DEPTH-BOUNDED BOTTOM-UP EVALUATION OF LOGIC PROGRAMS *

JAN CHOMICKI

▷ We present here a *depth-bounded* bottom-up evaluation algorithm for logic programs. We show that it is sound, complete, and terminating for finite-answer queries if the programs are syntactically restricted to $Datalog_{nS}$, a class of logic programs with limited function symbols. $Datalog_{nS}$ is an extension of *Datalog* capable of representing infinite phenomena. Predicates in $Datalog_{nS}$ can have arbitrary unary and limited n-ary function symbols in one distinguished argument. We precisely characterize the computational complexity of depth-bounded evaluation for $Datalog_{nS}$ and compare depth-bounded evaluation with other evaluation methods, top-down and Magic Sets among others. We also show that universal safety (finiteness of query answers for any database) is decidable for $Datalog_{nS}$. ◁

1. INTRODUCTION

1.1. Non-termination of bottom-up query evaluation

Among query processing algorithms in deductive database systems bottom-up evaluation is one of the most popular, as evidenced by the survey of Bancilhon and Ramakrishnan [4] and the textbook of Ullman [47, 48]. Most deductive database systems in use today [27, 35, 50] are based on bottom-up evaluation. This algorithm starts from a finite set of *facts* (a database) and derives from it new facts using deductive Horn *rules*. The algorithm terminates when no new facts can be derived.

*Research partially supported by: NSF grants IRI-87-04614 and IRI-86-09170, ARO grant DAAL-03-88-K0087, and the University of Maryland Institute of Advanced Computer Studies.

Address correspondence to Jan Chomicki,Computing and Information Sciences, Kansas State University, Manhattan, KS 66506-2302, E-mail: chomicki@cis.ksu.edu.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

It should be immediately clear that in the presence of function symbols and recursion in rules infinitely many facts may be derived, and consequently bottom-up evaluation may fail to terminate.

Example 1.1. The following rules describe the schedule of meetings of a given professor with his students:

$$meets(T, X) \leftarrow meets\_first(T, X).$$
$$meets(T + 1, Y) \leftarrow next(X, Y), meets(T, X).$$

The recursive rule expresses the following: *"if a student X meets the professor at time T and Y is the next student after X, then Y meets the professor at time $T+1$."* Let us consider a database containing the following facts:

$$meets\_first(0, emma).$$
$$next(emma, kathy).$$
$$next(kathy, emma).$$

The rules can be used to derive the following infinitely many facts from the database:

$$meets(0, emma).$$
$$meets(1, kathy).$$
$$meets(2, emma).$$
$$meets(3, kathy).$$
$$\ldots$$

where $1 = 0 + 1$, $2 = (0 + 1) + 1$, etc.

The termination problem has two distinct subproblems. First, can bottom-up evaluation of finite-answer queries be always made to terminate? In Example 1.1 the query *"List all students that meet the professor at time 1"* has a finite answer, namely *kathy*. So does the query *"List all students that meet the professor at some time"*. However, straightforward bottom-up evaluation of the above queries requires the computation of the infinite relation *meets*. The second subproblem involves queries with infinite answers, for example the query *"List all time instants when the professor meets emma"*.

1.2. $Datalog_{nS}$ - an extension of Datalog

In this paper we address the issue of termination of bottom-up evaluation of finite-answer queries in the context of $Datalog_{nS}$, a class of logic programs with limited function symbols [8, 10, 11, 12]. In $Datalog_{nS}$ programs the type and the occurrences of function symbols are restricted in the following way: function symbols can only be unary or limited n-ary (having a single distinguished argument), and they can appear in a single distinguished argument of predicates. In addition to $Datalog_{nS}$ we study its subset $Datalog_{1S}$ where only a single unary function symbol (written in postfix as $+1$) is allowed. The program in Example 1.1 is a valid $Datalog_{1S}$ program. $Datalog_{nS}$ is decidable [8, 16]. Moreover, in the context of $Datalog_{nS}$ infinite query answers can be finitely represented [11, 12].

$Datalog_{nS}$ programs have potentially many applications in knowledge-based systems, for example such as temporal reasoning, event scheduling, planning, and pathfinding. We also envision that $Datalog_{nS}$ programs will be used as an input language for intelligent office tools like a calendar or a personal planner. Subsequently, these tools can provide answers to user queries using the algorithms developed in this paper.

The syntactic restrictions introduced above can be motivated in the following way. The distinguished argument where function symbols may appear plays the role of a *state index* or a *time instant.* A set of facts with the same distinguished argument may be looked upon as a set of facts holding in a state. This set is finite and may be treated as a *snapshot* – a relational, function-free, database. Unary function symbols can be seen as denoting *operators* that map states to states (+1 maps a state to the *next* state). The above restrictions on the type and the occurrences of function symbols are essential to guarantee the termination of all finite-answer queries. If any of them is lifted, the resulting class of logic programs becomes undecidable.

Example 1.2. In Example 1.1 the distinguished variable T denoted a *time instant.* Here it plays the role of a *state* (situation). Function symbols correspond to *operators* available to a robot [17]. For example, $move(t, x, y)$ stands for " robot moving from position x to position y in situation t" (this is an example of a limited 3-ary function symbol, i.e., a function symbol with a single distinguished argument). A complex term corresponds to a sequence of robot moves (a path). The constant 0 denotes the empty path. The predicate $path(t, x, y)$ is true if t is a path connecting x and y. The predicate $mem(t, x, y)$ is true if (x, y) is an edge in the path t.

$path(0, X, X) \leftarrow position(X).$
$path(move(T, Y, Z), X, Z) \leftarrow connected(Y, Z), path(T, X, Y).$
$mem(move(T, Y, Z), Y, Z) \leftarrow path(T, X, Y), connected(Y, Z).$
$mem(move(T, Y, Z), U, V) \leftarrow path(T, X, Y), connected(Y, Z), mem(T, U, V).$
$w(X, Y, U, V) \leftarrow path(T, X, Y), mem(T, U, V).$

The query *"List all (x, y, u, v) such that $w(x, y, u, v)$ holds"* returns quadruples of points (x, y, u, v) such that the robot can move from position x to y through a path containing the edge (u, v).

We argue here that it makes sense to define and study syntactically restricted logic programming languages, particularly those for which query termination can be guaranteed, like *Datalog* or $Datalog_{nS}$. Such languages, although insufficient for general purpose programming, are suitable as *concept definition languages*. For example, the concept of *transitive closure* can be defined in *Datalog* and the concepts of *infinite periodic set* or *repeating path* can be defined in $Datalog_{nS}$. If query termination can not be guaranteed an undesirable situation arises in which the user can formulate rules to which no well-defined concept corresponds. In the cases of *Datalog* and $Datalog_{nS}$ query termination is guaranteed and every program corresponds to a well-defined concept. Moreover, in those cases specific bounds on the complexity of query evaluation are established. Also, the research in this direction should pave the way for query evaluation methods that not only terminate for syntactically restricted classes of logic programs but also work efficiently for arbitrary logic programs.

1.3. Depth-bounded evaluation

We want to be able to evaluate a finite-answer query without computing the entire least fixpoint which is infinite. To achieve this goal, we have to make use of the finiteness of the query answer. Consider only queries which are single atoms, i.e., $p(t_1, \ldots, t_n)$. In $Datalog_{nS}$ the finiteness of the answer is guaranteed if the first (distinguished) argument of the query atom is a ground term or an existentially quantified variable (see the queries in Example 1.1). Therefore, we have to find a way to propagate the information about the first argument from the query to the program.

This is a well-known problem in the area of deductive databases, studied under the names of *pushing selections* or *projections* into rules. There are many methods [4, 28, 48] that achieve either of the above goals by rule rewriting. Surprisingly enough, none of them seems to be able to guarantee the termination of bottom-up evaluation of finite-answer $Datalog_{nS}$ queries. Some of the methods, e.g., Magic Sets defined by Bancilhon, Maier, Sagiv and Ullman [3], introduce additional termination problems of their own, as noticed by Ramakrishnan [33] and Seki [39]. Therefore, we obtain termination in a different way. Essentially, we show that in the context of $Datalog_{nS}$, it is sufficient to consider ground refutations containing terms of bounded depth. Consequently, in bottom-up evaluation only facts with terms of bounded depth have to be generated. We provide tight upper and lower bounds on term depth in refutations. We call our algorithm *depth-bounded evaluation*. Depth-bounded evaluation is a bottom-up counterpart of bounded depth-first search proposed by Stickel [42].

Depth-bounded evaluation works well for queries with a ground or existentially quantified distinguished argument. However, in some cases even queries with the distinguished argument which is a free variable have finite answers. Therefore, depth-bounded evaluation should be complemented by a *safety tester* – an algorithm that checks whether the query answer is finite. We show that testing both *relative safety* (finiteness of the answer for a given set of rules and a given set of facts) and *universal safety* (finiteness of the answer for a given set of rules and every set of facts) is decidable for $Datalog_{nS}$. In fact, depth-bounded bottom-up evaluation can be used for testing relative safety (report *"unsafe"* when the generated facts get "too large"). This shows that $Datalog_{nS}$ is more akin to *Datalog* (for which both kinds of safety are decidable) than to full Prolog (for which neither is recursively enumerable [19, 40]).

It is interesting to compare depth-bounded evaluation with other query processing algorithms for $Datalog_{nS}$[8]. The data complexity[1] of the latter algorithms matches the corresponding lower bounds: query evaluation is PSPACE-complete for $Datalog_{1S}$, EXPTIME-complete for $Datalog_{nS}$. This is not the case for depth-bounded evaluation which requires exponential time for $Datalog_{1S}$ and double exponential time for $Datalog_{nS}$. However, we have shown elsewhere [9] that for $Datalog_{1S}$

[1] Informally: complexity as a function of the database size, not the size of the rules.

a polynomial bound on refutation size guarantees that depth-bounded bottom-up evaluation will terminate in polynomial time. This property was used to identify several polynomial-time computable subclasses of $Datalog_{1S}$.

There are also other reasons for studying depth-bounded bottom-up evaluation. First, bottom-up evaluation, like other resolution-based algorithms, constructs bindings for existentially quantified variables in the query. Therefore, it can be used to provide yes-no (boolean) answers, as well as enumerate all the answer substitutions. This is not the case for the lower bound algorithms (mentioned above) which yield only yes-no answers. Second, rule optimization methods [4, 28, 48] work under the assumption that queries are evaluated bottom-up. We would like to be able to combine those methods with depth-bounded evaluation to obtain a practical query processing system for $Datalog_{nS}$. Third, it seems better to have a single evaluation mode (bottom-up or top-down) that works for arbitrary logic programs rather than many incompatible procedures applicable only to restricted classes of logic programs.

