Finite Representation of Infinite Query Answers *

Jan Chomicki† Tomasz Imieliński‡
Kansas State University Rutgers University

November 9, 1992

Keywords: query processing, logic programming, *Datalog*, decidability, safety, computational complexity, non-standard query answers, non-Herbrand models.

1989 CR Classification: F.2.2, F.4.1, H.2.3, I.2.3.

Abstract: We define here a formal notion of finite representation of infinite query answers in logic programs. We apply this notion to $Datalog_{nS}$ (*Datalog* with n successors): an extension of *Datalog* capable of representing infinite phenomena like flow of time or plan construction. Predicates in $Datalog_{nS}$ can have arbitrary unary and limited n-ary function symbols in one fixed position. This class of logic programs is known to be decidable. However, least Herbrand models of $Datalog_{nS}$ programs may be infinite and consequently queries may have infinite answers.

We present a method to finitely represent infinite least Herbrand models of $Datalog_{nS}$ programs as *relational specifications.* A relational specification consists of a finite set of facts and a finitely specified congruence relation. A relational specification has the following desirable properties. First, it is *explicit* in the sense that once it is computed, the original $Datalog_{nS}$ program (and its underlying computational engine) can be forgotten. Given a query to be evaluated, it is easy to obtain from the relational specification finitely many answer substitutions that represent infinitely many answer substitutions to the query. The method involved is a combination of a simple, unification-less, computational mechanism (graph traversal, congruence closure, or term rewriting) and standard relational query evaluation methods. Second, a relational specification is *effectively computable* and its computation is no harder, in the sense of the complexity class, than answering yes-no queries.

Our method is applicable to every range-restricted $Datalog_{nS}$ program. We also show that for some very simple non-$Datalog_{nS}$ logic programs, finite representations of query answers do not exist.

*To appear in *ACM Transactions on Database Systems.* A preliminary version of this paper was presented at the ACM SIGMOD International Conference on Management of Data, May 1989, Portland, Oregon. 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: Department of Computing and Information Sciences, 234 Nichols Hall, Kansas State University, Manhattan, KS 66506-2302. E-mail: `chomicki@cis.ksu.edu`.

‡Address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903. E-mail: `imielins@cs.rutgers.edu`.

1 Introduction

A lot of attention has been recently devoted to logic programs with *function-free* Horn rules (*Datalog*) [CH85, Ull88]. While *Datalog* constitutes an important and practical class of logic programs, there are phenomena which cannot be expressed in *Datalog*. Such processes as flow of time, state transitions, construction of plans and recursive data structures (see the next section for the examples of these) require limited use of function symbols. Storing data about infinite processes in a database and reasoning about it requires extending *Datalog* to allow limited occurrences of function symbols.

We postulate two criteria for an extension of *Datalog* to be acceptable as a database query language. First, yes-no queries should be decidable. Second, infinite query answers that necessarily arise in the presence of function symbols in rules should be finitely representable. If both of the above criteria are satisfied, a database user can interact with the system implementing the extension in a similar way as she does with a relational (or a *Datalog*) system: all queries terminate and print complete answers in finite time.

Example 1.1 The rule r_0 schedules the meetings of graduate students with their common advisor (the fact $meets(t, x)$ means that the student x meets his advisor on the day t):

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

Assume the database D contains the following facts (0 stands for the first day of school, 1 for the second etc.):

$meets(0, emma)$.
$follows(emma, kathy)$.
$follows(kathy, emma)$.

The query $meets(T, emma)$ when evaluated over the database D using the rule r_0 returns an answer consisting of infinitely many answer substitutions:

$$T = 0, T = 2, T = 4, \ldots$$

where 2 is an abbreviation for $(0 + 1) + 1$ etc.

In this paper, we address the issue of finite representation of infinite query answers. We provide a precise definition of this notion and apply it to $Datalog_{nS}$ (*Datalog* with n successors), a class of logic programs with limited function symbols postulated in [CI89, Cho90b] (it was called *functional deductive databases* there). $Datalog_{nS}$ is a superset of *Datalog* and a subset of (pure) Prolog. In [Cho90a, Cho90b], we show that $Datalog_{nS}$ is decidable using the methods of [Für81]. The type and the occurrences of function symbols in $Datalog_{nS}$ programs are restricted in the following way: function symbols can only be unary (*successors*) or limited k-ary [1], and they can only appear in one distinguished argument of

[1] As we will show, non-unary function symbols in $Datalog_{nS}$ programs can be replaced by unary function symbols.

predicates. So the example above is a valid $Datalog_{nS}$ program (+1 is the unary function symbol). The syntactic restrictions are natural for the class of applications that we intend to handle: the distinguished argument corresponds to a *state* and function symbols correspond to *operators* mapping states to states. An important special case is when only a single unary function symbol, written as +1, is allowed. We call it $Datalog_{1S}$ (*Datalog* with *one* successor). In [CI88, Cho90b, Cho90c], we used the term *temporal deductive databases* for $Datalog_{1S}$. $Datalog_{1S}$ is particularly well suited for the description of temporal phenomena. It is well known, e.g., Tarnlund [Tar77], that logic programs with unlimited function symbols are undecidable. In fact, lifting any of the syntactic restrictions imposed in the definition of $Datalog_{nS}$ leads to an undecidable class.

In this paper, we show that for $Datalog_{nS}$ finite representations of infinite query answers exist and can be effectively constructed.

It is generally accepted, e.g., see Lloyd [Llo87], that the intended meaning of a logic program is expressed by its *least Herbrand model.* In the presence of function symbols in rules, this model may be infinite. To achieve finite representation of infinite query answers, we seek a finite, non-Herbrand, structure which is equivalent to the least model in the sense that yes-no (Boolean) queries evaluate to the same truth value in both. This equivalence is guaranteed by the Homomorphism Theorem [End72] if the finite structure is the (strong) homomorphic image of the least Herbrand model. Then queries with free variables may be evaluated in the finite structure. There will be finitely many answer substitutions which can be formally characterized as *compositions* of the original answer substitutions (coming from the least model) with the homomorphism.

Example 1.2 Returning to our initial example, let's treat the least Herbrand model of the program as a first-order structure. Its domain consists of all Herbrand terms built from the constants and the function symbols in rules, i.e.,:

$$emma, kathy, 0, 1, 2, \ldots$$

We do not consider terms like $emma + 1$ because we think of them as not well-typed (this will be made more precise later). Functions are defined in the standard way: a function symbol f is interpreted as a function mapping a term t to the term $f(t)$. Finally, relations are defined as the extensions of predicates in the least model, i.e.,:

$$\begin{array}{l} follows(emma, kathy). \\ follows(kathy, emma). \\ meets(0, emma). \\ meets(1, kathy). \\ meets(2, emma). \\ meets(3, kathy). \\ \ldots \end{array}$$

Clearly, the relation corresponding to the predicate *meets* is infinite.

Now the crucial point in the construction of a finite structure equivalent to the least model is the interpretation of function symbols as *finite functions.* Here we define a function map_{+1} (corresponding to the function symbol $+1$) in the following way:

$$map_{+1}(0) = 1$$
$$map_{+1}(1) = 0$$

The domain consists now of four elements: *emma*, *kathy*, 0, and 1. The relations corresponding to predicates are finite and contain:

$follows(emma, kathy).$
$follows(kathy, emma).$
$meets(0, emma).$
$meets(1, kathy).$

It can be easily verified that there is a homomorphism H between the least model and the above finite structure. The query $meets(T, emma)$ can be evaluated in the finite structure to yield a single answer substitution τ:

$$T = 0.$$

This substitution is a composition of any of the original answer substitutions:

$$T = 0, T = 2, T = 4, \ldots$$

and the homomorphism H. Informally, we can say that the substitution τ *stands for* all the infinitely many original answer substitutions.

Therefore, we consider the task of finite representation to be the construction of a finite homomorphic image of the least Herbrand model. Once this finite structure is obtained, the original logic program can be forgotten and queries answered solely on the basis of the finite structure. This structure describes the correspondence between the finitely many "new" answer substitutions and the infinitely many "old" answer substitutions, so it should be returned as a part of the query answer. In fact, it is enough to return only the graphs of finite functions for the user to be able to reconstruct the correspondence between those two kinds of substitutions.

We will be using the criterion of finite representation in a positive and in a negative way. First, we will show that answers for queries to $Datalog_{nS}$ programs *can* be finitely represented. Second, we will show simple examples of non-$Datalog_{nS}$ programs for which query answers *can not* be finitely represented.

For every $Datalog_{nS}$ program, the least Herbrand model is "repetitive" (we observed this fact first in the context of $Datalog_{1S}$ in [CI88]), and can be homomorphically mapped to a very natural finite structure that we call the *quotient model.* Essentially, the elements of the quotient model are equivalence classes of a congruence relation $\cong$ between terms. In the above example, there are two such classes:

$$a_1 = \{0, 2, 4, \ldots\}$$
$$a_2 = \{1, 3, 5, \ldots\}$$

The elements of each class are intuitively equivalent because they induce similar truth values of facts. For example:

$$\forall x,\ meets(0, x) \equiv meets(2, x) \equiv meets(4, x) \ldots$$

The construction of the quotient model relies in a crucial way on the fact that in $Datalog_{nS}$ programs, function symbols can appear only in one distinguished argument of predicates.

Instead of introducing a new constant for every class of the above congruence, we select one *representative* term from each congruence class and make it an element of the domain of the quotient model. In the example above, the representative terms are 0 and 1. In this way, answer substitutions contain only terms from the Herbrand universe of the original $Datalog_{nS}$ program: a property important from the user's point of view.

The main algorithmic task that we address in this paper is the construction of a *specification* of the quotient model for $Datalog_{nS}$ programs. The relations interpreting predicates are represented explicitly by listing their tuples. We will call them the *primary* database. In the *graph* specification, finite functions are specified by their (finite) graphs. In the *equational* specification, the congruence $\cong$ is specified by a finite set of ground equations. For $Datalog_{nS}$, both formulations are equivalent and the resulting specifications have the same complexity. Due to the special role of the congruence relation, we use the term *relational specification* to cover both graph and equational specifications. To illustrate the difference between them, consider the problem of evaluating a ground atomic yes-no query in the quotient model.

Example 1.3 Returning to the original example, consider the atomic query $meets(4, emma)$.

In the graph specification, we apply four times the finite function map_{+1} to 0 obtaining 0 and check whether the fact $meets(0, emma)$ is in the primary database. It is there, so the answer is "yes". Similarly, the query $meets(3, emma)$ returns the answer "no" because the fact $meets(1, emma)$ is not in the primary database ($1 = map_{+1}(map_{+1}(map_{+1}(0)))$).

In the equational specification, we will have the single pair $0 \cong 2$ instead of the graph of the finite function map_{+1}. Because $\cong$ is a congruence, it can be derived that $1 \cong 3$, $2 \cong 4$ and $0 \cong 4$. Thus, as before, evaluating whether $meets(4, emma)$ can be reduced to checking whether the fact $meets(0, emma)$ is in the primary database. Similar comments apply to the other query ($meets(3, emma)$).

So the evaluation of a query in a specification of the quotient model involves the following steps: the reduction of arbitrary terms to representative terms by means of graph traversal or congruence closure, and the subsequent evaluation of the resulting relational query in the primary database. Note that both steps are considerably simpler than the computational mechanism, e.g., SLD-resolution, underlying the execution of logic programs. We can say that the relational specifications of the quotient model are more *explicit* that the original program. For example, they bring to the surface the *periodicity* present in the least model of the program. The specifications can be effectively constructed by an algorithm that uses an algorithm for yes-no query processing as a subroutine. In fact, data complexity of constructing a relational specification is in the same complexity class as data complexity of answering yes-no queries: DEXPTIME-complete for $Datalog_{nS}$, PSPACE-complete for $Datalog_{1S}$.

A standard solution to the problem of infinite answers is to detect queries that may have infinite answers and simply disallow them [RBS87]. In [CI88], we argued that such solution is unnatural. Intuitively, we'd like to be able to finitely represent infinite objects, like *"every second day"* in the example above, and return the representation as the answer to a query. In [CI88], we presented a technique to represent infinite answers in a finite way for a very restricted subset of $Datalog_{1S}$. But that approach was less general than the present one: for example, it was unable to handle the example discussed above.

The paper is organized as follows. In Section 2, we show further applications of $Datalog_{nS}$. In Section 3, we define $Datalog_{nS}$. In Section 4, we formally define the notions of *finite representation* and *quotient model.* We show simple examples of non-$Datalog_{nS}$ programs for which a finite representation of query answers does not exist. In Section 5, we show how to construct the *graph specification* of the quotient model. In Section 6, we define the *equational specification* and show that it is equivalent to the graph specification. In Section 7, we discuss computational complexity issues. In Section 8, we compare our work with other related research, including our own [CI88]. Finally, we draw conclusions and speculate about further extensions of this work in Section 9.

