Undecidable Optimization Problems for Database Logic Programs

HAIM GAIFMAN

Hebrew University of Jerusalem, Jerusalem, Israel

HARRY MAIRSON

Brandeis University, Waltham, Massachusetts

YEHOSHUA SAGIV

Hebrew University of Jerusalem, Jerusalem, Israel

AND

MOSHE Y. VARDI

IBM Almaden Research Center, San Jose, California

Abstract. *Datalog* is the language of logic programs without function symbols. It is used as a database query language. If it is possible to eliminate recursion from a Datalog program P, then P is said to be *bounded*. It is shown that the problem of deciding whether a given Datalog program is bounded is undecidable, even for *linear* programs (i.e., programs in which each rule contains at most one occurrence of a recursive predicate). It is then shown that every semantic property of Datalog programs is undecidable if it is stable, is strongly nontrivial, and contains

An earlier version of this work appeared under the same title in the *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (Ithaca, N.Y.). IEEE, New York, 1987, pp. 106–115.

Most of the research reported here was done while H. Gaifman was visiting the AI Center of SRI International whose support he wishes to acknowledge. He also wishes to thank IBM Watson Research Center and IBM Almaden Research Center for support in the summer of 1989, when the concluding work on this paper was done.

The research reported here was done partly while H. Mairson was at the Computer Science Department of Stanford University and was supported by the Office of Naval Research (ONR) contract N00014-85-C-0731 and partly while he was at the Programming Research Group of Oxford University.

The research reported here was done while Y. Sagiv was visiting the Computer Science Department of Stanford University and was supported by a grant of AT & T Foundation, a grant of IBM Corporation and the National Science Foundation (NSF) grant IST 84-12791.

Authors' addresses: H. Gaifman and Y. Sagiv, Hebrew University, Jerusalem 91904, Israel; H. Mairson, Department of Computer Science, Brandeis University, Waltham, MA 02254; M. Y. Vardi, IBM Almaden Research Center, K53-802, 650 Harry Road, San Jose, CA 95120-6099.

boundedness. In particular, the property of being first-order is undecidable and (assuming that PTIME is different from LOGSPACE and from NC) the same holds for the property of being equivalent to a linear program and for the properties of being in LOGSPACE and of being in NC

Categories and Subject Descriptors: D.3.4 **[Programming Languages]**: Processors—*optimization*; F.4.1 **[Mathematical Logic and Formal Languages]**: Mathematical Logic—*logic programming*; H.2.3 **[Database Management]**: Languages—*query languages*

General Terms: Languages, Theory

Additional Key Words and Phrases: Boundedness, database, Datalog, Logic Program, optimization, query language, recursion, undecidability

1. *Introduction*

It has been recognized for some time that first-order database query languages are lacking in expressive power [3, 16, 42]. Since then many higher-order query languages have been investigated [6–8, 20, 40]. A language that has received considerable attention recently is *Datalog*, the language of logic programs (known also as Horn-clause programs) without function symbols [4, 9, 16, 17, 37], which is essentially a fragment of fixpoint logic [9, 28]. A canonical example of Datalog is the following program that computes transitive closure, where we think of the database as a directed graph.

$$path(X,Y) :- edge(X,Y).$$
$$path(X,Y) :- path(X,Z), path(Z,Y).$$

In this example, we take *edge* to be an *extensional database* (*EDB*) *predicate*, that is, representing basic facts stored in the database. For example, $edge(1,5)$ is an EDB fact stating that there is an edge between vertices 1 and 5. The *intensional database* (*IDB*) *predicate path* represents facts deduced from the database via the logic program above: the first rule says every directed edge forms a path, and the second rule tells how paths can be joined together. We can now query, for instance, $path(1,7)$ or $path(2,V)$ to determine, respectively, whether there is a path from vertex 1 to vertex 7, or what vertices V are connected to vertex 2 by a path.

Recent works have addressed the problems of finding efficient evaluation methods for Datalog queries (Bancilhon and Ramakrishnan [4] provides a good survey on this topic) and developing optimization techniques for Datalog [5, 21, 23, 30, 31]. By "efficient," we refer to *data complexity*, where cost of query operations is measured in terms of the size of the database. A typical approach to the problem of efficient evaluation involves identifying "nice" properties of Datalog programs that facilitate efficient computation of programs with these properties. For example, Ullman and Van Gelder [39] have identified the *polynomial fringe property* of Datalog programs, where every proof involves a number of EDB facts (at the "fringe" of the proof tree) at most polynomial in the size of the database. Although the complexity of evaluating arbitrary Datalog programs can be *PTIME-complete* [20, 40], Ullman and Van Gelder have shown that the complexity of Datalog programs with the polynomial fringe property is in *NC*, that is, all facts can be deduced in parallel time polynomial in the logarithm of the size of the database, given a number of processors polynomial in the size of the database.

