A UNIFIED APPROACH TO LOGIC PROGRAM EVALUATION

by

Jeffrey F. Naughton and Raghu Ramakrishnan

Computer Sciences Technical Report #889

November 1989

A Unified Approach to Logic Program Evaluation

Jeffrey F. Naughton*and Raghu Ramakrishnan†
Computer Sciences Department
University of Wisconsin-Madison, WI 53706, U.S.A.

November 1, 1989

Abstract

The Prolog evaluation algorithm has become the standard for logic program evaluation, and bottom-up methods have long been considered impractical because they compute irrelevant facts. Recently, however, bottom-up evaluation algorithms that retain the focusing property of top-down evaluation have been proposed, and in view of these algorithms the choice between top-down and bottom-up methods needs to be re-examined.

In order to motivate a closer look at bottom-up methods, we identify certain classes of logic programs for which bottom-up evaluation provides polynomial time evaluation algorithms where Prolog takes exponential time. We also demonstrate that techniques such as *predicate factoring* can provide a further $O(n)$ improvement, when they are applicable. We argue that no one evaluation method is uniformly preferable, and suggest that a choice of the appropriate method must be made by the compiler based on the given program. We present several results that shed light on this choice.

The bottom-up approach can be refined in a number of ways, and we show how various results in the literature can be combined to provide a coherent evaluation framework. Further, we indicate how ideas in the *tabulation* literature for functional programs can be adapted to improve the memory utilization of bottom-up methods. We also consider the program transformation techniques pioneered by Burstall and Darlington, and study their relationship to deductive database program transformations such as Magic Templates. The comparison indicates that the bottom-up approach, with the refinements discussed in the paper, often achieves the same gains as these sophisticated transformation systems, and thus illustrates the power of the approach.

To keep this paper self-contained, we have included brief surveys of all the techniques that we discuss. We have not attempted to be comprehensive in our survey; it is our hope that the reader will be motivated to pursue these ideas in greater detail by following up on the references.

1 Introduction

Traditionally, logic programming language implementations have evaluated programs top-down. Recent developments in evaluation of database queries have provided a means of evaluating logic

*Supported by NSF grant IRI-8909795
†Supported in part by an IBM Faculty Development Award and NSF grant IRI-8804319.

programs bottom-up while still retaining the focusing properties of top-down evaluation, based in large part on optimizing program transformations.

Example 1.1 As a simple example of the power of the bottom-up approach, consider a definition of transitive closure

$$\begin{array}{lll} t(X,Y) & :- & e(X,W), t(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(5,Y). \end{array}$$

If e is cyclic, Prolog will not terminate on this program. However, even e is not cyclic, there are values for e such that Prolog takes time $O(2^n)$ to find all answers to the query and terminate.

To evaluate the original query bottom-up, we first obtain the following program by first applying the Magic Templates[1] transformation and then factoring:

$$\begin{array}{lll} m\_t^{bf}(W) & :- & ft(W). \\ m\_t^{bf}(5). & & \\ ft(Y) & :- & m\_t^{bf}(X), e(X,Y). \\ query(Y) & :- & ft(Y). \end{array}$$

Evaluating this program using Seminaive bottom-up always terminates and computes the answer to the query in time linear in the answer size.

Essentially the same program results if Magic Templates plus factoring is applied to versions of the transitive closure expressed with different forms of the recursive rule, including the left-recursive and binary recursive forms upon which Prolog does not terminate for any values of e. This example is presented in detail in Section 5. □

This is an example where the bottom-up approach based on program transformations has notable success. Even on programs for which the transformations do not achieve such large gains, the Magic Templates transformation ensures that no irrelevant goals or facts are generated, and in many cases significant advantages accrue when bottom-up evaluation is used, primarily because:

- With memoing, it implements a form of dynamic programming, which eliminates a great deal of redundant computation, and
- It is sound and complete, and thus the declarative semantics of the program is preserved. This non-procedural approach allows for much more flexibility in program transformation and optimization.

In this paper, we survey and extend recent results that make the bottom-up approach practical. Our objective is two-fold: We wish to provide a broad introduction to the results in this area, and we also wish to demonstrate that the state of the art in logic program evaluation has advanced to the point where we now have available several complementary evaluation methods. These alternative methods use a variety of control strategies, and thus we see a real exploitation of the separation between logic and control that is possible with logic programs.

[1] Earlier versions of the algorithm were called Magic Sets. See Section 3 for details.

It is our thesis that future implementations should consider a wide range of options and use them effectively by making intelligent compile-time decisions about what evaluation method to use. We present several results that shed light on this choice, but these should be viewed as a starting point; the problem is a challenging one.

There are also several results in the functional programming literature that are applicable to the evaluation of logic programs, or that suggest ways in which methods for logic program evaluation, especially bottom-up methods, can be further refined. These connections currently are not well understood. An important link to the bottom-up approach, which is based on memoing intermediate results, is provided by the studies on tabulation [Bir80, Coh83]. These studies examine how memory can be intelligently utilized in memoing computations, based on a careful compile-time analysis of the source program. These schemes are quite ingenious, but are of limited generality. We suggest that the basic ideas could be developed in the context of the Magic Templates transformation, gaining generality at the expense of less than optimal memory utilization. We do not explore this in detail here, due to space constraints, but provide illustrative examples of the potential gains.

We also consider the relationship between program transformation systems proposed for functional programs — primarily the one proposed by Burstall and Darlington [BD77] — and the program transformations used in the bottom-up approach. We bring out some important differences in the underlying assumptions and objectives, and compare their relative power. Not surprisingly, the functional program transformation systems are capable of more sophisticated transformations, but they require programmer guidance, and are very sensitive to changes in the program. The bottom-up transformations are more robust, in that they are typically independent of the set of facts in predicates that are not defined by rules of the program, and they are carried out entirely by the compiler. The surprising aspect of this comparison, rather, is that they do remarkably well relative to the more sophisticated functional transformations, especially if refinements that are suggested by the tabulation studies are incorporated.

The rest of this paper is organized as follows. There are three broad parts: (1) A review of the bottom-up approach, including program transformations and fixpoint evaluation, (2) A discussion on the choice of methods, including a comparison of Prolog and bottom-up evaluation, and (3) A review of results from the functional programming literature, and the refinements that they suggest for the bottom-up approach.

We introduce notation and preliminary definitions in Section 2, and present a summary of the bottom-up evaluation method based on the Magic Templates program transformation and Seminaive fixpoint evaluation in Section 3. In Section 4, we demonstrate that the bottom-up approach does significantly better than Prolog on certain classes of programs. We present a survey of some related program transformations used in the bottom-up approach in Section 5. The survey in Section 5 is not intended to cover the deductive database literature; rather it is a selective treatment of a body of work that together offers a coherent framework for bottom-up evaluation.

In Section 6, we examine bottom-up fixpoint evaluation more closely, and present some important program properties that are used subsequently to guide the choice of method for computing fixpoints. Section 7 extends the properties introduced in Section 6 and applies them to the comparison of the behavior of several bottom-up methods and Prolog.

We consider results from the functional programming literature in Section 8, first presenting the ideas on intelligent tabulation and indicating how they could be applied in the bottom-up approach, and then presenting some program transformation systems and comparing them with the bottom-up approach. We present our conclusions in Section 9.

Due to space limitations, we have not considered any of the recent results on parallelizing bottom-up computations (e.g., [CW89, Don86, GST89, Ram90, VG86, WS89]). This is one of the areas in which we believe that the bottom-up approach shows great promise.

2 Notation and Preliminary Definitions

The language considered in this paper is that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols. A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A tuple of terms is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$.

A substitution is an idempotent mapping from the set of variables of the language under consideration to the set of terms, that is, the identity mapping at all but finitely many points. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \varphi \circ \sigma$. Substitutions are denoted by lower case Greek letters θ, σ, ϕ, etc. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier that is unique up to renaming of variables.

A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body*. A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $\langle P, Q \rangle$ where P is a set of predicate definitions and Q is the *input*, which consists of a query, or goal, and possibly a set of facts for "database predicates" appearing in the program.

We follow the convention in deductive database literature of separating the set of rules with non-empty bodies (the set P) from the set of facts, or unit clauses, which appear in Q and are called the *database*. P is referred to as the *program*, or the set of rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. The meaning of a logic program is given by its least Herbrand model [vEK76].

Following the syntax of Edinburgh Prolog, definite clauses (rules) are written as

$$p :- q_1, \ldots, q_n.$$

read declaratively as q_1 and q_2 and ...and q_n implies p. Names of variables begin with upper case letters, while names of non-variable (i.e., function and predicate) symbols begin with lower case letters.

We will use *derivation trees* in several proofs:

Definition 2.1 Given a program P and input Q, derivation trees in $\langle P, Q \rangle$ are defined as follows:

- Every fact h in Q is a derivation tree for itself, consisting of a single node with label h.
- Let r be a rule: h :- $b_1, b_2, \ldots, b_k$ in P, let d_i, $i = 1 \ldots k$ be atoms with derivation trees t_i, and let θ be the *mgu* of $(b_1, \ldots, b_k)$ and $(d_1, \ldots, d_k)$. Then, the following is a derivation tree for $h\theta$: The root is a node labeled $h\theta$, and each t_i, $i = 1 \ldots n$, is a child of the root. Each arc from the root to a child has the label r.

Note that the substitution θ is not applied to the children of $h\theta$ in the second part of the above definition. Thus, a derivation tree records which set of (previously generated) facts is used to generate a new fact using a rule, rather than the set of substitution instances of these facts that instantiated the rule. The *height* of a derivation tree is defined to be the number of nodes in the longest path (which is always from the root to a leaf).

3 The Bottom-Up Approach

The bottom-up approach that we consider consists of a two-part process. First, the program is rewritten in a form so that the bottom-up fixpoint evaluation of the program will be more efficient; next, the fixpoint of the rewritten program is computed by a bottom-up iteration. Subsection 3.1 describes the initial rewriting, while Subsection 3.2 investigates the computation of the fixpoint of the rewritten program.

Both these steps can be refined further. In Section 5, we discuss several program transformations that can be used in conjunction with the Magic Templates algorithm in the rewriting step. We consider refinements of fixpoint evaluation in Sections 6 and 8.2.1.

3.1 The Magic Templates Rewriting Algorithm

As described in [BR87, Ram88], the initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or sips. For each rule, the associated sip determines the order in which the body literals are evaluated.

We first present a simplified version of the Magic Templates algorithm, tailored to the case that sips correspond to left-to-right evaluation with all bindings propagated, as in Prolog. We generalize to allow a different choice of sip for different goals at the end of this section. The reader is referred to [Ram88] for a more general algorithm capable of implementing more sophisticated sip choices, and also for a detailed discussion of bottom-up fixpoint computation in the presence of non-ground facts.

The idea is to compute a set of auxiliary predicates that contain the goals. The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples.

Definition 3.1 The Magic Templates Algorithm
We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. Create a new predicate $magic\_p$ for each predicate p in P. The arity is that of p.
2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body.
3. For each rule r in P with head, say, $p(\bar{t})$, and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body contains all literals that precede q_i in the sip associated with this rule, and the literal $magic\_p(\bar{t})$.
4. Create a *seed* fact $magic\_q(\langle \bar{c} \rangle)$ from the query.

Example 3.1 Consider the following program.

$$\begin{array}{lll} sg(X,Y) & :- & flat(X,Y). \\ sg(X,Y) & :- & up(X,U), sg(U,V), down(V,Y). \\ sg(john,Z)? & & \end{array}$$

For a choice of sips that orders body literals from left to right, as in Prolog, the Magic Templates algorithm rewrites it as follows:

$$\begin{array}{lll} sg(X,Y) & :- & magic\_sg(X,Y), flat(X,Y). \\ sg(X,Y) & :- & magic\_sg(X,Y), up(X,U), sg(U,V), down(V,Y). \\ magic\_sg(U,V) & :- & magic\_sg(X,Y), up(X,U). \\ magic\_sg(john,Z). & & \end{array}$$

□

We present some results that characterize the transformed program P^{mg} with respect to the original program P, from [Ram88]. The following theorem ensures soundness.

Theorem 3.1 ([Ram88]) *$\langle P, Q \rangle$ is equivalent to $\langle P^{mg}, Q \rangle$ with respect to the set of answers to the query.*

Definition 3.2 Let us define the *Magic Templates Evaluation Method* as follows:

1. Rewrite the program $\langle P, Q \rangle$ according to the choice of sips using the Magic Templates algorithm.
2. Evaluate the fixpoint of the rewritten program.

We hope that the slight abuse of notation in having the same name for the evaluation method and the rewriting algorithm will not lead to confusion; the distinction should be should be clear from the context. The second step above is presented in more detail in the next subsection. The next theorem states that the computation generates no irrelevant facts or goals, and is complete with respect to the least Herbrand model semantics.

Theorem 3.2 ([Ram88]) *The Magic Templates Evaluation Method is a complete sip-method.*

Informally, this means that the method evaluates all answers, and only generates goals and facts that are required according to the choice of sips. For example, if we choose left-to-right sips for all rules, this means that no goal or fact is generated that is not also generated by Prolog (in the course of computing all answers, and assuming that Prolog does not enter an infinite loop because of its depth-first search strategy).

The careful reader will notice that some joins are repeated in the bodies of rules defining magic predicates and modified rules. The *supplementary* version of the rewriting algorithm essentially identifies these common sub-expressions and stores them (with some optimizations that allow us to delete some columns from these intermediate, or supplementary, relations). We refer the reader to [BR87] for details.

We now describe a generalization of the Magic Templates algorithm that uses the notion of *adornments*, similar to *modes*, to specialize rules for different goals. This generalization is necessary in order to discuss the factoring optimization of Section 5.2.

An *adornment* for an n-ary predicate is defined to be a string of 1's and 0's. Argument positions that are treated as free variables in the goal are designated as 0's, and the other argument positions, which are potentially restricted in the goal, are designated as 1's.

Initially, the query is given an adornment in which every argument position that contains only a variable that appears nowhere else is designated 0, and all other argument positions are designated 1. Subsequently, for each rule that defines a predicate p, for every adorned version p^a of p (that is reachable from the adorned query predicate by this analysis), we create an adorned version with head predicate p^a. The choice of a sip — we may choose one sip per rule per head adornment — induces adornments for each body literal: An argument position is 0 if it contains just a variable, and that variable appears nowhere else in any literal that precedes the given literal in the sip.

This algorithm generates a set of adorned rules from P, which we call P^{ad}. The Magic Templates algorithm can be generalized as a two-step transformation in which we first obtain P^{ad} and then apply the algorithm of Section 3.1. Note that there is now a magic predicate associated with each predicate in P^{ad}, instead of with each predicate in P. A simple optimization is to delete all argument positions corresponding to 0's from the magic predicates, based on the observation that these positions always contain distinct variables. In the sequel, we will refer to the above two step transformation with this optimization as the Magic Templates algorithm.

Example 3.2 To see the result of the above modifications, observe that P^{mg} for the program of Example 3.1 is now:

$$
\begin{array}{lll}
sg^{10}(X,Y) & :- & magic\_sg^{10}(X), flat(X,Y). \\
sg^{10}(X,Y) & :- & magic\_sg^{10}(X), up(X,U), sg^{10}(U,V), down(V,Y). \\
magic\_sg^{10}(U) & :- & magic\_sg^{10}(X), up(X,U). \\
magic\_sg^{10}(john). & &
\end{array}
$$

□

3.2 Iterative Fixpoint Evaluation

We describe two refinements of ordinary bottom-up fixpoint evaluation, called Seminaive and Not-So-Naive evaluation. Seminaive fixpoint evaluation, which is cited as the method of choice

in the deductive database literature, makes the improvement that derivations are not repeated in subsequent iterations, by ensuring that in each iteration, only rule instantiations that utilize a new fact generated in the previous iteration are considered. Not-So-Naive evaluation performs a similar optimization, considering only rule instantiations that utilize a fact generated in the previous iteration. However, it does not do duplicate elimination, and so a rule instantiation may be redundantly considered if a fact generated in the previous iteration was also generated earlier. This represents a trade-off in that we avoid the cost of duplicate elimination, but face the possibility of redundant derivations. For some classes of programs, it can be shown that redundant derivations will not arise, and Not-So-Naive evaluation is then the method of choice.

We follow the presentation in [MR90] in the rest of this section, with some simplifications.

Let us first define a binary operator W_P, whose role is similar to that of the well-known T_P operator of [vEK76]:

$$W_P(X,Y) = \{h\theta \mid \begin{array}{l} h :- b_1,\ldots,b_k \text{ is a rule of } P, \\ \theta \text{ is mgu of } (b_1,\ldots,b_k) \text{ and } (d_1,\ldots,d_k), \\ Y \subseteq X, \{d_1,\ldots,d_k\} \subseteq X, \text{ and} \\ k \geq 0 \text{ and } \{d_1,\ldots,d_k\} \cap Y \neq \emptyset.\} \end{array}$$

Intuitively, W_P only allows deductions from the set of facts X that use the "new" facts Y. We now define Seminaive iteration. In the following definition, *set* is an operator that takes a multiset and returns a set, and *subs* is an operator that takes a multiset and returns an irredundant set. (An irredundant set, or *irrset*, is a set of elements such that no element subsumes another.)

Definition 3.3 Seminaive Iteration (SN)
Let $S_{-1} = S_0 = \delta_0 = \mathcal{F}$.

$$\begin{array}{ll} \delta_{n+1} & = dup\_elim(W_P(S_n, \delta_n - S_{n-1})) \\ S_{n+1} & = dup\_elim(S_n \cup \delta_{n+1}) \\ S & = \lim_{n\to\infty} S_n \\ GC & = set(S) \end{array}$$

where $\mathcal{F}$ is the set of facts, or rules with empty bodies, in the program P, and *dup_elim* is either *set* or *subs*. We refer to the variant with *dup_elim* = *set* as SN_S, and the variant with *dup_elim* = *subs* as SN_I.

In Seminaive iteration, the set of facts produced in iteration n (δ_n) is compared with the set of known facts (S_n) to identify the new facts produced ($\delta_n - S_{n-1}$). Duplicates generated within the same iteration are eliminated by the *set* operation. Only derivations that use one of these new facts are carried out in iteration $n+1$. This avoids generating many duplicate facts by avoiding repeated derivations. The algorithm terminates (at step $n+1$) when $S_{n+1} = S_n$; we must test whether $\delta_{n+1} \subseteq S_n$. Consequently, Seminaive iteration terminates if and only if S is finite.