2 Examples

We describe here several applications of $Datalog_{nS}$.

Scheduling periodic activities. Both fixed and database dependent periodicity can be expressed. For an example of the first, consider the following rules governing the schedule of backups in a distributed system:

$$backup(T+24, X) \leftarrow backup(T, X).$$
$$backup(T, Y) \leftarrow dependent(X, Y), backup(T, X).$$

The first rule states that a backup on a machine should be taken *every 24 hours.* The second rule requires that backups should be taken simultaneously on all *dependent* machines (e.g., sharing files). The second, database dependent, type of periodicity was illustrated in the introduction.

Situation-calculus planning. In the previous examples, the distinguished first argument denoted a *time instant.* Here it plays the role of a *state* (situation). Function symbols correspond to *operators* available to a robot [Gre69]. For example, $move(t, x, y)$ stands for " robot moving from position x to 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).

$$at(0, c_0).$$
$$at(move(T, X, Y), Y) \leftarrow connected(X, Y), at(T, X).$$

The least Herbrand model of this set of rules may be infinite (in the presence of a cycle in the relation corresponding to the *connected predicate*). So may be the answer to the query

$at(T, c_1)$ which asks about all sequences of moves that can lead the robot to position c_1. Both can be finitely represented because there are only finitely many positions that the robot can assume (all positions appear in a finite database). The notion of periodicity is no longer applicable and is replaced by a more general notion of *congruence*. Intuitively, once the robot is again in the same position, it faces the same set of possible moves. On every possible infinite path, there must be a cycle. Representing one cycle traversal is enough: any subsequent traversals can be derived from this representation.

Simple list processing. Here the distinguished argument represents a list. The term $ext(t, x)$ corresponds to the list t extended by the element x (this is just a version of the more familiar *cons* with the arguments reversed to follow the syntactic convention used throughout this paper). The term 0 stands for the empty list. The relation *null* contains only one fact: $null(0)$. The following rules describe lists containing elements from a finite set specified by a unary relation p:

$$member(ext(T, X), X) \leftarrow p(X), null(T).$$
$$member(ext(T, Y), Y) \leftarrow p(Y), member(T, X).$$
$$member(ext(T, Y), X) \leftarrow p(Y), member(T, X).$$

When two lists t_1 and t_2 have identical elements ($\forall x \;\; member(t_1, x) \equiv member(t_2, x)$), their extended versions $ext(t_1, c)$ and $ext(t_2, c)$ also have identical elements. Two lists with identical sets of elements, e.g., $ext(ext(0, a), b)$ and $ext(ext(0, b), a)$, belong to the same congruence class. There are only finitely many congruence classes and the congruence can be finitely specified, yielding a finite representation for the infinite relation *member*. We elaborate on this example in Section 5.

3 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 [vEK76, Llo87, Ull88]. 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 here *functional* terms).

3.1 Syntax

An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol 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. 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 ($1 \leq i \leq m$) are called *subgoals*. A *goal* is a formula written as $\leftarrow B_1, \ldots, B_m$ (in clausal form: $\neg B_1 \vee \cdots \vee \neg B_m$). A *logic program* is a finite set of rules together with a database. All variables in rules and goals are implicitly universally quantified.

Datalog[Ull88] 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. We distinguish between *functional* predicates (those with a functional argument) and *data* predicates (those with data arguments only).

Often the definition of *Datalog* allows two special built-in predicates: equality ($=$) and inequality ($\neq$). In $Datalog_{nS}$, equalities and inequalities are allowed only between data terms.

We study separately $Datalog_{1S}$ programs which are $Datalog_{nS}$ programs with exactly one unary function symbol ($+1$). This class was first defined in [CI88].

A *query* is a first order formula built from functional and data atoms, standard logical connectives and quantifiers. We assume that both existential and universal quantifiers come in two different sorts: one quantifying over the functional domain ($\forall^f$,$\exists^f$) and the other - over the data domain ($\forall^d$,$\exists^d$). No query can contain an equality or an inequality between functional terms. A *fact query* is a ground atomic query.

3.2 Restrictions

We make a number of assumptions:

- rules are *range-restricted*, i.e., every variable is *limited* [Ull88]. 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). $Datalog_{nS}$ programs that are not range-restricted have, in the presence of infinite data domains, infinite least Herbrand models that can not be finitely represented using our approach.

- 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.

- rule bodies are non-empty.

- 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 literal. A $Datalog_{nS}$ rule r is *normal* if it is semi-normal and non-ground 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 in [Cho90b, Cho90a].

- rules do not contain ground terms. Such terms can be eliminated by introducing additional predicates.
- 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. An example of this transformation is given in Section 5. 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.

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 character and 0 as the empty string, we can treat ground functional terms that use only successors as strings.

In the sequel, we will present various technical results in two different versions. First, we will assume that certain simplifying assumptions (*basic form*) about programs hold. Second, we will study the problems in full generality.

We say that a *Datalog*$_{nS}$ program is in *basic form* if:

- all the predicates are functional.
- the program contains at most two different function symbols: s_1 and s_2.
- 0 is the only ground functional term in the facts of the program.

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

We are ready now to present the main technical contributions of this paper: a formal notion of finite representation of query answers, its application to *Datalog*$_{nS}$ which yields the notion of *quotient model*, and effective methods to compute relational specifications of the quotient model.

4 Finite representation

We study here the problem of *finite representation* of query answers in a completely general setting, applicable to any range-restricted logic program. Then we apply the developed criterion in a positive and in a negative way. First, we will show that answers for queries to *Datalog*$_{nS}$ programs *can* be finitely represented. Second, we will show simple examples of non-*Datalog*$_{nS}$ programs for which query answers *can not* be finitely represented. We define now first-order semantics and the notion of query answer.

4.1 First-order semantics

The *parameters* of a first-order language $\mathcal{L}$ consist of constant, function and predicate symbols in $\mathcal{L}$, and the quantifier $\forall$. In the context of logic programs, it is customary to fix the first-order language $\mathcal{L}$ as the finite language $\mathcal{L}_S$ of a given logic program S, e.g., [Llo87, p.16]. We assume, as usual, that the language $\mathcal{L}_S$ contains at least one constant.

A *structure* [End72] for a first-order language $\mathcal{L}_S$ is a mapping that assigns values to the parameters of $\mathcal{L}_S$. It is convenient to think of a structure as a quadruple $A = (U_A, C_A, F_A, R_A)$ where U_A is the domain which provides the assignment to $\forall$, C_A is the assignment to constant symbols, F_A is the assignment to function symbols and R_A is the assignment to predicate symbols.

A *Herbrand* structure $M = (U_M, C_M, F_M, R_M)$ for $\mathcal{L}_S$ is defined as follows:

- U_M is the set of terms built from constant and function symbols in $\mathcal{L}_S$.
- C_M is the identity mapping.
- F_M: a function symbol f of arity k is assigned a mapping defined as:

$$(t_1, \ldots, t_k) \to f(t_1, \ldots, t_k).$$

- R_M: a predicate symbol of arity k is assigned a subset of $(U_M)^k$.

The notion of *satisfaction* in a structure and *model* are defined in the standard way [End72].

Every logic program is consistent and has a Herbrand model. Herbrand models of a given logic program S can differ only as far as relations are concerned. They can be ordered by set containment. Under this ordering, there is a least element M_S, as shown by van Emden and Kowalski [vEK76]. It is generally agreed that the least Herbrand model M_S expresses the intended meaning of a logic program S.

4.2 Query answers

We will write $\models_M \Phi$ to mean "M is a model of Φ " and $\Pi \models \Phi$ to mean "Π entails Φ".

A substitution θ to the free variables of a query Q is an *answer substitution* w.r.t. a structure A if $Q\theta$ is ground and $\models_A Q\theta$. In the context of a logic program, it is usually assumed that $A = M_S$. Then the above formulation is equivalent to query evaluation under Closed World Assumption [She88], i.e., to

$$cwa(S) \models Q\theta.$$

If $Q\theta$ is a fact query, the latter condition is equivalent to the condition $S \models Q\theta$. If Q is open, then the *answer* to Q w.r.t. A is the set of all answer substitutions to Q w.r.t. A. If Q is closed, then the answer to Q w.r.t. A is "yes" if $\models_A Q$, "no" - otherwise.

4.3 General notion of finite representation

Let $A = (U_A, C_A, F_A, R_A)$ and $B = (U_B, C_B, F_B, R_B)$ be two structures for $\mathcal{L}_S$. A function H from U_A to U_B is a *homomorphism* if:

1. for each constant symbol c:

   $$H(c_A) = c_B$$

   where c_A (c_B) is the constant assigned to c in A (resp. B).

2. for each function symbol f of arity k and each k-tuple $(a_1, \ldots, a_k)$ of elements of U_A:

   $$H(f_A(a_1, \ldots, a_k)) = f_B(H(a_1), \ldots, H(a_k))$$

   where f_A (f_B) is the function assigned to f in F_A (resp. F_B).

3. for each predicate symbol p of arity k and each k-tuple $(a_1, \ldots, a_k)$ of elements of U_A:

   $$(a_1, \ldots, a_k) \in p_A \text{ iff } (H(a_1), \ldots, H(a_k)) \in p_B$$

   where p_A (p_B) is the relation assigned to p in R_A (resp. R_B).

This notion of homomorphism is sometimes called *strong* homomorphism because of "iff" in the last condition.

We define *finite representation* of a logic program S to be a *homomorphism* from the least Herbrand model M_S of S *onto* a finite structure N_S.

The existence of such a homomorphism guarantees that the finite structure is equivalent to the least Herbrand model. This is made precise by the following proposition.

Proposition 4.1 *[End72, p.91][Mon76, p.475] Let H be a homomorphism from a structure A onto a structure B. For every first-order formula Φ without equality and every substitution θ that maps the free variables of Φ to the elements of U_A:*

$$\models_A \Phi\theta \text{ iff } \models_B \Phi\tau$$

where $\tau = H \circ \theta$.

Therefore, if we take $A = M_S$ and $B = N_S$ a finite structure, then the presence of a homomorphism from M_S onto N_S guarantees that M_S and N_S can not be distinguished by equality-free queries. Thus the evaluation of queries on M_S (as defined earlier) can be reduced to their evaluation on N_S. Instead of infinitely many answer substitutions w.r.t. M_S, we obtain only finitely many answer substitutions w.r.t. N_S (the domain of N_S is finite). The correspondence between those two types of substitutions is precisely captured by the homomorphism H, thus a specification of H should form a part of a query answer.

It is not difficult to see that if H is not one-to-one, then there are (first-order) queries with equality that evaluate to different truth values on M_S and on N_S. "Equality" means here *explicit* equality of the form $s = t$, not *implicit* equality - mere multiple occurrences of the same variable. Later in this section, we will deal with the issue of explicit vs. implicit equality in the context of $Datalog_{nS}$.

Next, we turn to the task of defining the homomorphism H for $Datalog_{nS}$ programs. We will call the image of M_S under H the *quotient model* because it will turn out to be a model of S. In the next section, we will explore several ways of finitely specifying the quotient model and present effective methods to compute these specifications.

4.4 Least models of $Datalog_{nS}$ programs

We define here a finite structure which is a homomorphic image of the least model M_S of a $Datalog_{nS}$ program S. This is possible because the least model of a $Datalog_{nS}$ program has a regular, repetitive structure. We capture the regularity through the notions of *state equivalence* and *state congruence*.

But first we have to fully clarify the semantics of $Datalog_{nS}$ programs. We have to find a semantic counterpart for the syntactic restrictions on the occurrences of function symbols. To do that, we look at M_S (the least Herbrand model of a $Datalog_{nS}$ program S) as a two-sorted structure. It has not one but two domains: the functional domain $U^f_{M_S}$ and the data domain $U^d_{M_S}$. The functional domain maps the quantifier $\forall^f$ to the set of all ground functional terms built using constant and function symbols in S. The data domain maps the quantifier $\forall^d$ to the set of all data constants in S. Moreover, the tuples assigned by R_{M_S} to predicates should be *well-typed*:

- if a tuple $(t_1, a_1, \ldots, a_k)$ is in a relation assigned to a functional predicate symbol, then $t_1 \in U^f_{M_S}$ and $a_1, \ldots, a_k \in U^d_{M_S}$.
- if a tuple $(a_1, \ldots, a_k)$ is in a relation assigned to a data predicate symbol, then $a_1, \ldots, a_k \in U^d_{M_S}$.

Well-typedness is guaranteed if the rules in S are range-restricted (as defined in the previous section) and the facts are ground.