At first, the problem of optimizing Datalog queries does not seem to be too difficult, since every rule in a Datalog program can be viewed as a *conjunctive*

query. Conjunctive queries constitute a fragment of the class of first-order queries for which the optimization problem is completely solved [2, 10]. Unfortunately, it is the recursive application of the rules that makes Datalog queries hard to evaluate. Let us consider some examples of how one might go about optimizing Datalog queries.

One might attempt to simplify a Datalog program by eliminating unnecessary variables or predicates. As an illustration, Naughton has given the following illuminating example [30]:

$$buys(X,Y) :- cheap(Y), likes(X,Y),$$
$$buys(X,Y) :- cheap(Y), knows(X,Z), buys(Z,Y).$$

In this Datalog program, $cheap(Y)$ can be eliminated from the second rule, and the resultant program can make the same deductions as the previous one. On the other hand, if $cheap(Y)$ is replaced throughout the program by $rich(X)$, then $rich(X)$ cannot be eliminated from the second rule.

In the last two examples, *path* and *buys* are recursive predicates. Sometimes it is possible to change occurrences of recursive predicates to nonrecursive ones. For example, in the above transitive closure program, $path(Z,Y)$ can be changed to $edge(Z,Y)$. Sometimes *all* recursions can be eliminated:

$$buys(X,Y) :- likes(X,Y);$$
$$buys(X,Y) :- trendy(X), buys(Z,Y).$$

In this example, $buys(Z,Y)$ can be changed to $likes(Z,Y)$ in the body of the second rule [29]. Note that when a program is without recursion then, for every database D, *every* fact derived by the program from D can be derived in constant time, independent of D. Hence, the problem of finding when recursion is eliminable (i.e., when an equivalent recursion-free program exists) is of considerable interest.

It can be shown that the recursion can be eliminated iff there is a fixed upper bound on the number of iterations (in a bottom-up execution) that are needed to answer queries. When this is the case, we say that the program is *bounded*. This notion has been introduced in the context of the universal relation database model [27] and has been studied since that time by several researchers [12, 21, 22, 29, 32, 34].

The above examples give a glimmer of hope that a research intended to develop general optimization techniques along these lines might be fruitful, but the purpose of this paper is in part to quash such optimism. In particular, we show that the problem of deciding whether a Datalog program is bounded, that is, whether it is possible to have all its recursions removed, is undecidable. We show boundedness to be undecidable even for programs containing only a single IDB predicate which are moreover *linear* (i.e., the body of every rule contains at most one occurrence of an IDB predicate).

Furthermore, we prove that a strong version of boundedness, where we view a Datalog program as having initial facts (stored in the database) for all predicates, rather than just for EDB predicates, is also undecidable.

We also show that boundedness is undecidable for linear programs with one monadic IDB predicate, provided that we include $\neq$ (interpreted in the standard way) among the EDB predicates. This is interesting in view of the fact that for programs (without $\neq$) containing only monadic IDB predicates boundedness is decidable [11].

Using the proof technique of the first result, we obtain a Rice-style theorem by showing that every strongly nontrivial, stable, and semantic property of Datalog programs that is always true for bounded programs must be undecidable.[1] Our results show that the syntactical simplicity of Datalog can be quite misleading, in the sense that the problem of optimizing Datalog queries can be just as hard as optimizing programs in any general-purpose programming language.

The rest of the paper is organized as follows: The basic concepts and the undecidability of boundedness are proved in Section 2. In Section 3, we prove the undecidability of the strong boundedness. Section 4 treats the case of a monadic IDB predicate with $\neq$, and Section 5 treats the general undecidability of properties satisfying the above-mentioned conditions. We conclude with a summary and survey of open problems in Section 6.

2. *Undecidability of Boundedness*

2.1 PRELIMINARIES. A *Datalog program* is a collection of function-free Horn clauses, that is, clauses of the form:

$$H :- B_1, \ldots, B_n,$$

where H and $B_1, \ldots, B_n$ are atomic formulas, also called *atoms*. In Datalog, atoms are simply first-order predicates over variables. We refer to the left- and right-hand sides of a clause as its *head* and *body*. A clause is logically interpreted as the universal closure of the implication $B_1 \wedge \cdots \wedge B_n \rightarrow H$. Our undecidability results will clearly extend to the wider class of programs where constants are allowed.