In the following definition of Not-So-Naive iteration, note that S and δ are in general multisets. Also, we use "$S_1 \oplus S_2$" to denote the operation of adding to the multiset S_1 all elements of S_2 that do not already appear in S_1.

Definition 3.4 Not-So-Naive Iteration (NSN)
Let $S_{-1} = S_0 = \delta_0 = \mathcal{F}$.

$$\begin{aligned}
\delta_{n+1} &= W_P(S_n, \delta_n) \\
S_{n+1} &= S_n \oplus \delta_{n+1} \\
S &= \lim_{n\to\infty} S_n \\
GC &= set(S)
\end{aligned}$$

In Not-So-Naive iteration, only derivations that use one of facts produced in iteration n (δ_n) are carried out in iteration $n+1$. Note that neither the comparison with the set of previously known facts nor the *set* operation is carried out, unlike in Seminaive iteration.

The set GC is the set of *generated consequences* of the program.

Let *ground* be an operator that takes a set of possibly non-ground facts and a domain $\mathcal{D}$ and returns the set of ground facts containing only constants from $\mathcal{D}$ that are instances of the input set. The following result shows that the above iterative methods are consistent with the usual least Herbrand model semantics of [vEK76]; here implicitly $\mathcal{D}$ is the domain for the program in question. We denote the least Herbrand model of the program by $\mathcal{M}$.

Proposition 3.1 *The set of generated consequences GC of a program computed using Seminaive (SN_S or SN_I) or Not-So-Naive Iteration is such that $ground(GC) = \mathcal{M}$.*

We note that ordinary fixpoint evaluation — frequently called "Naive" (N) iteration in the deductive database literature — corresponds to the case where the second argument of W_P in the definition of Seminaive Iteration is replaced by S_n.

Example 3.3 The following program illustrates the difference between N, SN and NSN.

$$\begin{aligned}
d &\text{ :- } c. \\
c &\text{ :- } b. \\
b &\text{ :- } a. \\
a.& \\
b.&
\end{aligned}$$

Note that SN_S and SN_I behave identically on this program. Consider SN and NSN. At the second step, both evaluation strategies generate c and a second occurrence of b. SN eliminates the duplicate b. In the next step SN generates only d, since c was the only new fact found at the previous step. In contrast NSN generates both d and a second occurrence of c, since both b and c were most recently generated. The final result of the computation for each strategy is $[a, b, b, c, c, d, d]$ for NSN, and $[a, b, c, d]$ for SN_S. NSN evaluation takes one more step to terminate than SN. To see the difference with respect to N, note that NSN, unlike N, does not generate a in the second step and b in the third step. □

3.3 Related Work

We present a brief discussion of related work. The reader is referred to [BR86] and to the recent deductive database literature for more details.

The first version of the Magic Templates algorithm was introduced in [BMSU86]. As defined in that paper, it was only applicable to linear Datalog rules; it was generalized to range-restricted logic programs in [BR87]. These versions of the algorithm were called Magic Sets. It was generalized to full logic programs in [Ram88], and given the name Magic Templates. In

this paper, although many of the examples we consider could be handled by the Magic Sets version rather than the Magic Templates algorithm, we will still refer to the algorithm used as "Magic Templates."

The generalization from Magic Sets to Magic Templates utilizes non-ground tuples, and in the database context, it is desirable to restrict the computation to deal only with ground tuples. It was shown in [Ull89] that the Magic Sets transformation, in conjunction with some additional rewriting, could deal with all Datalog programs — even programs that are not range-restricted — without generating non-ground tuples.

Several other variants of the Magic Sets idea have also been proposed. For example, it is possible to compute supersets of the magic sets without compromising soundness. Although this results in some irrelevant computation, it may be possible to compute supersets more efficiently than the magic sets themselves [SS88]. Another variant is based on combining the computation for different rules by computing the union of certain relations, taking advantage of the structure of the Magic Sets transformation [HL89].

We have chosen to present the Magic Templates algorithm in detail since it can be described simply as a source-to-source program transformation, and allows for easy consideration of a number of related program transformations and fixpoint evaluation techniques. However, it should be noted that several related methods have been proposed for restricting a bottom-up computation to generate only facts relevant to the query, and much of our discussion carries over to these methods as well.

The Alexander method was proposed independently of the Magic Sets approach in [RLK86]. It is essentially the *supplementary* variant of the Magic Templates method, described in [BR87]. The main difference between the Alexander method and this variant — both of which deal only with range-restricted programs — is that the former does not utilize adornments, and is restricted to use a single, left-to-right sip for each rule, for all possible goals. The Alexander method also does not deal with function symbols, although this is easily remedied. Seki has generalized the method to deal with non-ground facts and function symbols, and has called the generalized version Alexander Templates [Sek89]. This generalization is also restricted to use a single left-to-right sip for each rule, for all possible goals.

The Magic and Alexander methods are based on program transformations. Other methods use a combination of top-down and bottom-up control to propagate bindings. Pereira and Warren presented a memoing top-down evaluation procedure based on Earley deduction [PW83]. This evaluation procedure may be viewed as a top-down evaluation procedure that incorporates memoing. Vieille has proposed a method called QSQ [Vie87, Vie86] that can be viewed as follows. Goals are generated with a top-down invocation of rules, as in Prolog. However, there are two important differences: 1) whenever possible, goals and facts are propagated set-at-a-time, and 2) all generated goals and facts are memoed. If a newly generated goal is already memoed, this is recognized by duplicate elimination.

In Alexander Templates, facts are generated bottom-up in response to previously identified goals, similar to the Magic Templates method. However, there is a significant difference with respect to the Magic Templates method as well. Not only are all generated goals and facts stored, the parent-child relationship between each goal and its immediate subgoals is also recorded. We can think of the method as generating a directed acyclic graph (dag) of goals starting with the

query goal. (This is a dag, rather than a tree, since each new goal is compared with the set of previously known goals to eliminate duplicates. This could potentially lead to incompleteness if a goal generates itself recursively, but this problem is dealt with by iterating the generation of goals and facts until no new goals and facts can be generated.) This dag structure is used to send answers back to the parent goal, and thus serves as an index. This should be contrasted with the Magic Templates method, which does not build the dag structure. The parent-child connections are essentially recovered through "joins" in the rules.

We remark that Vieille has also proposed an optimization, called *global query optimization*, that seeks to exploit the dag structure at the expense of some additional run-time overhead. An intriguing issue is how QSQ with global optimization and Magic Templates with factoring — which is an optimization described in later sections — compare. Dietrich and Warren have proposed a method called Extension Tables that is very similar to QSQ [DW87].

Finally, Kifer and Lozinskii have proposed a method called Filtering, which is based on constructing a rule-goal graph [KL86, KL88]. There is a node in the graph for each predicate, and for each rule, and arcs from predicate nodes to each rule node in whose body it appears, and from rule nodes to the predicates that they define. The idea is to compute the fixpoint by propagating tuples along these arcs, and to restrict the computation by attaching "filters" to arcs. The role of filters is similar to that of magic facts, although their generation is governed by a set of rules for propagating filter conditions through arcs, rather than a compile-time program transformation.

4 The Case for Bottom-Up Evaluation

In the past, bottom-up methods have not been seriously considered for the evaluation of logic programs because of a serious drawback: no techniques were known that avoided computing an unbounded number of irrelevant facts. Now that such techniques are known, it is worth reconsidering bottom-up approaches.

In this section we highlight some advantages that can be obtained due to bottom-up evaluation. These advantages are striking enough to suggest that in many cases bottom-up evaluation will be the method of choice; however, we reiterate that we do not claim that this will always be the case.

4.1 Declarative Semantics and Ease of Programming

A fundamental difference between Prolog evaluation and bottom-up evaluation is that the bottom-up approaches do not compromise the declarative semantics. It is frustrating and confusing to a novice Prolog programmer to discover that

$$\begin{aligned} t(X,Y) &\text{ :- } e(X,W), t(W,Y). \\ t(X,Y) &\text{ :- } e(X,Y). \end{aligned}$$

and

$$\begin{aligned} t(X,Y) &\text{ :- } t(X,W), e(W,Y). \\ t(X,Y) &\text{ :- } e(X,Y). \end{aligned}$$

do not produce the same result under Prolog.

The issue of the desirability of a declarative semantics for logic programs has been discussed at length elsewhere, and we do not have much to add here except to note that if declarative semantics can be provided with as much or greater efficiency as procedural semantics, there is little reason not to maintain the "pure" declarative semantics of the language.

4.2 Redundant Derivations and Memoing

The Prolog evaluation algorithm has the property that it does not remember which facts have already been derived. This means that if there are many ways in which a given fact can be derived, in searching for answers Prolog may redundantly search large portions of the problem space.

Since bottom-up evaluation memos all facts, it never repeats a derivation. On programs P such that Prolog multiply derives facts in P, bottom-up evaluation can provide dramatic improvements in running time.

The relationship between memoing and redundant derivations is discussed more fully in Section 8. Here, we state two propositions about classes of programs for which bottom-up evaluation is polynomial time while Prolog is exponential time.

The first class we consider is motivated by the transitive closure example. Here the problem is that for some values of the database (facts) in the program P, there are an exponential number of ways a given fact can be proven by P. The transitive closure has received a great deal of attention in the deductive database literature — in fact, the inability of relational database systems to express the transitive closure was one of the original reasons for extending database technology to provide logic-based query languages [AU79]. If logic-based languages are to succeed as database query languages, they must evaluate queries on the transitive closure efficiently.

As a demonstration of the inefficiency of Prolog on some instances of the transitive closure problem, consider the following example.

Example 4.1 Consider again the following form of the transitive closure.

$$\begin{array}{lll} t(X,Y) & :- & e(X,W), t(W,Y). \\ e(X,Y). & & \end{array}$$

and the query

$$? - t(1,Y).$$

Furthermore, suppose we wish to find all answers to the query, as is usually the case in database semantics. First, if e contains cycles, Prolog will not terminate in its search for all answers. However, even without cycles, there are sets of facts on which Prolog is extremely inefficient. For example, suppose that e contains the facts

$$\begin{array}{cccc} e(1,2). & e(1,3). & e(2,4). & e(3,4). \\ \vdots & \vdots & \vdots & \vdots \\ e(4n-3,4n-2). & e(4n-3,4n-1). & e(4n-2,4n). & e(4n-1,4n). \end{array}$$

In searching for all answers Prolog's backtracking strategy produces an $\Omega(2^n)$ evaluation procedure. Note that it is not essential that one search for all answers at the top level in order to see this inefficiency — if the t goal occurs in some rule to the left of a goal that cannot be satisfied, the same behavior results even under single answer semantics. □

If the transitive closure is rewritten using the Magic Templates transformation, we obtain the program

$$\begin{array}{lll} m(1). & & \\ m(W) & :- & m(X), e(X,W). \\ & & \\ t(X,Y) & :- & m(X), e(X,W), t(W,Y). \\ t(X,Y) & :- & m(X), e(X,Y). \end{array}$$

One can verify that evaluating the resulting program bottom-up by the Seminaive method results in a polynomial time algorithm.

The following definition generalizes the previous example.

Definition 4.1 A program P and query q is *right-linear* if it is of the form

$$\begin{array}{lll} t(\overline{X},\overline{Y}) & :- & \mathcal{L}(\overline{X},\overline{U},\overline{W}), t(\overline{W},\overline{Y}) \\ t(\overline{X},\overline{Y}) & :- & \mathcal{E}(\overline{X},\overline{Y}). \end{array}$$

where $\overline{X}$, $\overline{Y}$, $\overline{W}$, and $\overline{U}$ are vectors of variables, $\mathcal{L}$ and $\mathcal{E}$ are conjunctions made up of predicates that are not mutually recursive with t, and q binds the columns of t corresponding to the vector of variables X.

Proposition 4.1 *Let P be a right-linear Datalog recursion defining a predicate t, and suppose that $\mathcal{E}$ and $\mathcal{L}$ are satisfiable. Then there exists a database for P of size n such that Prolog is $\Omega(2^n)$ on P.*

Proof Consider the following database for P

$$\begin{array}{cccc} \mathcal{L}(\overline{c_1},\overline{c_2}). & \mathcal{L}(\overline{c_1},\overline{c_3}). & \mathcal{L}(\overline{c_2},\overline{c_4}). & \mathcal{L}(\overline{c_3},\overline{c_4}). \\ \vdots & \vdots & \vdots & \vdots \\ \mathcal{L}(\overline{c_{4n-3}},\overline{c_{4n-2}}). & \mathcal{L}(\overline{c_{4n-3}},\overline{c_{4n-1}}). & \mathcal{L}(\overline{c_{4n-2}},\overline{c_{4n}}). & \mathcal{L}(\overline{c_{4n-1}},\overline{c_{4n}}). \end{array}$$

$$\mathcal{B}(\overline{c_{4n}},\overline{c_{4n}}).$$

and let the query q be

$$q(\overline{Y}) \quad :- \quad t(\overline{c_1},\overline{Y}).$$

Then there are $\Omega(2^n)$ derivations of $t(\overline{c_1},\overline{c_{4n}})$, and Prolog will search each one. □

Note that since bottom-up evaluation is polynomial time on any Datalog program, it is polynomial on right-linear programs P.

Another class of programs for which bottom-up computation is polynomial time while Prolog is exponential is exemplified by the standard definition of the Fibonacci numbers.

Example 4.2 . Consider the following definition $fib(N, X)$, where X is the Nth Fibonacci number.

$$
\begin{array}{l}
fib(0,1). \\
fib(1,1). \\
fib(N, X_1 + X_2) \quad :- \quad N > 1, fib(N-1, X_1), fib(N-2, X_2).
\end{array}
$$

On this program, Prolog makes $\Omega(\phi^n + \hat{\phi}^n)$ calls to *fib*, where $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2$. If we rewrite using Magic Templates, we get

$$
\begin{array}{lll}
m\_fib(N). & & \\
m\_fib(N-1) & :- & m\_fib(N), N > 1. \\
m\_fib(N-2) & :- & m\_fib(N), N > 1.
\end{array}
$$

$$
\begin{array}{l}
fib(0,1). \\
fib(1,1). \\
fib(N, X_1 + X_2) \quad :- \quad m\_fib(N), N > 1, fib(N-1, X_1), fib(N-2, X_2).
\end{array}
$$

which can be evaluated in polynomial time by Seminaive iteration. □

The cause of this type of redundancy is different than that exhibited on right-linear rules. Here, while there may be only one derivation tree for any fact implied by the program, that single derivation tree contains the same subtree replicated an exponential number of times.

One common situation in which this form redundancy occurs is in programs with a recursive rule with two recursive subgoals such that in any derivation of a sufficiently large fact, the subtrees rooted at these recursive subgoals contain the same subgoal. In the following, we consider programs in which the head contains a term T, and the two recursive subgoals in the body contain the terms $c(T)$ and $d(T)$, where c and d are functions that reduce T to subterms of smaller size. (We will return to this class of programs in Section 8, where c and d are called "descent conditions," after [Coh83].)

Definition 4.2 Let r be a recursive definition of the form

$$
\begin{array}{lll}
p(T,R) & :- & \neg \mathcal{B}(T), p(c(T), R_1), p(d(T), R_2), \mathcal{R}(R_1, R_2, R). \\
p(T, r_0) & :- & \mathcal{B}(T).
\end{array}
$$

Then r is *n-callable* on a term t if

- For $i \geq n$, $\mathcal{B}(c^i(t))$ and $\mathcal{B}(d^i(t))$ are true, and
- For $0 \leq i < n$, $c^i(t)$ and $d^i(t)$ are well-defined and at least one of $\mathcal{B}(c^i(t))$ and $\mathcal{B}(d^i(t))$ is false, and
- For any values of R_1 and R_2, there is a value of R such that $\mathcal{R}(R_1, R_2, R)$ holds.

For example, the Fibonacci program is $(n-2)$-callable on any positive integer n.

Proposition 4.2 *Let r be the recursion*

$$
\begin{array}{lll}
p(T,R) & :- & \neg \mathcal{B}(T), p(c(T), R_1), p(d(T), R_2), \mathcal{R}(R_1, R_2, R). \\
p(T, r_0) & :- & \mathcal{B}(T).
\end{array}
$$

Suppose that there exist constants k_1 and k_2 such that for any sufficiently large term t we have that $c^{k_1}(t) = d^{k_2}(t)$, and suppose that there is a constant a such that for any term t of size an, the recursion r is n-callable. Then the number of subgoal calls made by Prolog in the evaluation of the goal $p(t, R)$ is exponential in the size of t.

Proof The number subgoal calls made by Prolog in evaluating r on an n-callable term is given by the recurrence

$$\begin{aligned} T(n) &= T(n - c_1) + T(n - d_1); \\ T(1) &= 1. \end{aligned}$$

where c_1 and c_2 are constants such that if t is of size n, $c(t)$ is of size $n - c_1$ and $d(t)$ is of size $n - d_1$. If we set $k = \max(c_1, d_1)$, we can write

$$\begin{aligned} T'(n) &= 2T'(n - k); \\ T'(1) &= 1. \end{aligned}$$

$$T(n) \geq T'(n).$$

Since $T'(n) = \Omega(2^n)$, the proposition is proved. □

Note that this implies that on any program satisfying the conditions of Proposition 4.2, Prolog is exponential time in the size of the term T, whereas bottom-up evaluation is polynomial.

4.3 Optimizations Made Possible by the Declarative Semantics

One of the most important benefits of a purely declarative semantics is that it allows powerful optimizations. A typical Prolog optimizer has to ensure not only that the optimized program defines the same set of facts as the original program, but also that the Prolog interpreter has the same behavior on both programs. That is, both programs must produce the answers in the same order, and, more importantly, the interpreter cannot terminate on the original program yet go into an infinite loop on the optimized program. These restrictions severely limit the optimizations that can be considered.

One example of a powerful optimization made possible by the declarative semantics of bottom-up is *factoring* [NRSU89a]. For example, Prolog is $O(\Omega(2^n))$ on the transitive closure program, and Magic Templates is $O(\Omega(n^2))$; factoring the Magic program results in a program whose Seminaive evaluation computes only $O(n)$ facts. This is typical of the improvement over Magic Templates when the factoring optimization applies. Factoring and several other program optimizations that are made possible by adopting declarative semantics are discussed in Section 5.