1.4. Summary of the paper

The following is the summary of the paper. In Section 2 we define the syntax and the semantics of $Datalog_{nS}$. In Section 3 we introduce depth-bounded evaluation and show upper and lower bounds on its execution time for $Datalog_{nS}$. In Section 4 we compare depth-bounded evaluation with other query evaluation algorithms for $Datalog_{nS}$. In particular, we consider the lower bound algorithms [8] and the algorithms that construct finite representations of infinite query answers [11, 12]. We also discuss the algorithm of Joyner [18] and top-down algorithms based on resolution. We show a close connection between depth-bounded evaluation and bounded depth-first search. Finally, we discuss Magic Sets [3]. In Section 5 we show that both relative and universal safety is decidable for $Datalog_{nS}$. We also relate our results to other works dealing with the safety problem. Finally, in Section 6 we summarize the results of this paper and discuss the prospects for further work.

2. BASIC NOTIONS

We begin by recalling some standard definitions and terminology used in logic programming and *Datalog*. For further detail, the reader is referred to the books of Lloyd [26] and Ullman [47]. The definitions below attempt to capture the intuition that in $Datalog_{nS}$ we deal with two separate domains: the domain of standard, unstructured database constants (called here *data* terms) and the domain of inductively built objects (called *functional* terms).

2.1. Syntax

An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and each t_i is a term (in the usual first-order logic sense). A term or an atom is said to be *ground* if it is variable-free. A *fact* is a ground atom. A *database* is a finite set of facts. Given a set of facts D, the *extension* of a predicate symbol p of arity n is the set of all facts of the form $p(t_1, \ldots, t_n)$ in D. A *rule* is a formula written as $A \leftarrow B_1, \ldots, B_m$ (in clausal form: $A \vee \neg B_1 \vee \cdots \vee \neg B_m$), where $A, B_1, \ldots, B_m$

are atoms; A is called the *head* and $B_1, \ldots, B_m$ the *body* of the rule; individual B_i ($1 \le i \le m$) are called *subgoals*. A *logic program* is a finite set of rules together with a database. All variables in rules are implicitly universally quantified.

A *query* is an atom in which the functional argument is either a variable or a ground term. We will also consider queries closed with existential quantifiers. A query with at least one free variable is *open*; it is *closed* otherwise. A *fact* query is a ground atomic query. A *simple* query is an atom whose arguments are all distinct free variables. (Although every query defined above can be represented as a simple query and an additional rule, we have found it more convenient not to make this assumption in the first part of the paper, namely Sections 3 and 4.) A *formula* is a logic program with the possible addition of the negation of a query (thus a formula is in clausal form).

We distinguish *extensional database* (*EDB*) and *intensional database* (*IDB*) predicates. The EDB predicates correspond to the relations defined by the database, whereas the IDB predicates correspond to the derived relations, that is, those that are defined by the rules. EDB predicates may thus appear in database facts and in bodies of rules, whereas IDB predicates can only appear in rules and queries, and not in database facts.

Datalog [47] is the language of *function-free* logic programs, that is, logic programs in which the only terms are constants or variables. We call such terms *data* terms.

Datalog$_{nS}$ is an extension of *Datalog* in which atoms and terms may have a single distinguished (first) argument, which is said to be *functional*, in addition to the usual *data* arguments. Functional arguments are *functional terms*, which are built from a distinguished functional constant 0, data constants, functional variables, data variables (distinct from functional variables), and function symbols. Other (data) arguments of an atom or a term can only be data terms. Also, data terms can appear only in data arguments. This implies that every functional term contains either 0 or a single occurrence of a single functional variable. For instance, if T is a functional variable, then 0, T, $f(T)$ and $g(T, a)$ are functional terms but $g(T, T)$ is not. A functional term or atom can be viewed as a tree where each node has at most one son that is not a leaf. We distinguish between *functional* predicates (those with a functional argument) and *data* predicates (those with data arguments only). The *depth* of a variable or a constant is 0. The depth of a complex functional term is equal to one plus the depth of its functional argument.

Often the definition of *Datalog* allows two special built-in predicates: equality ($=$) and inequality ($\neq$). Most of the results in this paper hold if equalities and inequalities between data terms are allowed. The proof of the decidability of universal safety, however, requires the exclusion of inequalities.

We study separately *Datalog*$_{1S}$ programs which are *Datalog*$_{nS}$ programs with exactly one unary function symbol ($+1$). This class was first defined in our earlier work [10] and has the same expressive power as function-free *Templog* studied by Abadi, Manna, and Baudinet [1, 5].

2.2. Restrictions

We make a number of assumptions:

- rules are *range-restricted*, i.e., every variable is *limited* [47]. A variable is limited if it appears in a literal in the body of the rule or is equated by an equality to a limited variable (we assume no constants in rules).
- equalities are eliminated by picking one variable for every class of equated variables and substituting it for all these variables. After this transformation, in a range-restricted rule every variable has to appear in some literal in the body.
- rules do not contain ground terms. Such terms can be eliminated by introducing additional predicates.
- rules are *normal*. A $Datalog_{nS}$ rule r is *semi-normal* if r contains at most one functional variable and if this variable appears in r, it has to appear as the functional argument of some atom. A $Datalog_{nS}$ rule r is *normal* if it is semi-normal and functional terms in r are of depth at most 1. For convenience, we also assume that in a set of normal rules functional variables in different rules are identical, i.e., there is just one functional variable in a program, usually named T. For every set of $Datalog_{nS}$ rules, there is an equivalent set of normal $Datalog_{nS}$ rules. We show how to obtain it elsewhere [8].
- non-unary function symbols have been eliminated by instantiating data variables appearing in functional terms with all data constants appearing in the program and creating a new unary function symbol for every such combination. After this transformation, the number of function symbols and rules in the program may be database-dependent. This, however, does not affect any of the results obtained here, except universal safety.

The last assumption justifies the name $Datalog_{nS}$ (*Datalog* with n successors), as we are considering only programs with unary function symbols (*successors*). Treating every different successor as a letter and 0 as the empty string, we can view ground functional terms that use only successors as strings over a finite alphabet.

Notation: we generally follow the Prolog notation. $f_k \cdots f_1(t)$ denotes the complex term $f_k(\cdots f_1(t)\cdots)$, n denotes $(\cdots((0 \underbrace{+1) + 1)\cdots + 1)}_{n\ times}$, and $T + n$ denotes $(\cdots((T \underbrace{+1) + 1)\cdots + 1)}_{n\ times}$. A vector of terms is written as $\bar{x}$, $\bar{X}$ (if the terms are variables), or $\bar{a}$ (if the terms are ground).

2.3. Semantics

By an *interpretation* we mean a Herbrand interpretation of a formula (which is identified with a subset of its Herbrand base), and by a *model* – a *Herbrand model*. Both the Herbrand universe and the Herbrand base of a $Datalog_{nS}$ formula which has at least one function symbol are infinite. By the results of Van Emden and Kowalski[51], every $Datalog_{nS}$ program S has a least (Herbrand) model M_S which is the intersection of all Herbrand models of S and contains all the facts implied by S. The least model M_S is considered the intended meaning of a program S. This model can be infinite in the presence of function symbols – see Example 1.1.

Van Emden and Kowalski define a mapping T_S from Herbrand interpretations to Herbrand interpretations:

$$T_S(I) = \{A : A\ is\ a\ fact\ in\ S\ or\ A_1 \in I, \ldots, A_k \in I\ and\ A \leftarrow A_1, \ldots, A_k$$

is a ground instance of a rule in $S\}$

The successive iterations of T_S are defined as follows:

$T_S^0 = \emptyset$.
$T_S^{k+1} = T_S(T_S^k)$.
$T_S^\omega = \bigcup_{k \geq 0} T_S^k$.

T_S^ω is equal to the least fixpoint of T_S denoted by $lfp(T_S)$. Moreover, $lfp(T_S) = M_S$ [51].

2.4. Query evaluation

Assuming M is an interpretation and both Π and Φ are formulas, $M \models \Phi$ means "M is a model of Φ " and $\Pi \vdash \Phi$ means "Π implies Φ".

A substitution θ to the free variables of a query Q is an *answer substitution* w.r.t. a set of facts X if $Q\theta$ is ground and $X \models Q\theta$. If Q is open, then the *answer* to Q w.r.t. X, denoted by $ans(X, Q)$, is the set of all answer substitutions to Q w.r.t. X. If Q is closed, then the answer to Q w.r.t. X is "yes" if $X \models Q$, "no" - otherwise. The *evaluation* of Q in X is the computation of the answer of Q w.r.t. X. Queries considered in this paper are clearly monotonic, i.e., $ans(X, Q) \subseteq ans(Y, Q)$ if $X \subseteq Y$.

For a logic program S and query Q, the answer substitutions of interest are those w.r.t. the least Herbrand model M_S, i.e., $ans(M_S, Q)$. If function symbols are present in rules, there may be infinitely (but countably) many ground answer substitutions w.r.t. M_S. An algorithm evaluating a query Q is *sound* if every computed substitution is an element of $ans(M_S, Q)$; *complete* if every substitution in $ans(M_S, Q)$ is computed in finite time; *terminating* if its computation terminates. The task of obtaining all answer substitutions to a query w.r.t. M_S (or a finite representation of those) is called *generation*, the task of providing a yes-no answer w.r.t. M_S – *recognition*. A query q is a *finite-answer* query if $ans(M_S, Q)$ is finite.

Because M_S may be infinite, it is not explicitly given and answer substitutions w.r.t. M_S can not be computed directly from the definition. Nevertheless, if Q is a quantifier-free positive query, the condition $M_S \models \exists Q$ (where $\exists Q$ is Q closed with existential quantifiers) is equivalent to $S \vdash \exists Q$. In particular, if Q is already ground, $M_S \models Q$ is equivalent to $S \vdash Q$.

The condition $S \vdash \exists Q$ is equivalent to the unsatisfiability of the formula (the set of clauses) $S \cup \{\neg \exists Q\}$. Thus, procedures that determine the existence of a Herbrand model of $S \cup \{\neg \exists Q\}$ can be used for recognition.