Predicates are divided into *EDB* ("*extensional database*") *predicates* and *IDB* ("*intensional database*") *predicates*. The former cannot occur in the heads of clauses, so that ground instances of EDB predicates are given explicitly (extensionally) by the database, while ground instances of IDB predicates may in addition be *implied* by the query program. If R is a predicate, then an R *fact*, or a *fact about* R, is a ground atom whose predicate is R. Ground atoms are atomic formulas without variables. A *database* D is a finite collection of facts about the EDB predicates.

The same program can be applied to different databases. The application of P to D consists in running $P \cup D$ as a logic program. In this context, we consider bottom-up executions. For any set of facts S, let $P(S)$ be the set of all facts derivable from S by an application of some rule of P:

> $F \in P(S)$ iff either $F \in S$, or for some clause $H :- B_1, \ldots, B_n$ in P and some facts $B'_1, \ldots, B'_n$ in S, there exists a substitution of variables by constants which maps each B_i to B'_i and H to F.

[1] A property is strongly nontrivial, if, when relativized to the class of *ordered databases*, it is neither universal nor empty. It is stable if any program that satisfies it in the class of all databases satisfies it in the class of ordered databases. It is semantic if, with respect to each class of databases, it is preserved under program equivalence. "Undecidable" means that it is undecidable whether a program has the property in the class of all databases. We use here a notion of property that is relativized to classes of databases, that is, a program may or may not have the property relative to some class of databases. The full definitions are given in Section 5.

Let $P^0(D) = D$ and $P^{i+1}(D) = P(P^i(D))$. Then $P^i(D)$ consists of all facts derivable from D by at most i iterations of P. Each member of $P^i(D)$ has an associated *proof tree*, constructed in the obvious way. If f is a fact in $P^i(D) - P^{i-1}(D)$, then we say that its *proof height* is i. Let

$$P^\infty(D) = \bigcup_{i \geq 0} P^i(D).$$

Evidently $P^\infty(D)$ is the least fixpoint of P on D, that is the least fixpoint of P containing D as a subset.

If we put $P' = P \cup D$, then $P(D)$ is what, in the context of logic programming, is denoted as $T_{P'}(\varnothing)$ and $P^\infty(D)$ is $T_{P'} \uparrow \omega$ [26]. The Herbrand universe is in our case simply the set of all constants in D.

Since the Herbrand universe is infinite, there is some k such that $P^k(D) = P^\infty(D)$. This k depends, as a rule, on D. If there exists a k such that the equality holds for *all* databases, then we say that P is *bounded.* More generally, if there exists a k such that the equality holds for all members of $\mathscr{C}$, where $\mathscr{C}$ is some class of databases, we say that *P is bounded in $\mathscr{C}$.*

Usually, a particular IDB predicate is designated as a *goal predicate.* The *answer* of P on D consists of all facts about the goal predicate that are in $P^\infty(D)$. A program is *bounded on the goal predicate* if for some fixed k, independent of D, the answer is contained in $P^k(D)$. The boundedness of the program implies its boundedness on the goal predicate, but not vice versa. The two notions coincide if there is only one IDB predicate. Since our undecidability results are proved for programs with one IDB predicate, the distinction between the two types of boundedness does not matter.

We prove that boundedness is undecidable, by constructing a reduction from the halting problem for 2-counter machines. We spell out the proof in detail, because later we have to modify it to prove a similar result about *strong* boundedness, where the database may be initialized with arbitrary IDB facts.

A 2-counter machine (2CM) is a finite-state deterministic machine with two nonnegative counters. Machine configurations consist of the state of the finite control and the states of the counters, where each counter is either *empty* (i.e., equal to 0, a situation denoted by $=$) or is *nonempty* (i.e., greater than 0, a situation denoted by $>$). The major component of the machine is the transition function which determines the changes in machine configurations. If Σ is the finite set of states, then the transition function δ can be characterized as

$$\delta: \Sigma \times \{=, >\} \times \{=, >\} \to \Sigma \times \{pop, push\} \times \{pop, push\},$$

where, for example, $\delta(a, >, =) = (b, pop, push)$ means: *if in state* a *with counter* 1 *greater than zero and counter* 2 *equal to zero, then shift into state* b, *subtracting* 1 *from counter* 1 *and adding* 1 *to counter* 2. Initially, the 2CM M is in a distinguished *initial state*; the input is the initial value of the counters. M *halts* if it reaches a distinguished *halting state*. M is said to *diverge* if it does not halt; in this case, the computation is infinite, since by assumption there are transitions from all states except the halting state. The halting problem for 2-counter machines is: given a 2CM M, to decide whether M halts or diverges.