Range-restrictedness of rules and groundness of facts give us another important property: domain independence. So far, M_S was defined relative to the fixed language $\mathcal{L}_S$ which contained only constant, function and predicate symbols in S. Therefore, queries to S could only use the same symbols. This is inconvenient because we don't want to restrict the symbols used in queries. At the same time however, we would like M_S not to change if we extend $\mathcal{L}_S$ without changing the program S. This property is satisfied if the rules are range-restricted and facts ground. It guarantees that M_S can be constructed in advance without the knowledge of future queries and subsequently, those queries can be answered on the basis of M_S.

We will denote by L_S the set of facts corresponding to tuples assigned by R_{M_S} to predicate symbols. The correspondence is as follows:

$$p(a_1, \ldots, a_k) \in L_S \text{ iff } (a_1, \ldots, a_k) \in p_{M_S}$$

where p_{M_S} is the relation assigned to p in M_S. In a more traditional logic programming terminology, L_S is just the least fixpoint of S.

4.5 State equivalence

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 contain only finitely many different constant and function symbols. In particular, L_S (defined above) has this property.

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

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

$M(t_0)$ 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 *data part* M^d of M as the set of all the data facts in M.

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

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

$M[t_0]$ 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 we know $M[t]$, we can reconstruct the entire set M.

Recall that L_S is the set of facts corresponding to tuples assigned by M_S to predicate symbols. Define the *state-equivalence* relation $t_1 \sim_{L_S} t_2$ between ground functional terms t_1 and t_2:

$$t_1 \sim_{L_S} t_2 \text{ iff } L_S[t_1] = L_S[t_2].$$

When S is clear from the context, we will write $\sim$ instead of $\sim_{L_S}$.

The state-equivalence relation is indeed an equivalence relation: reflexive, symmetric and transitive. The number of equivalence classes of $\sim_{L_S}$ is called the *equivalence scope* of L_S and denoted by $scope_\sim(L_S)$. Every equivalence class of $\sim_{L_S}$ corresponds to a different state of L_S and vice versa. Thus, $scope_\sim(L_S)$ is finite. In Section 7, we provide an upper bound on its value.

Intuitively, we would like to collapse equal states of the set of facts L_S and obtain a finite structure that is a homomorphic image of M_S. But this is only possible if function symbols in S can be interpreted as functions mapping equivalence classes to equivalence classes, i.e., when the equivalence relation is a *congruence*. We show now that for $Datalog_{nS}$ programs in basic form, the state equivalence $\sim_{L_S}$ is indeed a congruence. Later, we study the complications resulting from allowing $Datalog_{nS}$ programs not in basic form.

4.6 $Datalog_{nS}$ programs in basic form

Theorem 4.1 *Assume S is a $Datalog_{nS}$ program in basic form. The state equivalence relation $\sim_{L_S}$ is a congruence, i.e., for all function symbols f*

$$t_1 \sim_{L_S} t_2 \textit{ implies } f(t_1) \sim_{L_S} f(t_2).$$

Proof: See Appendix A. □

Clusters are the congruence classes of $\sim_{L_S}$. The number of clusters is equal to $scope_{\sim}(L_S)$. We are ready now to define the *quotient model* $N_S = (U^f_{N_S}, U^d_{N_S}, C_{N_S}, F_{N_S}, R_{N_S})$ where:

- $U^f_{N_S}$ is a finite set of distinct *representatives*, one for each cluster.
- $U^d_{N_S} = U^d_{M_S}$ is the set of data constants in S.
- $C_{N_S} = C_{M_S}$ is the identity (assigning constant symbols to themselves). In particular, 0 is assigned to 0.
- F_{N_S} assigns to a function f a finite function map_f from $U^f_{N_S}$ to $U^f_{N_S}$. The value $map_f(x)$ is defined as follows: if t_0 is the representative of the cluster of a functional term t, then $map_f(t_0)$ is the representative of the cluster of $f(t)$. The value of $map_f(x)$ is well defined because of Theorem 4.1.
- R_{N_S} assigns to a functional predicate symbol p a relation consisting of such tuples $(t_0, a_1, \ldots, a_k)$ that there is a tuple $(t_1, a_1, \ldots, a_k)$ in the relation assigned by R_{M_S} to p and t_0 is the representative of the cluster of t_1. On data predicate symbols, R_{N_S} agrees with R_{M_S}.

It is easy to see that N_S is finite and that it is also the image of M_S under a homomorphism H. The homomorphism H is an identity on data constants and maps a functional term t to the representative corresponding to the cluster of t.

In order to apply Proposition 4.1, we have to reformulate it in the context of the two-sorted structures M_S and N_S. In particular, we should consider now "first-order formulas without equality between functional terms". The proof in [End72] still goes through.

Corollary 4.1 *For every first-order formula Φ without equality between functional terms and every substitution θ that maps the free variables of Φ to the elements of U_{M_S}:*

$$\models_{M_S} \Phi\theta \textit{ iff } \models_{N_S} \Phi\tau$$

where $\tau = H \circ \theta$ is a substitution mapping the free variables of Φ to the elements of U_{N_S}.

By the definition of $Datalog_{nS}$, equalities between functional terms can not appear in rules or queries. Thus, N_S is a (non-Herbrand) model of the program S and no yes-no query can distinguish between M_S and N_S.

What are representatives of clusters? We can choose arbitrary names for them. However, from the user's point of view it is important to relate the representatives to the terms from the domain of M_S. After all, those are the terms constructed by the program S. Thus, we pick as a representative one functional term from each cluster and call it a *representative term.* Notice that the above homomorphism is an identity on representative terms and data constants. Therefore, answer substitutions w.r.t. N_S form a subset of answer substitutions w.r.t. M_S. Representative terms should be as small as possible. For every cluster, we choose as the *representative term* the smallest of all congruent terms in the precedence ordering $\prec$. We define now this ordering.

Assume that distinct function symbols in the program S are numbered from 1 to k, i.e., they are $s_1, \ldots, s_k$. Now the precedence ordering $\prec$ is constructed in the following way. First, it is consistent with the ordering of terms by depth i.e.,:

$$depth(t_1) < depth(t_2) \text{ implies } t_1 \prec t_2.$$

Second, the terms of the same depth are ordered using the conditions:

$$t_1 \prec t_2 \text{ implies } s_i(t_1) \prec s_j(t_2)$$
$$i < j \text{ implies } s_i(t_1) \prec s_j(t_1)$$

for all ground functional terms t_1 and t_2 of the same depth, and $i, j = 1, \ldots, k$.

It is easy to see that the above conditions define a unique total ordering on (an enumeration of) ground functional terms built with function symbols from S. Moreover

$$t_1 \prec t_2 \text{ implies} f(t_1) \prec f(t_2)$$

for all function symbols f.

Example 4.1 In the presence of two function symbols s_1 and s_2, the precedence ordering is as follows:

$$0 \prec s_1(0) \prec s_2(0) \prec s_1(s_1(0)) \prec s_2(s_1(0)) \prec s_1(s_2(0)) \prec s_2(s_2(0)) \ldots$$

Clearly, for different numberings of function symbols, different precedence orderings and consequently, different representative terms and different quotient models may be obtained. However, all such quotient models are homomorphic images of M_S. Any of them will do for the purpose of finite representation. Therefore, in the next section we discuss the specification of a particular quotient model corresponding to a given precedence ordering.

Proposition 4.2 *For a Datalog$_{nS}$ program S, there can be no finite structure which is a homomorphic image of M_S and has fewer elements in its functional domain than N_S.*

Proof: Assume otherwise. Thus there is a homomorphism H such that for two distinct ground functional terms t_1 and t_2, $L_S[t_1] \neq L_S[t_2]$ and $H(t_1) = H(t_2)$. Then for some predicate symbol p and vector of ground data terms $\bar{a}$, $p(t_1, \bar{a}) \in L_S$ and $p(t_2, \bar{a}) \notin L_S$, or vice versa. Assume the former. Thus $(H(t_1), H(\bar{a})) \in p_{N_S}$ and $(H(t_2), H(\bar{a})) \notin p_{N_S}$, which is a contradiction in view of the fact that $H(t_1) = H(t_2)$. $\Box$

So in the above sense N_S is minimal. There can be, though, a finite structure which is a homomorphic image of M_S and has fewer elements in the *data* domain.

4.7 $Datalog_{nS}$ programs in general form

It is easy to see that if data predicates are present in the program S or the number of function symbols is greater than two, state-equivalence $\sim_{L_S}$ is still a congruence.

However, this does not have to be the case if there are facts in S that contain ground functional terms not equal to 0.

Example 4.2 The program S contains only two facts:

$p(0)$.
$p(1)$.

Clearly, $0 \sim 1$ but not $1 \sim 2$. The states corresponding to 0 and 1 are both equal to $\{p\}$. However, the state corresponding to $1(= 0+1)$ is equal to $\{p\}$, while the state corresponding to $2(= 1+1)$ is empty. Therefore, $\sim_{L_S}$ is not a congruence.

However, we can define now the *state congruence* relation $\cong$ on the basis of the state equivalence $\sim$.

Theorem 4.2 *Let L_S be defined as before. The relation $\cong_{L_S}$ defined as:*

$$t \cong_{L_S} t$$

if t is a subterm of a term in a fact in S, and

$$t_1 \cong_{L_S} t_2$$

if $t_1 \sim_{L_S} t_2$ and neither of t_1 and t_2 is a proper subterm of a term appearing in a fact in S, is a congruence, i.e., it is an equivalence relation and for all function symbols f:

$$t_1 \cong_{L_S} t_2 \text{ implies } f(t_1) \cong_{L_S} f(t_2).$$

Proof: The same as the proof of Theorem 4.1, except that a slight generalization of Lemma A.1 should be used. □

We will omit the subscript in $\cong_{L_S}$ when the program S is clear from the context. We will call the congruence classes of $\cong_{L_S}$ *clusters*, as before. The clusters defined by the first condition ("t is a subterm of a term in a fact in S") will be called *initial*, the remaining ones – *repetitive*. The number of congruence classes (clusters) of $\cong_{L_S}$ will be called the congruence scope $scope_{\cong}(L_S)$ of L_S. Because the congruence $\cong_{L_S}$ is a refinement of the equivalence $\sim_{L_S}$, we have that:

$$scope_{\sim}(L_S) \leq scope_{\cong}(L_S).$$

The number of clusters for $Datalog_{nS}$ programs in general form is still finite. A finite quotient model N_S can be constructed exactly in the same way as before. However, this model is no longer minimal in the sense mentioned earlier: there can be a finite structure which is a homomorphic image of M_S and has fewer elements in the functional domain.

4.8 Problems with explicit equality

As before, let H be the homomorphism between the least Herbrand model M_S and a quotient model N_S. In the presence of explicit equality between functional terms in queries, an answer substitution τ w.r.t. N_S can be the homomorphic image of a substitution θ' which is not an answer substitution w.r.t. M_S.

Example 4.3 Consider the following $Datalog_{nS}$ program S:

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

The domain of N_S consists of the single element 0. Take the query $p(T_1) \wedge p(T_2) \wedge T_1 = T_2$. The substitution θ':

$$T_1 = 0, T_2 = 1$$

is clearly not an answer substitution w.r.t. M_S. But $\tau = H \circ \theta'$ is given as:

$$T_1 = 0, T_2 = 0.$$

Thus τ is an answer substitution w.r.t. N_S.

In such a situation, if τ is returned to the user, she might infer incorrectly that θ' is an answer substitution w.r.t. M_S. Therefore, only answers to queries without an explicit equality (or inequality) between functional terms are preserved by the homomorphism H. Explicit equality between data terms does not present any difficulties because on such terms H is an identity and the above phenomenon can not be reproduced.

At first glance, all this seems paradoxical. Explicit equality can be replaced by implicit equality, at least in the example above, giving a query which is "equivalent" to the original one.

Example 4.4 Now the query is $p(T)$ and its answer is preserved by H: for every substitution θ to the free variables of this query, we have that θ is an answer substitution w.r.t. M_S iff $H \circ \theta$ is an answer substitution w.r.t. N_S.

We explain the above in the following way. Replacing explicit equality by an implicit one eliminates variables. The substitutions that are not answer substitutions w.r.t. M_S but are homomorphically mapped to answer substitutions w.r.t. N_S can no longer arise when the variables that appear in them are eliminated.

4.9 Non-existence of finite representation

We show here examples of very simple logic programs leading to query answers which can not be finitely represented. The programs clearly do not satisfy the syntactic restrictions imposed on function symbols in the definition of $Datalog_{nS}$.

Example 4.5 The first program S defines equality between functional terms of *Datalog*$_{1S}$. It has a unary function symbol in more than one component:

$eq(0,0)$.
$eq(T_1+1, T_2+1) \leftarrow eq(T_1, T_2)$.