3. DEPTH-BOUNDED BOTTOM-UP EVALUATION

In this section, we show a bottom-up evaluation algorithm that computes the least fixpoint of any set of Horn rules. Because of the presence of function symbols in rules that may lead to an infinite least fixpoint, the algorithm may fail to terminate for $Datalog_{nS}$. We then show a depth-bounded bottom-up evaluation algorithm for $Datalog_{nS}$ that terminates for finite-answer queries. We prove its correctness and analyze its complexity. In the next section, we compare this algorithm with other evaluation algorithms for $Datalog_{nS}$. In the following section, we study the problem of detecting infinite query answers for $Datalog_{nS}$ (safety).

3.1. Bottom-up evaluation

We describe here the standard algorithm for bottom-up evaluation of logic programs. This algorithm is a straightforward implementation of the T_S operator [51] defined in the previous section. It may also be seen as a form of *hyperresolution* [36, 51]. In the sequel, we will assume that a logic program $Z \cup D$ consists of a finite set of rules Z and a finite database D.

An application $Z(I)$ of a set of rules Z to a database I is defined on a rule-by-rule basis. Take a rule $A \leftarrow A_1 \ldots A_m \in Z$. For every ground substitution θ such that for all $i = 1 \ldots m$, $A_i\theta \in I$, we have that $A\theta \in Z(I)$. Figure 1 describes the bottom-up algorithm **B**. After ω iterations of the loop, $L = lfp(T_{Z\cup D}) = M_{Z\cup D}$. If $M_{Z\cup D}$ is infinite, ω iterations are necessary to compute it.

```
L' := D
repeat
    L := L'
    L' := Z(L)
until L = L'
```

Figure 1. Bottom-up evaluation: algorithm **B**

This algorithm can be easily adapted to evaluate a query Q in the least Herbrand model $M_{Z\cup D}$. It is enough to add the evaluation of Q w.r.t. L *inside* the loop. This is correct because in every iteration $L \subseteq M_{Z\cup D}$ and the queries that we consider are monotonic. Moreover, after finitely many iterations L is finite because the rules are range-restricted. Therefore, Q can be evaluated in L as in a relational database.

The above algorithm is usually termed *naive* evaluation [47], and can be improved in many ways. In particular, *semi-naive* evaluation [47] is an incremental variant of this algorithm. In the rest of this paper, we will, however, assume naive evaluation for simplicity. Our results can be easily adapted to its variants, like semi-naive evaluation.

Clearly, if a query has an infinite answer, generation of this answer will not terminate. But even recognition can not be guaranteed to terminate if the logic programming language under consideration is capable of expressing the computations of an arbitrary Turing machine. This is the case of unrestricted logic programs, as shown by Andreka and Nemeti [2], and Tärnlund [45].

$Datalog_{nS}$ is an intermediate case. There, least fixpoints and query answers may be infinite (see Example 1.1), but there is a terminating, sound, and complete recognition algorithm [8, 16]. This algorithm is very different from bottom-up evaluation. It is not even based on resolution or hyperresolution. On the other hand, bottom-up evaluation does not terminate for $Datalog_{nS}$ because it requires the computation of the entire least fixpoint which may be infinite. So the question is: can bottom-up evaluation be made to terminate while preserving soundness and completeness in the case of $Datalog_{nS}$? We are interested in the termination of recognition and, more generally, in the termination of the evaluation of finite-answer queries.

To achieve this goal, we introduce the notion of *depth-bounded* bottom-up evaluation. Intuitively, we are going to impose a finite bound on the depth of atoms that are derived during bottom-up evaluation. The bound has to be large enough for the query evaluation to remain complete. Such a finite bound does not exist for arbitrary logic programs, but it does exist for $Datalog_{nS}$ due to the syntactic restrictions imposed on the type and the occurrences of function symbols.

Assume m is a non-negative integer. Define the m-bounded version of a $Datalog_{nS}$ rule $r \in Z$ as the rule $r_{|m}$ which is identical to r except that its body contains also the subgoal $depth(t) \leq m$, called a *depth constraint*, where t is the term that appears in the functional argument of the head atom of r. For example, the rules from Example 1.1 will have the following m-bounded version:

$$meets(T, X) \leftarrow meets\_first(T, X), depth(T) \leq m.$$
$$meets(T + 1, Y) \leftarrow next(X, Y), meets(T, X), depth(T + 1) \leq m.$$

In a similar way, define $Z_{|m}$ – the m-bounded version of a set of $Datalog_{nS}$ rules Z. We have to slightly extend the notion of rule application – only those ground substitutions will be considered that make the added depth constraint true.

Now *depth-bounded* evaluation is defined in the following way. In the algorithm **B** (Figure 1) the application of Z is replaced by the application of $Z_{|m}$ - the m-bounded version of Z. The resulting algorithm always terminates because only finitely many facts are generated. When used to evaluate a finite-answer query, it is *sound* for any value of m. To achieve *completeness*, however, m has to be sufficiently large. Therefore, in order to find an appropriate value for m, it is necessary to look closer at various parameters of a $Datalog_{nS}$ program.

3.2. Deriving the depth bound

At first glance it is not obvious that m-bounded bottom-up evaluation for any finite m can be a complete evaluation procedure for finite-answer $Datalog_{nS}$ queries. For arbitrary logic programs it isn't, however least fixpoints of $Datalog_{nS}$ programs have a "repetitive" structure and this fact can be used to bound the depth of terms appearing in bottom-up evaluation.

To formally characterize the "repetitiveness", we introduce several new notions. In the following definitions, assume that M is a set of functional and data facts, and t_0 is a ground functional term. M does not have to be finite. However, it has to be finitely generated: the number of different constant and function symbols appearing in the elements of M should be finite. The least fixpoint M_S (defined earlier) has this property.

Define the *snapshot* $M(t_0)$ of M as:

$$M(t_0) = \{p(t_0, \bar{a}) : p(t_0, \bar{a}) \in M)\}.$$

$M(t_0)$ may be thought of as the result of the selection $\sigma_{\$1=t_0}(M)$. Additionally, $M(t_0)$ is always finite because data arguments can assume only finitely many values.

Define the *state* $M[t_0]$ of M as:

$$M[t_0] = \{p(\bar{a}) : p(t_0, \bar{a}) \in M)\}.$$

$M[t_0]$ may be thought of as a result of "projecting out" the functional arguments in the predicates in $M(t_0)$. Therefore, it is a finite, function-free database. Every

M has only finitely many different states. Moreover, if for every t $M[t]$ is known, the entire set M can be reconstructed.

Define the *data part* M^d of M as the set of all the data facts in M. This set is also finite.

Consider a $Datalog_{nS}$ program consisting of a set of rules Z and a database D, and a query Q. The following parameters of the program and the query are identified:

k - the maximal arity of a predicate in Z and D.
n - the number of facts in the database D.
d - the number of different data constants in D.
c - the maximum depth of a (ground) functional term in D ($c = 0$ if there is no such term).
h - the depth of the functional term in Q ($h = 0$ if there is no such term).

Denote by $size_f$ the number of facts that a state $M_{Z \cup D}[t]$ may contain (only facts built with symbols appearing in $Z \cup D$ are considered). If there are s_i functional predicates of arity i ($1 \leq i \leq k$) in $Z \cup D$, then

$$size_f = \Sigma_{i=1}^{k} s_i d^{i-1}.$$

Similarly, if there are t_i data predicates of arity i ($0 \leq i \leq k$) in Z and D, then the data part ${M_{Z \cup D}}^d$ may contain at most $size_d$ facts where

$$size_d = \Sigma_{i=0}^{k} t_i d^i.$$

For a fixed set of rules Z, both $size_f$ and $size_d$ are polynomial functions of the number of data constants in the database D (and therefore also polynomial functions of the number of facts in the database).

Define the *range* of $Z \cup D$ as the number of different states in $M_{Z \cup D}$. Because the number of facts in any state is bounded by $size_f$:

$$range(Z \cup D) \leq 2^{size_f}$$

where $size_f$ is a polynomial function of the number of facts in the database. However, there are special cases [9] where $range(Z \cup D)$ can be bounded from above by a polynomial in the size of the database D.

As mentioned in the previous section, evaluating a ground query Q in (the least model of) a logic program $Z \cup D$ is equivalent to testing the unsatisfiability of the formula $Z \cup D \cup \{\neg Q\}$. Evaluating a query Q with free variables consists of constructing all ground substitutions θ such that $Z \cup D \cup \{\neg Q\theta\}$ is unsatisfiable.

Consider now only closed queries. Define the *basis* of a $Datalog_{nS}$ formula $\Phi = Z \cup D \cup \{\neg Q\}$ in the following way:

$$basis(\Phi) = max(c, h) + range(Z \cup D).$$

Therefore:

$$basis(\Phi) \leq max(c, h) + 2^{size_f}.$$

We show now how to reduce the satisfiability of a $Datalog_{nS}$ formula Φ to the existence of a finite model for a formula Φ_m derived from Φ. Define a *functionally-grounded instance* $\Phi\{t\}$ of a $Datalog_{nS}$ formula Φ as the result of substituting the ground functional term t into the single functional variable T in Φ. The instance $\Phi\{t\}$ does not have to be ground – data variables may be still unbound. It is clear that M is a model of Φ iff M is a model of every functionally-grounded instance $\Phi\{t\}$.

Example 3.1. Assume the rule r is as follows:

$$p(f(T), X) \leftarrow p(T, X).$$

The functionally-grounded instance $r\{0\}$:

$$p(f(0), X) \leftarrow p(0, X).$$

For a $Datalog_{nS}$ formula Φ, define Φ_m $(m = 1, 2, \ldots, \omega)$ as follows:

$$\Phi_m = \bigcup_{t:depth(t)<m} \Phi\{t\}.$$

Only ground functional terms t built from 0 and function symbols appearing in Φ are considered. Notice that Φ_m is a finite formula if $m < \omega$.

Lemma 3.1. Φ is satisfiable iff Φ_m is satisfiable for $m = basis(\Phi)$.

PROOF. This seems to be a folk result, referenced in several papers [13, 16]. We haven't been able to locate its proof, so we reprove it in Appendix A. □

One more result is needed – a fundamental property of least models of $Datalog_{nS}$ programs.