4.4 The Down-Side of Bottom-Up

As we've tried to illustrate in this section, bottom-up evaluation can in many cases provide orders of magnitude improvements in efficiency over Prolog. However, there are also drawbacks associated with evaluating bottom-up.

When only one answer is required, the deterministic control strategy of Prolog may allow the programmer to order the search so as to minimize the time taken to find the first answer; this is not possible with bottom-up evaluation unless some additional facility for specifying control is provided. Providing such a facility — for example, in the form of regular expressions that determine the order of rule applications in each iteration — could, in general, compromise the completeness of bottom-up evaluation. (Of course, Prolog's depth-first strategy is not complete either.)

This must be tempered by several factors. First, the search for a single answer at one level often requires searching for all answers at a lower level in the computation tree. Second, the explicit use of Prolog's control strategy to direct the search requires that the programmer think in terms of the Prolog implementation, rather than the declarative semantics of the program. Further, the search space may be such that depth-first search does more work to find the first answer than a breadth-first search (which is typical of bottom-up evaluation). Third, optimization strategies such as *projection pushing*, discussed in Section 5.4, improve the performance of bottom-up evaluation significantly for some single-answer queries.

A perhaps more important issue to consider is the following. While it is known that the approach of Magic Templates rewriting plus Seminaive bottom-up evaluation is never worse than Prolog by more than a constant factor [Ram88, Ull89] (under the assumption of constant time table lookup operations), the constant factors in the running time for bottom-up evaluation may be larger than the associated constant factors in the top-down Prolog evaluation of the same program.

As one example, consider the transitive closure program; each time Prolog applies the recursive rule, it "remembers" the current instantiation of $e(X, W)$ on the run-time stack. Suppose, for example, that Prolog has the instantiations $e(1,2)$, $e(2,3)$, ..., $e(n-2, n-1)$ on the stack, then applies the nonrecursive rule to deduce $t(n-1, n)$. Now in order to infer $t(i, n)$, for $1 \leq i < n$, Prolog can just read the e facts off the stack.

The situation in bottom-up evaluation is different — instead of reading the corresponding e facts off the stack, we must look up the e facts in relation e. Given appropriate indexing, this overhead can be minimized, but it is still there. The problem is worse in programs that compute terms built up of function symbols, where indexing is not as straight forward.

If bottom-up evaluation is asymptotically better than Prolog, the constants are not significant as larger and larger instances of the problem are considered. Even if c_1 is much larger than c_2,the parameter n does not have to be very large before $c_1 n^2$ is smaller than $c_2 2^n$. Also, as bottom-up evaluation technology matures, we can expect the difference in the constants in the asymptotic running time expressions for Prolog and bottom-up evaluation to grow smaller. However, it is probable that there will be significant classes of programs — for example, those that deterministically manipulate structures such as lists and trees — for which the constants result in faster Prolog execution.

An especially significant source of overhead in bottom-up evaluation is the duplicate elimination that is done on every iteration of Seminaive evaluation. An interesting new area of research deals with when this duplicate elimination can itself be eliminated. In this paper we review early results in this direction; the interesting general trend is that in cases where Prolog can be expected to perform well, duplicate elimination is not necessary. We remark that the

need for duplicate elimination is not peculiar to bottom-up evaluation; for example, Prolog can be made complete by adding loop-detection, which is essentially duplicate elimination on the set of goals that are generated.

5 Some Important Program Optimizations

In this section, we survey a number of optimizations that apply to positive Horn programs and guarantee query equivalence in the least fixpoint model. That is, the same set of answers according to the declarative semantics of logic programs is computed. Note that no further guarantees are offered: no equivalence is guaranteed with respect to other program predicates, nor is any ordering preserved in the generation of answers.

The optimizations considered here are the *Counting* algorithm, the *factoring* optimization, techniques for deleting redundant rules and literals, and techniques by which "existential" queries (queries for which a single answer — any answer — suffices) can be optimized. These optimizations, like the Magic Templates optimization, attempt to rewrite the program so that the computation of the fixpoint of the rewritten program is more efficient than the computation of the fixpoint of the original program. The Counting algorithm may be viewed as an alternative to the Magic Templates algorithm that is less generally applicable, but that sometimes performs better. We present a detailed comparison in Section 7.1. Factoring is a program optimization that is often applicable to programs that are generated by the Magic Templates algorithm; it always improves the program when it applies. The other optimizations are independent of the Magic Templates algorithm, and can be used freely in conjunction with it; these optimizations also have the desirable property that they always improve the program if they are applicable.

This survey is not intended to be comprehensive, but rather to provide an indication of the flexibility that is available when we only need to preserve the declarative semantics. Ensuring that the declarative semantics is preserved has long been viewed as a bar to efficient evaluation; the results surveyed here indicate that it can sometimes be considerably more efficient to guarantee this semantics, rather than an operational semantics.

5.1 Counting

Counting is a refinement of the Magic Set approach. Whereas the Magic Set method restricted the computation to relevant facts, Counting additionally computes indices for each fact that indicate why it is relevant, and this additional information is used to delete some literals from rule bodies and to reduce the arity of some predicates by deleting some argument positions.

Counting was originally proposed in [BMSU86], and was refined in [SZ86]. As in the case of Magic Sets, these versions of the algorithm only applied to (programs containing only) range-restricted linear rules; the algorithm was generalized to deal with all range-restricted rules in [BR87].

Counting can be understood as a two-step refinement of Magic Sets. Syntactically, the version presented in [BR87], called Generalized Counting (GC), simply adds some index fields to predicates in the Magic program and is applicable to any logic program. The additional

indices enable a further optimization, called the *semijoin optimization*[2], for some programs. Since the indices require additional computation, the method is obviously useful only when this optimization applies, and is thus only useful for a proper subset of the programs on which Magic Sets can be used. The applicability is further limited because the computation of the indices may sometimes cause non-termination.

We refer the reader to [BR87] for a detailed presentation of the method, and present only the intuition behind the index fields. We also sketch the semijoin optimization, and present an illustrative example.

Recall that we limit ourselves in this paper to a left-to-right choice of sips. We can therefore think in terms of (a loop-checking, complete version of) Prolog executing the adorned program P^{ad} in order to understand how goals are generated. Thus, for every goal that is generated, there is a chain of goals such that the following holds: The first goal is the original query, and each goal is generated by instantiating a literal in the body of a rule; the rule must have been invoked by unifying the head literal with a predecessor goal. In each goal that is generated, we include an index value that essentially encodes the chain associated with it. Let goal G unify with the head of rule r, and solutions to the first $k-1$ body literals instantiate the kth literal to generate goal G1. Let the index value I be associated with goal G. The index value for G1 is essentially the pair (r,k) concatenated to I. The index value for the original query is simply "0".

Goals correspond to magic facts, as we noted earlier. In the Counting program, magic facts are called *count* facts. The index value for a non-count fact is simply the index value of the goal for which it was generated as an answer.

Index values can be encoded in various ways; we will not consider the details.

We now explain the semijoin optimization, for the case of left-to-right sips, and an adorned program that contains a single recursive rule. Consider an occurrence $p^a(\bar{t})$ of the recursive predicate in the body of the recursive rule. Suppose that the following holds for every such occurrence:

1. If a variable appears in a bound argument position of $p^a(\bar{t})$, it does not appear to the right of $p^a(\bar{t})$, or in a free argument position of the head literal.

2. If a variable appears in a free argument position of the head literal or in a body literal to the left of $p^a(\bar{t})$, it does not appear to the right of $p^a(\bar{t})$, or in a free argument position of the head literal.

Then, the following optimization can be applied to the Generalized Counting program:

1. The arity of p^a is reduced by deleting the bound argument positions.

2. All body literals to the left of the right-most occurrence of p^a in the recursive rule are deleted.

[2]This name is unfortunate; there is an eponymous well-known database optimization, and while there is some similarity in the intuition — hence the name — the two are distinct.

Example 5.1 Consider the program of Example 3.2 again. Before the semijoin optimization, the Counting algorithm generates:

$$
\begin{array}{lll}
sg^{10}(X,Y,I) & :- & count\_sg^{10}(X,I), flat(X,Y). \\
sg^{10}(X,Y,I) & :- & count\_sg^{10}(X,I), up(X,U), sg^{10}(U,V,I+1), down(V,Y). \\
count\_sg^{10}(U,I+1) & :- & count\_sg^{10}(X,I), up(X,U). \\
count\_sg^{10}(john,0). & &
\end{array}
$$

Note that the index values have been encoded in a simple way that takes advantage of the fact that there is a single, linear recursive, rule; thus, we need only record the number of goals in the chain from the query goal. The semijoin optimization applies, and the resulting program is:

$$
\begin{array}{lll}
sg^{10}(Y,I) & :- & count\_sg^{10}(X,I), flat(X,Y). \\
sg^{10}(Y,I) & :- & sg^{10}(V,I+1), down(V,Y). \\
count\_sg^{10}(U,I+1) & :- & count\_sg^{10}(X,I), up(X,U). \\
count\_sg^{10}(john,0). & &
\end{array}
$$

□

Unless otherwise stated, we assume that the semijoin optimization is applicable and has been performed, and in the sequel, simply refer to the Generalized Counting program with this optimization as the Counting program P^{cnt}.

5.2 Predicate Factoring

The basic idea behind predicate factoring is to replace a predicate by two predicates of strictly smaller arity. We present a simplified description that is tailored to the case of programs obtained by applying the Magic Templates algorithm described in Section 3.1. Our presentation is through the use of examples, and we do not describe sufficient conditions for the optimization to apply in general. We refer the reader to [NRSU89a] for a detailed treatment.

In essence, we seek to take advantage of the magic predicates to replace the original predicate in P^{mg} by its projection onto its 0 argument positions. Thus, the original (non-magic) predicate in P^{mg} is factored into the magic predicate, which computes the 1 argument positions, and this new predicate, which computes the 0 argument positions.

Example 5.2 We begin with a familiar example, transitive closure. While efficient algorithms are known, the rewriting algorithms presented in [NRSU89b] were the first to automatically derive unary programs for single-selection queries, for all three forms (left-linear, right-linear, non-linear) of the recursive rule. We achieve the same result here by first applying the Magic Templates transformation and then factoring the rewritten program. To illustrate the technique, we consider a single program that includes all three forms of the recursive rule, although any one would suffice. It should be evident that we would also obtain a unary program if the original program contained just one of the recursive rules. Consider the program and single-selection

query below:

$$\begin{array}{lll} t(X,Y) & :- & t(X,W), t(W,Y). \\ t(X,Y) & :- & e(X,W), t(W,Y). \\ t(X,Y) & :- & t(X,W), e(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(5,Y). \end{array}$$

The Magic Templates algorithm rewrites this to:

$$\begin{array}{lll} m\_t^{10}(W) & :- & m\_t^{10}(X), t^{10}(X,W). \\ m\_t^{10}(W) & :- & m\_t^{10}(X), e(X,W). \\ m\_t^{10}(5). & & \\ t^{10}(X,Y) & :- & m\_t^{10}(X), t^{10}(X,W), t^{10}(W,Y). \\ t^{10}(X,Y) & :- & m\_t^{10}(X), e(X,W), t^{10}(W,Y). \\ t^{10}(X,Y) & :- & m\_t^{10}(X), t^{10}(X,W), e(W,Y). \\ t^{10}(X,Y) & :- & m\_t^{10}(X), e(X,Y). \\ query(Y) & :- & bt(5), ft(Y). \end{array}$$

If we identify m_t^{10} tuples with goals in a top-down evaluation, we see that only the last occurrence of t^{10} in a rule body generates new goals, and further, the answer to a new goal is also an answer to the goal that invoked the rule. In fact, every answer to a subgoal is also an answer to the query goal m_t^{10}. A second observation is that if c is generated as an answer to a subgoal, then a new subgoal $m\_t^{10}(c)$ is also generated. These observations lead us to conclude that it does not matter to which subgoal an answer corresponds; its role in the computation is the same in any case. That is, $t^{10}(X,Y)$ can be factored into $bt(X)$ and $ft(Y)$ in the Magic program. Doing this yields:

$$\begin{array}{lll} m\_t^{10}(W) & :- & m\_t^{10}(X), bt(X), ft(W). \\ m\_t^{10}(W) & :- & m\_t^{10}(X), e(X,W). \\ m\_t^{10}(5). & & \\ bt(X) & :- & m\_t^{10}(X), bt(X), ft(W), bt(W), ft(Y). \\ bt(X) & :- & m\_t^{10}(X), e(X,W), bt(W), ft(Y). \\ bt(X) & :- & m\_t^{10}(X), bt(X), ft(W), e(W,Y). \\ bt(X) & :- & m\_t^{10}(X), e(X,Y). \\ ft(Y) & :- & m\_t^{10}(X), bt(X), ft(W), bt(W), ft(Y). \\ ft(Y) & :- & m\_t^{10}(X), e(X,W), bt(W), ft(Y). \\ ft(Y) & :- & m\_t^{10}(X), bt(X), ft(W), e(W,Y). \\ ft(Y) & :- & m\_t^{10}(X), e(X,Y). \\ query(Y) & :- & bt(5), ft(Y). \end{array}$$

Applying some simple syntactic optimizations, which are discussed in [NRSU89a], we finally obtain the following unary program:

$$\begin{array}{lll} m\_t^{10}(W) & :- & ft(W). \\ m\_t^{10}(5). & & \\ ft(Y) & :- & m\_t^{10}(X), e(X,Y). \\ query(Y) & :- & ft(Y). \end{array}$$

□

The above example is illustrative of a general approach to optimizing programs, in which we first apply the Magic Templates transformation and then factor. When we factor a Magic program and separate the 1 and 0 arguments, we can no longer associate each subgoal with its answers. In effect, we compute a set of goals and a set of facts such that each is an answer to some goal. To show that this optimization is applicable, we must establish two things:

- Every answer to a subquery is also an answer to the original query.
- No spurious subqueries or answers are generated.

We remark that the factoring technique is applicable to logic programs, although the sufficient conditions presented in [NRSU89a] are only applicable to Datalog programs. To deal with logic programs, we must first transform them into Extended Datalog programs, that is, Datalog programs with infinite base relations. This allows us to use the sufficient conditions of [NRSU89a].

The following algorithm takes as input a program P and produces as output an Extended Datalog program P^{ext}.

Definition 5.1 (Extended Datalog Program) Let P be a Datalog program. Then the corresponding Extended Datalog Program is constructed from P by doing the following for every rule r in P and for every term t in r.

- Replace every occurrence of t in r by a new variable, say X.
- Let t contain n distinct variables and constants. Then, add a literal with predicate name r_t and $n+1$ arguments to the body of r. The first n arguments are the variables and constants of t, and the last argument is the variable X. These new predicates all denote base relations (with a possibly infinite number of tuples in them).

It is easy to extend the proofs of [NRSU89a] to show that if the sufficient conditions for factorability are satisfied by P^{ext}, then P can also be factored.

Example 5.3 Suppose we want to find all postfixes of a list. The following Prolog program is a straight forward way to do so.

$$\begin{array}{l} pf(X,X). \\ pf([H|T],T1) \ \text{:-} \ \ pf(T,T1). \end{array}$$

and suppose the query is $pf([x_1,x_2,...,x_n],Y)$?. Prolog, if asked to compute all answers, and Magic Templates without factoring will establish the facts $pf([x_i,...,x_n],[x_j,...,x_n])$ for $1 \leq i \leq n$ and $i \leq j \leq n$. Thus, Prolog and Magic Templates establish n^2 facts.

Now consider the Extended Datalog version of the program:

$$\begin{array}{l} pf(X,X). \\ pf(X,Y) \ \ \text{:-} \ \ list(H,T,X), pf(T,Y). \end{array}$$

where $list(H,T,X)$ is true if X is the list $[H|T]$. Applying Magic Templates to the query $pf([x_1,...,x_n],Y)$, we get:

$$
\begin{array}{lll}
m\_pf^{10}([x_1,...,x_n]). & & \\
m\_pf^{10}(T) & :- & m\_pf^{10}(X), list(H,T,X). \\
 & & \\
pf^{10}(X,Y) & :- & m\_pf^{10}(X), eq(X,Y). \\
pf^{10}(X,Y) & :- & m\_pf^{10}(X), list(H,T,X), pf^{10}(T,Y).
\end{array}
$$

Now we factor to obtain:

$$
\begin{array}{lll}
m\_pf^{10}([x_1,...,x_n]). & & \\
m\_pf^{10}(T) & :- & m\_pf^{10}(X), list(H,T,X). \\
 & & \\
bpf(X) & :- & m\_pf^{10}(X), eq(X,Y). \\
fpf(Y) & :- & m\_pf^{10}(X), eq(X,Y). \\
 & & \\
bpf(X) & :- & m\_pf^{10}(X), list(H,T,X), bpf(T), fpf(Y). \\
fpf(Y) & :- & m\_pf^{10}(X), list(H,T,X), bpf(T), fpf(Y).
\end{array}
$$

Optimizing gives

$$
\begin{array}{lll}
m\_pf^{10}([X_1,...,X_n]). & & \\
m\_pf^{10}(T) & :- & m\_pf^{10}(X), list(H,T,X). \\
 & & \\
bpf(X) & :- & m\_pf^{10}(X), eq(X,Y). \\
fpf(Y) & :- & m\_pf^{10}(X), eq(X,Y). \\
 & & \\
bpf(X) & :- & m\_pf^{10}(X), list(H,T,X), bpf(T), fpf(Y).
\end{array}
$$

(This last rule can be further optimized.) Evaluating this program bottom-up will compute the answer to the query, but only establishes $O(n)$ facts. □

Another example is the program:

$$
\begin{array}{lll}
member(X,[X|T]). & & \\
member(X,[H|T]) & :- & member(X,T).
\end{array}
$$

and the query $member(X,[x_1,...,x_n])$? Prolog and Magic Templates without factoring compute $member(x_j,[x_i,...,x_n])$ for $1 \leq i \leq n$ and $i \leq j \leq n$, which is n^2 facts, whereas Magic Templates with factoring only computes $O(n)$ facts.

5.3 Bounded Recursion and Redundant Literals

In many cases providing a purely declarative semantics allows for optimizations much more far-reaching than would be possible under a procedural semantics. One interesting optimization that has been studied in some detail by the theoretical database community is that of *bounded*

recursion, where a recursive program is bounded be if it can be replaced by a nonrecursive program.