We prove that there is no homomorphism between the least model M_S and a finite structure N_S. Assume the contrary. Then there are two ground functional terms t_1 and t_2 such that $H(t_1) = H(t_2)$ and $t_1 \neq t_2$. Now we have:

$$(t_1, t_1) \in eq_{M_S} \text{ iff } (H(t_1), H(t_1)) \in eq_{N_S} \text{ iff } (H(t_1), H(t_2)) \in eq_{N_S} \text{ iff} (t_1, t_2) \in eq_{M_S}$$

where eq_{M_S} (eq_{N_S}) is the relation assigned to the predicate eq in M_S (resp. N_S). The last statement in this sequence of equivalences is false, thus we obtain a contradiction.

Example 4.6 The second program U has a full-fledged binary function symbol:

$p(0)$.
$p(g(T,T)) \leftarrow p(T)$.

As before, assume there is a homomorphism H between M_U and a finite structure N_U. There are infinitely many terms that belong to the unary relation p_{M_U} assigned by the least model M_U to the predicate p. Therefore, there are two ground functional terms t_1 and t_2 such that:

$$H(t_1) = H(t_2) \text{ and } t_1 \neq t_2 \text{ and } t_1 \in p_{M_U}.$$

Now we have:

$$t_1 \in p_{M_U} \text{ implies } g(t_1, t_1) \in_{M_U} \text{ iff } H(g(t_1, t_1)) \in p_{N_U} \text{ iff } g(H(t_1), H(t_1)) \in p_{N_U}$$
$$\text{iff } g(H(t_1), H(t_2)) \in p_{N_U} \text{ iff } H(g(t_1, t_2)) \in p_{N_U} \text{ iff } g(t_1, t_2) \in p_{M_U}.$$

Once again, the last statement in this sequence is false, leading to a contradiction.

So we can see that relaxing the restrictions on the type and occurrences of function symbols in the definition of *Datalog*$_{nS}$ leads to non-existence of finite representation of query answers. Therefore, in a sense *Datalog*$_{nS}$ is a maximal class of logic programs enjoying finite representability.

It is particularly interesting to see that both of the above programs have recursive (decidable) least Herbrand models. Thus, a natural question to ask is whether finite representability implies decidability. We conjecture that this is indeed the case but do not undertake the task of proving the conjecture in this paper.

5 Graph specification

In the previous section, we introduced the notion of a *quotient model* N_S: a finite structure which is a homomorphic image of the least Herbrand model M_S of a *Datalog*$_{nS}$ program S. There are several ways in which a quotient model may be specified. In this section, we study the most obvious one where the elements of N_S are simply *listed*. We call it *graph specification.*

5.1 Definition and properties

We assume here that S is in basic form and contains exactly two unary function symbols s_1 and s_2. This restriction may be lifted in a straightforward way.

The *graph specification* $G(N_S)$ of N_S is a triple (T, F, B) such that:

1. T is the set of *representative terms*, one for each cluster. As mentioned before, a representative term is the smallest element of its cluster in the precedence ordering.

2. F contains graphs of two finite functions map_{s_1} and map_{s_2}. Those functions are identical to the functions assigned by F_{N_S}. It is convenient to think of F as a single graph with multiply labelled edges.

3. B is a finite *primary database*:

   $B = \bigcup_{t \in T} L_S(t) \cup {L_S}^d$

   where L_S is the set of facts corresponding to tuples assigned by M_S to predicates in S.

For an example of a graph specification, see the introduction. It is shown there that a graph specification can be used instead of the original logic program to evaluate fact queries. To check whether $\models_{M_S} q(t, \bar{a})$ where $t = f_k \cdots f_1(0)$ is a ground functional term [2] and $\bar{a}$ is a vector of data constants, find a representative term $t_0 \in T$ such that $t_0 = map_{f_k} \cdots map_{f_1}(0)$. Such a term can be easily found because F contains explicit graphs of successor mappings. Then check whether $q(t_0, \bar{a}) \in B$. This is correct because N_S is a homomorphic image of M_S and the homomorphism maps t to t_0.

Proposition 5.1 *Once the graph specification $G(N_S)$ is computed, testing whether $\models_{M_S} q(t, \bar{a})$ can be done in time linear in the depth of the term t (t and $\bar{a}$ are ground).*

Evaluating more complex queries in $G(N_S)$ is somewhat more complicated. First, according to Corollary 4.1 only queries without explicit equality between functional terms preserve their answers when we move from M_S to the finite structure N_S and its graph specification. Second, if a query contains function symbols, they should be interpreted as finite functions represented by the graph F. This is simple for ground terms (as shown above) but considerably more complex for non-ground terms. Therefore, the latter case is not discussed here. However, if a query is function-free, then it is just a relational calculus query and can be evaluated in the finite database B in the standard way.

Suppose a query Q is evaluated in B and returns an answer substitution τ:

$T_1 = t_1, \ldots, T_k = t_k, X_1 = a_1, \ldots, X_m = a_m$

[2] Each of f_i, $i = 1, \ldots, k$, is either s_1 or s_2.

where $T_1, \ldots, T_k$ are functional and $X_1, \ldots, X_m$ data variables. This substitution stands for possibly infinitely many answer substitutions θ w.r.t. M_S, i.e., answer substitutions obtained from the original program S. Each such θ is of the following form:

$$T_1 = u_1, \ldots, T_k = u_k, X_1 = a_1, \ldots, X_m = a_m$$

where u_1 corresponds to a path in F, not necessarily simple, from 0 to t_1, u_2 corresponds to a path from 0 to t_2 etc. Therefore, given τ we can enumerate all the original substitutions θ that it represents. Conversely, given a substitution θ w.r.t. M_S, the substitution τ that represents θ can be obtained by replacing all functional terms by their representatives. Note that we consider only *ground* substitutions.

We can also extract from $G(N_S)$ other useful information about M_S. For example, we can check *relative safety*, i.e., whether M_S is finite. It is easy to see that this is the case iff every cycle [3] in F is trivial, i.e., of the form $map_f(t) = t$, and for all such t, $B(t) = \emptyset$. The existence of a non-trivial cycle can be easily verified in time polynomial in the size of the graph. Therefore:

Proposition 5.2 *Once the graph specification $G(N_S)$ is computed, relative safety can be determined in time polynomial in the size of $G(N_S)$.*

5.2 Constructing the graph specification

In Figure 1, we present an algorithm to compute the graph specification $G(N_S)$ for a given $Datalog_{nS}$ program S.

We will use three sets of ground functional terms: ACTIVE, POTENTIAL and DUPLICATES. ACTIVE will contain representative terms and POTENTIAL candidates for representative terms. DUPLICATES will contain every term t in POTENTIAL for which there is a term t_0 in ACTIVE such that $t \sim t_0$. A finite database PDB will contain the primary database. There will also be two sets S1 and S2 representing graphs of functions map_{s_1} and map_{s_2} resp. Additionally, there will be a set STATES of states seen so far by the algorithm. The algorithm considers terms of growing depth and the states corresponding to them. If after considering all terms of a given depth all terms in POTENTIAL are in ACTIVE or in DUPLICATES, the algorithm terminates.

We assume that given the precedence ordering $\prec$, we have the following functions (A is a set of ground terms) that return ground terms:

- First(A, i) – the first element of A in the ordering of depth equal to i.
- Next(A, i, t) – the element of A following t in the ordering (assuming $t \in A$) of depth equal to i.

Theorem 5.1 *For a $Datalog_{nS}$ program S in basic form and a given precedence ordering, algorithm* **QF** *terminates and computes the graph specification $G(N_S)$.*

[3] The cycle may consist of edges from different successor mappings.

```
/* Initial phase. */
POTENTIAL := {0};
ACTIVE := ∅;
DUPLICATES := ∅;
PDB := ∅;
S1 := ∅;
S2 := ∅;
STATES := ∅;
i := 0;

/* Main phase. */
repeat
   t := First(POTENTIAL, i);
   done := false;
   repeat
      if t ∈ POTENTIAL then
         if L_S[t] ∈ STATES
         then DUPLICATES := DUPLICATES ∪ {t}
         else begin
            ACTIVE := ACTIVE ∪ {t};
            POTENTIAL := POTENTIAL ∪ {s_1(t), s_2(t)};
            STATES := STATES ∪ {L_S[t]};
            end;
      if ∃t_1 t_1 = Next(POTENTIAL, i, t)
         then t := Next(POTENTIAL, i, t)
         else done := true
   until done;
   i := i + 1
until POTENTIAL = ACTIVE ∪ DUPLICATES;

/* Primary database. */
forall t ∈ ACTIVE do
   PDB := PDB ∪ L_S(t);

/* Construction of the mappings. */
forall t ∈ ACTIVE do
   forall u ∈ ACTIVE do
      begin
      if L_S[u] = L_S[s_1(t)] then S1 := S1 ∪ {(t, u)};
      if L_S[u] = L_S[s_2(t)] then S2 := S2 ∪ {(t, u)}
      end
```

Figure 1: Algorithm **QF**: basic form.

Proof: Because S is in basic form, the state equivalence relation $\sim$ and the state congruence relation $\cong$ are identical.

The algorithm **QF** has the following property: after the iteration i (the first iteration has number 0) of the outer loop, every term t of depth no greater than i in POTENTIAL is either in ACTIVE or in DUPLICATES. Also, after iteration i the maximum depth of terms in POTENTIAL is at most $i + 1$.

Let's call the block of instructions that update ACTIVE, POTENTIAL and STATES the *critical section.* If the critical section is executed in the iteration i of the outer loop, a new term is added to ACTIVE, a new state is added to STATES and two new terms of depth $i + 1$ are added to POTENTIAL. Thus, the critical section is executed altogether at most m times where $m = scope_{\sim}(L_S)$ (the number of different states in L_S). If the critical section is executed in the iteration i_0 but not in the iteration $i_0 + 1$, then in the iteration $i_0 + 1$ no new terms are added to POTENTIAL and all terms of depth $i_0 + 1$ in POTENTIAL are inserted into DUPLICATES. Terms in POTENTIAL of depth less than $i_0 + 1$ are already in ACTIVE or DUPLICATES. So after iteration $i_0 + 1$:

$$\text{POTENTIAL} = \text{ACTIVE} \cup \text{DUPLICATES}$$

and the algorithm **QF** terminates. Therefore, the outer loop is executed at most $m + 1$ times.

Note also that at the beginning of every iteration i of the outer loop there is always a term of depth i in POTENTIAL. If $i = 0$, this term is 0. If $i > 0$, such a term was inserted into POTENTIAL in the previous iteration. Therefore, First(POTENTIAL, i) is well defined.

In the above analysis, we have assumed that the state $L_S[t]$ can be computed in finite time. This is indeed the case because $Datalog_{nS}$ is decidable. We show how to do it and discuss relevant complexity results in Section 7 (Lemma 7.2).

In the following, when speaking about the objects computed by **QF** we mean the values of these objects after the termination of **QF** (the final value of ACTIVE etc).

We are going to prove now that the algorithm **QF** computes the graph specification $G(N_S)$. We will make the set of representative terms T equal to ACTIVE, the graphs of finite functions map_{s_1} and map_{s_2} equal to S1 and S2, and the primary database B equal to PDB.

We show first that S1 and S2 define functions from T to T. The proofs for S1 and S2 are identical, so we will consider only S1.

First, we show that for every term $t_0 \in$ ACTIVE, there is a pair $(t_0, t_1) \in$ S1 such that $t_1 \in$ ACTIVE. If $s_1(t_0) \in$ ACTIVE, then we can take $t_1 = s_1(t_0)$. Otherwise, $s_1(t_0) \in$ DUPLICATES because $s_1(t_0) \in$ POTENTIAL and

$$\text{POTENTIAL} = \text{ACTIVE} \cup \text{DUPLICATES}.$$

Therefore, there is a term $t_1 \in$ ACTIVE such that $t_1 \sim s_1(t_0)$. Consequently, $(t_0, t_1) \in$ S1.

Second, we prove that if $(t_0, t_1) \in$ S1 and $(t_0, t_2) \in$ S1, then $t_1 = t_2$. Clearly, both t_1 and t_2 are in ACTIVE. Moreover, $t_1 \sim s_1(t_0)$ and $t_2 \sim s_1(t_0)$. Thus, $t_1 \sim t_2$ which is possible only if $t_1 = t_2$ (an element is added to ACTIVE only if it is not state-equivalent to another element of ACTIVE).

Therefore, S1 and S2 are graphs of functions mapping T to T. For simplicity, we will use now the functional notation for the elements of S1 and S2.

Clearly, for each cluster there is at most one term in ACTIVE that belongs to this cluster. We write $t_1 \preceq t_2$ to express that $t_1 \prec t_2$ or $t_1 = t_2$. We show now that for every ground functional term $t = f_k \cdots f_1(0)$ (each of f_i, $i = 1, \ldots, k$, is either s_1 or s_2), there is a term $t_0 \in$ ACTIVE such that $t_0 \sim t$ and $t_0 \preceq t$. In fact, $t_0 = map_{f_k} \cdots map_{f_1}(0)$. The proof is by induction on the depth of terms.