Lemma 3.2. [8, 12] Let $Z \cup D$ be a $Datalog_{nS}$ program and c the maximum depth of a functional term in D. For all ground functional terms t_1 and t_2 of depth greater than or equal to c, and every function symbol f:

$$(M_{Z\cup D}[t_1] = M_{Z\cup D}[t_2]) \Rightarrow (M_{Z\cup D}[f(t_1)] = M_{Z\cup D}[f(t_2)])$$

Corollary 3.1. Let $p(t_0, \bar{a})$ be a fact in $M_{Z\cup D}$. If

$$depth(t_0) \geq c + range(Z \cup D)$$

then there are infinitely many different terms $t_1, t_2, \ldots$ such that for all i, $p(t_i, \bar{a})$ is in $M_{Z\cup D}$.

PROOF. Because there are $range(Z \cup D)$ different states in $M_{Z\cup D}$, then there must be two different terms w_1 and w_2 of depth greater than or equal to c and such that (a) w_1 is a proper subterm of w_2 which is a subterm of t_0, and (b) $M_{Z\cup D}[w_1] = M_{Z\cup D}[w_2]$. Therefore, using Lemma 3.2 repeatedly, we can conclude that there is a proper subterm w_0 of t_0 of depth no less than c such that $M_{Z\cup D}[w_0] = M_{Z\cup D}[t_0]$. Assume $t_0 = f_1 f_2 \cdots f_k(w_0)$. Using Lemma 3.2 repeatedly, we obtain that $M_{Z\cup D}[t_1] = M_{Z\cup D}[t_0]$ for $t_1 = f_1 f_2 \cdots f_k(t_0)$. Therefore, the fact $p(t_1, \bar{a})$ is in $M_{Z \cup D}$. In this way infinitely many facts satisfying the thesis are obtained. □

We prove now the main results of this section. Theorem 3.1 shows soundness and completeness of depth-bounded bottom-up evaluation of yes-no queries. Theorem 3.2 shows that with a somewhat larger, but still finite bound, finite-answer queries can also be completely evaluated. In the following we use the notation introduced earlier in this section.

Theorem 3.1. For $m = basis(\Phi)$, m-bounded bottom-up evaluation of a yes-no (closed) query Q is sound and complete.

Proof. Let L be the (finite) set of facts computed by m-bounded bottom-up evaluation. If $L \models Q$, then also $M_{Z \cup D} \models Q$ (because $L \subseteq M_{Z \cup D}$ and Q is monotonic), and for every Herbrand model M of $Z \cup D$, $M \models Q$. Thus, $\Phi = Z \cup D \cup \{\neg Q\}$ is unsatisfiable (soundness).

If Φ is unsatisfiable, so is

$$\Phi_m = \bigcup_{t:depth(t)<m} \Phi\{t\}$$

by Lemma 3.1. The formula Φ'_m, the fully grounded version of Φ_m, is also unsatisfiable. Moreover, Φ'_m is finite, as we obtain it from Φ_m using only the data constants of Φ and there is only a finite number of those. By completeness of ground hyperresolution [36], there is a ground hyperresolution refutation of Φ'_m. By construction, this refutation consists of data facts and also of functional facts with functional terms of depth at most m. By induction on the length of the refutation, it is easy to see that this refutation is also obtained by m-bounded bottom-up evaluation. In this way completeness is obtained. □

Theorem 3.2. For $m = max(c, h) + 2 \cdot range(Z \cup D)$, m-bounded bottom-up evaluation of a finite-answer query Q is sound and complete.

Proof. If only data variables in Q are free, then the above analysis still applies. So we have to examine only the case when the functional variable is free. Soundness is immediate because if the evaluation of Q in L gives a substitution θ, then also $M_{Z \cup D} \models Q\theta$.

Assume now that θ is an answer substitution w.r.t. $M_{Z \cup D}$, i.e., $M_{Z \cup D} \models Q\theta$. Denote by t_0 the ground functional term to which the functional variable in Q is mapped by θ. If $depth(t_0) \geq c + range(Z \cup D)$, then there are infinitely many different answer substitutions w.r.t. $M_{Z \cup D}$ that differ from θ only in the terms assigned to the functional variable (Corollary 3.1). Therefore, the answer to Q w.r.t. M_S is infinite – a contradiction. So we can conclude that $depth(t_0) < c + range(Z \cup D)$. Consequently:

$$\begin{aligned} basis(Z \cup D \cup \{\neg Q\theta\}) &= max(c, depth(t_0)) + range(Z \cup D) \\ &< c + 2 \cdot range(Z \cup D). \end{aligned}$$

Thus there is a ground refutation of $Z \cup D \cup \{\neg Q\theta\}$ containing only facts with terms of depth at most $c + 2 \cdot range(Z \cup D)$. This refutation is obtained by m-bounded bottom-up evaluation for $m = c + 2 \cdot range(Z \cup D)$ (see the proof of the previous theorem). □

Note that in view of Theorem 3.2, in the context of $Datalog_{nS}$ the answer to a query is finite iff it can be computed by a terminating algorithm. Therefore, in this context the notions of *safety* (query answer finiteness) and *capturability* [20] (existence of a terminating algorithm to compute the answer, also called *effective computability* [4]) coincide.

In Figure 2, we finally present the depth-bounded evaluation algorithm **BF**. Note that it is m'-bounded evaluation for $m' = max(c, h) + 2^{size_f+1}$ rather than m-bounded evaluation for $m = max(c, h) + 2 \cdot range(Z \cup D)$. The reason for this modification is as follows: m can not be calculated directly from the text of the program $Z \cup D$, while m' can. Sometimes a tighter bound on $range(Z \cup D)$ than 2^{size_f+1} can be provided and then this bound may be used in the algorithm **BF** [9].

$L' := D$
repeat
    $L := L'$
    $L' := Z_{|m'}(L)$
until $L = L'$
evaluate Q in L

Figure 2. Depth-bounded bottom-up evaluation: algorithm **BF**

The termination of depth-bounded evaluation can also be obtained in a different way. Instead of introducing depth constraints, the termination condition in **B** can be changed to the following: *no atoms with terms of depth less than or equal to* $basis(\Phi)$ *are in* $L' - L$.

3.3. Computational complexity

In estimating the execution time of depth-bounded evaluation for $Datalog_{nS}$, we will assume that the set of rules Z is fixed and only the database D and the query Q vary. This is a common assumption in database theory, called *data complexity* (Chandra and Harel [7] and Vardi [53]).

Theorem 3.3. The execution time of m*-bounded evaluation where* $m = basis(\Phi)$ *can be bounded from above by a function double exponential in the number of facts in the database. In the case of* $Datalog_{1S}$ *this bound can be reduced to single exponential.*

PROOF. In every loop iteration, at least *one* new fact is obtained. The number of generated facts is thus bounded from above by:

$$nr = (m + 1) \cdot \#(m) \cdot size_f + size_d$$

where $\#(i)$ is the number of ground terms of depth i built using the constant and function symbols appearing in Φ.

Now $m = basis(\Phi)$ is exponential in the size of the database D. In the presence of just one unary function symbol $(+1)$ there is exactly one ground term of any given depth. However, if there is more than one unary function symbol, the number of terms of any given depth i is exponential in i. Thus the time bound on the execution of **BF** is single exponential for $Datalog_{1S}$ and double exponential for $Datalog_{nS}$. $\Box$

Theorem 3.4. There is a set of $Datalog_{1S}$ rules Z, a ground $Datalog_{1S}$ query Q, and an infinite family of $Datalog_{1S}$ databases $\mathcal{D}$ with the following property: for every n, there is a $D_n \in \mathcal{D}$ of size polynomial in n such that to test the unsatisfiability of $\Phi = Z \cup D_n \cup \{\neg\exists Q\}$ the interpreter **BF** *will derive a set of facts whose cardinality is exponential in n.*

PROOF. The basic idea is to encode adding 1 to an n-bit number. The following predicates are used ($1 \leq i \leq n$):

$one(t, a_i) \equiv$ "at time t, the i-th bit is 1".
$zero(t, a_i) \equiv$ "at time t, the i-th bit is 0".
$chng(t, a_i) \equiv$ "at time t, the i-th bit is changed".
$unchng(t, a_i) \equiv$ "at time t, the i-th bit is not changed".

The database D_n contains the facts $zero(0, a_i)$ for every $1 \leq i \leq n$ ($a_i \neq a_j$ for $i \neq j$), $next(a_i, a_j)$ for every $1 \leq i, j \leq n$ and $j = i + 1$, and additionally the fact $chng(0, a_1)$. The set of rules Z is as follows:

$chng(T + 1, a_1) \leftarrow chng(T, a_1).$
$chng(T + 1, Y) \leftarrow next(X, Y), one(T, X), chng(T + 1, X).$

$unchng(T + 1, Y) \leftarrow next(X, Y), zero(T, X), one(T + 1, X).$
$unchng(T + 1, Y) \leftarrow next(X, Y), unchng(T, X).$

$one(T + 1, X) \leftarrow chng(T + 1, X), zero(T, X).$
$one(T + 1, X) \leftarrow unchng(T + 1, X), one(T, X).$

$zero(T + 1, X) \leftarrow chng(T + 1, X), one(T, X).$
$zero(T + 1, X) \leftarrow unchng(T + 1, X), zero(T, X).$

$sometimes\_one(X) \leftarrow one(T, X).$

The first pair of rules describes the pattern of bits that change between t and $t+1$. The second pair of rules describes the pattern of bits that do not change between t and $t + 1$. The third pair of rules describes the bits that become 1 at time $t + 1$, and the fourth pair of rules describes the bits that become 0 at time $t + 1$. The first rule contains a constant, but such rule can be replaced by a constant-free rule and an additional database fact.

The yes-no query $Q = sometimes\_one(a_n)$ requires the derivation of a set of facts whose cardinality is exponential in n. $\Box$

Theorem 3.5. There is a set of $Datalog_{nS}$ rules Z, a ground $Datalog_{nS}$ query Q, and an infinite family of $Datalog_{nS}$ databases $\mathcal{D}$ with the following property: for every n, there is a $D_n \in \mathcal{D}$ of size polynomial in n such that to test the unsatisfiability of $\Phi = Z \cup D_n \cup \{\neg\exists Q\}$ the interpreter **BF** *will derive a set of facts whose cardinality is double exponential in n.*

Proof. The database D is identical to the one in the proof of the Theorem 3.4. For the set of rules Z, we take two copies of the rules from the same proof: one with the function symbol f replacing $+1$, the other with g replacing $+1$. Additionally, we add two rules:

$r(T) \leftarrow one(T, a_n).$
$r(T) \leftarrow r(f(T)), r(g(T)).$

The query is now simply $Q = r(0)$. For every ground functional term t where $depth(t) = 2^{n-1}$, the fact $r(t)$ will be generated by the interpreter **BF**. All of them are necessary to derive $r(0)$, therefore the execution of **BF** will now require the derivation of a double exponential number of facts. □

4. COMPARISON OF EVALUATION ALGORITHMS

Here, we survey a number of approaches to evaluating $Datalog_{nS}$ queries. We will compare depth-bounded bottom-up evaluation with a number of evaluation algorithms for $Datalog_{nS}$ that we presented in our earlier work. We will analyze the sources of differing computational properties of those algorithms. Moreover, we will briefly comment on an algorithm of Joyner [18] and, more extensively, on top-down evaluation (SLD-resolution). In particular, we are going to show a close connection between depth-bounded evaluation and bounded depth-first search. Finally, we will discuss the impact of rule rewriting methods like Magic Sets on the termination of bottom-up evaluation.