Note that here we are not asking when recursion can be replaced by iteration; rather, we are asking when a logic program containing recursive clauses has an equivalent finite logic program in which no clause is recursive.

Example 5.4 The following example is from [Nau89a].

$$\begin{aligned} buys(X,Y) &\text{ :- } likes(X,Y). \\ buys(X,Y) &\text{ :- } trendy(X), buys(Z,Y). \end{aligned}$$

In English, a person X buys a product Y if either X likes Y, or X is trendy and a person Z has bought Y. This recursive program is equivalent to the following nonrecursive program:

$$\begin{aligned} buys(X,Y) &\text{ :- } likes(X,Y). \\ buys(X,Y) &\text{ :- } trendy(X), likes(Z,Y). \end{aligned}$$

□

In general, detecting bounded recursions is undecidable. A rich classification of sets of programs for which detecting boundedness is decidable or undecidable has been developed in the literature; see, for example, [CGKV88, Ioa86, GMSV87, Nau89a, NS87, Var88].

A related question is when a literal is redundant in a given program. Naughton [Nau89b] defined a predicate p appearing in a clause C to be *recursively redundant* if, for any set of base facts, there is a constant k such that any fact provable by that clause has a derivation tree in which there are at most k instances of p. If a predicate is recursively redundant, then the program can be rewritten so that no instance of that predicate appears in any recursive rule.

Example 5.5 The following example is taken from [Nau89b].

$$\begin{aligned} buys(X,Y) &\text{ :- } likes(X,Y). \\ buys(X,Y) &\text{ :- } knows(X,W), buys(W,Y), cheap(Y). \end{aligned}$$

Here the predicate *cheap* is recursively redundant; an equivalent program is

$$\begin{aligned} buys1(X,Y) &\text{ :- } likes(X,Y). \\ \\ buys2(X,Y) &\text{ :- } knows(X,W), likes(W,Y), cheap(Y). \\ buys2(X,Y) &\text{ :- } knows(X,W), buys2(W,Y). \\ \\ buys(X,Y) &\text{ :- } buys1(X,Y). \\ buys(X,Y) &\text{ :- } buys2(X,Y). \end{aligned}$$

□

In related work, Sagiv [Sag88] defined a literal to be redundant if the semantics of the program are unchanged by deleting the literal. His algorithm for detecting redundant literals is interesting in that it is based on the *chase* procedure for inferring data dependencies. More detail about detecting and eliminating redundancy from recursive Datalog programs appears in [NS89].

5.4 Projecting Arguments

Query optimization for relational database queries exploits the commutativity of *selection* and *projection* operators with respect to the *join* operator whenever possible, in order to reduce the size of relations that are being joined. This is often referred to as "pushing" selections and projections. Pushing selections is achieved through the use of the Magic Templates transformation; the introduction of recursion makes it necessary to compute auxiliary sets. Pushing projections has also been explored, and the gains can be significant. We will illustrate the idea through examples, and refer the reader to [RBK88] for details.

Example 5.6 Consider the transitive closure program of Example 1.1, but with the query $query(Y)$:- $t(\_, Y)$. The underscore "_" indicates that we do not care about the value in the first argument position; we simply want the set of values that appear in the second argument position of t (with some arbitrary value in the first argument position). Note that such queries are likely to arise during the course of query optimization. □

An adornment that distinguishes don't-care argument positions (d) from the rest (needed or n) was used in [RBK88] to push projections. An adorned program is generated using an algorithm similar to that described in Section 3.1. The different nature of the adornments is reflected in how we determine adornments for body literals from the adornment for the rule head: In choosing an adornment for a body literal, an argument position is d if it contains only a variable that does not appear anywhere else in the rule, except possibly as the argument in a d position of the head; all other argument positions are n.

Example 5.7 Consider a simplified version of the transitive closure program:

$$\begin{array}{lll} t(X,Y) & :- & t(X,W), e(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(\_,Y). \end{array}$$

The adorned program — using n and d adornments — is:

$$\begin{array}{lll} t^{dn}(X,Y) & :- & t^{dn}(X,W), e(W,Y). \\ t^{dn}(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t^{dn}(\_,Y). \end{array}$$

It is easy to see that the d argument positions can be uniformly deleted, and this leaves us with a program in which the recursive predicate is unary. □

The previous example illustrated how the adornment algorithm can sometimes push the projection through recursion and thereby reduce the arity of recursive predicates. (The observation that pushing projections could reduce arity of recursive predicates was first made by Aho and Ullman [AU79], and later Kifer and Lozinskii [KL86]. The adornment algorithm above generalizes their results.) The acute reader will have observed that more can be achieved — the recursive rule may be deleted entirely.

Algorithms for deleting redundant rules and literals can be utilized to detect this; in fact, since such opportunities are frequently created by pushing projections, we can devise special algorithms for rule and literal deletion that exploit this. The following example illustrates the power of the techniques developed in [RBK88] for this purpose.

Example 5.8 Consider the transitive closure program of Example 1.1 with the query

$$query(Y) \text{ :- } t(\_, Y).$$

The adorned program — using n and d adornments, and deleting d argument positions — is:

$$\begin{array}{lll}
t^{dn}(Y) & \text{:-} & t^{dn}(W), t^{nn}(W,Y). \\
t^{dn}(Y) & \text{:-} & e(X,W), t^{nn}(W,Y). \\
t^{dn}(Y) & \text{:-} & t^{dn}(W), e(W,Y). \\
t^{dn}(Y) & \text{:-} & e(X,Y). \\
 & & \\
t^{nn}(X,Y) & \text{:-} & t^{nn}(X,W), t^{nn}(W,Y). \\
t^{nn}(X,Y) & \text{:-} & e(X,W), t^{nn}(W,Y). \\
t^{nn}(X,Y) & \text{:-} & t^{nn}(X,W), e(W,Y). \\
t^{nn}(X,Y) & \text{:-} & e(X,Y). \\
 & & \\
query(Y) & \text{:-} & t^{dn}(Y).
\end{array}$$

Notice that the adorned program contains more rules than the original program. (This could happen with 1 and 0 adornments as well.) The additional information in the adornments can, however, be used to delete several of the rules. The following program is finally obtained using the techniques of [RBK88]:

$$\begin{array}{lll}
t^{dn}(Y) & \text{:-} & e(X,Y). \\
query(Y) & \text{:-} & t^{dn}(Y).
\end{array}$$

□

5.5 Linearizing Programs

An interesting class of program transformations has recently been explored by a number of researchers [IW88, ZYT, Sar89, RSUV89]. The objective is to transform a program that contains non-linear rules into an equivalent one that contains only linear rules; this may make some of the other transformations surveyed in this paper applicable, or permit simplifications in the implementation of the fixpoint evaluation phase. We do not consider these results here due to space limitations.

6 Derivation Trees and Refinements of Fixpoint Evaluation

In Section 5 we discussed optimizing program transformations that seek to produce a transformed program whose fixpoint can be computed more efficiently than the fixpoint of the original program. In this section we consider properties of programs that allow optimizations in the actual computation of the fixpoint.

The set of derivation trees for a program can be used to abstract and analyze many operational aspects of program execution, and is referred to extensively in this section. (Recall that derivation trees were defined in Section 2.) Foremost among these are duplicate and redundant

derivations, and an analysis of the set of derivation trees can reveal a variety of potential optimizations. Examples of such optimizations include elimination of checks for duplicate facts and identification of properties such as commutativity and linearizability of rules. In addition, even if such strong properties do not hold for a given program, it may be possible to avoid many redundant derivations through the judicious use of control during fixpoint evaluation. These issues are discussed in more detail in the following sections.

6.1 Duplicate Freedom and Finite Forest Properties

We observe that the collection of derivation trees for a program is in general a multiset if we admit the possibility of the same rule (including rules with empty bodies, i.e. facts) appearing twice in the original program. Let P be a program and let a be a fact. The multiset of all derivation trees for a in P is denoted by $DT(P,a)$. The multiset of all derivation trees in P is $DT(P) = \bigcup_a DT(P,a)$.

Let $atoms(t)$ be the label on the root of derivation tree t, and let us extend this to multisets of trees by $atoms(X) = [atoms(x) \mid x \in X]$. (We use $[\ldots]$ as the multiset constructor.) The function $atoms$ abstracts the generated atoms from the derivation trees that generate them.

Definition 6.1 A program P is *subsumption-free* if $atoms(DT(P))$ is an irrset. A program P is *duplicate-free* if $atoms(DT(P))$ is a set.

Example 6.1 Consider the following program.

$$\begin{array}{lll} p(X,Y) & :- & X = 5. \\ p(X,Y) & :- & Y = 5. \end{array}$$

This program is subsumption-free. However, the following program is not:

$$\begin{array}{lll} q(X) & :- & p(X,Y), X = 5, Y = 5. \\ p(X,Y) & :- & X = 5. \\ p(X,Y) & :- & Y = 5. \end{array}$$

□

The following result, from [MR90], shows that NSN evaluation — which performs no expensive check for duplicates — performs as well as SN evaluation for programs that are subsumption-free.

Theorem 6.1 ([MR90]) *In terms of the number of inferences,*

1. SN_I = *NSN for subsumption-free programs* P.
2. SN_S = *NSN for duplicate-free programs* P.

A multiset has *finite character* if every element has finite multiplicity. A multiset has the *finite subsumption* property if it has finite character and every element subsumes only finitely many elements. A program P has the *finite forest* property if $atoms(DT(P))$ has finite character. A program P has the *finite subsumption* property if $atoms(DT(P))$ has the finite subsumption property.

Note that the following trivial rule in a program would destroy the finite subsumption and forest properties whenever the relation p in $M = ground(GC)$ is non-empty: $p(X) :- p(X)$.

These properties are important for termination.

Theorem 6.2 ([MR90]) *1. If SN_I terminates, then NSN terminates, for programs P with the finite subsumption property.*

2. If SN_S terminates, then NSN terminates, for programs P with the finite forest property.

Sufficient conditions for subsumption-freedom are presented in [MR90]. These conditions seem to apply to a large class of Prolog programs; intuitively, it is required that no two rules produce the same fact, and that the evaluation of a rule be deterministic. In [MFPR89], it is shown that if the original program is subsumption-free, this property can be utilized effectively in the bottom-up evaluation of the program rewritten according to the Magic Templates algorithm. More precisely, it is shown that the rewritten program can be evaluated in a particular way — that essentially differs from NSN in that subsumption checks are performed on all magic predicates — that computes the same set of derivation trees as NSN evaluation of the original program. Thus, if the original program is subsumption-free, no subsumption checks need be performed on any non-magic predicates in the rewritten program.

6.2 Determinacy and Functional Goals

The determinism required by the sufficient conditions for subsumption-freedom can be formulated in terms of functional dependencies [MR90]. An important class of such dependencies are *functional goals*, that is, goals for which there is a single answer. Sufficient conditions for a goal to be functional have been developed in [Red84, DW89]. Such goals can be optimized in a number of ways; for example, backtrack points need not be maintained in a Prolog-style evaluation. Indeed, if only the declarative semantics is to be preserved, it may be possible to simply apply evaluation techniques for functional programs, rather than logic programs. We remark that although functionality was only considered for programs that generated ground facts in [Red84, DW89], the techniques in [MR90] — developed in the somewhat different context of establishing sufficient conditions for subsumption-freedom — enable us to identify functional computations that generate non-ground goals and facts.

6.3 Algebraic Properties of Programs

The fixpoint evaluation of a logic program can be refined by taking certain algebraic properties of the program into consideration. Such refinements, and techniques for detecting when they are applicable, have been investigated by several researchers [Hel88, IW88, Mah85, Nau88b, RSUV89]. We discuss these ideas briefly through examples.

Example 6.2 We begin with an example that illustrates *commutativity* of rules. The idea has been studied in all of the references cited above, and several sufficient conditions are known for detecting when this property holds.

$$\begin{aligned} r_1 : p(X,Y) &\text{ :- } a(X,Z), p(Z,Y). \\ r_2 : p(X,Y) &\text{ :- } p(X,Z), b(Z,Y). \\ r_3 : p(X,Y) &\text{ :- } c(X,Y). \end{aligned}$$

It can be shown that applications of rules r_1 and r_2 commute. That is, $r_1.r_2 = r_2.r_1$. This fact can be utilized to avoid many redundant derivations by evaluating the fixpoint of the program

as follows: first apply rule r_3, then apply r_2 as often as possible, and finally apply r_1 as often as possible. □

Example 6.3 This example illustrates *decomposability*, which was studied by Maher [Mah85]. Informally, a program is decomposable if its fixpoint can be computed as the union of the fixpoints of two programs that are subsets of the original program. Consider the following program, which generates all positive even numbers:

$$\begin{array}{lll} r_1 : even(X) & :- & even(s^2(X)). \\ r_2 : even(s^2(X)) & :- & even(X). \\ r_3 : even(s^{2k}(0)). & & \end{array}$$

It can be decomposed into the following programs:

$$\begin{array}{lll} r'_1 : even(X) & :- & even(s^2(X)). \\ r'_2 : even(s^{2k}(0)). & & \end{array}$$

$$\begin{array}{lll} r'_3 : even(s^2(X)) & :- & even(X). \\ r'_4 : even(s^{2k}(0)). & & \end{array}$$

The first program, r'_1, r'_2, computes all positive even numbers less than s^{2k}, and the second program, r'_3, r'_4, computes all even numbers greater than s^{2k}. □

Ioannidis presents an algebraic formulation of Datalog programs that is particularly suited to reasoning about such properties of programs [IW88].

Naughton [Nau88b] defines a class of recursions called *separable recursions*. While commutativity is not explicit in the definition of the class, the rules of a separable recursion do in fact commute, and the algorithm presented for separable recursion evaluation depends upon commutativity for its correctness. A discussion of the relationship between commutativity and separability appears in [Ioa89].

We can view SN and NSN iteration as acting on derivation trees instead of atoms. With appropriate definitions for the operations used in an evaluation, the multiset of atoms S computed in an evaluation is simply an abstraction of the multiset of derivation trees S' computed by the same evaluation. We note that the properties we have discussed in this section can be understood as essentially assuring us that it is sufficient to construct a set of derivation trees that is a proper subset of the set of all derivation trees of the program. For example, commutativity allows us to avoid generating derivation trees in which applications of rules r_1 and rule r_2 are interleaved. An approach based on analyzing derivation trees and proving containment theorems between sets of derivation trees is presented in [RSUV89].

The derivation tree approach is also explored by Helm [Hel88]. In contrast to [RSUV89], he does not attempt to directly establish properties such as commutativity. Rather, he performs a compile-time analysis based on directly testing containments of (fragments of) derivation-trees, and uses the results to guide the development of *iterative control expressions* that govern the fixpoint evaluation of the program, thereby avoiding redundant derivations. His approach is noteworthy in that it is able to avoid some redundant derivation even when the program does not satisfy algebraic properties such as commutativity. Further, the results of the more powerful

containment tests of [RSUV89] and other sufficient conditions for various algebraic properties can be incorporated into the approach by using them to guide the generation of iterative control expressions.

7 On Choosing an Evaluation Method

In this section we explore in more detail properties of logic programs that help determine when a particular evaluation method can be expected to perform better than another. For clarity of exposition, in this section we assume that the the rewritten programs (Magic Templates and Counting) are range-restricted. This implies that only ground goals and facts are generated.

We begin by investigating the choice between Magic Templates and Counting. We show that, informally, if a program P is such that there is only one way to derive each fact in the Counting program, Counting and Magic Templates derive the same number of facts in the evaluation of P. This is significant because it has been suggested that since Counting can be extremely inefficient in the presence of duplicates, it should only be used when there are no duplicates. Our result suggests that using Counting provides the greatest benefit when there are a small but nonzero number of duplicate derivations in the counting program.

Next we turn to the question of comparing Magic Templates bottom-up with Prolog top-down. Here we are considering the "pure" subset of Prolog — that is, with occur-check, and without cut, assert, meta-level predicates, and so forth. Since the structure of a computation in Magic Templates and in Prolog are so different, we must first develop some common framework in which the two can be compared. Toward this end we define *Prolog Trees*, and the program P^{opt}. The Prolog trees for a given program and query capture the behavior of Prolog in evaluating that program and query in a static set of derivation trees. P^{opt} is a simple modification of P^{mg} such that the set of NSN trees for P^{opt} (generated from a program P and a query q) correspond closely to the set of Prolog trees for P and q.

With the definitions of Prolog trees and the program P^{opt}, we compare the behavior of Prolog and bottom-up on several classes of programs.

7.1 Counting and Magic

In many cases, the Counting rewriting algorithm produces programs that are more efficient than the corresponding Magic program [BR88, MSPS87]. However, if there are facts in the program that can be generated in multiple ways, the Counting rewriting can be worse than even the most straightfoward bottom-up approach — ignoring the query and computing the entire queried relation using the original, unrewritten program program. The following example shows how this can occur.

Example 7.1 Consider the following program.

$$
\begin{array}{lll}
t(X,Y) & \text{:-} & a_1(X,W), t(W,Z), b_1(Z,Y). \\
t(X,Y) & \text{:-} & a_2(X,W), t(W,Z), b_2(Z,Y). \\
t(X,Y) & \text{:-} & t_0(X,Y).
\end{array}
$$

$$
q(Y) \quad \text{:-} \quad t(c_0,Y)?
$$

The corresponding counting program is

$$
\begin{array}{lll}
cnt\_t(c0, 0). & & \\
cnt\_t(W, 2 * I) & :- & cnt\_t(X, I), a_1(X, W). \\
cnt\_t(W, 2 * I + 1) & :- & cnt\_t(X, I), a_2(X, W). \\
 & & \\
t\_c(Y, J) & :- & cnt\_t(X, J), t_0(X, Y). \\
t\_c(Y, I/2) & :- & t\_c(Z, I), b_1(Z, Y). \\
t\_c(Y, (I-1)/2) & :- & t\_c(Z, I), b_2(Z, Y). \\
 & & \\
q(Y) & :- & t\_c(Y, 0).
\end{array}
$$

(Here for clarity of exposition we have deleted some unnecessary indices that would be produced by the full Generalized Counting rewriting.)