First, assume $depth(t) = 0$. Thus $t = 0$ and it is easy to see that $0 \in$ ACTIVE. Assume the thesis for all terms of depth at most k. Take a term t of depth $k+1$. For some function symbol f and term $t_1 = f_k \cdots f_1(0)$, $t = f(t_1)$. From the inductive assumption, there is a term $t_2 \in$ ACTIVE such that $t_2 = map_{f_k} \cdots map_{f_1}(0)$, $t_2 \sim t_1$ and $t_2 \preceq t_1$. Consider the term $t_0 = map_f(t_2)$. Clearly $t_0 \in$ ACTIVE. We will show that $t_0 \sim t$ and $t_0 \preceq t$. We have that $f(t_2) \sim t$ and $f(t_2) \preceq t$ (because $t_2 \preceq t_1$). From the fact that $t_2 \in$ ACTIVE, we can infer that $f(t_2) \in$ POTENTIAL. Now if $f(t_2) \in$ ACTIVE, $f(t_2) = t_0$ because $f(t_2) \sim t_0$ and map_f is a function. Otherwise, $f(t_2) \in$ DUPLICATES and there must be a term $t_3 \in$ ACTIVE such that $t_3 \sim f(t_2)$ and $t_3 \prec f(t_2)$. But then $t_3 \sim t_0$ which means that $t_3 = t_0$. In both cases, we obtain that $t_0 \sim t$ and $t_0 \preceq t$ which proves the inductive thesis.
□

The algorithm **QF** is not based on resolution and uses a yes-no query processing algorithm (interpreter) as a subroutine to compute states. In principle, any interpreter can be used. However, to achieve the upper complexity bound, the algorithm **QF** should use the upper bound interpreter for $Datalog_{nS}$ described in [Cho90b, Cho90a]. In Section 7, we discuss the computational complexity of **QF**.

5.3 General form

To allow more then two function symbols, the algorithm can be adapted in a straightforward way. If data predicates are allowed, the primary database PDB is constructed in a similar way except that it is now extended by the data part ${L_S}^d$ of L_S. This part can be computed similarly to a state using a yes-no query processing algorithm.

If arbitrary size terms are allowed in the facts of the program S, notice that the initial clusters consist of exactly one term and this term is explicitly given in the definition of $\cong$ (Theorem 4.2). Obviously, this term is a representative term. A ground term is *maximal* if it is present as an argument of an atom in the program S but it is not a proper subterm of any other term in S. For a non-maximal term t which is a representative of an initial cluster, we can assume:

$$(t, s_1(t)) \in \text{S1}$$

and similarly for other function symbols. Moreover, we change the initial value of POTENTIAL from 0 to the set of maximal terms. No other change is necessary.

We will not reproduce here the Revised Algorithm **QF**, but just mention that the arguments about termination and correctness remain essentially the same.

5.4 Example

We will follow now the work of the Revised Algorithm **QF** on the example of list construction from Section 2.

Example 5.1 A $Datalog_{nS}$ program S consists of the following rules:

$$member(ext(T,X),X) \leftarrow p(X), null(T).$$
$$member(ext(T,Y),Y) \leftarrow p(Y), member(T,X).$$
$$member(ext(T,Y),X) \leftarrow p(Y), member(T,X).$$

and the facts:

$$p(a).$$
$$p(b).$$
$$null(0).$$

Note that we will have now two new function symbols ext_a and ext_b instead of ext (elimination of non-unary function symbols). The transformed set of rules:

$$member(ext_a(T),a) \leftarrow p(a), null(T).$$
$$member(ext_b(T),b) \leftarrow p(b), null(T).$$
$$member(ext_a(T),a) \leftarrow p(a), member(T,X).$$
$$member(ext_b(T),b) \leftarrow p(b), member(T,X).$$
$$member(ext_a(T),X) \leftarrow p(a), member(T,X).$$
$$member(ext_b(T),X) \leftarrow p(b), member(T,X).$$

We will use the notation $[a_1,\ldots,a_k]$ for the term $ext_{a_1}\cdots ext_{a_k}(0)$, i.e., the list consisting of $a_1,\ldots,a_k$.

The following facts are in L_S:

$p(a).$
$p(b).$
$null(0).$
$member([a], a).$
$member([a, a], a).$
$member([b], b).$
$member([b, b], b).$
$member([b, a], a).$
$member([b, a], b).$
$member([a, b], a).$
$member([a, b], b).$
$member([a, b, a], a).$
$member([a, b, a], b).$
$member([b, b, a], a).$
$member([b, b, a], b).$
$\ldots$

Therefore $[a] \sim [a, a]$, $[b] \sim [b, b]$, $[a, b] \sim [b, a]$, $[b, a] \sim [a, b, a]$ and $[b, a] \sim [b, b, a]$. Assuming ext_a is numbered with 1 and ext_b with 2, the precedence ordering $\prec$ between terms [4] is as follows:

$$0 \prec [a] \prec [b] \prec [a, a] \prec [b, a] \prec [a, b] \prec [b, b] \prec [a, b, a] \prec [b, b, a].$$

In S1 the graph of the function map_{ext_a} is computed and in S2 the graph of the function map_{ext_b}.

The algorithm will compute:

ACTIVE : $0, [a], [b], [b, a]$
POTENTIAL : $0, [a], [b], [a, a], [b, a], [a, b], [b, b], [a, b, a], [b, b, a]$
DUPLICATES : $[a, a], [a, b], [b, b], [a, b, a], [b, b, a]$
S1 : $(0, [a]), ([a], [a]), ([b], [b, a]), ([b, a], [b, a])$
S2 : $(0, [b]), ([a], [b, a]), ([b], [b]), ([b, a], [b, a])$.

The representative terms are: $0, [a], [b], [b, a]$. They correspond to four clusters – elements of the quotient model N_S. The primary database PDB is as follows:

$p(a).$
$p(b).$
$null(0).$
$member([a], a).$
$member([b], b).$
$member([b, a], a).$
$member([b, a], b).$

[4] Other terms, like $[a, a, a]$ are not listed in the ordering because they will be ignored by the algorithm, as not belonging to POTENTIAL.

The set ACTIVE of representative terms, the primary database PDB and the graphs S1 and S2 constitute the graph specification of the quotient model N_S.

6 Equational specification

We have shown one way of finitely representing the quotient model N_S. Here we show an alternative method which makes more significant use of $\cong$ being a congruence.

6.1 Finite representation of an infinite closure

The congruence $\cong$ is an infinite relation. However, it can be represented as a *closure* W^* of a finite relation W. The closure is defined in the following way:

- *Initialization:* $(t_1, t_2) \in W \Rightarrow\ (t_1, t_2) \in W^*$.
- *Reflexivity:* $(t, t) \in W^*$.
- *Symmetry:* $(t_1, t_2) \in W^* \ \Rightarrow\ (t_2, t_1) \in W^*$.
- *Transitivity:* $(t_1, t_2) \in W^*$ and $(t_2, t_3) \in W^* \ \Rightarrow\ (t_1, t_3) \in W^*$.
- *Congruence:* $(t_1, t_2) \in W^* \Rightarrow (f(t_1), f(t_2)) \in W^*$ where f is a unary function symbol appearing in the program S.

Our notion of closure is the infinite version of (finite) *congruence closure* which is used in the congruence closure procedure [Koz77, NO80, DST80, KR90]. Instead of a finite graph, we use an infinite one.

The *equational specification* $E(N_S)$ of a quotient model N_S is a triple (T, W, B) where T is a set of representative terms and B is the primary database. Both are defined in the same way as in the definition of the graph specification in the previous section. However, instead of the graph F of successor mappings, we have now a finite binary relation W between ground functional terms such that W^* is equal to $\cong$.

Example 6.1 Consider Example 1.1 with $W = \{(0, 2)\}$. W^* contains all pairs (t_1, t_2) such that $|t_1 - t_2| = 2i$ for some $i = 0, 1, \ldots$

W^* is not explicitly computed but implicitly present in W. Only if after having computed the specification $E(N_S) = (T, W, B)$ we want to verify whether $\models_{M_S} q(t_1, \bar{a})$, we compute the (finite) set $V = \{t\ :\ q(t, \bar{a}) \in B\}$. Then we verify whether for some $t_2 \in V$, $(t_1, t_2) \in W^*$. The last test is performed by the congruence closure procedure. Although the entire W^* is infinite, the test $(t_1, t_2) \in W^*$ needs to examine only finitely many terms because of the finiteness of B and W. Later, we will show a faster method to answer yes-no queries on the basis of an equational specification.

Example 6.2 In the example 1.1, try to verify whether $meets(4, emma)$ and $meets(3, emma)$ hold in M_S. i.e., whether $(0, 4) \in W^*$ and $(0, 3) \in W^*$. We obtain $(1, 3) \in W^*$, $(2, 4) \in W^*$ and $(0, 4) \in W^*$ but not $(0, 3) \in W^*$.

6.2 Computing the equational specification

We will now show how to obtain the relation W. For simplicity, we restrict ourselves to programs in basic form (where $\cong$ is identical to $\sim$).

Consider Algorithm **QF** (Figure 1). We define for any function symbol f:

$$(t_1, f(t_2)) \in W \text{ iff } t_1 \in \text{ACTIVE and } t_2 \in \text{ACTIVE and } t_1 \sim f(t_2) \text{ and } t_1 \neq f(t_2).$$

Theorem 6.1 *The closure W^* of the relation W is equal to the congruence $\cong$.*

Proof:

By the definition of W and the soundness of rules defining W^* , we obtain that W^* is contained in $\sim$. To show:

$$t_1 \sim t_2 \Rightarrow (t_1, t_2) \in W^*$$

it is enough to show that for any t_0 and t such that t_0 is the representative term of t, $(t_0, t) \in W^*$. If this is the case and t_0 is the representative term of both t_1 and t_2, we have that $(t_0, t_1) \in W^*$ and $(t_0, t_2) \in W^*$. The desired conclusion $(t_1, t_2) \in W^*$ can then be derived by symmetry and transitivity.

We proceed by induction on the depth of terms. If it is 0, then clearly $(0, 0) \in W^*$ because $0 \in$ ACTIVE and reflexivity holds. Consider the term $t = f(t_1)$. By the inductive assumption, we have $(t_3, t_1) \in W^*$, where t_3 is the representative term of t_1. Thus, $(f(t_3), f(t_1)) \in W^*$ can be immediately derived by the congruence rule. If $f(t_3) \in$ ACTIVE, then $f(t_3)$ is the representative term of $f(t_1)$ which gives the inductive thesis. Otherwise, take t_4, the representative term of $f(t_3)$. We have:

$$t_4 \in \text{ACTIVE and } t_3 \in \text{ACTIVE and } t_4 \sim f(t_3) \text{ and } t_4 \neq f(t_3).$$

Therefore, we have that $(t_4, f(t_3)) \in W$ and by the initialization rule $(t_4, f(t_3)) \in W^*$. Now from $(f(t_3), f(t_1)) \in W^*$, we can derive $(t_4, f(t_1)) \in W^*$ by transitivity. Moreover, t_4 is the representative term of $f(t_1)$ which gives the inductive thesis.

For *Datalog*$_{nS}$ programs in general form, the proof can be carried out similarly. □

6.3 Rewrite rule interpretation

Note that by the above construction of W: if $(t_1, t_2) \in W$, then $t_1 \prec t_2$. Thus, a pair $(t_1, t_2) \in W$ can be interpreted as right-to-left ground rewrite rule ("rewrite t_2 to t_1"). We will write $t \overset{W}{\leadsto} t_0$ to indicate that the term t can be rewritten to t_0 using W (possibly in many steps) and t_0 can not be further rewritten using W.

Theorem 6.2 *For every ground functional term t, $t \overset{W}{\leadsto} t_0$ where t_0 is the representative term of t.*

Proof: We prove the theorem in several steps. First, we show that using W, a term is rewritten to a congruent term. Second, we show that a representative term can not be further rewritten. Third, we show that if a ground term is not a representative, there is always exactly one pair in W that can be used to rewrite it. Fourth, we show that the rewriting using W always terminates. Taken together, all the steps give the desired thesis. For simplicity, we assume that the $Datalog_{nS}$ program S is in basic form and the state equivalence is a congruence.

Take a term $t = f_k \cdots f_1(t_2)$ which is being rewritten using a pair (t_1, t_2). The resulting term $t = f_k \cdots f_1(t_1)$ is clearly congruent to t because t_1 is congruent to t_2.

Denote by $I(t', t'')$ the set of all terms that are both proper superterms of t' and subterms (not necessarily proper) of t'' (assuming t'' is a superterm of t').