4.1. Evaluation algorithms for $Datalog_{nS}$

Lemma 3.1 immediately suggests a method to check the satisfiability of a $Datalog_{nS}$ formula $\Phi = Z \cup D \cup \{\neg Q\}$. The formula Φ_m does not contain non-ground functional terms and therefore may be considered a function-free formula. Thus, to establish satisfiability of Φ_m it is sufficient to consider finite Herbrand interpretations. In fact, we have to consider only the interpretations M defined by the states $M[t]$ such that $depth(t) \leq m$.

So the first approach will be to simply enumerate such interpretations and for each check whether it is a model of Φ_m. We will not present the details of this approach but just mention that it leads to recognition algorithms of PSPACE data complexity for $Datalog_{1S}$ (algorithm **T**) and EXPTIME data complexity for full $Datalog_{nS}$ (algorithm **F**). The algorithms are presented in our earlier work [8]. They are based on the work by Plaisted [30] and Fürer [16]. The above bounds can not be improved because unsatisfiability of $Datalog_{1S}$ (resp. $Datalog_{nS}$) is PSPACE-data-complete [8, 13] (resp. EXPTIME-data-complete [8, 25]).

In our previous work [11, 12] we presented another approach – a method to effectively construct a finite representation of the least model $M_{Z \cup D}$. Lemma 3.2 provides a theoretical foundation for this method. The finite representation can be subsequently used to evaluate queries. We have shown that this method of evaluation (called hereafter algorithm **QF**) can also achieve the best possible complexity bounds mentioned above. The algorithm **QF** uses some recognition algorithm as a subroutine, thus we have two variants of it: **QF/F** (**QF** using **F**) and **QF/BF** (**QF** using **BF**).

We are interested in finding out:

1. whether the algorithm under consideration can be used for recognition, for generation (computing the entire answer), or for both,
2. whether the algorithm terminates and achieves the lower complexity bound (i.e., PSPACE for $Datalog_{1S}$, EXPTIME for $Datalog_{nS}$),
3. whether the algorithm scales down, i.e., runs in polynomial time for restricted classes of rules,
4. whether the algorithm works for supersets of $Datalog_{nS}$.

For a recognition algorithm to be used for generation, the algorithm has to accept existentially quantified variables in queries and to evaluate such queries by computing the bindings for the variables.

The recognition algorithm **F** can not be used for generation because it answers an existential query without computing bindings for the existentially quantified variables in the query. It terminates and achieves the lower bound but does not scale down. We know of no non-trivial restriction on rules (a trivial one is, for example, the exclusion of function symbols in recursive rules) that would guarantee that **F** runs in polynomial time. The algorithm **F** works also for an extension of $Datalog_{nS}$ in which clauses are not required to be Horn. The recognition algorithm **T** for $Datalog_{1S}$ shares all the above properties of **F**.

Depth-bounded evaluation (algorithm **BF**) can be used for recognition and also for generation if the query answer is finite (see Section 5 for methods to check this condition). The algorithm **BF** always terminates but does not achieve the lower bound. On the example from Theorem 3.4, both the algorithm **T** and the algorithm **BF** execute an exponential number of steps. However, in contrast with **T** it can not be guaranteed that **BF** runs in polynomial space. The algorithm **F** works in *single* exponential time, thus **BF** which requires in general *double* exponential time is clearly deficient. However, if a polynomial bound on $basis(\Phi)$ is guaranteed, **BF** will run in polynomial time on $Datalog_{1S}$ programs. This is not the case for either **F** or **T** because of the enumeration approach inherent in both. Depth-bounded evaluation requires that the clauses be Horn, so in this sense **BF** is less general than **F** (or **T** if only $Datalog_{1S}$ programs are considered).

The generation algorithm **QF/F** terminates and achieves the lower bound for both closed and open queries. It is complete even for queries with infinite answers because it constructs a finite representation of the answer. However, it does not scale down because it uses **F** as the recognition algorithm.

The generation algorithm **QF/BF** terminates, does not achieve the lower bound but scales down and like **BF** runs in polynomial time for $Datalog_{1S}$ if a polynomial bound on $basis(\Phi)$ is guaranteed [9].

Both **QF/F** and **QF/BF** require that the rules be Horn, because they construct a finite representation of the *least* Herbrand model of a program.

All the above algorithms presuppose that the input program is a $Datalog_{nS}$ program. Some of them can handle simple extensions of $Datalog_{nS}$, e.g., non-Horn clauses. They are not, however, complete refutation procedures for arbitrary logic programs.

Recently, many rule rewriting methods were introduced [4, 28, 48]) in order to more efficiently evaluate a query using some additional information from the query itself (variable bindings or existential quantification). Those methods are applicable and yield significant gains in performance if the rules are executed bottom-up. So the algorithm **BF** should be able to benefit from them. It is unclear whether the algorithms **F**, **T**, and **QF** will be able to make use of those methods.

Finally, let's look at the possible sources in the differences in complexity of the above algorithms.

In the case of $Datalog_{1S}$ **BF** constructs sometimes an exponentially-sized ground hyperresolution refutation (Theorem 3.4). It is clear that any ground resolution refutation of the formula constructed in the proof of Theorem 3.4 has to contain clauses with ground terms of any given depth between 0 and 2^{n-1}. Thus, the size of such a refutation is exponential. For non-ground resolution the issue is more subtle. There are exponentially-sized ground refutations whose non-ground counterparts are polynomial.

Example 4.1. Take the following $Datalog_{nS}$ program

$p0(0).$
$p(T) \leftarrow p0(T).$
$p(T+1) \leftarrow p(T).$

together with the query:

$p(2^n).$

To evaluate this query bottom-up, $2^n + 2$ steps are needed. This is also the minimal size of any *ground* resolution refutation. However, if the rule can be resolved with itself, a much shorter, $(n+3)$-step refutation suffices. A sequence of n clauses is obtained:

$p(T+1) \leftarrow p(T).$
$p(T+2) \leftarrow p(T).$
$p(T+4) \leftarrow p(T).$
$\dots$
$p(T+2^n) \leftarrow p(T).$

and the last clause yields unsatisfiability in three steps. If terms are encoded in binary, the size of the above refutation is polynomial.

However, our lower bound example (Theorem 3.4) contains *non-linear* clauses where such "acceleration" is not possible because the size of the obtained *non-ground* resolvents grows exponentially. We will not pursue this issue further here, but we conjecture that our lower bound example for $Datalog_{1S}$ (Theorem 3.4) will produce exponentially-sized refutations for *any* variant of resolution.

In the case of full $Datalog_{nS}$ the interpreter **F** works in *single* exponential time, while **BF** requires in general double exponential time. We can trace this deficiency to the inability of ground resolution or hyperresolution to deal with many *isomorphic* subrefutations. It is an open problem whether a resolution-based method can match the upper bound achieved by **F**.

Joyner [18] proposed a resolution procedure R_2 that uses atom ordering and clause condensation (which is essentially tableau minimization [48]). This procedure has several attractive properties. First, it is a complete recognition procedure for arbitrary logic programs. (In fact, it is also a refutation procedure for arbitrary first-order formulas). Second, it is terminating for $Datalog_{nS}$ queries with finite answers and does not require for termination that a bound (calculated from the text of the program) be supplied. However, if R_2 is combined with SLD-resolution, the resulting procedure is incomplete. For completeness, R_2 seems to require the ability to resolve rules among themselves. It is not clear how to efficiently implement such a facility.

Recently, a number of researchers pursued further Joyner's ideas. Tammet [44], Zamov [56], Leitzsch [24] and Fermüller [14] considered various classes of formulas in clausal form that can be decided by resolution or hyperresolution. In particular, the book [15] gives a hyperresolution decision procedure that can be applied to $Datalog_{nS}$. This procedure requires, similarly to depth-bounded bottom-up evaluation, that a program-dependent bound be supplied.

4.2. Top-down evaluation

Top-down evaluation (SLD-resolution) in its practical incarnations, e.g., the Prolog evaluation procedure, introduces additional problems with termination. Guaranteeing termination through memoing [43, 55], breadth-first search, or ground loop checking, is possible for *Datalog* programs or, more generally, for logic programs with finite least fixpoints. However, in the presence of infinite least fixpoints those methods are insufficient to obtain termination of all finite-answer $Datalog_{nS}$ queries. A technique is necessary which would make use of the finite bound on the size of ground refutations. *Bounded depth-first search*, proposed by Stickel [42], is such a technique. We present here its version due to O'Keefe [29] by means of an example.

Example 4.2. We extend here Example 1.1. The predicate names are shortened.

$$
\begin{array}{l}
n(a,b).\\
n(b,a).\\
m0(0,a).\\
m(T,X) \leftarrow m0(T,X).\\
m(T+1,Y) \leftarrow n(X,Y), m(T,X).\\
r(T,X) \leftarrow m(T,X).\\
r(T,X) \leftarrow r(T+1,X).
\end{array}
$$

and the query $r(0,b)$. O'Keefe's method yields the following rules:

$$
\begin{array}{l}
m(T,X,s(N),M) \leftarrow m0(T,X,N,M).\\
m(T+1,Y,s(N),M) \leftarrow n(X,Y,N,K), m(T,X,K,M).\\
r(T,X,s(N),M) \leftarrow m(T,X,N,M).\\
r(T,X,s(N),M) \leftarrow r(T+1,X,N,M).
\end{array}
$$

The added arguments count the number of resolution steps in a refutation. The database is also modified and contains the following facts:

$$
\begin{array}{l}
n(a,b,s(T),T).\\
n(b,a,s(T),T).\\
m0(0,a,s(T),T).
\end{array}
$$

The facts contain variables, but this is not a problem because we are considering top-down evaluation here. Finally, the query is as follows:

$r(0, b, k, W)$.

where k is equal to the number of steps of SLD-resolution that are necessary to evaluate the query. This number is less than or equal to g^{nr} where g is the maximum number of literals in the body of a rule and nr is equal to the number of distinct facts that can be generated by depth-bounded bottom-up evaluation of the program (Theorem 3.3). (In the context of *Datalog* it was shown by Naughton and Ramakrishnan [28] that top-down evaluation can in fact be exponentially less efficient than bottom-up evaluation.)

