : −N1 is N − 1,
    fact(N1, Atom),
    user_clause(H, Bs),
    select(Atom, Bs, Rest),
    prove(Rest),
    cond_assert(fact(N, H)).

select(Fact, [Fact|R], R).
select(Fact, [B|Bs], [B|R]) ←
    select(Fact, Bs, R).

prove([X|Xs]) ← fact(_, X),
    prove(Xs).
prove([ ]).
```

The "logic"

Figure 6: A Prolog interpreter for Semi-naive evaluation.

of the interpreter of Figure 8 we first choose a new fact and then call *solve_induced*. The evaluation considers two cases depending on if the new fact is a *call* or an *answer*.

Eager Induced Magic-sets

Figure 9 depicts an interpreter for eager bottom-up evaluation which is distilled from the presentation in [28]. The idea is that instead of looking in each iteration for a new fact, this interpreter uses new facts as soon as they are discovered. Note that the code for `cond_assert(Atom)` is designed to fail if the `Atom` is not new. There is no code to "iterate" over the program clauses. This is managed by the underlying control. The resulting evaluation strategy is close to semi-naive but does repeat computations

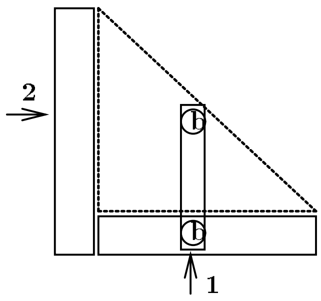

Figure 7: The principle of Semi-naive Induced Magic

$$
\begin{array}{l}
operator(N) \leftarrow N1\ is\ N-1,\ fact(N1, NewAtom),\\
\quad solve\_induced(N, NewAtom).\\
solve\_induced(N, call(H)) \leftarrow user\_clause(H, Body),\\
\quad prove\_right(N, Body),\ cond\_assert(fact(N, ans(H))).\\
solve\_induced(N, ans(A)) \leftarrow user\_clause(H, Body),\\
\quad inbody(A, Body, Left, Right),\ fact(\_, call(H)),\ prove(Left),\\
\quad prove\_right(N, Right),\ cond\_assert(fact(N, ans(H))).\\
inbody(Fact, [Fact|Bs], [\,], Bs).\\
inbody(Fact, [B|Bs], [B|Left], Right) \leftarrow inbody(Fact, Bs, Left, Right).\\
prove\_right(\_, [\,]).\\
prove\_right(N, [X|Xs]) \leftarrow\\
\quad cond\_assert(fact(N, call(X))),\ fact(\_, ans(X)),\ prove\_right(N, Xs).
\end{array}
$$

Figure 8: A Prolog Interpreter for Semi-naive Induced Magic.

avoided by a semi-naive strategy. The advantage is that it is not necessary to augment facts by time stamps nor to search for new facts. The initial call should contain the first "new" fact. This could be a call of the form `eager(true)` for standard bottom-up evaluation, or a call of the form `eager(query_p)` if the program is a magic program and the initial call is `p`.

$$
\begin{array}{l}
eager(Atom) \leftarrow cond\_assert(fact(Atom)),\\
\quad user\_clause(Head, Body),\ select(Atom, Body, Rest),\\
\quad prove(Rest),\ eager(Head).\\
eager(\_).\\
cond\_assert(F) \leftarrow \backslash\!+ in\_database(F),\ assert(F), !.
\end{array}
$$

Figure 9: A Prolog interpreter for Eager evaluation.

Figure 10 illustrates a Prolog interpreter for induced magic which applies an eager goal dependent evaluation strategy to a given program. Note the call to `ind_eager(call(X))` in `solve_right/2`.

$ind\_eager(call(Atom)) \leftarrow cond\_assert(fact(call(Atom))), user\_clause(Atom, Body),$
$\quad solve\_right(Body), ind\_eager(ans(Atom)).$
$ind\_eager(ans(Atom)) \leftarrow cond\_assert(fact(ans(Atom))), user\_clause(H, Body),$
$\quad inbody(Atom, Body, Left, Right), fact(call((H))),$
$\quad prove(Left), solve\_right(Right), ind\_eager(ans(H)).$
$ind\_eager(\_).$

$solve\_right([]).$
$solve\_right([X|Xs]) \leftarrow ind\_eager(call(X)), fact(ans(X)), solve\_right(Xs).$

Figure 10: A Prolog interpreter for Induced Magic Eager evaluation.