Suppose that we have the facts $a_1(c_0, c_1)$, $a_1(c_1, c_2)$, ..., $a_1(c_{n-1}, c_n)$ and the facts $a_2(c_0, c_1)$, $a_2(c_1, c_2)$, ..., $a_2(c_{n-1}, c_n)$. Then the bottom-up evaluation of the rewritten program will generate $\Omega(2^n)$ facts in cnt_t. Note that simply evaluating the original unrewritten program bottom-up can never generate more then n^2 facts in t. □

This exponential behavior of Counting is not only manifested over specially designed relations — it occurs even over randomly generated sparse relations [Nau88a]. Intuitively, the inefficiency arises when some facts can be derived in many different ways, because counting stores a fact corresponding to each derivation.

The next theorem establishes the somewhat surprising fact that if the Counting program is duplicate-free, then it computes at least as many facts as the corresponding Magic program. Since the Counting transformation can sometimes degrade performance significantly, even introducing non-termination, it is important to be able to determine when it obtains an improvement. It has been noted that Counting can be shown to terminate, by a compile-time analysis, for several programs that recursively manipulate structures like lists. Programs like list reverse and append are good examples; in general, this class includes many deterministic programs in which the recursive predicates contain an argument position in which the values are monotonically decreasing in successive iterations. Since the Counting version of such programs is typically duplicate-free, the following theorem indicates that on such programs we should not expect to see an improvement over the Magic program in terms of the number of facts computed — which was the original motivation for the Counting algorithm.

Theorem 7.1 *If the Counting program for goal Q is duplicate-free, then the Magic program computes no more facts than the Counting program.*

Proof In the following, for a predicate p in the original program, we will use p^{mg} to denote the adorned relation in the program produced by the Magic Templates transformation, and will use mag_p to denote the corresponding magic predicate. Similarly, we will use p^{cnt} to denoted the adorned relation in the program produced by the Counting transformation, and cnt_p to denote the corresponding counting relation.

We begin by partitioning the adorned relation p^{mg} by grouping the tuples in p^{mg} according to the vector of values in the free columns. Consider a partition $p^{mg}(\overline{a_1}, \overline{b})$, $p^{mg}(\overline{a_2}, \overline{b})$, ...,

$p^{mg}(\overline{a_n}, \overline{b})$. We show that either the Counting program has duplicates or else there are n distinct facts in the Counting program corresponding to these n facts in the Magic program.

Because the n facts $p^{mg}(\overline{a_1}, \overline{b})$, $p^{mg}(\overline{a_2}, \overline{b})$, ..., $p^{mg}(\overline{a_n}, \overline{b})$ appear in p^{mg}, by definition of the Magic Templates transformation the n facts $p(\overline{a_1}, \overline{b})$, $p(\overline{a_2}, \overline{b})$, ..., $p^{mg}(\overline{a_n}, \overline{b})$ appear in p in the original program. Also by definition of the Magic Templates transformation, in order for the n facts $p^{mg}(\overline{a_1}, \overline{b})$, $p^{mg}(\overline{a_2}, \overline{b})$, ..., $p^{mg}(\overline{a_n}, \overline{b})$ to appear in p^{mg}, the n facts $mag\_p(\overline{a_1})$, $mag\_p(\overline{a_2})$, ... $mag\_p(\overline{a_n})$ must appear in mag_p.

But by definition of the Counting and Magic transformations, for every tuple $mag\_p(\overline{a})$ in mag_p, there exists an I such that the tuple $cnt\_p(\overline{a}, I)$ is in cnt_p. This follows because the rules defining cnt_p are just the rules defining mag_p with the addition of index fields. Also, from the definition of the Counting transformation, if $cnt\_p(\overline{a}, I)$ appears in the counting relation for p, and $p(\overline{a}, \overline{b})$ is a fact in p in the original program, then $p^{cnt}(\overline{b}, I)$ appears in p^{cnt}.

Hence we must have the n facts $cnt\_p(a_1, I_1)$, $cnt\_p(a_2, I_2)$, ..., $cnt\_p(a_n, I_n)$ and the n facts $p^{cnt}(b, I_1)$, $p^{cnt}(b, I_2)$,..., $p^{cnt}(b, I_n)$ in the counting program.

Now there are two cases to consider. Either the I_j are distinct, in which case for every fact in the Magic program there is a corresponding fact in the Counting program; or they are not, in which case the Counting program is not duplicate free. □

Theorem 7.1 demonstrates that with respect to the number of facts produced, the advantages of Counting over Magic Templates must appear in programs for which Counting is not duplicate-free. However, the number of facts generated is not the only issue determining performance. Counting introduces integer index fields that can be used effectively as a physical index. In programs that manipulate list structures, for example, this may make Counting better than Magic Templates.

7.2 Bottom-up Evaluation and Prolog

In this subsection we consider the relationship between bottom-up evaluation and Prolog. First, in Subsection 7.2.1 we develop some definitions and technical results that are useful in this comparison; next, Subsection 7.2.2 we use these definitions and results to define and discuss well-behaved and strictly well-behaved programs.

7.2.1 Prolog Trees and P^{opt}

We now turn our attention to properties of programs that determine the relationship between top-down Prolog evaluation method and bottom-up Magic Templates evaluation. In order to state these properties, we need to provide a basis for comparing the work done by the two strategies.

For this purpose derivation trees are not sufficient, for they only contain information about successful derivations, and ignore the process of finding those derivation trees. A more detailed set of trees is necessary; we call such trees *Prolog trees.*

There are two types of nodes in "Prolog" trees. *Goal* nodes reflect goals that have been set up through the course of the evaluation; *fact* nodes reflect facts that have been deduced. In order to make the connection between Prolog top-down and Magic Templates bottom-up

computation more explicit, we label a goal corresponding to a prolog call $q(c)$ with $m\_q(c)$. The intuition here is that the magic facts for q are exactly the q goals that Prolog must try to solve.

Definition 7.1 Let P be a logic program, and let $q(c)$ be a top-level goal (query) on P. Then the set of *Prolog trees* corresponding to Prolog evaluation of $\langle q, P\rangle$, which we will denote by $\mathcal{T}_P(P, q)$, are defined follows. Initially, we start with $\mathcal{T}_P(P, q)$ empty.

- First, add to $\mathcal{T}_P(P, q)$ a tree (consisting of a single goal node) corresponding to the top-level goal, $m\_q(c)$.
- Let r be a rule $p_0(X_0)$:- $p_1(X_1), \ldots, p_k(X_k)$, and suppose that at some point Prolog invokes r with the goal $p_0(c)$?. This means that at this point $\mathcal{T}_P(P, q)$ contains a tree with root $m\_p_0(c)$. Let T_0 be the subtree rooted at $m\_p_0(c)$. Furthermore, there must be a substitution θ_1 such that $\theta_1 = mgu(X_0, c_0)$. Then add the following tree to $\mathcal{T}_P(P, q)$.
  - The root of the new tree is $m\_p_1(c_1)$, where $c_1 = X_1\theta_1$.
  - The node $m\_p_1(c_1)$ has a single child. The node for the child is $m\_p(c_0)$; this child node itself has the tree T_0 its only child.
- If the subgoal $m\_p_1(c_1)$ returns with failure, exit this rule invocation (at this point no more trees are added to $\mathcal{T}_P(P, q)$ due to this rule invocation.)
- Suppose that for $1 < i \leq k$, subgoal $m\_p_i(c_i)$ succeeds with answer $p_i(c_i')$. Then it must be the case that for $1 \leq j < i$, goal $m\_p_j(c_j)$ succeeded with some answer $p_j(c_j')$, where c_j' can be written as $X_j\theta_1\theta_2 \ldots \theta_j$. Then add the following goal tree to $\mathcal{T}_P(P, q)$.
  - The root is $m\_p_i(c_i)$, where $c_i = X_i\theta_1\theta_2 \ldots \theta_{i-1}$.
  - There are i children $p_j(c_j')$, for $0 \leq j < i$, where $c_j' = X_j\theta_1\theta_2 \ldots \theta_j$. The derivation trees T_j, for $0 \leq j < i$, are rooted at their corresponding nodes.
- If rule r succeeds on goal $m\_p_0(c_0)$ with $p_0(c_0')$, and with the ith literal in the body instantiated to $p_i(c_i')$, then it must be the case that $c_i' = X_i\theta_1 \ldots \theta_i$. Then add the following tree to $\mathcal{T}_P(P, q)$.
  - The root is $p_0(c_0')$, where $c_0' = X_0\theta_1\theta_2 \ldots \theta_k$.
  - There are k children $p_1(c_1')$, ..., $p_k(c_k')$. The trees $T_1, \ldots, T_k$ are rooted at the corresponding nodes.

Note that the set of Prolog trees for a given program and query may be infinite. While we have described the set of Prolog trees procedurally, based upon the behavior of Prolog in evaluating a program, we do not mean to imply that the set of all Prolog trees for a given computation can be enumerated by following a Prolog execution of the program. This is because in general, some Prolog trees will be generated "after" Prolog finishes an infinite branch of the computation. Our intended interpretation of the preceding definition is that the set of Prolog trees for a computation is exactly the set of trees corresponding to a Prolog execution in which all infinite branches are completely evaluated, possibly generating an infinite set of trees.

Example 7.2 Consider the program

$$\begin{array}{lll} p(X) & :- & p(X), r(X). \\ p(1). & & \\ r(1). & & \end{array}$$

$$q(X) \quad :- \quad p(X).$$

Here Prolog will loop forever on the initial call to $p(X)$. The set of Prolog trees is infinite. One infinite subset of trees within this set, defined inductively, consists trees of the following form: the tree for $k = 0$ is just the node $p(1)$. For all $k > 0$, the root of tree k is the fact $p(1)$. The root of tree has has two children: the left child is the tree for k -- 1, while the right subtree is the fact $r(1)$. □

In the following we wish to prove theorems relating the behavior of Prolog on a given program P and query q to certain properties of the NSN trees for P^{mg}, which we denote by $\mathcal{T}_{NSN}(P^{mg}, q)$. Technically speaking, the "q" in $\mathcal{T}_{NSN}(P^{mg}, q)$ is redundant, because enough information to determine q is contained in P^{mg}. However, we retain the q for consistency with our way of denoting the Prolog trees for P and q: $\mathcal{T}_P(P, q)$. The situation is complicated because the NSN trees for P^{mg} can contain spurious infinite trees due to "loops" created by the interaction between the magic predicate instances in the modified rules of P and the magic predicate instances in the magic rules.

Example 7.3 Consider the following program P:

$$\begin{array}{lll} q(X,Y) & :- & p(X,Z), p(Z,Y). \\ p(X,Y) & :- & b(X,Y). \\ b(5,5). & & \end{array}$$

$$query(Y) \quad :- \quad q(5,Y).$$

The corresponding magic program P^{mg} is

$$\begin{array}{llll} 1) & mq(5). & & \\ 2) & mp(X) & :- & mq(X). \\ 3) & mp(Z) & :- & mq(X), p(X,Z). \end{array}$$

$$\begin{array}{llll} 4) & q(X,Y) & :- & mq(X), p(X,Z), p(Z,Y). \\ 5) & p(X,Y) & :- & mp(X), b(X,Y). \end{array}$$

$$6) \quad b(5,5).$$

Now consider evaluating this program bottom-up using NSN. On the first iteration we will infer the fact $mp(5)$ using Rule 2). On the second iteration we will infer $p(5,5)$ using Rule 5). This in turn will imply $mp(5)$ on the following iteration, from Rule 3). But now we have entered a loop where $mp(5)$ derives $p(5,5)$, which derives $mp(5)$, ad infinitum. □

To avoid this situation we define a new program, P^{opt}. The following definition distinguishes between the *magic rules* and the *modified original rules* of the program P. A *magic rule* is a rule in which some predicate introduced by the Magic Templates rewriting algorithm appears in the head.

Definition 7.2 *Let P be a program, and P^{mg} be the corresponding magic program. Then P^{opt} is formed by deleting all magic predicates from the non-magic rules of P^{mg}.*

The program P^{opt} has the desirable property that the spurious loops mentioned above do not arise. However, it has the undesirable property that many facts that can be proven in P^{opt} are irrelevant — that is, they do not appear in P^{mg}. This can be avoided by focussing our attention on the *relevant* portions of P^{opt}, that is, the NSN trees in P^{opt} that have roots that appear as facts in $GC(P^{mg})$.

Definition 7.3 Let P be a program, and q be a query on some predicate appearing in P. Then the *relevant trees with respect to Q* in P^{opt}, denoted $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, are all trees t in $\mathcal{T}_{NSN}(P^{opt})$ such that the root of t is in $GC(P^{mg})$.

Lemma 7.1 *For a program P and a query q, there is a tree of height k with a goal $m\_p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ if and only if there is a tree of height k with $m\_p(c)$ at the root in $\mathcal{T}_P(P, q)$. Similarly, there is a tree of height k with a fact $p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ if and only if there is a tree of height k with $p(c)$ at the root in $\mathcal{T}_P(P, q)$.*

Proof Given in Appendix A. □

7.2.2 Well-Behaved and Strictly Well-Behaved Programs

We now apply the notions of P^{opt} and Prolog trees to relate the behavior of top-down Prolog and bottom-up Magic Templates on classes of programs.

The following property, *well-behaved*, is related to the finite forest property of Section 6.1. The key difference is that well-behaved considers only the facts that are relevant to a particular query. The motivation for this restriction is that both Prolog and Magic Templates only compute relevant facts, so their respective behavior on irrelevant facts has no bearing on the choice of which method to use.

Definition 7.4 (Well-Behaved) Consider a program P (including a goal q), and P^{mg} according to the Prolog sips. Let P^{opt} be P^{mg} with the magic predicates deleted from the modified original rules of P. Let $\mathcal{T}_{NSN}(P^{opt}, q)$ be the set of NSN trees for P^{opt}. P is *well-behaved* if each fact in $GC(P^{mg})$ unifies with the roots of only a finite number of trees of $\mathcal{T}_{NSN}(P^{opt}, q)$.

The following property, *strictly well-behaved*, is stronger than *well-behaved*.

Definition 7.5 (Strictly Well-Behaved) Consider program P (including a goal q), and P^{mg} according to the Prolog sips. Let P^{opt} be P^{mg} with the magic predicates deleted from the modified original rules of P. Let $\mathcal{F}$ be the set of facts in $GC(P^{mg})$. Let $\mathcal{T}_{NSN}(P^{opt}, q)$ be the set of NSN trees for P^{opt}. P is *strictly well-behaved* if there are no two trees t_1 and t_2 in $\mathcal{T}_{NSN}(P^{opt}, q)$ such that the roots of both trees appear in $GC(P^{mg})$ and the root of t_1 is the same fact as the root of t_2.

Theorem 7.2 *P^{mg} is duplicate-free if and only if P is strictly well-behaved.*

Proof (P^{mg} duplicate-free $\rightarrow$ P is strictly well-behaved.) Assume P is not strictly well-behaved. Then there is some fact in $\mathcal{F}$ with two distinct trees. By adding subtrees for magic facts to these trees, we get two distinct trees in P^{mg}. This uses the fact that there is a magic fact in P^{mg} corresponding to each goal that appears in a tree of $\mathcal{T}_P(P, q)$, which in turn follows from Lemma 7.1.

(P strongly well-behaved $\rightarrow$ P^{mg} duplicate-free.) In this direction we use the notion of the δ-height of a tree. The δ-height of a tree in P^{mg} is the height of the tree after all magic facts other than the root (and their subtrees) are deleted.

Assume that P^{mg} is not subsumption free. Then there must be two distinct trees in P^{mg} with the same root, which is a fact in $\mathcal{F}$. If these trees have different δ-heights, then clearly there are two trees for the same fact in P^{opt}.

Consider the case where the trees have the same δ-height. Let d be the minimal height such that there exist two distinct trees in P^{mg} with the same root in $\mathcal{F}$ and of δ-height d. If these trees differ in a subtree not rooted in a magic fact, then clearly there are corresponding trees in P^{opt}, so P is not strictly well-behaved. If these trees differ in a subtree rooted in a magic fact, then we have a contradiction to the fact that d was minimal. □

Theorem 7.2 is important for two reasons. First, it may be of use in a test to decide if P^{mg} is duplicate-free, since in general both the magic rules and the original program P are simpler than P^{mg} (witness the interaction between the magic rules and modified original rules in Example 7.3.) Second, Theorem 7.2 gives us insight into the relationship between P and P^{mg}. As noted, in general there can be complex interactions between the magic rules and the modified original rules; the theorem tells us that these interactions cannot give rise to violations of subsumption freedom where there were none in the original program or in the magic rules.

Lemma 7.2 *For a program P and a query Q, if P is strictly well-behaved, then for each tree with root r in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, there is exactly one tree with root r in $\mathcal{T}_P(P, q)$.*

Proof Given in Appendix A. □

The following theorem is useful in relating the termination and completeness of Prolog on P to properties of the NSN trees of P^{opt}.

Theorem 7.3 (Prolog Computation 1) *Prolog generates each goal and fact only a finite number of times if and only if P is well-behaved.*

Proof If Prolog generates a goal or a fact infinitely often, then there must be Prolog trees of unbounded height for this goal/fact. (There are Prolog trees of unbounded height for a given goal or fact if for any k, there is a Prolog tree T with the goal or fact at the root such that T is of height greater than k.) This follows since

- For a given Prolog program, there are only finitely many Prolog trees of a given bounded height, and
- Prolog only generates a goal or fact a finite number of times with the same Prolog tree.

Let f be a goal or fact that is generated infinitely often by Prolog. By Lemma 7.1, this implies that there are NSN trees of unbounded height for f in $\mathcal{T}_P(P^{opt}, q)$, and that f appears in $GC(P^{mg})$. This in turn implies, by the definition of well-behaved, that P is not well-behaved.

The other direction is again symmetric — if P is not well-behaved, then there must be some fact f such that f appears in $GC(P^{mg})$ and f is the root of trees of unbounded height in $\mathcal{T}_{NSN}(P^{opt}, q)$. This in turn implies by Lemma 7.1 that there are unbounded Prolog trees in $\mathcal{T}_P(P, q)$ with f at the root, so Prolog generates f an infinite number of times. □

The following theorem is useful in determining when bottom-up evaluation can be guaranteed to terminate.

Theorem 7.4 *If P is well-behaved, all base predicates are finite, and all magic predicates are safe in P^{mg}, then all predicates are safe in P^{mg}.*

Proof If P is well-behaved, then by definition of well-behaved any fact in $GC(P^{mg})$ appears as the root of only a finite number of trees in $\mathcal{T}_{NSN}(P^{opt}, q)$. This in turn implies that for any fact f appearing in $GC(P^{mg})$, there is a bound on the maximum height of any derivation tree for f in P^{opt}.