Top-down bounded depth-first search terminates because it counts the number of resolutions. If only a bound on term depth in top-down refutations is imposed, loop detecting techniques are still required to obtain termination. In the method above, we had to come up with the correct value for the bound k. Can we dispense with this knowledge? The *iterative deepening* algorithm [42, 29] invokes bounded depth-first evaluation with consecutive integer bounds. At some point, the value of the bound reaches g^{nr} and by that time, any possible refutation is found. However, it is not clear how to terminate iterative deepening if the correct value of nr is not given in advance. Using a saturation test, i.e., terminating when no new answer substitutions are obtained in the current iteration, leads to incompleteness even for $Datalog_{nS}$.

4.3. Rule rewriting methods

We argue here that the termination of bottom-up evaluation of (finite-answer) $Datalog_{nS}$ queries can not in general be obtained by using existing rule rewriting methods whose goal is to push selections or projections into rules [4, 48, 28]. Clearly, we will not be able to consider all such methods. We will study only Magic Sets [3, 33], one of the best known methods of pushing selections into rules. We will try to show that methods of this kind seem incapable of guaranteeing termination and that they actually introduce additional sources of non-termination.

Example 4.3. Consider the program from Example 4.2 together with the query $r(0, X)$. The result of applying Magic Sets is as follows:

$n(a, b)$.
$n(b, a)$.
$m0(0, a)$.
$m(T, X) \leftarrow m0(T, X)$.
$m(T+1, Y) \leftarrow magic\_m(T+1), n(X, Y), m(T, X)$.
$r(T, X) \leftarrow magic\_r(T), r(T+1, X)$.
$r(T, X) \leftarrow magic\_r(T), m(T, X)$.

$magic\_m(T) \leftarrow magic\_r(T)$.
$magic\_m(T) \leftarrow magic\_m(T+1)$.
$magic\_r(0)$.
$magic\_r(T+1) \leftarrow magic\_r(T)$.

The query $r(0, X)$ has a finite answer, but its bottom-up evaluation using the transformed version of the program does not terminate because the relation $magic\_r$ is infinite.

To achieve completeness in the above example the relation $magic\_r$ has to contain a finite, but database-dependent, number of facts. It does not seem possible to achieve this effect using Magic Sets or any other rule rewriting method. In addition to magic predicates and rules we need a "magic" bound that will limit the magic facts generated bottom-up to a finite number.

It should be mentioned that the inherent difficulty in evaluating programs like the one in Example 4.3 has been noticed in the literature. Kifer and Lozinskii [20] call it "a diverging cycle feeding into a converging one". They present a technique, called *signatures*, that is helpful in the opposite situation, i.e., a converging cycle feeding into a diverging one. Seki [39] presents a variant of the same technique in the context of Magic Sets. The technique actually originates in the work of Sato and Tamaki [38, 43]. Finally, Ramakrishnan [33] also recognizes the additional sources of non-termination introduced by the Magic Sets transformation. Clearly, for some $Datalog_{nS}$ programs and queries, Magic Sets will provide termination. In Example 4.3, if the rules for r were not present and the query was $m(1, X)$, then we would have the "magic" fact $magic\_m(1)$, $magic\_m$ would be a finite relation, and bottom-up evaluation of the program would terminate.

The program in Example 4.3 can be transformed by hand to a program whose bottom-up evaluation terminates. However, if we deal with a more complicated program, for example one resulting from adding the following clauses to the program used in the proof of Theorem 3.4:

$q(T) \leftarrow one(T, X).$
$q(T) \leftarrow q(T + 1).$

it is no longer clear how to obtain from it an equivalent terminating program that evaluates the query $q(0)$ or even whether such a program exists at all. We conjecture that there are $Datalog_{nS}$ programs and finite-answer queries that can not be rewritten in a database-independent way to yield a program whose bottom-up evaluation terminates. If this claim is true, it will highlight an inherent limitation of rule rewriting methods when applied to the task of improving the termination behavior of logic programs. Nevertheless, proving or disproving the claim seems to be a formidable task and we leave it for further research.

It should be noted that rule rewriting methods can be applied to improve the execution time of $Datalog_{nS}$ programs. We only claim that combining these methods with standard bottom-up evaluation will not by itself guarantee termination of finite-answer $Datalog_{nS}$ queries. To achieve termination, bottom-up evaluation has to be replaced by its depth-bounded version.

5. SAFETY

In this section we deal with the issue of infinite query answers. If the answer to a query is infinite, depth-bounded bottom-up evaluation is not complete. We discuss two approaches to this problem.

In the first approach, we are interested in *relative safety* [19], i.e., whether the answer to a query is finite for a given set of rules and a given database. Relative safety can be determined by depth-bounded evaluation with a sufficiently large bound. So the user will get in addition to a finite set of answer substitutions one of the following replies: "those are the only answer substitutions" or "there are infinitely many more answer substitutions".

In the second approach, a set of rules and a query are tested for *universal safety* [19], i.e., whether the answer to the query is finite for every database. If a query is universally safe, the user will know that depth-bounded evaluation of this query with an appropriate bound is complete. A query has to be retested for universal safety only if the rules change. Updates to the database which are much more common than rule changes do not influence universal safety by definition. We show that testing universal safety for $Datalog_{nS}$ is decidable, similarly to *Datalog*.

We consider only *simple* queries (queries consisting of a single atom whose arguments are all different variables). This is not a restriction because if a query Q is not simple, we can always add a new rule whose body will be Q and whose head will be an atom with a new IDB predicate symbol and the arguments all distinct and equal to the free variables of Q.

5.1. Relative safety

Theorem 5.1. Let $Z \cup D$ be a $Datalog_{nS}$ program and c the maximum depth of a functional term in D. Let $m = c + 3 \cdot range(Z \cup D)$. The answer to a simple $Datalog_{nS}$ query Q is infinite iff m-bounded bottom-up evaluation of Q returns a substitution where the functional variable is mapped to a ground term t_0 such that

$$c + range(Z \cup D) \leq depth(t_0) \leq c + 2 \cdot range(Z \cup D).$$

PROOF. The right-to-left direction follows immediately from Corollary 3.1. Assume now that m-bounded evaluation does not return a substitution satisfying the above condition. If such an answer substitution existed, m-bounded evaluation would return it (Theorem 3.2). Therefore, we can conclude that it does not exist. Can there be an answer substitution with the functional variable mapped to a ground term t_1 of depth greater than $c + 2 \cdot range(Z \cup D)$? Such a substitution might be "missed" by m-bounded bottom-up evaluation. It is an easy consequence of Lemma 3.2 that there is a term t_2 such that

$$c + range(Z \cup D) \leq depth(t_2) \leq c + 2 \cdot range(Z \cup D).$$

and $M_{Z \cup D}[t_1]{=}M_{Z \cup D}[t_2]$. Therefore, if there is an answer substitution θ that maps the functional variable to t_2, there is also one identical to θ except that it maps the functional variable to t_1 – contradiction. Consequently, no substitutions are "missed" by m-bounded evaluation and the answer to Q is finite. □

There is another approach to testing relative safety of $Datalog_{nS}$ queries. The finiteness of the query answer can be detected from the finite representation of the least model $M_{Z \cup D}$ [12]. That algorithm works in exponential time for $Datalog_{nS}$ and polynomial space for $Datalog_{1S}$, thus it seems superior to the above method whose complexity is double exponential time (Theorem 3.5) and single exponential time, respectively (Theorem 3.4). However, testing relative safety by bottom-up depth-bounded evaluation has its advantages. For example, bottom-up evaluation scales down better, as pointed out in the previous section. Also, rule rewriting may be used to improve the execution time.

5.2. Universal safety

Theorem 5.2. Let Z be a set of $Datalog_{nS}$ rules and $Q = r(x_1, \ldots, x_n)$ a simple $Datalog_{nS}$ query. It is decidable whether $ans(M_{Z \cup D}, Q)$ is finite for every database D.

Proof. The proof consists of two major steps: a reduction to *Monadic* $Datalog_{nS}$ (a subset of $Datalog_{nS}$ where predicates have arity at most 1), and a proof that universal safety is decidable for *Monadic* $Datalog_{nS}$.

If $ans(M_{Z \cup D}, Q)$ is infinite for some database D, $ans(M_{Z \cup D_0}, Q)$ is also infinite for a database D_0 obtained from D by replacing every data constant in D by an arbitrary single data constant. (The reverse implication is trivially true.) This is easily seen because bottom-up computation of $M_{Z \cup D_0}$ mimicks that of $M_{Z \cup D}$. If the latter requires infinitely many iterations to compute the answer to the query Q, so does the former. It is crucial here that rules are range-restricted and do not contain the inequality symbol.

Define the *skeleton* $S(A)$ of a $Datalog_{nS}$ term (atom) A to be the result of removing all data arguments of A. This definition is extended in an obvious way to clauses and sets of clauses.

We claim that $ans(M_{Z \cup D_0}, Q)$ is infinite iff $ans(M_{S(Z \cup D_0)}, S(Q))$ is infinite. It is easily seen that bottom-up computation of $M_{S(Z \cup D_0)}$ simulates that of $M_{Z \cup D_0}$ step-by-step.

The skeleton of $Z \cup D_0$ is a *Monadic* $Datalog_{nS}$ program. We show now that detecting universal safety is decidable for *Monadic* $Datalog_{nS}$ by formulating universal safety as a sentence of the *monadic* second-order theory of the infinite n-ary tree (SnS), shown decidable by Rabin [32]. We are going to show the construction on the following example:

$$p(T) \leftarrow p(f(T)), w(T).$$
$$p(g(T)) \leftarrow r(T).$$

The query Q is $p(T)$. Consider the formula $\phi(P, R, W)$ where P, R, and W are second-order variables (they correspond to predicates p, r, and w, respectively):

$$\phi(P, R, W) \equiv \forall x\ (f(x) \in P \wedge x \in W \Rightarrow x \in P) \wedge (x \in R \Rightarrow g(x) \in P)$$

This formula expresses the property that P contains the result of the query Q on the database consisting of R and W. We are interested in the smallest P satisfying $\phi(P, R, W)$. This can be expressed as:

$$smallest(A, R, W) \equiv \phi(A, R, W) \wedge (\forall X\ \phi(X, R, W) \Rightarrow A \subseteq X)$$

Now universal safety can be expressed as:

$$\neg\exists P, R, W\ \mathit{finite}(R) \wedge \mathit{finite}(W) \wedge \neg \mathit{finite}(P) \wedge \mathit{smallest}(P, R, W).$$

Thomas [46] shows how to express $\mathit{finite}(X)$ (meaning X is finite) using the lexicographic ordering of strings and how this ordering can in turn be expressed in SnS.