Note that all subterms of a representative term t_0 are also representative. Otherwise, there is a proper subterm t_1 of t_0, which is not a representative term. It is easy to see that no term in $I(t_1, t_0)$, including t_0, will be considered as a candidate representative term (an element of POTENTIAL) by the algorithm **QF**. This contradicts the fact that t_0 is a representative term.

A representative term can not be the second argument of a pair in W. Otherwise, by the definition of W we would have two congruent representative terms: a contradiction. Therefore, neither a representative term itself nor any of its subterms (which are also representative terms) can be rewritten using W.

Consider now a non-representative ground term t. The term t has a proper subterm t_0 such that t_0 is a representative term and all terms in $I(t_0, t)$ are not representative. This follows from the fact that 0 is always a representative term and from the properties of the algorithm **QF** discussed above. Take t_2 to be the least term in $I(t_0, t)$. We have that $t_2 = f(t_0)$ for some function symbol f and t_2 is congruent to some representative term t_1 ($t_1 \neq t_2$). Moreover, t_0 is a representative term. Thus, by the definition of W, the pair $(t_1, t_2) \in W$. Take any term $t_3 \in I(t_0, t)$ other than t_2. The term t_3 is equal to $f(t_4)$ for some function symbol f and term $t_4 \in I(t_0, t)$. But t_4 is not representative, and therefore t_3 can not be the second argument of a pair in W. Moreover, any subterm term of t_0, including t_0, is representative and thus can not be the second argument of a pair in W. We conclude then that only one subterm of a non-representative term can be rewritten using W. The rewriting is deterministic.

To prove termination, we have to use the fact that the precedence ordering $\prec$ satisfies the condition:

$$t_1 \prec t_2 \text{ implies} f(t_1) \prec f(t_2)$$

for all function symbols f.

We have that $t_1 \prec t_2$ for every pair $(t_1, t_2) \in W$. Therefore, if a term t is rewritten to another term t_3, $t_3 \prec t$. There is only a finite number of terms with bounded depth, thus the precedence ordering is well-founded. Consequently, every rewriting using W terminates. $\square$

Theorem 6.2 suggests a fast, deterministic method to determine whether $\models_{M_S} q(t, \bar{a})$ given $E(N_S) = (T, W, B)$: rewrite t obtaining t_0 which can not be further rewritten using W, then check whether $q(t_0, \bar{a}) \in B$.

6.4 Periodicity

Note that in the case of *Datalog$_{1S}$*, there is only one finite function in the graph specification $G(N_S)$, whose graph is a single line ending with a cycle. In the equational specification $E(N_S)$, the set of pairs W consists of just one pair capturing the *periodicity* of the least model. For programs in basic form, the first element of this pair is 0.

We are going to say that a pair (t_0, m) is a *period* of a (possibly infinite) set of *Datalog$_{1S}$* facts M if:

$$\forall t \; t \geq t_0 \text{ implies } M[t] = M[t+m].$$

A set is *periodic* if it has a finite period.

Let L_S be the set of facts that correspond to tuples assigned by M_S to predicates in a *Datalog$_{1S}$* program S (in other words, the least fixpoint of S). We show now that L_S is periodic.

Proposition 6.1 *The set L_S is periodic with a period $(b+c, m)$ such that*

$$b + m \leq scope_{\sim}(L_S)$$

and c is the maximal ground functional term in S.

Proof: This proposition follows from the proof of the Theorem 5.1 and the way the equational specification is constructed. Note that the only rewrite rule will be $(b+c, b+c+m)$. □

6.5 Canonical form

The results on finite representability in this section have another interesting interpretation: they provide a canonical form for *Datalog$_{nS}$* programs. Define a set of rules *CONGR* consisting of:

- recursive rules describing W^* listed at the beginning of this section.
- recursive rules, one for each predicate p, of the form:

$$p(T_1, \bar{X}) \leftarrow p(T_2, \bar{X}), T_1 \cong T_2.$$

The rules in *CONGR* do not satisfy the definition of *Datalog$_{nS}$*: the predicate $\cong$ has two functional arguments and there are two functional variables (T_1 and T_2).

For a *Datalog$_{nS}$* program S, consider a logic program S' consisting of the set of rules *CONGR*, the primary database B and the relation W where $E(N_S) = (T, W, B)$. The relations that $M_{S'}$ and M_S assign to the predicates from S are identical.

Therefore, we can treat the logic program S' as a canonical form of S. This interesting property of *Datalog*$_{nS}$ programs, which probably can not be generalized to logic programs with arbitrary function symbols, makes them analogous to *Datalog* programs. Any *Datalog* program has a canonical form - the least Herbrand model which is always a finite relational database. Here, any *Datalog*$_{nS}$ program has also a canonical representation which is always finite (although it is no longer a relational database!)

It is important to note that the set of rules $CONGR$ depends on the set of predicates in S but not on the actual rules in S. Therefore, it might be possible to develop optimization techniques for *Datalog*$_{nS}$ programs that will optimize $CONGR$ rules independently of the set of rules in S.

7 Computational complexity

In this section, it will be convenient to treat a *Datalog*$_{nS}$ program S as a union of a set of *Datalog*$_{nS}$ rules Z and a *Datalog*$_{nS}$ database D, written as $Z \cup D$.

7.1 Data complexity

The graph $Gr(Z, Q)$ of a set of *Datalog*$_{nS}$ rules Z and an open atomic query $Q \equiv q(\bar{x})$ is defined as:

$$Gr(Z, Q) = \{(\bar{a}, D) : \models_{M_{Z \cup D}} q(\bar{a})\} = \{(\bar{a}, D) : Z \cup D \models q(\bar{a})\}$$

where $\bar{a}$ is a vector of ground terms of appropriate arity (such that $q(\bar{a})$ is ground) and D is a *Datalog*$_{nS}$ database. The complexity of $Gr(Z, Q)$ for a fixed Z and Q will be called the data complexity [CH80, Var82] of *recognition* (yes-no query processing). The complexity of computing the finite representation of $M_{Z \cup D}$ for a fixed Z will be called the data complexity of *generation* (computing the entire least Herbrand model).

We will identify the following parameters of a *Datalog*$_{nS}$ program $Z \cup D$ and a fact query $q(\bar{a})$:

- s - the number of predicates in $Z \cup D$.
- k - the maximum arity of a predicate in $Z \cup D$.
- 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 no such term).
- h - the depth of the functional term in $\bar{a}$ ($h = 0$ if no such term).

As discussed above, the data complexity assumption means that s and k will be fixed, while n, d, c and h will vary. If the program is in basic form, $c = 0$.

Theorem 7.1 *[Für81] For $Datalog_{nS}$ programs, recognition is DEXPTIME-data-complete.*

In [Cho90b], we show a stronger result by constructing a deterministic interpreter for $Datalog_{nS}$ programs working in time exponential in $max(n, \log c, \log h)$, where n, c and h are defined above.

Theorem 7.2 *[Pla84, CI88] For $Datalog_{1S}$ programs, recognition is PSPACE-data-complete.*

In [Cho90b], we show a stronger result by constructing a deterministic interpreter for $Datalog_{1S}$ programs working in space polynomial in $max(n, \log c, \log h)$.

Both strengthenings are crucial for our results about the complexity of computing relational specifications.

7.2 Number of states

As before, denote by $L_{Z \cup D}$ the set of facts corresponding to tuples assigned by $M_{Z \cup D}$ to predicates. We need to know precisely how many atoms (facts) a state of $L_{Z \cup D}$ can contain. Denote that number by $size_f$. We can restrict our attention to well-typed functional facts built with symbols appearing in $Z \cup D$. If there are s_i functional predicates of arity i $(1 \leq i \leq k)$ in Z and 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 of $L_{Z \cup D}$ can contain at most $size_d$ facts where:

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

Because we are dealing with 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).

The number of different states in $L_{Z \cup D}$ is at most 2^{size_f}. Thus, the equivalence scope of $L_{Z \cup D}$ (as defined in Section 4)

$$scope_{\sim}(L_{Z \cup D}) \leq 2^{size_f}$$

where $size_f$ is a polynomial function of the number of facts in the database.

Below, we show that also the congruence scope $scope_{\cong}(L_{Z \cup D})$ is finite and at most exponential in the database size.

Lemma 7.1 *Let $Z \cup D$ be a $Datalog_{nS}$ program. For a fixed Z, the congruence scope $scope_{\cong}(L_{Z \cup D})$ is at most exponential in n - the number of facts in D and polynomial in c - the maximum depth of terms in D.*

Proof: It is clear that the number of repetitive clusters is at most exponential in $size_f$, and consequently exponential in n. Thus, only the number of initial clusters remains to be bounded. This number is equal to the number of subterms of functional terms in D (no ground terms appear in Z) and is easily seen to be polynomial in $max(n, c)$. $\Box$

7.3 Computing states

From the following lemma, it follows that for a $Datalog_{nS}$ program both $\sim$ and $\cong$ are decidable.

Assume n is the number of facts in a $Datalog_{nS}$ database D, c - the maximum depth of terms in D, t - a ground functional term and $b = max(c, depth(t))$.

Lemma 7.2 *For every $Datalog_{nS}$ program $Z \cup D$, computing the state $L_{Z\cup D}[t]$ can be done:*

- *in time exponential in $max(n, \log b)$,*
- *in space polynomial in $max(n, \log b)$ if $Z \cup D$ is in $Datalog_{1S}$*

under the data complexity assumption.

Proof: Recognition for $Datalog_{nS}$ is decidable in time exponential in $max(n, \log b)$ (strengthened Theorem 7.1). For $Datalog_{1S}$, it is decidable in space polynomial in $max(n, \log b)$ (strengthened Theorem 7.2). To compute the state $L_{Z\cup D}[t]$, verify whether

$$Z \cup D \models p(t, \bar{a})$$

for every predicate p and every vector $\bar{a}$ built from data constants appearing in $Z \cup D$. There are only finitely (polynomially) many such vectors. □

7.4 Computing relational specifications

Theorem 7.3 *Algorithm* **QF** *(Figure 1) works in deterministic exponential time (data complexity).*

Proof: Assume n is the number of facts in the database D.

The crucial observation is that only the terms in POTENTIAL are considered by the algorithm. All the terms in ACTIVE correspond to different states, therefore the cardinality of ACTIVE is equal to m where $m{=}scope_{\sim}(L_{Z\cup D})$. The cardinalities of POTENTIAL and DUPLICATES are polynomial functions of m. The depth of terms in ACTIVE is bounded by m and the depth of terms in POTENTIAL and DUPLICATES is bounded by $m+1$ because the number of iterations of the outer loop is at most $m+1$. No term is added to ACTIVE in the last iteration. The state $L_{Z\cup D}[t]$ is computed only if $t \in$ POTENTIAL. We have that $depth(t)$ is then at most exponential in n (see above). Thus using Lemma 7.2, we conclude that $L_{Z\cup D}[t]$ can be computed in time exponential in n. Only an exponential number of states (those corresponding to terms in POTENTIAL) is ever computed. Thus the main phase of **QF** takes exponential time (data complexity).

Similarly, the primary database PDB, and the graphs of the finite functions S1 and S2 can also be computed in time exponential in n. □

Proposition 7.1 *The Revised Algorithm* **QF** *runs in time exponential in the number of facts in the database D and polynomial in the depth of terms in D (data complexity).*

Proof: The result follows from Theorem 7.3 and the definition of the algorithm. □

The algorithm **QF** can be easily improved, e.g., by using pairs $(t, L_{Z\cup D}[t])$ instead of separate sets of terms and states. Such optimizations will remove some redundant computations. By choosing the smallest possible representative terms, we make a relational specification as small as possible. One could use a more economical, polynomial size, representation for representative terms because there are only exponentially many of them. However, computing a relational specification requires exponential time in the worst case because the size of the specification may be exponential.

Theorem 7.4 *The upper and the lower bound on the size of the graph specification $G(N_{Z\cup D})$ and the equational specification $E(N_{Z\cup D})$ of a quotient model $N_{Z\cup D}$ is exponential in the number of facts in D (under the data complexity assumption).*

Proof:

The overall size of the relational specification is at most exponential (upper bound) by Theorem 7.3. We show now a set of $Datalog_{1S}$ rules and an infinite family of $Datalog_{1S}$ databases that lead to relational specifications whose size is exponential in the number of facts in the database. The same example was used in [CI88] to demonstrate an exponential lower bound on the execution time of a resolution-based interpreter for $Datalog_{1S}$.