Similarly, since we are given that all magic predicates are safe in P^{mg}, there is a bound on the maximum height of any derivation tree for any magic fact in $GC(P^{mg})$. But then since every derivation tree in P^{mg} is simply a derivation tree in P^{opt}, possibly with the addition of nodes for magic facts (and their associated subtrees), there is a bound on the maximum height of any derivation tree for any fact in P^{mg}. This, together with the finite number of rules in P^{mg} and the finite number of facts in the base relations for P, implies that all predicates are safe in P^{mg}. □

Corollary 7.1 (Prolog Computation 2) *Prolog is complete if P is well-behaved, every base predicate is finite, and every magic predicate is safe in P^{mg}.*

Proof By Theorem 7.4, if P is well-behaved, every base predicate is finite, and every magic predicate is safe in P^{mg}, then there is a bound on the maximum height on any derivation tree in $\mathcal{T}_{NSN}(P^{opt})$ for a fact f in $GC(P^{mg})$. This, by Lemma 7.1, implies that there is a bound on the maximum height of any tree in $\mathcal{T}_P(P, q)$. If every tree in $\mathcal{T}_P(P, q)$ is of bounded height, and every base relation is finite, then there are only a finite number of trees of finite height for Prolog to search, hence Prolog must be complete. □

Corollary 7.2 *Completeness of Prolog is decidable for range-restricted Datalog programs in which base relations contain only ground facts.*

Proof All range-restricted Datalog programs in which base relations contain only ground facts are safe. Maher and Ramakrishnan [MR90] have proven that for Datalog programs, the finite forest property is decidable; this proof can be modified to show that for Datalog programs, well-behavedness is decidable. Hence by Theorem 7.4, completeness of Prolog is decidable for ground Datalog programs. □

Lemma 7.3 *Seminaive bottom-up evaluation of P^{mg} terminates if and only if all predicates are safe in P^{mg}.*

Proof Suppose some predicate p in P^{mg} is not safe. By definition of safety, this implies that there are an infinite number of facts for p in $GC(P^{mg})$. Since bottom-up evaluation is complete, and computes all of $GC(P^{mg})$, and computes only a finite number of tuples on each iteration, bottom-up evaluation of P^{mg} cannot terminate.

Suppose that all predicates in P^{mg} are safe. By definition of safety, this implies that there is only a finite number of facts for each predicate in P^{mg}. Since Seminaive bottom-up evaluation either computes new facts on each iteration or terminates, it must eventually compute all facts and then terminate. □

This leads to the observation that bottom-up evaluation is preferable if P is not well-behaved — Prolog is not complete, and bottom-up evaluation (of P^{mg}) is complete.

Now, a more positive side to Prolog:

Theorem 7.5 *If P^{mg} is duplicate-free, each fact and goal is generated at most once by Prolog.*

Proof By Theorem 7.2, if P^{mg} is duplicate-free, then P is strictly well-behaved. Then by Lemma 7.2, there is a unique Prolog tree for each fact. Thus each fact and goal must be generated at most once by Prolog. □

Corollary 7.3 *If P^{mg} is duplicate-free, Prolog is linear in the size of the set of facts computed.*

Corollary 7.4 *If P^{mg} is duplicate-free, Prolog is polynomial in the size of the EDB.*

However, note that Prolog may not be complete unless P^{mg} is also safe.

8 Connections to Functional Program Transformations and Tabulations

In previous sections we reviewed the bottom-up approach to logic program evaluation, observed that there are several alternatives within this approach, and examined its relationship to a top-down method. We have tried to establish the thesis that there are a number of comparable choices of evaluation method, and a compiler must make an intelligent choice based on the given program. In the same spirit, we now study some results from the functional programming literature, and illustrate how methods for logic program evaluation can be refined using similar techniques. Our study only indicates the possibilities, and makes no attempt to cover the literature comprehensively.

Our discussion is in three parts. We begin by introducing some of the ideas in functional program transformation, drawing primarily upon Burstall and Darlington's pioneering work [BD77]. This work has somewhat different objectives and assumptions from the program transformations in the bottom-up evaluation literature, and it is interesting to compare them. Next, we present work on tabulation, or memoing, which is closely related to the bottom-up approach of retaining all derived facts. This work points to an important new direction of research in bottom-up evaluation, which is the effective utilization of memory through compile-time garbage collection. We follow the presentation in [Bir80, Coh83]. Finally, we consider how this work relates to bottom-up evaluation of logic programs. We apply the tabulation results to refine the Magic Templates approach in a simple way, and then compare this with the Burstall-Darlington approach using a number of examples.

Our conclusions can be summarized as follows:

1. The functional approaches use specialized techniques and hints from the programmer to effect very sophisticated transformations. The objective is to build systems that enable

a programmer to develop efficient programs from declarative functional specifications. The bottom-up approach deals with logic programs, and also achieves somewhat greater generality since it is independent of the set of facts in base relations. It is more appropriately viewed as a collection of techniques for compiler optimization than as a system for developing programs from specifications (although the distinction is one of degree).

2. The bottom-up approach typically matches the functional approach in time complexity, but uses considerably more space.

3. Applying techniques from the tabulation literature to the bottom-up approach shows promise in improving the space utilization of the bottom-up approach.

8.1 Functional Program Transformations

In this section, we describe the transformation approach proposed in [BD77].

Burstall and Darlington propose to transform functional programs, expressed as recursion equations, using a library of transformation rules. We begin by establishing a correspondence between recursion equations and the formalism of logic programs. It is actually easier to show such a correspondence for adorned programs. (See Section 3.1 for a definition of adorned programs.)

Any set of recursion equations can be thought of as an adorned program P^{ad}. However, the reverse is not true — an adorned program that corresponds to a set of recursion equations must satisfy a number of constraints not satisfied by adorned programs in general.

Intuitively, if an adorned program corresponds to a set of recursion equations, each adorned predicate denotes a function; $f(c) = d$ is represented as a tuple $p^{10}(c, d)$. For each adorned literal $p^a(\bar{t})$, there is at most one rule with head predicate p^a such that the bound arguments of the head unify with the bound arguments of $\bar{t}$. That is, if the function denoted by p^a is invoked with some arguments, at most one program rule is applicable.

Additionally, in the relation associated with p^a in the least Herbrand model, the set of bound arguments must functionally determine the set of free arguments, for all adorned predicates p^a. That is, p^a is indeed a function from the set of bound arguments to the set of free arguments. (Recall that there are techniques to test these conditions; see Section 6.2.)

In the remainder of this subsection, we will only consider adorned programs that satisfy these additional restrictions which guarantee that they correspond to recursion equations.

Burstall and Darlington allow the following transformation rules: *Definition, Instantiation, Unfolding, Folding* and *Abstraction.* In addition, laws such as commutativity and associativity can be used. Definition adds new rules to the adorned program P^{ad} (that preserve the conditions for restriction to recursion equations). Instantiation adds a substitution instance of an existing rule to P^{ad}. Unfolding adds a rule that is formed by taking a rule of P^{ad} and expanding a body literal using a rule (also in P^{ad}) that defines it. Folding is essentially the inverse of unfolding; it adds a new rule that is formed from a rule of P^{ad} by replacing a set of body literals $b_i \ldots b_j$ with a single literal h. (There must be some rule r in P^{ad} such that $h :\!- b_i \ldots b_j$ is a substitution instance of r.) Abstraction puts a rule into a canonical form by replacing all occurrences of

a term t by a new variable X and adding the equation $X = t$ to the body. This allows us to expose the structure of the rule in a way that facilitates generalization.

The repeated application of these transformation rules is guaranteed to preserve soundness in that any inferred answer is in the Herbrand model; however, it may introduce non-termination. Related sets of transformation rules, for the case of logic programs, have been investigated by Tamaki and Sato [TS84] and by Maher [Mah89]. The Tamaki-Sato transformation system is shown to preserve the least Herbrand model, and the Maher system is shown to preserve the Clark completion of the program.

Burstall and Darlington suggest the following strategy for applying their transformation rules: Make necessary definitions, instantiate, and repeatedly try unfolding followed by application of laws, abstraction and folding. They note that introducing the definitions and choosing appropriate instantiations and abstractions often require hints from the user.

In earlier work, Burstall and Darlington developed a system in which transformation rules converted recursive schemas into iterative schemas [DB73]; the system that we have discussed arose out of a desire to transform programs extensively before eliminating recursion. In [DB73], a transformation rule is specified as a schematic rewriting system with constraints on the instances of the schemas for which the rule is valid. This is formalized using using second order unification in [HL78].

We now present several examples, from [BD77], to illustrate the use of transformation rules; the reader is urged to consult [BD77] for a detailed treatment.

Example 8.1 Consider the Fibonacci example:

$$\begin{array}{lll} fib^{10}(0,1). & & fib^{10}(1,1). \\ fib^{10}(N, X_1 + X_2) & :- & N > 1, fib^{10}(N-1, X_1), fib^{10}(N-2, X_2). \end{array}$$

We have seen that a top-down evaluation of this program is exponential time unless the results of all calls are saved. The Burstall-Darlington system dramatically improves this program by combining the two recursive calls into one. This requires a hint from the user, who must supply the following definition:

$$g^{100}(N-2, X_1, X_2) \;:-\; N > 1, fib^{10}(N-1, X_1), fib^{10}(N-2, X_2).$$

Given this hint, the system tries a variety of applications of the transformation rules, and produces the following program that runs in linear time and space:

$$\begin{array}{lll} fib^{10}(0,1). & & fib^{10}(1,1). \\ fib^{10}(N+2, U+V) & :- & N \geq 0, g(N,U,V). \\ g^{100}(0,1,1). & & \\ g^{100}(N+1, U+V, U) & :- & N \geq 0, g(N,U,V). \end{array}$$

□

Example 8.2 The following program computes factorials over the domain of non-negative integers:

$$\begin{array}{lll} fact^{10}(0,1). & & \\ fact^{10}(N+1, X) & :- & fact^{10}(N,M), X = (N+1) * M. \end{array}$$

This is a recursive program that is typically implemented in a top-down system with a stack. As we return from calls, we must perform a series of multiplications. The execution takes linear space and time. This can be improved by introducing an *accumulator*, which is a new field that is used to carry along intermediate results. This requires intervention from the user in the form of the following definition:

$$g^{110}(N, U, X) \;\text{:-}\; fact^{10}(N, M), X = U * M.$$

The system then uses the transformation rules to obtain:[3]

$$\begin{array}{lll} fact^{10}(N, M) & \text{:-} & g^{110}(N, 1, M). \\ g^{110}(0, U, U). & & \\ g^{110}(N+1, U, M) & \text{:-} & g^{110}(N, U*(N+1), M). \end{array}$$

This is a tail recursive program, and is easily translated into an iteration; for example, by applying the Magic Templates algorithm and factoring. (Note that the Magic Templates algorithm when applied to the original program yields a program that cannot be factored.) The program can be executed in linear time and constant space. □

Example 8.3 The following program computes a list of factorials:

$$\begin{array}{lll} flist^{10}(0, nil). & & \\ flist^{10}(N+1, M.L) & \text{:-} & fact^{10}(N+1, M), flist^{10}(N, L). \\ fact^{10}(0, 1). & & \\ fact^{10}(N+1, X) & \text{:-} & fact^{10}(N, M), X = (N+1) * M. \end{array}$$

We have used the infix "." to denote the *cons* operator. This program is inefficient in that it makes no use of the computation of the first N factorials in computing the next factorial. The Burstall-Darlington system again needs assistance in the form of the following definition:

$$g^{100}(N, U, V) \;\text{:-}\; fact^{10}(N+1, U), flist^{10}(N, V).$$

This enables it to produce the following program, which improves upon the original by combining the two recursions into one:

$$\begin{array}{lll} flist^{10}(N+1, M.L) & \text{:-} & g^{100}(N, M, L). \\ g^{100}(0, 1, nil). & & \\ g^{100}(N+1, (N+2)*M, M.L) & \text{:-} & g(N, M, L). \end{array}$$

□

Example 8.4 We now consider the familiar naive reverse program:

$$\begin{array}{lll} rev^{10}(nil, nil). & & \\ rev^{10}(A.X, L) & \text{:-} & rev^{10}(X, L_1), app^{110}(L_1, A.nil, L). \\ app^{110}(nil, L, L). & & \\ app^{110}(X.L_1, L_2, X.L) & \text{:-} & app^{110}(L_1, L_2, L). \end{array}$$

[3]This actually requires an extra transformation rule called Redefinition.

Prolog executes this in linear time and space. This program can also be improved through the use of the accumulator technique. Given the following definition as a hint:

$$g^{110}(X, U, L) \quad :- \quad rev^{10}(X, L), app^{110}(L_1, U, L).$$

the Burstall-Darlington system will produce:

$$\begin{array}{lll} rev^{10}(A.X, L) & :- & g^{110}(X, A.nil, L). \\ g^{110}(A.L, U, W) & :- & app^{110}(A.nil, U, Z), g^{110}(L, Z, W). \\ g^{110}(nil, U, U). & & \end{array}$$

This is again a tail recursive program. It can be executed in linear time and constant space. □

This completes our review of functional program transformations. We will compare these results with the bottom-up approach in Section sec:perfmtab, after considering how the memory utilization of the latter can be improved in the following subsection.

8.2 Tabulation and Compile-Time Garbage Collection

Evaluation methods may be viewed as identifying subgoals and generating solutions for them, and often, it is the case that subgoals are identified in multiple ways. If there is a record of what goals have been identified, and what solutions have been generated, it is possible to re-use this information instead of computing it repeatedly. On the other hand, a table of goals and solutions represents overhead in terms of both space and additional time to maintain it, although the time spent on maintaining the table is likely to be significantly less than the time gained through avoiding repeated computation. Thus, tabulation represents a time-space trade-off in most situations; Prolog is an example of one extreme that does no tabulation, and memoing methods represent the other extreme, tabulating all identified goals and solutions for the remainder of the computation.

We will refer to methods that save intermediate results as *tabulation methods*, using Bird's terminology [Bir80]. There is a wide range of tabulation methods that differ in how much intermediate computation they save. Elsewhere, we have used the term *memoing method*; we will reserve this to refer to a subset of tabulation methods that save *all* intermediate results.

Bottom-up evaluation methods for logic programs typically save all generated goals and solutions to these goals, whereas top-down methods — for instance, Prolog — often do not. Tabulation is not limited to bottom-up methods; indeed, top-down methods that aim to be complete must do some form of tabulation (for example, loop-checking in Prolog-style evaluation).

In this section, we review work on tabulation that aims to determine at compile-time how goals and solutions that are tabulated can be discarded as computation progresses. This allows the development of techniques to allocate memory at compile-time in a way that re-utilizes freed space. We will follow [Bir80] for the most part, and rely upon [Coh83] for the rest of our overview. These papers deal with functional programs, and we will adapt the treatment to logic programs where necessary.

A tabulation method that does not repeat any computation must have the following properties: No information (goals or solutions) that is still needed is discarded, and there is the

capability to check if a new goal is already known, and if so to retrieve the set of known solutions. Bird refers to a particular class of evaluation methods as *exact tabulations*; these memoing methods are top-down methods that save all generated goals and solutions until the execution is completed. Bird's description of exact tabulations is limited to the case of functional programs that are well-defined; this implies that $f(c)$ is never defined in terms of $f(c)$, for any c.

When we consider logic programs, not only must we consider relational mappings rather than functional mappings, which implies that at any time only a partial set of solutions may be known, we must also contend with the fact that such cyclic definitions arise frequently — for instance, Example 1.1 over a cyclic graph. To deal with these differences, it is necessary to extend tabulation as defined by Bird; methods such as Vieille's QSQ [Vie86, Vie87] are precisely such generalizations (although they were derived independently). The Magic Templates method is a bottom-up evaluation method that tabulates the same set of goals and solutions as QSQ, although the control strategy is entirely bottom-up, and so can also be thought of as an instance of (generalized) exact tabulation.

Exact tabulations are optimal in terms of time, but, to use Cohen's forceful phrase, suffer from "profligate and inefficient use of storage" and the associated disadvantages of slow table access and table maintenance overhead. The challenge is to see if we can improve space utilization through compile-time analysis to determine when tabulated goals and solutions are no longer needed.

The basis of such an analysis is the *dependency graph* of a computation.

Definition 8.1 (Dependency Graph) The dependency graph $\mathcal{D}$ for a logic program and a given query indicates how goals depend on each other, and is defined as follows.

1. The source node is the initial query $q?$.
2. The nodes of the graph correspond to goals. (These are "magic" facts in the context of the Magic Templates algorithm.)
3. There is an arc from node g_1 to node g_2 if a solution to g_1 depends directly on a solution to g_2. (The goal g_2 is generated from an instance of a magic rule in which g_1 is the (only) magic fact in the body.)

Bird requires dependency graphs to be acyclic, and this is necessary for a function to be well-defined, but in general dependency graphs for logic programs need not be acyclic. (Indeed, each node in the graph may be part of a cycle!) However, we will only consider programs with acyclic dependency graphs in the rest of this paper.

For acyclic dependency graphs, the nodes can be arranged in a linear order such that $i < j$ whenever there is a path from n_j to n_i in the graph. Once such an order is identified, a tabulation method can proceed by evaluating goals in ascending order, using previously computed values as necessary. Further, this provides a starting point for determining when goals and solutions can be discarded: If there is a constant N such that there is no arc from goal n_j to any goal n_i, for $i < j - N$, then we need only retain the previous N goals and their solutions at any point in the computation.

This simple estimate can be improved through *pebble games* played on the dependency graph. A supply of labeled pebbles, which can be thought of as memory units, is given and

at each move of the game, a pebble is placed on a node. The game ends when a pebble is placed on the source node. The following rule must be obeyed: A pebble can be placed on a node only when there are pebbles on all immediate predecessors (if any), and no node may be pebbled more than once. The smallest number of pebbles with which we can cover the graph corresponds to the minimum amount of memory needed to execute the program without repeating any computation, and the sequence of moves describes how memory is to be assigned in such an execution.

Consider the Fibonacci program from Example 8.1. Each goal corresponds to an integer, and there is an arc from node I to the nodes $I-1$ and $I-2$. We leave it to the reader to pebble this dependency graph using just two pebbles; two units of memory are all we need to execute this program. Variations on the pebble game allow us to explore time-space trade-offs; for example, relaxing the requirement that no node be pebbled twice permits some repeated computation and in general reduces the number of pebbles required. We will not consider any variations in this paper.