The above construction is possible because of the correspondence between ground functional terms and strings over a finite alphabet. It may appear at first that non-unary function symbols allowed in the definition of $Datalog_{nS}$ may create additional problems. Such symbols can not be eliminated here because the result of the elimination is database-dependent and universal safety is a database-independent property. However, the skeleton construction can be easily generalized to $Datalog_{nS}$ programs with non-unary function symbols. The resulting *Monadic* $Datalog_{nS}$ programs will have a database-independent set of unary function symbols. □

Note. One can provide an elementary (fixed-height exponential) upper bound on the time complexity of the universal safety problem for $Datalog_{nS}$ as follows. First, notice that for any $Datalog_{nS}$ program the SnS formula that expresses universal safety of a simple query involving a predicate of this program contains only a bounded number of negations (after the elimination of $\forall$, $\wedge$, and first order variables). Then one can use the recent result of Klarlund [22] that the complementation of Rabin tree automata can be done in single exponential time and the well known correspondence between these automata and SnS [46] to conclude that the complexity of universal safety for $Datalog_{nS}$ is elementary. The exact complexity of universal safety for $Datalog_{nS}$ remains an open problem.

5.3. Related work on safety

Shmueli [40] and Kifer [19] show that relative and universal safety for *Datalog* are decidable, while for arbitrary logic programs neither is recursively enumerable. As far as we know, $Datalog_{nS}$ is the only class of logic programs with function symbols for which both problems are decidable.

The notion of safety was also studied by, among others, Ramakrishnan, Bancilhon and Silberschatz [34], Kifer, Ramakrishnan and Silberschatz [21], and Sagiv and Vardi [37]. However, a different model was assumed: instead of function symbols – infinite EDB relations with finiteness and monotonicity constraints. This model is usually called *Extended Datalog*. (The extension of an EDB predicate r satisfies a finiteness constraint $r : X \rightsquigarrow Y$ where X and Y are sets of argument numbers if for every tuple w in it the set of Y-values in the tuples that agree with w on X is finite. The extension of an EDB predicate r satisfies a monotonicity constraint $r : N \prec M$ if for every tuple w the value in the argument N is less than that in the argument M.)

When comparing these two models (infinite relations vs. function symbols), two issues have to be addressed. First, the *error* of the approximation, namely which queries that are safe in the function symbol model fail to be safe in the infinite relation model [2]. Second, the computational complexity of safety testing in both

[2] Any query that is safe in the infinite relation model is also safe in the function symbol model because function symbols provide special instances of infinite relations with appropriate constraints.

models. However, to reflect the special role of the functional argument in $Datalog_{nS}$ predicates, we should compare *Monadic* $Datalog_{nS}$ with *Monadic Extended Datalog* (monadic IDB predicates). Universal safety for *Monadic Extended Datalog* with finiteness constraints can be checked in polynomial time [37]. In this case, however, the approximation provided by constraints is too crude and many safe queries will be missed.

Example 5.1. Consider the following $Datalog_{nS}$ program:

$$p(T) \leftarrow p0(T).$$
$$p(T) \leftarrow q(s(T)).$$
$$q(T) \leftarrow p(T).$$

and the query $Q = q(T)$. This program corresponds to the following *Extended Datalog* program:

$$p(X) \leftarrow p0(X).$$
$$p(X) \leftarrow q(Y), s(X, Y).$$
$$q(X) \leftarrow p(X).$$

The binary EDB relation s captures the successor function symbol. The finiteness information is expressed by the following finiteness constraints:

$$p0 : \emptyset \leadsto 1$$
$$s : 1 \leadsto 2$$
$$s : 2 \leadsto 1.$$

However, universal safety in this model requires that the answers to a query are finite not only for all extensions of $p0$ that satisfy the first constraint (i.e., are finite) but also for all extensions of s that satisfy the remaining two constraints. One of the latter codes exactly the successor but there are others as well. For example, consider the following infinite extension of s:

$$s(1, 0).$$
$$s(2, 1).$$
$$s(3, 2).$$
$$\ldots$$

and the extension of $p0$ consisting of $p0(0)$. The answer to Q for such EDB is infinite. Thus, Q is classified as unsafe in the infinite relation model, although it is clearly safe.

Monotonicity constraints do not really help, as witnessed by the next example.

Example 5.2. Consider the following $Datalog_{nS}$ program:

$$p(T) \leftarrow p0(T).$$
$$p(f(T)) \leftarrow p(T).$$
$$q(T) \leftarrow q0(T).$$
$$q(g(T)) \leftarrow q(T).$$
$$r(T) \leftarrow p(T), q(T).$$

When this program is modelled as an *Extended Datalog* program in which the function symbols f and g are replaced by binary relations f and g, respectively, with the monotonicity constraints $f : 1 \prec 2$ and $g : 1 \prec 2$ and the appropriate finiteness constraints, the property that the intersection of the extensions of p and q is always finite is lost. Consequently, the query $r(X)$ will be classified in this model as unsafe, although it is clearly safe.

Moreover, the complexity (or even decidability) of testing universal safety for *Monadic Extended Datalog* in the presence of monotonicity constraints (in addition to finiteness constraints) has not been, to our knowledge, established yet. The decidability of universal safety for *Extended Datalog* is unknown [37]. Therefore, various sufficient conditions for universal safety have been developed [21, 23].

6. CONCLUSIONS AND FURTHER WORK

We have shown a bottom-up query evaluation algorithm for $Datalog_{nS}$, a class of logic programs with limited function symbols. The algorithm is always sound and terminating; it is also complete for all finite-answer queries. We have also shown that both relative and universal safety for $Datalog_{nS}$ are decidable. Using several criteria, we have compared different query evaluation methods for $Datalog_{nS}$. This analysis should be complemented by a comparison of practical implementations of those methods. For example, we have only classified the algorithms according to the membership in specific complexity classes, which is a very rough measure. In general, depth-bounded evaluation looks promising because it can make use of various implementation techniques developed for logic programs, for example incremental evaluation and rule rewriting methods. Following our earlier results for $Datalog_{1S}$[9], tractable classes of $Datalog_{nS}$ should also be identified.

It is interesting to see whether other classes of logic programs can be evaluated using a depth-bounded approach. Our depth bound can be easily obtained from the text of the program and is validated by referring to non-trivial properties of $Datalog_{nS}$. There might be other ways for obtaining depth bounds, ways that rely on the analysis of a single program rather than on properties of the entire class of programs. In this context, it may be fruitful to explore the connections to the work of Ullman, Van Gelder, and Sohn [41, 49, 52], Plümer [31], and Brodsky and Sagiv [6] on deriving constraints among argument sizes in logic programs. The simple notion of a depth bound used in this paper, namely an inequality of the form $depth(t) < m$ may be generalized to *bound constraints* – constraints built from arithmetic expressions involving various parameters of terms like depth or number of subterms. Similar ideas were pursued in the context of top-down evaluation by Vasak and Potter [54]. It is also challenging to establish the power of depth-bound evaluation vis-a-vis rule rewriting methods like Magic Sets. Can they replicate its termination behavior? A positive or a negative answer to this question would be very interesting. Finally, it would also be interesting to compare the exact and approximate (using the infinite relation model) safety analyses for various syntactically restricted classes of logic programs. The properties of interest include error of the approximation and computational complexity of the analyses.

We believe that the research reported in this paper is a step towards finding a general evaluation method for logic programs that will also terminate for interesting, syntactically defined classes of programs.

7. ACKNOWLEDGMENTS

This paper is a revised version of a part of my Ph.D. dissertation written at Rutgers University under the supervision of Professor Tomasz Imieliński. I gratefully acknowledge his participation at every stage of my doctoral research. Early versions of some of the results of this paper were presented in our joint paper at the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1988, Austin, Texas. I am also very grateful to Professor Jack Minker for supporting the continuation of this research at the University of Maryland and to Dr. Damian Niwiński for the discussions leading to the proof of Theorem 5.2. Detailed comments of the anonymous referees are also gratefully acknowledged.

REFERENCES

1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8(3), September 1989.
2. H. Andreka and I. Nemeti. The Generalised Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica*, 4(1):3–10, 1978.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1986.
4. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In M. Stonebraker, editor, *Readings in Database Systems*. Morgan Kaufmann, 1988.
5. M. Baudinet. Temporal Logic Programming is Complete and Expressive. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1989.
6. A. Brodsky and Y. Sagiv. Inference of Inequality Constraints in Logic Programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1991.
7. A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.
8. J. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols.* PhD thesis, Rutgers University, New Brunswick, New Jersey, January 1990. Also Laboratory for Computer Science Research Technical Report LCSR-TR-142.
9. J. Chomicki. Polynomial-Time Computable Queries in Temporal Deductive Databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, Tennessee, April 1990.
10. J. Chomicki and T. Imieliński. Temporal Deductive Databases and Infinite Objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Texas, March 1988.
11. J. Chomicki and T. Imieliński. Relational Specifications of Infinite Query Answers. In *ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May 1989.
12. J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems*, 18(2):181–223, June 1993.
13. L. Denenberg and H.R. Lewis. Logical Syntax and Computational Complexity. In *Computation and Proof Theory*, pages 101–115. Springer-Verlag, Lecture Notes in Mathematics 1104, 1984.
14. Ch. Fermüller. A Resolution Variant Deciding some Classes of Clause Sets. In E. Börger, H. Kleine Büning, M.M. Richter, and W. Schönfeld, editors, *Workshop on Computer Science Logic*, pages 128–144. Springer-Verlag, LNCS 533, 1990.