We 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 contains the facts $zero(0, a_i)$ for every $1 \leq i \leq n$, $next(a_i, a_j)$ 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:

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

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

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

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

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 least model $M_{Z \cup D}$ is infinite and has exponentially many (2^n) different states and therefore the number of clusters is exponential. Even if the representative terms are encoded using polynomial space, the size of the primary database will still be exponential (it will contain exponentially many nonempty snapshots). □

It is easy to obtain from the *Datalog*$_{1S}$ program given above another program whose least model is finite and of exponential size. For example, carry copying can be made conditional upon having at least one zero bit in the current number (a condition easily expressible in *Datalog*$_{1S}$). In fact, if the least model $M_{Z \cup D}$ of a *Datalog*$_{1S}$ program is finite, the primary database in the relational specification of $N_{Z \cup D}$ is equal to the relations assigned by $M_{Z \cup D}$ to predicates. This does not have to be the case for arbitrary *Datalog*$_{nS}$ programs where *finite* least models can be of double exponential size, while relational specificatons are always at most single exponential, thus providing a more economical representation. Consider the following example:

Example 7.1 The database D is identical to the one in the proof of the preceding Theorem. For the set of rules Z, we take two copies of the rules from the same proof: one with the function symbol s_1 replacing $+1$, the other with s_2 replacing $+1$. We also apply the modification suggested above to obtain a finite least model, but now its size is double exponential.

7.5 *Datalog*$_{1S}$

In the case of *Datalog*$_{1S}$, the relation W in the equational specification $E(N_{Z \cup D}) = (T, W, B)$ contains just one pair capturing the periodicity of the least fixpoint. The set of facts B can be, however, exponentially sized.

Note that for *Datalog*$_{nS}$ programs, it is not harder (in the sense of complexity class) to compute a finite representation of the least fixpoint than to answer yes-no queries! Is it also the case for *Datalog*$_{1S}$? The following theorem provides an affirmative answer.

Theorem 7.5 *The equational specification $E(N_{Z \cup D})$ of a Datalog$_{1S}$ program $Z \cup D$ can be computed in space polynomial in the number of tuples in the database D and polylogarithmic in the depth of functional terms in D (under the data complexity assumption).*

Proof: We show how to modify the Algorithm **QF** so that it works in polynomial space for any *Datalog*$_{1S}$ program S.

Note that there is now only one term of a given depth (only one unary function symbol), thus the representative terms form a sequence: $0, 1, \ldots, t_0$ for some term t_0. It is enough to keep only one current term t and work as follows.

At the beginning, let the current term $t = 0$. Compute $L_S[t]$ and compare it with the states of L_S corresponding to the proper subterms of t. Each of these states is recomputed anew. If none of these comparisons succeeds, t is a representative term and can be output as such. In this case, $t+1$ is made the current term and the procedure repeated. Otherwise, let t_1 be the subterm of t such that $L_S[t_1] = L_S[t]$ (there can be only one such term). Output $\{(t_1, t)\}$ as the set of rewrite rules W, compute the snapshots of L_S corresponding to the proper subterms of t (this will give the primary database), and terminate.

The term t, altough exponential, can be represented in binary in polynomial space. A state can be computed and compared with another state for equality in polynomial space (Lemma 7.2). It is crucial here that the terms considered by the algorithm are at most exponential in the number of facts in the database.

The extension of the above algorithm to programs which are not in basic form is straightforward. The space complexity of such an extension will be polylogarithmic in the depth of terms in D. □

7.6 Finiteness of least models

From Theorem 7.4 and Proposition 5.2, the following corollary follows immediately:

Corollary 7.1 *Finiteness of the least model $M_{Z \cup D}$ of a Datalog$_{nS}$ program $Z \cup D$ can be determined in time exponential in the number of facts in D and polynomial in the depth of functional terms in D (under the data complexity assumption).*

For *Datalog*$_{1S}$, it is enough to see whether for the equational specification $E(N_{Z \cup D}) = (T, W, B)$, the single pair in W is of the form $(t_0, t_0 + 1)$ and the snapshot $B(t_0)$ is empty. If both these conditions are satisfied, $M_{Z \cup D}$ is finite. Otherwise, it is infinite. Thus, in view of Theorem 7.5:

Corollary 7.2 *Finiteness of the least model $M_{Z \cup D}$ of a Datalog$_{1S}$ program $Z \cup D$ can be determined in space polynomial in the number of facts in D and polylogarithmic in the depth of functional terms in D (under the data complexity assumption).*

8 Related work

8.1 Non-standard query answers

There are many possible definitions of a query answer in logic programs. In this paper, we start from the concept of a *ground* answer. In logic programming, the answers are not necessarily ground but rather *most general*. However, for range-restricted rules and ground databases, ground and most general answers coincide.

Several researchers have recognized the need for higher-level, intensional, implicit query answers. However, the motivation and the applicability of various methods differ. Imielinski [Imi87b] and Cholvy and Demolombe [CD87] propose different intensional representations, in terms of rules [Imi87b] or formulas [CD87], for answers to queries. The approach of [CD87] tries to find a (first-order) formula that will express the query answer. It is clear that no such formula exists when recursive rules are allowed, and consequently this approach has limited applicability. The approach tries to obtain formulas true in every database state through a variant of resolution. Recursive formulas for the query can not be obtained in this way because they are not implied by the rules. However, intuitively they can be inferred, because they are true in every least Herbrand model.

Example 8.1 Take the following set of rules:

$$p(X,Z) \leftarrow w(Y,Z), p(X,Y).$$
$$q(Y) \leftarrow p(X,Y).$$

If the query is defined as $q(Y)$, the query answer can be finitely represented using a predicate q defined using the following rules:

$$q(Y) \leftarrow p(X,Y).$$
$$q(Z) \leftarrow w(Y,Z), q(Y).$$

However, this set of rules is not implied by the original set of rules. To see this take the rules in conjunction with a database consisting of the fact $w(a,b)$. Now the interpretation containing $\{q(a), w(a,b)\}$ but not $q(b)$ is a model of the original rules and the database, but is not a model of the recursive rule

$$q(Z) \leftarrow w(Y,Z), q(Y).$$

The least Herbrand model contains just $w(a,b)$ and does not depend on which set of rules is chosen.

The approach of [Imi87b] is more general as it can generate recursive rules in query answers (it will give the intuitively correct answer in the example above). Still, the class of queries and rules which can be handled by this method is limited to *Datalog*. In [CI89, Cho90b], we discuss a similar approach. Infinite least models (of $Datalog_{1S}$ programs) are represented there as facts with *infinite objects*: elements of an extended Herbrand universe. These objects are returned in query answers. However, not every $Datalog_{1S}$ program can be handled in this way and the restrictions imposed by this method are rather arbitrary. Moreover, even for simple rules the resulting infinite objects are very complex.

The methods of [Imi87b, CI88] mentioned above are syntactic in nature: they seek to replace the bottom-up computation of the original set of rules Z (which could yield an infinite answer) by a bottom-up computation of some other set of rules Z_1 (which will terminate). The transformation leading from Z to Z_1 is performed independently of the database D. Our approach in this paper is semantic: we provide a method to homomorphically map the least model $M_{Z \cup D}$ of a $Datalog_{nS}$ program to a finite structure $N_{Z \cup D}$. Of course, such representation will be database dependent. On the other hand, it is query independent (so is the method of [CI88]) because answer substitutions w.r.t. $N_{Z \cup D}$ can be easily obtained from a relational specification of $N_{Z\cup D}$. We conjecture that there is no method to finitely (and explicitly) represent the least model of every $Datalog_{nS}$ program in a database independent way. To prove this conjecture, one needs to formalize the notion of *database independence* in our current framework.

It is interesting to see whether the approach of [CI88] can be interpreted within the general framework developed in Section 4. The construction in [CI88] does not give a homomorphic image of the least model because *object functions* introduced there do not correspond to function symbols from the original program but rather capture the way in which functional terms are constructed during the bottom-up execution of the program. Moreover, because infinite objects represent infinite sets of functional terms, answer substitutions returning those objects are not really first-order.

Our approach can be viewed as an instance of a more general paradigm outlined in [Imi87b]: providing answers in terms of *concepts* acceptable to the user. Here, the relevant concept is clearly *homomorphism* and in the context of $Datalog_{nS}$, *congruence* relation. We claim that the notion of congruence, particularly if presented pictorially, can be made understandable for the user.

Our method of finite representation of infinite answers may also be seen as an analogue of *domain abstraction* defined in a different context by Imielinski [Imi87a]. Instead of dealing with infinite sets, we deal with their representatives. This abstraction is not approximate but precise: fact queries preserve their logical values. There is another important difference. In [Imi87a] the domain abstraction was given a priori, here it is constructed anew for each $Datalog_{nS}$ program.

An interesting idea suggested by Corella [Cor87] and developed by Shum and Muntz [SM88] is to use additional information, presented e.g., as taxonomies of concepts, to return a more compact and more useful answer to a query. So far, this idea has been applied only to *Datalog*.

8.2 Finite representation

Since the first version of this paper was presented [CI89], several papers have appeared that also deal with the issue of finite representation of infinite query answers: [HJ90b, HJ90a, FSVY91, CH91]. The approaches pursued in those papers are, however, different from ours.

Heintze and Jaffar [HJ90b, HJ90a] use *set constraints* to represent approximations of least Herbrand models of logic programs. Their approximation is precise for a class of logic programs that properly contains $Datalog_{nS}$, thus their method is more general than ours. Heintze and Jaffar propose to represent least Herbrand models as collections of set equations in explicit form. As they notice, such representations are equivalent to *non-deterministic tree automata* [GS84] and therefore can be used to answer queries. For example, verifying whether a given fact is true in the least Herbrand model of a logic program can be done by checking whether the automaton corresponding to the program accepts the fact. Collections of set equations or tree automata are, however, less explicit than the representations proposed in this paper. Heintze and Jaffar's approach does not, for example, make the periodicity implicit in a logic program (c.f., Example 1.1) explicit.

Frühwirth, Shapiro, Vardi and Yardeni [FSVY91] propose an approach similar in spirit to that of Heintze and Jaffar. Instead of set constraints, they use unary logic programs. Their approach is slightly more general than Heintze and Jaffar's and they also supply relevant computational complexity results.

Interestingly, the example programs that do not have a finite representation in our sense (Section 4) are not handled by the above mentioned methods either. We conjecture that when the methods of [HJ90b, HJ90a, FSVY91] produce an exact finite representation of a least Herbrand model M_S of a logic program S, the program S has a finite model homomorphic to M_S (thus a finite representation in our sense).

Chen and Hsiang [CH91] introduce ω-terms: abstractions corresponding to infinite sets of terms. It seems that every least Herbrand model of a *Datalog*$_{nS}$ program can be represented as a finite set of facts containing ω-terms. However, constructing this set of facts on the basis of a *Datalog*$_{nS}$ program is tantamount to constructing a relational specification. For example, if a user wants to represent a periodic phenomenon using ω-terms, she has to determine the period first and only then encode the periodic set just obtained as an ω-term. This should be contrasted with our approach where a user supplies a logic program and the periodicity is extracted from the program by the computational mechanism. However, the approach of Chen and Hsiang deals with a larger class of programs than *Datalog*$_{nS}$. It seems capable of handling at least Example 4.5 for which a finite representation in our sense does not exist.

8.3 Constraints

Constraints, introduced in the context of logic programming as the language CLP by Jaffar and Lassez [JL87], address also the issue of finite representation. For example, the set of even numbers can be represented by the constraint:

$$x \bmod 2 = 0.$$

To represent the set of ground terms t such that $\models_{M_S} p(t)$ for a *Datalog*$_{1S}$ program S, the periodicity of the least Herbrand model M_S has to be determined first. Only then one can write a constraint representing this set of terms as:

$$t = s_1 \text{ or } \ldots \text{ or } t = s_m \text{ or } \exists i_1, \ldots, i_k \; (t = t_1 + i_1 d \text{ or } \ldots \text{ or } t = t_k + i_k d)$$

where d is the period of M_S and $s_1, \ldots, s_m, t_1, \ldots, t_k$ are representative terms. Such a constraint can also be constructed if p is defined by a *Datalog*$_{nS}$ program with more than one function symbol. Of course, different algebraic operations defined on strings of function symbols have to be used instead of addition and multiplication.

It is interesting to note that the constraints of the type outlined above are treated as *database objects* by Kabanza, Stevenne and Wolper [KSW90] (the constraints are called *linear repeating points*). In this paper, relational calculus and relational algebra are extended to handle such objects. Also, inequality constraints are introduced to represent temporal intervals. In a follow-up work, Baudinet, Niézette and Wolper [BNW91] combine *Datalog* with linear repeating points to obtain a very expressive language for which, however, the termination of query evaluation is not guaranteed. In the latter paper, the authors formally establish the relationship between the expressive power of the relational calculus (or algebra) on relations with linear repeating points, and $Datalog_{1S}$ (which is equivalent to the function-free version of Templog, the language studied by Abadi, Manna and Baudinet [AM89, Bau89]). They distinguish between *data* and *query* expressiveness. Data expressiveness characterizes the power of a language to represent temporal data. In this respect, a restricted version of the calculus and $Datalog_{1S}$ are equally expressive. Query expressiveness characterizes the power of a language to extract data from a database. In this respect, the calculus and $Datalog_{1S}$ are incomparable. The first can express all *star-free* ω-regular languages [Tho90], the second – all *finitely-regular* ω-languages (as shown by Baudinet [Bau91]).