The problem that we must address is therefore to analyze the dependency graph for a program at compile-time and to devise a pebbling strategy that uses as few pebbles as possible. The analysis proposed in [Coh83] rests upon a study of *descent functions*, which describe how immediate subgoals are derived from a goal. In our discussion of Cohen's work, we will restrict ourselves to adorned programs that are equivalent to recursion equations, since this class properly includes all the program schemas that he considers.

Consider the following program schema $\mathcal{S}$, which generalizes the Fibonacci program:

$$\begin{array}{lll} f^{10}(N,X) & :- & p^{1}(N), a^{10}(N,X). \\ f^{10}(N,X) & :- & \neg p^{1}(N), c^{10}(N,N_1), f^{10}(N_1,X_1), d^{10}(N,N_2), f^{10}(N_2,X_2), b^{110}(X_1,X_2,X). \end{array}$$

The functions c and d are the descent functions for this schema. It is instructive to examine the magic rules that are generated for the above program:

$$\begin{array}{lll} m\_f^{10}(N_1) & :- & \neg p^{1}(N), m\_f^{10}(N), c^{10}(N,N_1). \\ m\_f^{10}(N_2) & :- & \neg p^{1}(N), m\_f^{10}(N), d^{10}(N,N_2). \end{array}$$

These rules reflect the descent functions very directly; we return to this point in Section 8.2.1.

Cohen considers several conditions on descent functions for the schema $\mathcal{S}$, and shows how a program that is an instance of $\mathcal{S}$ can be pebbled by transforming it into an equivalent program. (The transformations could, however, introduce non-termination.)

To translate his conditions from function to predicate notation, we use the following notation. We say that $c = d$ if for any N, if N_1 and N_1 are such that we have $c(N,N_1)$ and $d(N,N_2)$, then $N_1 = N_2$. Also, we write $c^n(N,N_1)$, where $n \geq 2$, as shorthand for

$$c(N,W_1)c(W_1,W_2)\ldots c(W_N,N_1)$$

and $d^n(N,N_1)$ for

$$d(N,W_1)d(W_1,W_2)\ldots d(W_N,N_1)$$

We say that c and d commute if $c(N,W_1)d(W_1,N_1)$ and $d(N,W_2)c(W_2,N_2)$ implies $N_1 = N_2$.

The conditions considered by Cohen are the following: (1) $c = d$, (2) there is some function g and integers m, n such that $c = g^m$ and $d = g^n$, (3) there are integers m, n such that $c^m = d^n$, and c and d commute, (4) c and d commute. The four conditions form a hierarchy of strictly weaker requirements, and the associated schema transformations are correspondingly more complex. For three of the conditions, the resulting program uses constant space (with the same time complexity as exact tabulation); for the last condition, linear space is necessary.

The transformations have the same general structure. Intuitively, they partition the dependency graph into an ordered set of subgraphs such that nodes in one subgraph depend only upon nodes in the next. (Recall that the dependency graph for a function is required to be acyclic.) Then, they use a linear recursive program scheme to evaluate a node by first evaluating solutions to nodes in the successor subgraph. Finally, the linear recursion is eliminated to obtain an iterative program, which intuitively starts by computing solutions for nodes in subgraphs with no successors (the base cases), and then proceeds to evaluate predecessor subgraphs one by one.

It should be noted that in addition to the descent conditions, these transformations rely upon some *frontier* conditions. The frontier conditions intuitively ensure that there is a clean separation between terminal and non-terminal nodes in the (ordered) dependency graph; that is, there is a terminal node such that all successors are terminal nodes and all predecessors are non-terminal nodes. This separation simplifies the structure of the transformed program, but it is not essential; in effect, initializing the iteration is harder, and subsequently, we must test the condition p at each step.

As an example, consider the Fibonacci program. It is an instance of schema $\mathcal{S}$ that satisfies descent condition (2), with g being the predecessor function. Further, it has the property that $p(x)$ implies $p(g(x))$, for all x, thereby separating the base cases from the recursive goals. This is the frontier condition. Given a query $fib^{10}(n, Y)?$, we must first determine the least k such that $p(g^k(n))$ is true. This allows us to initialize the iteration with the tuple $fib^{10}(g^k(n), a(g^k(n)))$. The computation then proceeds by repeatedly applying the following rule until we generate $fib^{10}(n, y)$:

$$fib^{10}(N, X) \;\text{:-}\; c^{01}(N, N_1), fib^{10}(N_1, X_1), d^{01}(N, N_2), fib^{10}(N_2, X_2), b^{110}(X_1, X_2, X).$$

We rely upon the fact that g is *invertible*, that is, given $g(n)$ we can determine n. In the above rule, this amounts to using c and d with adornment 01 in this phase of the computation. Note that the frontier condition enables us to delete the test for $\neg p$. Further, we can always discard all but the two most recently generated tuples.

The class of transformations that Cohen describes are based on pebbling the dependency graph, but it may often be the case that this graph has no uniform structure. A compromise then is to pebble a graph that is uniform and has embedded in it the dependency graph. This in general requires us to compute goals that we are not required to compute by the original program; it is the addition of these irrelevant goals that gives us the desired uniformity. Bird calls such schemes *overtabulation methods.*

We will not describe these transformations in greater detail here; the reader is asked to consult [Bir80, Coh83].

8.2.1 Magic Tabulation

The rules in a program produced by the Magic Templates algorithm (a magic program, for brevity) are of two kinds: "magic" rules, which define "magic" predicates, and "modified" rules, which define adorned predicates and use the magic predicates to restrict the set of generated facts. We observed a close connection between the descent functions and the magic rules introduced by the Magic Templates algorithm in our discussion of the Fibonacci example.

It is always the case that the magic rules correspond to the use of the descent conditions to generate subgoals. For certain programs, including instances of the schema considered by Cohen, it is possible to repeatedly use the magic rules and identify all magic facts — recall that they correspond to the set of goals — before using the modified rules. This separation may not always be possible; in particular, it may be necessary to generate some facts for adorned predicates in order to identify new subgoals. Such a program is shown in Example 7.3. However, where this separation is possible, the connection between the magic rules and the descent conditions is direct.

We will assume that the magic facts are defined over a countable domain $\mathcal{D}$ with an associated total order $<$. If $m, n \in \mathcal{D}, n < m$, we will use the notation $m - n$ for the number of elements l of $\mathcal{D}$ such that $n < l$ and $l < m$. Let us define a relation $\prec$ over the set of magic facts as $m_1 \prec m_2$ if and only if m_1 is generated by instantiating a magic rule such that m_2 is the (only) magic fact in the body of the instantiated rule. We require that the following conditions hold:

- There are no infinite decreasing chains or cycles according to $\prec$.
- If $m(x_1) \prec m(x_2)$, then $x_1 < x_2$, and we can identify a constant K such that $x_2 - x_1 < K$.
- Consider two magic facts $m(x_1)$ and $m(x_2)$. If there is no magic fact $m(x)$ such that $m(x) \prec m(x_1)$ (that is, x_1 is a base case), and $x_2 < x_1$, then there is no fact $m(y)$ such that $m(y) \prec m(x_2)$ (x_2 is also a base case).

Techniques for establishing monotonicity constraints (see e.g., [BS89]) can be adapted to show that these conditions hold. We say that programs satisfying these conditions are *ordered.*

Let $p \rightarrow q$ hold if there is a rule with a p literal in the head and a q literal in the body, and let $\stackrel{*}{\rightarrow}$ denote the transitive closure of $\rightarrow$. The predicates p and q are mutually recursive if $p \stackrel{*}{\rightarrow} q$ and $q \stackrel{*}{\rightarrow} p$. In particular, for every recursive predicate p, $p \stackrel{*}{\rightarrow} p$.

We now define an evaluation method that works on a special class of logic programs and is a refinement of the Magic Templates – Seminaive Iteration approach.

Definition 8.2 Magic Tabulation
Consider a magic program P^{mg} that satisfies the following:

1. If $p^a \stackrel{*}{\rightarrow} p^c$ and $p^c \stackrel{*}{\rightarrow} p^a$ in P^{mg}, then $m\_p^a \stackrel{*}{\rightarrow} p^c$ does not hold.
2. It is ordered.

Then, P^{mg} can be evaluated as follows:

1. Apply the magic rules repeatedly, retaining only the facts generated at the previous iteration, until no new fact is found. Additionally, retain all (magic) facts that generate no new fact, and denote this set as $\mathcal{M}_1$. (These are goals that correspond to the basis of the recursion.)

2. Apply the modified original rules repeatedly. Additionally:

   - On the first iteration, the set of magic facts is $\mathcal{M}_1$; on each subsequent iteration i, $i > 1$, if $m(x_1)$ appeared in the magic set on iteration i, on iteration $i+1$ replace it with $m(x_2)$, where x_1 immediately precedes x_2 in the total order associated with $\mathcal{D}$.
   - After iteration i, discard all facts generated in iteration j, $j < i - K$. (The constant K is determined from the "ordered program" conditions.)
   - Consider iteration i. If for every magic fact in $\mathcal{M}_i$, the argument t is such that $c < t$, where c is the argument of the original query, then stop.

Note that the first condition is sufficient to ensure that all magic facts can be computed without using any modified rules. Also, note that in general on iteration i there may be facts m in $\mathcal{M}_i$ such that m does not appear as a magic fact in P^{mg}. This does not affect correctness, but may affect efficiency. More sophisticated tabulation schemes that avoid this inefficiency are presented in [NR89].

We state the following theorem, which follows from results in [NR89], without proof.

Theorem 8.1 *The Magic Tabulation algorithm correctly computes all answers.*

8.3 Performance of Magic Tabulation

We consider how the refined version of Magic Templates, which we dubbed Magic Tabulation, works on the programs that we used earlier to illustrate the power of the Burstall-Darlington system. We note that the Magic Tabulation approach does not rely upon user intervention, and is totally correct, that is, it does not introduce non-termination.

In all these example, we will assume that a "seed" magic fact corresponding to the given query is added to the magic program. Observe that the Magic Templates algorithm produces a program whose Seminaive evaluation always matches the time complexity of the program produced by the Burstall-Darlington system to within a constant factor. However, Seminaive evaluation of the magic program uses considerably more space; Magic Tabulation, when it is applicable, improves space utilization.

Example 8.5 Consider the Fibonacci program of Example 8.1. The Burstall-Darlington system, with some user intervention, produces a program that runs in linear time and space. The Magic Templates algorithm transforms the original program to:

$$
\begin{array}{lll}
m\_fib^{10}(N-1) & :- & m\_fib^{10}(N), N > 1. \\
m\_fib^{10}(N-2) & :- & m\_fib^{10}(N), N > 1. \\
fib^{10}(0,1) & :- & m\_fib^{10}(0). \\
fib^{10}(1,1) & :- & m\_fib^{10}(1). \\
fib^{10}(N, X_1 + X_2) & :- & m\_fib^{10}(N), N > 1, fib^{10}(N-1, X_1), fib^{10}(N-2, X_2).
\end{array}
$$

This program runs in linear time and space using Seminaive iteration; Magic Tabulation runs in linear time and constant space. □

Example 8.6 Consider the factorial program of Example 8.2. The Burstall-Darlington system transforms it to a program that runs in linear time and constant space. The Magic Templates algorithm transforms the original program to:

$$\begin{array}{lll} m\_fact^{10}(N) & :- & m\_fact^{10}(N+1). \\ fact^{10}(0,1) & :- & m\_fact^{10}(0). \\ fact^{10}(N+1,X) & :- & m\_fact^{10}(N+1), fact^{10}(N,M), X = (N+1)*M. \end{array}$$

Seminaive iteration executes this in linear time and space; Magic Tabulation runs in linear time and constant space. □

The previous two examples demonstrated that Magic Tabulation sometimes matches or even improves upon the program produced by the Burstall-Darlington system. The following example presents a program that is improved by the Burstall-Darlington system, but for which Magic Tabulation is inapplicable.

Example 8.7 Consider the reverse program of Example 8.4. The Burstall-Darlington system transforms it to a program that runs in linear time and constant space (assuming that the entire list fits in unit space). The Magic Templates algorithm rewrites the original program to:

$$\begin{array}{lll} m\_rev^{10}(X) & :- & m\_rev^{10}(A.X). \\ m\_app^{110}(L_1, A.nil) & :- & m\_rev^{10}(A.X), rev^{10}(X, L_1). \\ m\_app^{110}(L_1, L_2) & :- & m\_app^{110}(X.L_1, L_2). \\ rev^{10}(nil, nil) & :- & m\_rev^{10}(nil). \\ rev^{10}(A.X, L) & :- & m\_rev^{10}(A.X), rev^{10}(X, L_1), app^{110}(L_1, A.nil, L). \\ app^{110}(nil, L, L) & :- & m\_app^{110}(nil, L). \\ app^{110}(X.L_1, L_2, X.L) & :- & m\_app^{110}(X.L_1, L_2), app^{110}(L_1, L_2, L). \end{array}$$

Seminaive evaluation of the magic program also runs in linear time and constant space, if we assume that a structure-sharing implementation is used that stores a single copy of the list and maintains tuples using offsets into this list. However, it is slower than the Burstall-Darlington program by a factor of at least two, since the computation of the magic facts repeats much of the work in applications of the modified rules; similarly there is at least a factor of two in additional space.

Magic Tabulation is not applicable since Condition (1) is violated. However, the other conditions are preserved, which suggests the following execution scheme that can be seen as an extension of Magic Tabulation. Observe that: (1) app^{110} and rev^{10} calls can both be evaluated in constant space and linear time, and (2) Given a reverse fact, we can identify the corresponding call — an m_rev^{10} fact — and, by invertibility, the parent m_rev^{10} call. This parent call together with the rev^{10} fact is enough to initiate the next app^{110} computation. Further, only two facts (the m_rev^{10} and rev^{10} facts used to generate the app^{110} call) need be stored while waiting for the app^{110} call to return. This in turn generates a new rev^{10} fact, and we can iterate. □

The following example illustrates that the Burstall-Darlington system can significantly improve the program in some cases where Magic Tabulation is not applicable. Further, there does not seem to be any straightforward extension of Magic Tabulation that matches the program produced by the Burstall-Darlington system.

Example 8.8 Consider the list of factorials program of Example 8.3. The Burstall-Darlington system transforms it to the following program that runs in linear time and constant space (assuming that each *flist* fact takes only unit space):

$$
\begin{array}{lll}
flist^{10}(N+1, M.L) & :- & g^{100}(N, M, L). \\
g^{100}(0, 1, nil). & & \\
g^{100}(N+1, (N+2)*M, M.L) & :- & g(N, M, L).
\end{array}
$$

The Magic Templates algorithm transforms the original program to:

$$
\begin{array}{lll}
m\_fact^{10}(N) & :- & m\_fact^{10}(N+1). \\
m\_fact^{10}(N+1) & :- & m\_flist^{10}(N+1). \\
m\_flist^{10}(N) & :- & m\_flist^{10}(N+1), fact^{10}(N+1, M). \\
flist^{10}(0, nil) & :- & m\_flist^{10}(0). \\
flist^{10}(N+1, M.L) & :- & m\_flist^{10}(N+1), fact^{10}(N+1, M), flist^{10}(N, L). \\
fact^{10}(0, 1) & :- & m\_fact^{10}(0). \\
fact^{10}(N+1, X) & :- & m\_fact^{10}(N+1), fact^{10}(N, M), X = (N+1)*M.
\end{array}
$$

The magic program runs in linear time and space using Seminaive iteration. Magic Tabulation is not applicable since Condition (1) is violated, and we cannot separate the computation of the magic facts.

It looks like the best strategy using Magic Tabulation, or rather an extension of it, is to materialize $fact^{10}$, which can be done in linear time and space, and to then use Magic Tabulation to evaluate $flist^{10}$, essentially treating $fact^{10}$ as a base relation.

We could use an approach similar to that used in the naive reverse example. That is, briefly, we notice that both $fact^{10}$ and $flist^{10}$ calls can be set up and solved in constant space and linear time. However, throwing away all the intermediate facts generated in evaluating a $fact^{10}$ call causes a lot of redundant computation because the same call can be generated in many ways. □

9 Conclusion

We have reviewed a collection of results on bottom-up evaluation of logic programs, and attempted to place them in the perspective of a coherent approach to logic program evaluation. The main conclusions, in our opinion, are the following:

1. No one evaluation method is superior for all programs; an intelligent compiler must choose an appropriate method for a given program.

2. Efficient bottom-up evaluation methods are available that are sound and complete with respect to the declarative semantics.

Choosing a good evaluation method for a program is a hard problem, and we need good heuristics — based on analytical, as well as statistics-driven, cost-estimation techniques — to guide this choice. We have presented some results that shed light on the choice of Magic Templates, Counting, and Prolog evaluation methods for certain classes of programs. These results are intended to illustrate the issues, rather than to be incorporated into a compiler directly. Techniques that can be directly used in a compiler are sorely needed, and represent an area for research.

The bottom-up approach that we have presented — and there is related work to which much of our discussion applies — is based upon program transformations followed by bottom-up fixpoint evaluation. The Magic Templates transformation ensures that the resulting fixpoint computation generates no irrelevant facts. If a fixpoint method such as Seminaive evaluation, which does not repeat inferences, is used, this implies that the performance of the bottom-up approach is never worse than that of a top-down method (in particular, Prolog) by more than a constant factor, assuming that we have constant-time access to memoed facts.

This assumption of constant-time access is clearly warranted for ground facts, where expected constant-time access can be guaranteed by the use of an appropriate hashing scheme (see, for example, Carter and Wegman [CW79].) It is an important open problem whether the same efficiency can be achieved in practice in the presence of nonground facts, where instead of looking for an exact match or a given ground fact we need to search for general terms that subsume a given general term.

The bottom-up approach can be viewed as a tabulation strategy that saves all generated goals and facts until the end of the computation, and thus avoids repeated computation by looking up previously computed results. On certain programs, for example the fibonacci program, this results in polynomial execution where a non-tabulating method such as Prolog takes exponential time.

A number of additional optimizing program transformations are known, and in conjunction with Magic Templates, obtain significant additional gains in performance when applicable. These optimizations rely upon the declarative semantics of the program, and thus the freedom from an operational semantics often increases the efficiency of the computation.