15. Ch. Fermüller, A. Leitzsch, T. Tammet, and N.K. Zamov. *Resolution Methods for the Decision Problem.* Springer-Verlag, LNCS 679, 1993.
16. M. Fürer. Alternation and the Ackermann case of the decision problem. *L'Enseignement Mathematique*, 1981.
17. C. Green. Application of Theorem Proving to Problem Solving. In *International Joint Conference on Artificial Intelligence*, 1969.
18. W.H. Joyner Jr. Resolution Strategies as Decision Procedures. *Journal of the ACM*, 23(3):398–417, July 1976.
19. M. Kifer. On Safety, Domain Independence and Capturability of Database Queries. In *Data and Knowledge Base Conference*, Jerusalem, Israel, 1988.
20. M. Kifer and E.L. Lozinskii. SYGRAF: Implementing Logic Programs in a Database Style. *IEEE Transactions on Software Engineering*, 14(7):922–935, 1988.
21. M. Kifer, R. Ramakrishnan, and A. Silberschatz. An Axiomatic Approach to Deciding Query Safety in Deductive Databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 52–60, Austin, Texas, March 1988.
22. N. Klarlund. Progress Measures, Immediate Determinacy, and a Subset Construction for Tree Automata. In *IEEE Symposium on Logic in Computer Science*, 1992.
23. R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A Framework for Testing Safety and Effective Computability of Extended Datalog. In *ACM SIGMOD International Conference on Management of Data*, 1988.
24. A. Leitsch. Deciding Horn Clauses by Hyperresolution. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Workshop on Computer Science Logic*, pages 225–241. Springer-Verlag, LNCS 440, 1989.
25. H. Lewis. Complexity Results for Classes of Quantificational Formulas. *Journal of Computer and System Sciences*, 21:317–353, 1980.
26. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
27. S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.
28. J.F. Naughton and R. Ramakrishnan. Bottom-Up Evaluation of Logic Programs. In J-L. Lassez and G. Plotkin, editors, *Computational Logic*, pages 640–700. MIT Press, 1991.
29. R.A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.
30. D.A. Plaisted. Complete Problems in the First-Order Predicate Calculus. *Journal of Computer and System Sciences*, 29:8–35, 1984.
31. L. Plümer. Termination Proofs for Logic Programs based on Predicate Inequalities. In D.H.D. Warren and P. Szeredi, editors, *International Conference on Logic Programming*, pages 634–648. MIT Press, 1990.
32. M.O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions AMS*, 141:1–35, 1969.
33. R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Journal of Logic Programming*, 11(3&4):189–216, 1991.
34. R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of Recursive Horn Clauses with Infinite Relations. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 328–339, 1987.
35. R. Ramakrishnan, S. Srivastava, and S. Sudarshan. CORAL - Control, Relations and Logic. In L-Y. Yuan, editor, *International Conference on Very Large Data Bases*, pages 238–250, Vancouver, Canada, August 1992. Morgan Kaufmann.
36. J.A. Robinson. Automatic Deduction with Hyperresolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
37. Y. Sagiv and M.Y. Vardi. Safety of Datalog Queries over Infinite Databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*,

pages 160–171, Philadelphia, Pennsylvania, March 1989.
38. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.
39. H. Seki. On the Power of Alexander Templates. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–159, Philadelphia, Pennsylvania, March 1989.
40. O. Shmueli. Equivalence of Datalog Queries is Undecidable. *Journal of Logic Programming*, 15(3):231–241, February 1993.
41. K. Sohn and A. Van Gelder. Termination Detection in Logic Programs Using Argument Sizes. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1991.
42. M.E. Stickel. A Prolog Technology Theorem Prover. In *IEEE Symposium on Logic Programming*, pages 212–219, 1984.
43. S. Tamaki and T. Sato. OLD Resolution with Tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986.
44. T. Tammet. The Resolution Program, Able to Decide Some Solvable Classes. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, pages 300–312. Springer-Verlag, LNCS 417, 1990.
45. S-A. Tärnlund. Horn Clause Computability. *BIT*, 17:215–226, 1977.
46. W. Thomas. Automata on Infinite Objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–164. Elsevier/MIT Press, 1990.
47. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.
48. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.
49. J.D. Ullman and A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal of the ACM*, 35(2):345–373, 1988.
50. J. Vaghani, K. Ramamohanorao, D. Kemp, Z. Somogyi, and P. Stuckey. Design Overview of the Aditi Deductive Database System. In *IEEE International Conference on Data Engineering*, pages 240–247, 1991.
51. 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, 1976.
52. A. Van Gelder. Deriving Constraints Among Argument Sizes in Logic Programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 47–60, Nashville, Tennessee, April 1990.
53. M.Y. Vardi. The Complexity of Relational Query Languages. In *ACM SIGACT Symposium on Theory of Computing*, pages 137–146, 1982.
54. T. Vasak and J. Potter. Metalogical Control for Logic Programs. *Journal of Logic Programming*, 2(3):203–220, October 1985.
55. L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.
56. N.K. Zamov. Maslov's Inverse Method and Decidable Classes. *Annals of Pure and Applied Logic*, 42:165–194, 1989.

A. Finite satisfiability lemmas

We prove here Lemma 3.1 which asserts that a $Datalog_{nS}$ formula $\Phi = Z \cup D \cup \{\neg Q\}$ is satisfiable iff Φ_m is satisfiable for $m = basis(\Phi)$.

The non-trivial direction is right-to-left. The formula Φ_m for $m = basis(\Phi)$ does not contain non-ground functional terms, therefore it can be looked upon as a function-free formula. Therefore, if Φ_m is satisfiable, then it has a finite model

M. The maximum depth of ground functional terms in Φ_m is m, therefore the model M is completely characterized by its states $M[t]$ for ground terms t such that $depth(t) \leq m$. Other states of M are empty. Given M, we construct a model N for Φ which has a "repetitive" structure.

We start with a few definitions. A formula Φ is in *basic form* if it satisfies the following restrictions:

- all the predicates are functional.
- 0 is the only ground functional term in the facts of the program or the query.

Assume first that Φ is in basic form. A ground functional term t_0 is *initial* if:

$$\forall t, (t \in subterms(t_0) \wedge t \neq t_0 \Rightarrow N[t] \neq N[t_0]).$$

A ground functional term is *repeating* if it is not initial.

We construct N inductively by induction on k – the depth of terms.

If $k = 0$, we take $N[0]{=}M[0]$. Now assume $N[t]$ has been constructed for all terms t of depth smaller than k. Therefore, it can be determined for every such term whether it is initial or not. Take a term $t_0 = f(t_1)$ of depth k.

If t_1 is initial, put $N[t_0]{=}M[t_0]$ (*copying* from M). If t_1 is repeating, there is an initial subterm t_2 of t_1 such that $N[t_1]{=}N[t_2]$. We put $N[t_0]{=}N[f(t_2)]$ (*reusing* N).

It can be shown by easy induction that N (as constructed above) satisfies the following properties:

- $\forall t, w$, if t is not initial and t is a subterm of w, then w is not initial.
- $\forall t, w$, if t is initial and w is a subterm of t, then w is initial.

From the above it follows that all subterms of an initial term correspond to different states. Because only $range(Z \cup D)$ different states are considered (only the states from $M_{Z \cup D}$ are considered), we conclude that if t is initial, then

$$depth(t) < range(Z \cup D) = m.$$

Therefore, we see that only facts with terms of depth at most m are copied from M to N. The remainder of N is constructed by reusing N.

Finally, we show that N is a model of Φ. Take an instance $\Phi\{t_0\}$.

If t_0 is initial, then by the construction of N:

- $\forall t \in subterms(t_0)$, $N[t] = M[t]$ (in particular, $N[0] = M[0]$ and $N[t_0] = M[t_0]$).
- $N[f(t_0)] = M[f(t_0)]$ for all function symbols f.

Using Lemma A.1 (proved below), we can conclude the truth of $\Phi\{t_0\}$ in N from the truth of $\Phi\{t_0\}$ in M.

If t_0 is repeating, we can find a subterm t_1 of t_0 such that $N[t_0] = N[t_1]$ and t_1 is initial. Consequently, $M[t_1] = N[t_0]$. Moreover, by the construction of N, we have that $N[f(t_0)] = N[f(t_1)] = M[f(t_1)]$ for any function symbol f. Using Lemma A.1, we can conclude the truth of $\Phi\{t_0\}$ in N from the truth of $\Phi\{t_1\}$ in M.

If Φ is not in basic form, we have to guarantee that for all ground terms t in Φ, $N[t] = M[t]$ and the data parts of both agree: $N^d = M^d$. Therefore if

t_0 is a subterm of a term in Φ and $N[t_0] = N[t_1]$ for a subterm t_1 of t_0, we can not simply put $N[f(t_0)] = N[f(t_1)]$ as before, because that might result in $N[t] \neq M[t]$ for a term t in Φ. So to preserve the agreement between N and M, a bigger initial part of M will have to be "copied" to N. However, for t_0 and t_1 such that $depth(t_0) \geq max(c,h)$, $depth(t_1) \geq max(c,h)$, $N[t_0] = N[t_1]$ and t_1 is a subterm of t_0, we can again put $N[f(t_0)] = N[f(t_1)]$. Therefore, still only the facts with terms of depth at most m are copied from M to N (where now $m = basis(\Phi) = max(c,h) + range(Z \cup D)$). The rest of the argument is essentially the same. Lemma A.1 should be appropriately generalized.

Lemma A.1. Assume that M and N are Herbrand interpretations for a Datalog$_{nS}$ formula Φ in basic form such that for two ground functional terms t_M and t_N, $M[t_M] = N[t_N]$ and for every function symbol f, $M[f(t_M)] = N[f(t_N)]$. Then:

$$M \models \Phi\{t_M\} \textit{ iff } N \models \Phi\{t_N\}.$$

PROOF. Take a ground substitution θ such that $\Phi\{t_M\}\theta$ and $\Phi\{t_N\}\theta$ are ground (assume that data variables in different clauses are renamed apart). Any functional atom $p(t,\bar{a})$ in $\Phi\{t_M\}\theta$ is ground. Moreover, t is t_M, or t is $f(t_M)$ for some function symbol f. Therefore in view of the assumptions, the corresponding literal in $\Phi\{t_N\}\theta$ has the same truth value in N, as $p(t,\bar{a})$ in M. The substitution θ makes an equality or an inequality true in both $\Phi\{t_M\}$ and $\Phi\{t_N\}$, or false in both.

Thus we have that for all ground θ

$$M \models \Phi\{t_M\}\theta \text{ iff } N \models \Phi\{t_N\}\theta$$

and the thesis follows. $\square$