Another relevant approach is that of Kanellakis, Kuper and Revesz [KKR90, Rev90]. There, various constraint query languages based on CLP are defined. Among them, *Datalog* with integer order [Rev90] is the closest to our approach. This language can not express infinite periodic sets because it lacks even a rudimentary arithmetic capability. On the other hand, $Datalog_{1S}$ can not express integer order.

8.4 Decidable classes of logic programs with function symbols

Here, we just mention the salient properties of our approach that distinguish it from others [Bla86, Bez89, AC89]: syntactic definition of the classes ($Datalog_{nS}$, $Datalog_{1S}$), existence of finite representation of infinite query answers, a natural class of database applications (storing and retrieving information about state transitions in infinite processes). In Section 4, we have shown several programs leading to query answers which can not be finitely represented. It is interesting to note that all of those programs fall into the decidable classes of recurrent programs [Bez89] and determinate programs [Bla86]. Those classes, are not, however, defined syntactically. In fact, determining whether a program is recurrent is Π^0_2-complete.

9 Conclusions

9.1 Summary

We have presented a general framework of finite representation of infinite query answers in logic programs. We have applied it to $Datalog_{nS}$, a class of logic programs with limited function symbols. For $Datalog_{nS}$, finite representations can be effectively computed. Our method handles every range-restricted $Datalog_{nS}$ program. We have also shown examples of non-$Datalog_{nS}$ logic programs for which finite representations of query answers do not exist.

9.2 Further work

It is well known that the least Herbrand model of a logic program constitutes an *initial structure* in the sense used in algebraic specifications, c.f., Makowsky [Mak85] and Padawitz [Pad88]. The quotient model that we study in this paper has an equally strong formal motivation: it constitutes a *final structure* [Wan79], obtained by treating the functional domain as an *invisible sort* [Pad88]. We claim that final structures, being less verbose than initial ones, are preferable from the computational point of view. It is very challenging to establish the existence or non-existence of *finite* final structures for logic programming languages richer than $Datalog_{nS}$.

We have also conjectured in this paper that finite representability implies decidability. This remains to be shown. Moreover, it might be wortwhile to study mappings that are weaker than homomorphism in our sense and obtain *approximate* finite representations for larger classes of logic programs.

For $Datalog_{nS}$ to be applicable in practice, several issues have to be resolved, in particular the definition of tractable (polynomial-time computable) subclasses and efficient query evaluation methods. In [Cho90c], we report some progress in the first direction. We obtain a number of tractable subclasses of $Datalog_{1S}$ using polynomial size of relational specifications as a sufficient condition for tractability. Unfortunately, our results there don't seem to generalize to full $Datalog_{nS}$. As far as the issue of efficient query evaluation is concerned, we think that rule optimization methods developed for *Datalog* [BR86, Ull89] can also be applied to $Datalog_{nS}$.

10 Acknowledgments

The work of the first author was mainly performed at the University of Maryland Institute for Advanced Computer Studies and the University of North Carolina, Chapel Hill. The financial support and the stimulating research environment provided by Jack Minker at Maryland are gratefully acknowledged. The detailed comments of the anonymous reviewers are greatly appreciated.

References

[AC89] L. Aiello and C. Cecchi. Adding a Closure Operator to the Extended Relational Algebra: a Further Step Towards the Integration of Database Techniques and Logic Programming. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, pages 179–200. Springer-Verlag, 1989.

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8(3), September 1989.

[Bau89] M. Baudinet. Temporal Logic Programming is Complete and Expressive. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1989.

[Bau91] M. Baudinet. On the Expressiveness of Temporal Logic Programming. Submitted, September 1991.

[Bez89] M. Bezem. Characterizing Termination of Logic Programs with Level Mappings. In *North American Conference on Logic Programming*, 1989.

[Bla86] H.A. Blair. Decidability in the Herbrand base. In *Workshop on Foundations of Deductive Databases and Logic Programming*, Washington,D.C., 1986. Manuscript.

[BNW91] M. Baudinet, M. Niézette, and P. Wolper. On the Representation of Infinite Temporal Data and Queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1991.

[BR86] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *ACM SIGMOD International Conference on Management of Data*, 1986.

[CD87] L. Cholvy and R. Demolombe. Querying a rule base. In L. Kerschberg, editor, *Expert Database Systems*, pages 477–485. Benjamin/Cummings, 1987.

[CH80] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

[CH85] A.K. Chandra and D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, 2(1):1–16, April 1985.

[CH91] H. Chen and J. Hsiang. Logic Programming With Recurrence Domains. In *International Colloquium on Automata, Languages and Programming*. Springer-Verlag, LNCS 510, 1991.

[Cho90a] J. Chomicki. A Decidable Class of Logic Programs with Function Symbols. Technical Report TR-CS-90-11, Computing and Information Sciences, Kansas State University, September 1990. 44 pp.

[Cho90b] 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.

[Cho90c] 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.

[CI88] 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.

[CI89] J. Chomicki and T. Imieliński. Relational Specifications of Infinite Query Answers. In *ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May 1989.

[Cor87] F. Corella. Semantic Retrieval and Levels of Abstraction. In L. Kerschberg, editor, *Expert Database Systems*, pages 477–485. Benjamin/Cummings, 1987.

[DST80] P. Downey, R. Sethi, and R.E. Tarjan. Variations on the Common Subexpression Problem. *Journal of the ACM*, 1980.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[FSVY91] T. Frühwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *IEEE Symposium on Logic in Computer Science*, 1991.

[Für81] M. Fürer. Alternation and the Ackermann case of the decision problem. *L'Enseignement Mathematique*, 1981.

[Gre69] C. Green. Application of Theorem Proving to Problem Solving. In *International Joint Conference on Artificial Intelligence*, 1969.

[GS84] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[HJ90a] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *IEEE Symposium on Logic in Computer Science*, 1990.

[HJ90b] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1990.

[Imi87a] T. Imieliński. Domain Abstraction and Limited Reasoning. In *International Joint Conference on Artificial Intelligence*, 1987.

[Imi87b] T. Imieliński. Intelligent Query Answering in Rule Based Systems. *Journal of Logic Programming*, 4(2), 1987.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[Koz77] D. Kozen. Complexity of Finitely Presented Algebras. In *ACM SIGACT Symposium on Theory of Computing*, pages 164–177, 1977.

[KR90] P.C. Kanellakis and P.Z. Revesz. On the Relationship of Congruence Closure and Unification. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 23–41. Addison-Wesley, 1990.

[KSW90] F. Kabanza, J-M. Stevenne, and P. Wolper. Handling Infinite Temporal Data. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 392–403, Nashville, Tennessee, April 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[Mak85] J.A. Makowsky. Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, pages 374–387. Springer-Verlag, LNCS 185, 1985.

[Mon76] J. Donald Monk. *Mathematical Logic.* Springer-Verlag, 1976.

[NO80] G. Nelson and D. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories.* Springer-Verlag, 1988.

[Pla84] D.A. Plaisted. Complete Problems in the First-Order Predicate Calculus. *Journal of Computer and System Sciences*, 29:8–35, 1984.

[RBS87] 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.

[Rev90] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. In *International Conference on Database Theory*, pages 187–201. Springer-Verlag, LNCS 470, 1990.

[She88] J.C. Shepherdson. Negation in Logic Programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[SM88] C-D. Shum and R. Muntz. Implicit Representation for Extensional Answers. In L. Kerschberg, editor, *International Conference on Expert Database Systems*, pages 257–273, Tysons Corner, Va., April 1988. George Mason University.

[Tar77] S-A. Tarnlund. Horn Clause Computability. *BIT*, 17:215–226, 1977.

[Tho90] 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.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[Var82] M.Y. Vardi. The Complexity of Relational Query Languages. In *ACM SIGACT Symposium on Theory of Computing*, pages 137–146, 1982.

[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, 1976.

[Wan79] M. Wand. Final Algebra Semantics and Data Type Extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.

A State equivalence

We prove here Theorem 4.1. Assume S is a $Datalog_{nS}$ program in basic form. We show that state equivalence relation $\sim_{L_S}$ is a congruence, i.e., for all function symbols f

$$t_1 \sim_{L_S} t_2 \text{ implies } f(t_1) \sim_{L_S} f(t_2).$$

Define a *functionally-grounded instance* $S\{t\}$ of a $Datalog_{nS}$ program S as the result of substituting the ground functional term t into the the single functional variable in S (the rules in S are normal). The instance $S\{t\}$ does not have to be ground: non-functional variables may be still unbound. It is clear that M is a model of S iff M is a model of every functionally-grounded instance $S\{t\}$.

We consider here only Herbrand structures of a given $Datalog_{nS}$ program S. Such structures can vary only in their assignment to predicates. Therefore, we will speak interchangeably about a structure M and the set of facts corresponding to tuples assigned by M to predicates.

Define $C(t)$ to be the set of all ground functional terms (built using constant and function symbols from S) that contain t as a subterm (not necessarily proper, i.e., $t \in C(t)$). Intuitively, we can think of the set of ground functional terms as cut into two parts relative to the term t.

We say that a mapping G from functional terms to functional terms is an *almost-homomorphism* [5] with respect to a functional term t_1 if:

1. $G(f(t)) = f(G(t))$ for $t \in C(t_1)$ and every function symbol f.
2. $G(t) = t$ otherwise, i.e., for $t \notin C(t_1)$.

The mapping G replaces all the occurrences of t_1 in a term by $G(t_1)$.

Let G_1 be a specific almost-homomorphism w.r.t. t_1 such that $G_1(t_1) = t_2$.

Consider the Herbrand interpretation L' defined as:

$$L'[t] = L_S[G_1(t)]$$

[5] This technical notion used only in this proof has nothing to do with the homomorphism between the least Herbrand model M_S and the finite structure N_S.

for all t, where L_S is the least Herbrand model of S. We will show that L' is a model of S. If this is the case, then

$$L_S[f(t_1)] \subseteq L'[f(t_1)]$$

because L_S is the least Herbrand model of S. But then

$$L'[f(t_1)] = L_S[G_1(f(t_1))] = L_S[f(G_1(t_1))]$$

because G_1 is a homomorphism w.r.t. t_1 and $f(t_1) \in C(t_1)$. Consequently

$$L_S[f(t_1)] \subseteq L_S[f(t_2)].$$

It is easy to see now that we can construct a symmetric proof that

$$L_S[f(t_2)] \subseteq L_S[f(t_1)].$$

using an almost-homomorphism G_2 w.r.t. t_2 such that $G_2(t_2) = t_1$. Therefore

$$L_S[f(t_1)] = L_S[f(t_2)].$$

To check whether L' is a model of S, we have to examine the entire population of functionally-grounded instances of S. Take any term t. We will show that $L' \models S\{t\}$. We will use the following Lemma.

Lemma A.1 *Assume that M and N are Herbrand structures for a Datalog$_{nS}$ program S 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:*

$$\models_M S\{t_M\} \textit{ iff } \models_N S\{t_N\}.$$

Proof: Take a ground substitution θ such that $S\{t_M\}\theta$ and $S\{t_N\}\theta$ are ground (assume that data variables in different clauses are renamed apart). Any functional atom $p(t, \bar{a})$ in $S\{t_M\}\theta$ 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 $S\{t_N\}\theta$ has the same truth value in N, as $p(t, \bar{a})$ in M.

Thus we have that for all ground θ

$$\models_M S\{t_M\}\theta \text{ iff } \models_N S\{t_N\}\theta$$

and the thesis follows. □

We return now to the proof that L' is a model of $S\{t\}$ for every ground functional term t.

Consider first $t \notin C(t_1)$. If $f(t) \notin C(t_1)$, then by definition of L_S, $L'[f(t)] = L_S[f(t)]$. Otherwise, $f(t) = t_1$. Then

$$L'[f(t)] = L'[t_1] = L_S[G_1(t_1)] = L_S[t_1] = L_S[f(t)].$$

Consequently, we can use Lemma A.1 with $M = L_S$, $N = L'$, and $t_M = t_N = t$, to infer that L' is a model of $S\{t\}$ (clearly L_S is a model of $S\{t\}$).

Now if $t \in C(t_1)$, we have that

$$L'[f(t)] = L_S[G_1(f(t))] = L_S[f(G_1(t))]$$

and by definition

$$L'[t] = L_S[G_1(t)].$$

Again, we can use Lemma A.1 with $M = L_S$, $N = L'$, $t_M = G_1(t)$, and $t_N = t$, to infer that L' is a model of $S\{t\}$ (clearly L_S is a model of $S\{G_1(t)\}$).