While the performance of the bottom-up approach is never worse than a top-down method by more than a constant factor, this constant can be significant for certain classes of programs, especially when we take the cost of duplicate elimination into account. (Prolog does no duplicate elimination, and is incomplete for this reason, in conjunction with its depth-first search.) Refinements in fixpoint evaluation techniques may reduce this factor, although it is likely that there will always be classes of programs in which an execution method such as Prolog is superior.

One promising refinement is to determine by a compile-time analysis that no duplicates will be generated, and thus avoid checking for duplicates at run-time. It seems likely that no duplicates are generated in a large class of Prolog programs; thus, such techniques could be widely applicable. Other useful refinements are based upon controlling the order of rule applications, and take advantage of properties such as commutativity.

A major drawback of the bottom-up approach is the amount of memory required, since all generated goals and facts are stored. However, it may often be possible to reduce the memory requirements by a compile-time analysis that indicates how we can discard facts that

are no longer needed and re-allocate the freed space. Techniques for such compile-time garbage collection have been considered in the functional programming literature, and adapting and extending them to the bottom-up approach for logic programs is an important area for future research.

References

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, 1979.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bir80] R. S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, December 1980.

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

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[BR88] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 439–517, Los Altos, California, 94022, 1988. Morgan Kaufmann.

[BS89] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in datalog programs. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, Philadelphia, Pennsylvania, March 1989.

[CGKV88] Stavros S. Cosmadakis, Haim Gaifman, Paris Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the Twentieth Symposium on the Theory of Computation*, pages 477–490, Chicago, Illinois, May 1988.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[CW89] S.R. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 200–216, Philadelphia, Pennsylvania, March 1989.

[DB73] John Darlington and R. M. Burstall. A system which automatically improves programs. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pages 479–485, 1973.

[Don86] G. Dong. On distributed processing of datalog queries by decomposing databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–35, Portland, Oregon, June 1986.

[DW87] Suzanne W. Dietrich and David S. Warren. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987. Unpublished Manuscript.

[DW89] S.K. Debray and D.S. Warren. Functional compositions in logic programs. *Transactions on Programming Languages*, page To appear, 1989.

[GMSV87] Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 106–115, Ithaca, New York, June 1987.

[GST89] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. Unpublished manuscript., 1989.

[Hel88] A. Richard Helm. Detecting and eliminating redundant derivations in deductive database systems. Technical Report RC 14244 (#63767), IBM Thomas Watson Research Center, December 1988.

[HL78] Gerard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HL89] J. Han and L. Liu. Processing multiple linear recursions. In *Proceedings of the North American Conference on Logic Programming*, page To appear, Cleveland, Ohio, October 1989.

[Ioa86] Yannis E. Ioannidis. Bounded recursion in deductive databases. *Algorithmica*, 1(4):361–385, October 1986.

[Ioa89] Yannis E. Ioannidis. Commutativity and its role in the processing of linear recursion. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 155–163, Amsterdam, The Netherlands, August 1989.

[IW88] Yannis E. Ioannidis and Eugene Wong. Towards an algebraic theory of recursion. Technical Report 801, Computer Sciences Department, University of Wisconsin-Madison, October 1988.

[KL86] Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive databases. In *Proceedings of the Advanced Database Symposium*, Tokyo, Japan, 1986.

[KL88] Michael Kifer and Eliezer L. Lozinskii. SYGRAF: Implementing logic programs in a database style. *IEEE Transactions on Software Engineering*, 1988.

[Mah85] Michael J. Maher. *Semantics of Logic Programs.* PhD thesis, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1985.

[Mah89] M.J. Maher. A transformation system for deductive database modules with perfect model semantics. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, page To appear, Bangalore, India, December 1989.

[MFPR89] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Extended magic sets. In preparation., 1989.

[MR90] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990. To appear.

[MSPS87] Alberto Marchetti-Spaccamela, Antonella Pelaggi, and Domenico Sacca. Worst-case complexity analysis of methods for logic query implementation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 294–301, San Diego, California, March 1987.

[Nau88a] Jeffrey F. Naughton. Benchmarking multi-rule recursion evaluation strategies. Technical Report CS-TR-141-88, Princeton University, 1988.

[Nau88b] Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.

[Nau89a] Jeffrey F. Naughton. Data independent recursion in deductive databases. *Journal of Computer and System Sciences*, 38(2):259–289, April 1989.

[Nau89b] Jeffrey F. Naughton. Minimizing function-free recursive definitions. *Journal of the Association for Computing Machinery*, 36(1):69–91, January 1989.

[NR89] Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget history without repeating it. In preparation., 1989.

[NRSU89a] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[NRSU89b] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–242, Portland, Oregon, May 1989.

[NS87] Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 227–236, San Diego, California, March 1987.

[NS89] Jeffrey F. Naughton and Yehoshua Sagiv. Minimizing expansions of recursions. In Hasan Ait-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 1, pages 321–349, San Diego, California, 1989. Academic Press, Inc.

[PW83] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the twenty-first Annual Meeting of the Association for Computational Linguistics*, 1983.

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

[Ram90] Raghu Ramakrishnan. Parallelism in logic programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990. To appear.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.

[Red84] U.S. Reddy. Transformation of logic programs into functional programs. In *Proceedings of the Symposium on Logic Programming*, pages 187–196, Salt Lake City, Utah, September 1984.

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

[RSUV89] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Vardi. Proof-tree transformation theorems and their applications. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, Pensylvannia, March 1989.

[Sag88] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698, Los Altos, California, 94022, 1988. Morgan Kaufmann.

[Sar89] Yatin Saraiya. Linearizing nonlinear recursions in polynomial time. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 182–189, Philadelphia, Pennsylvania, March 1989.

[Sek89] H. Seki. On the power of Alexander templates. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–159, Philadelphia, Pennsylvania, March 1989.

[SS88] Seppo Sippu and Eljas Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformations of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, July 1984.

[Ull89] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[Var88] Moshe Y. Vardi. Decidability and undecidablity results for boundedness of linear recursive queries. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 341–351, Austin, Texas, March 1988.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[VG86] Allen Van Gelder. A message passing framework for logical query evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 349–362, Washington, DC, May 1986.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, South Carolina, 1986.

[Vie87] Laurent Vieille. Database complete proof procedures based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, 1987.

[WS89] O. Wolfson and A. Silberschatz. Sharing the load of logic program evaluations. In *Proceedings of the 7th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, March 1989.

[ZYT] W. Zhang, C. T. Yu, and D. Troy. A necessary and sufficient condition to linearize doubly recursive programs in logic databases. Unpublished manuscript, Department of EECS, University of Illinois at Chicago.

A Proofs from Section 7

Lemma 7.1 *For a program P and a query q, there is a tree of height k with a goal $m\_p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ if and only if there is a tree of height k with $m\_p(c)$ at the root in $\mathcal{T}_P(P, q)$. Similarly, there is a tree of height k with a fact $p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ if and only if there is a tree of height k with $p(c)$ at the root in $\mathcal{T}_P(P, q)$.*

Proof The proof is by induction on k. For the "only if" direction, we use the following induction hypothesis: For a program P and a query Q,

- If there is a tree of height k with a goal $m\_p(c)$ at the root in $\mathcal{T}_P(P, q)$, then $m\_p(c)$ appears in $GC(P^{mg})$.
- There is a tree of height k with a goal $m\_p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ only if there is a tree of height k with $m\_p(c)$ at the root in $\mathcal{T}_P(P, q)$.
- There is a tree of height k with a fact $p(c)$ at the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ only if there is a tree of height k with $p(c)$ at the root in $\mathcal{T}_P(P, q)$.

For the basis, $k = 0$, the only trees of height zero are the base facts. By definition of $\mathcal{T}_P(P, q)$, the only Prolog tree of height 0 with a magic fact at the root is $m\_q(c)$, where $q(c)$? was the original query. By definition of P^{mg}, this fact appears in $GC(P^{mg})$. By definition of P^{opt}, there is a fact $m\_q(c)$ in P^{opt}, so by definition of $\mathcal{T}_{NSN}(P^{opt})$, there is a tree of height 0 with $m\_q(c)$ as the root in $\mathcal{T}_{NSN}(P^{opt})$. Also, by definition of $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, this tree is in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$.

Again by definition of $\mathcal{T}_P(P, q)$, the only trees of height 0 in $\mathcal{T}_P(P, q)$ with facts of the form $p(c)$ at the root are trees corresponding to the facts (rules without bodies) in P. By definition of P^{opt}, there is a fact $p(c)$. in P^{opt} for each such base fact $p(c)$ in P, and by definition of $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, there is a tree of height 0 in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ for each such fact. Hence for each tree of height 0 in $\mathcal{T}_P(P, q)$ with a fact $p(c)$ at the root, there is a corresponding fact in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$.

Now suppose that the induction hypothesis holds for all $i < k$. Consider a tree T_P in $\mathcal{T}_P(P, q)$ of height k with a goal $m\_p_j(c_j)$ at the root. By definition of $\mathcal{T}_P(P, q)$, there are two cases to consider:

1. The root $m\_p_j(c_j)$ has a single subtree with root $m\_p(c)$, where P contains the rule $p(X)$:- $p_j(X_j)$, ..., the rule was invoked with goal $m\_p(c)$, the unifier of X and c is θ, and $c_j = X_j\theta$. In this case P^{opt} contains the rule $m\_p_j(X_j)$:- $m\_p(X)$.

The height of this subtree must be $k-1$. But then by the induction hypothesis, $m\_p(c)$ must appear in $GC(P^{mg})$, hence so must $m\_p_j(c_j)$. Also by the induction hypothesis, there is a tree in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ of height $k-1$ with $m\_p(c)$ as the root. This, together with the definition of $\mathcal{T}_{NSN}(P^{opt}, q)$, implies that there is a tree of height k in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ with $m\_p_j(c_j)$ at the root.

2. The root $m\_p_j(c_j)$ has $j \geq 2$ subtrees with roots $m\_p(c)$, $p_1(c_1)$, ..., $p_{j-1}(c_{j-1})$, where P contains the rule $p(X) :- p_1(X_1),\ldots,p_{j-1}(X_{j-1}),\ldots$, and there are substitutions θ_1, ..., θ_{j-1} such that $c_1 = X_1\theta_1$, $c_2 = X_2\theta_2$, ..., $c_{j-1} = X_{j-1}\theta_1$. In this case P^{opt} contains the rule $m\_p_j(X_j) :- p_1(X_1), \ldots, p_{j-1}(X_{j-1})$.

   The height of the tallest subtree of $m\_p_j(c_j)$ is $k-1$, hence by the induction hypothesis each of of $m\_p(c)$, $p_1(c_1)$, ..., $p_{j-1}(c_{j-1})$ is in $GC(P^{mg})$, hence so is $m\_p_j(c_j)$. Also by the induction hypothesis there are trees for each of $m\_p(c), p_1(c_1), \ldots, p_{j-1}(c_{j-1})$ in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$. Furthermore, the height of each of these trees is the same as the height of its corresponding subtree in $\mathcal{T}_P(P, q)$. This, together with the rule $m\_p_j(X_j) :- p_1(X_1),\ldots,p_{j-1}(X_{j-1})$ and the definition of $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ implies that there is a tree in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ with $p_j(c_j)$ at the root of height k.

In either case, the lemma holds for the case k.

The only remaining case to consider is that of a tree in $\mathcal{T}_P(P, q)$ with a fact $p(c)$ at the root and of height k. By definition of $\mathcal{T}_P(P, q)$, the root of this tree will have as subtrees the j trees with roots $p_1(c_1),\ldots,p_j(c_j)$, where P contains the rule $p(X) :- p_1(X_1),\ldots,p_j(X_j)$.

Now in order for $p(c)$ to have been generated by Prolog, there must have been a goal $m\_p(c')$ generated such that there is a substitution θ and $c = c'\theta$ and such that θ can be extended to a substitution θ' such that $c_1 = X_1\theta'$, $c_2 = X_2\theta'$, ..., $c_j = X_j\theta'$. But this proves that the fact $p(c)$ must be in $GC(P^{mg})$.

Finally, the height of the tallest of these subtrees must be $k-1$, so by the induction hypothesis there are corresponding trees in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, hence there is a tree of height k with $p(c)$ as the root in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$.

The "if" direction of the lemma is symmetric, and hence is omitted here. □

Lemma 7.2 *For a program P and a query Q, if P is strictly well-behaved, then for each tree with root r in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, there is exactly one tree with root r in $\mathcal{T}_P(P, q)$.*

Proof Assume the contrary, that is, that P is strictly well-behaved, yet there is at least one tree with root r in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ such that there are two trees with root r in $\mathcal{T}_P(P, q)$. Let T of height k be the minimal height tree in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ such that there are two trees, say t_1 and t_2, with root r in $\mathcal{T}_P(P, q)$.

First we claim that t_1 and t_2 must both be of height k. This follows from Lemma 7.1, since if either of t_1 or t_2 were of height $k' \neq k$, there would be a tree in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ of height different than k with r at the root, contradicting the fact that P is strictly well-behaved.

Similarly, Lemma 7.1 and the assumption that T is minimal implies that for any subtree of t_1 or t_2, the fact or goal at the root of the subtree appears as the root of exactly one tree in $Rel(\mathcal{T}_{NSN}(P^{opt}, Q))$.

If r is a goal $m\_p_j(c_j)$ there are four cases to consider:

1. The root $m\_p_j(c_j)$ in t_1 has a single subtree with root $m\_p(c)$, where P contains the rule $p(X) :- p_j(X_j),\ldots$, the rule was invoked with goal $m\_p(c)$, the unifier of X and c is θ, and $c_j = X_j\theta$. Similarly, the root $m\_p_j(c_j)$ in t_2 has a single subtree with root $m\_p'(c')$, where P contains the rule $p'(X') :- p'_j(X'_j),\ldots$, the rule was invoked with goal $m\_p'(c')$, the unifier of X' and c' is θ', and $c'_j = X'_j\theta'$.

   In this case P^{opt} must contain the rules $m\_p_j(X_j) :- m\_p(X)$ and $m\_p_j(X_j) :- m\_p'(X')$, which would imply two distinct trees in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ with root $m\_p_j(X_j)$, contradicting the fact that P is strictly well-behaved.

2. The root $m\_p_j(c_j)$ in t_1 has $j \geq 2$ subtrees with roots $m\_p(c)$, $p_1(c_1)$, $\ldots$, $p_{j-1}(c_{j-1})$, where P contains the rule $p(X) :- p_1(X_1),\ldots,p_{j-1}(X_{j-1}),\ldots$, and there are substitutions $\theta_1,\ldots,\theta_{j-1}$ such that $c_1 = X_1\theta_1$, $c_2 = X_2\theta_2$, $\ldots$, $c_{j-1} = X_{j-1}\theta_{j-1}$. Similarly, the root $m\_p_j(c_j)$ in t_2 has $j' \geq 2$ subtrees with roots $m\_p'(c')$, $p'_1(c'_1)$, $\ldots$, $p'_{j-1}(c'_{j-1})$, where P contains the rule $p'(X) :- p'_1(X_1),\ldots,p'_{j-1}(X'_{j-1}),\ldots$, and there are substitutions $\theta'_1,\ldots,\theta'_{j-1}$ such that $c'_1 = X'_1\theta'_1$, $c'_2 = X'_2\theta'_2$, $\ldots$, $c'_{j-1} = X'_{j-1}\theta'_{j-1}$.

   In this case P^{opt} contains the rule $m\_p_j(X_j) :- m\_p(X), p_1(X_1),\ldots,p_{j-1}(X_{j-1})$ and the rule $m\_p_j(X_j) :- m\_p(X'), p'_1(X'_1),\ldots,p'_{j-1}(X'_{j-1})$, which, again, would give two trees in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ with root $m\_p_j(c)$, contradicting the fact that P is strictly well-behaved.

3. The root $m\_p_j(c_j)$ in t_1 has a single subtree with root $m\_p(c)$, where P contains the rule $p(X) :- p_j(X_j),\ldots$, the rule was invoked with goal $m\_p(c)$, the unifier of X and c is θ, and $c_j = X_j\theta$. On the other hand, the root $m\_p_j(c_j)$ in t_2 has $j' \geq 2$ subtrees with roots $m\_p'(c')$, $p'_1(c'_1)$, $\ldots$, $p'_{j-1}(c'_{j-1})$, where P contains the rule $p'(X) :- p'_1(X_1),\ldots,p'_{j-1}(X'_{j-1}),\ldots$, and there are substitutions $\theta'_1,\ldots,\theta'_{j-1}$ such that $c'_1 = X'_1\theta'_1$, $c'_2 = X'_2\theta'_2$, $\ldots$, $c'_{j-1} = X'_{j-1}\theta_1$.

   Then the rules $m\_p_j(X_j) :- m\_p(X)$ and $m\_p_j(X_j) :- m\_p(X'), p'_1(X'_1),\ldots,p'_{j-1}(X'_{j-1})$ must appear in P^{opt}, which, again, would give two trees in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$ with root $m\_p_j(c)$, contradicting the fact that P is strictly well-behaved.

4. This case is symmetric to the previous case, with the roles of t_1 and t_2 reversed.

In all cases, we reach a contradiction.

The only remaining case to consider is that of r being a fact, say $p(c)$, in which case the root of t_1 will have as subtrees the j trees with roots $p_1(c_1),\ldots,p_j(c_j)$, where P contains the rule $p(X) :- p_1(X_1), \ldots, p_j(X_j)$, and the root of t_2 will have as subtrees the j' trees with roots $p'_1(c'_1), \ldots, p'_{j'}(c_{j'})$, where P contains the rule $p(X) :- p'_1(X'_1), \ldots, p'_{j'}(X'_{j'})$,

Now in order for $p(c)$ to have been generated by Prolog, there must have been goals $m\_p(d)$ and $m\_p(d')$ generated such that there are substitutions θ and θ', and $c = d\theta$ and $c = d'\theta'$, and such that θ can be extended to a substitution ϕ such that $c_1 = X_1\phi$, $c_2 = X_2\phi$, $\ldots$, $c_j = X_j\phi$ while θ' can be extended to a substitution ϕ' such that $c'_1 = X_1\phi'$, $c'_2 = X'_2\phi'$, $\ldots$, $c'_{j'} = X'_{j'}\phi'$. But this proves that the fact $p(c)$ must be the root of two trees in $Rel(\mathcal{T}_{NSN}(P^{opt}, q))$, again contradicting the fact that P is strictly well-behaved.

In all cases we meet with contradiction, completing the proof. □