It is well known that 2-counter machines are sufficiently powerful to simulate any Turing machine; for details, see [20]. Hence, the halting problem for 2CM's is r.e. (recursively enumerable) complete. Without loss of generality, we restrict ourselves to consideration of the halting problem for 2CMs from an

initial state where counters are set to zero; we imagine the machines to be preprogrammed via their states with input. Given such a 2CM M, we construct a Datalog program P that "simulates" M, such that M halts if and only if P is bounded.

By a *configuration* of M, we mean a 4-tuple (t, s, c_1, c_2), such that in the computation of M, the machine is, at time t, in state s and the values of the counters are c_1 and c_2. Without loss of generality, let the states of the 2CM be $0, 1, \ldots, h$ where 0 is the initial state and h is the halting state.

The simulating program P will have one IDB predicate $CN(T, S, C_1, C_2)$ representing the configurations of M, where T is the time, S is the state and C_1 and C_2 are the counters. Since these four variables are integer-valued, the constants of the database must serve in some sense as integers. For this purpose, we assume two EDB predicates $zero(X)$ and $succ(X, Y)$. We think of the database as representing a finite initial segment of the natural numbers, with $zero(X)$ interpreted as "$X = 0$" and $succ(X, Y)$ interpreted as "$Y = X + 1$". There is no guarantee that this is a true representation, since the database is arbitrary. Our "simulating" program P will be sufficiently insensitive to the peculiarites of the database, so that the halting problem will reduce to the boundedness of P.

The program P has three types of rules: one *initialization rule* that simulates the initial configuration of M, several *transition rules* that simulate δ, and a *halting rule* that guarantees that when M halts (i.e., when a fact $CN(t, h, c_1, c_2)$ is derived), then all ground instances of $CN(X, Y, U, V)$ are obtained in the next iteration. Obviously, once CN is "saturated" in this way, we reach the fixpoint.

We now give in detail the rules of the query program. We use uppercase letters $X, X_1, T, S, C, \ldots$ for variables and lowercase letters $t, s, c, c_1, \ldots$ for individual constants.

Initialization. This rule derives the initial configuration of M. It says that any 4-tuple in which all arguments are zero is in CN:

$$CN(Z, Z, Z, Z) :- \text{zero}(Z).$$

Transitions. For every transition rule of the 2CM, we construct a Horn clause relating IDB facts $CN(T, S, C_1, C_2)$ and $CN(T', S', C_1', C_2')$, representing configurations of the 2CM immediately before and after a transition described by the rule.

For example, given the transition rule $\delta(j, >, =) = (j', pop, push)$, we construct the corresponding Horn clause:

$$\begin{aligned} CN(T', S', C_1', C_2') :- & \; CN(T, S, C_1, C_2), \\ & succ(T, T'), \\ & S = j, S' = j', \\ & succ(X, C_1), zero(C_2), \\ & succ(C_1', C_1), succ(C_2, C_2'). \end{aligned}$$

Observe how the Horn clause codes the preconditions given by the transition rule, as well as the action of the machine transition. To code "C_1 is greater than zero," we write $succ(X, C_1)$, that is, C_1 is the successor of some X. To

code that C_2 is zero, we write $zero(C_2)$. The incrementing and decrementing of the counters is represented by obvious use of the *succ* relation. Finally, we code the finite states ("the machine is in state j") by the following sequence of EDB atoms in the body of the clause, where Z and the A_t are additional distinct variables:

$$zero(Z), succ(Z, A_1), succ(A_1, A_1), \ldots, succ(A_{j-1}, S),$$

We abbreviate this expression as $S = j$.

Halting. We add the single rule

$$CN(A, B, C, D) :- CN(T, S, C_1, C_2), S = h.$$

Since A, B, C, D do not appear in the body of the rule, every atom $CN(a, b, c, d)$ is derivable by a single application of this rule, if and when the rule is applicable.

Before proving the properties of the above reduction, we need some definitions. For $k = 0, 1, \ldots,$ let D_k be the database whose set of elements is $\{0, 1, \ldots, k\}$ with the standard interpretation of zero and succ, that is, the facts consist exactly of $zero(0)$ and of $succ(i, i + 1)$, $0 \leq i < k$. Databases of the form D_k are referred to as *standard.*

A *chain (of length $k + 1$)* in a database D is any sequence $x_0, \ldots, x_k$ of elements in D such that $succ(x_t, x_{t+1})$ is a fact in D, for all $0 \leq i < k$. We refer to the chain as *x_0-based*, and call it *standard* if $zero(x_0)$ is a fact in D. Otherwise, we say that the chain is *nonstandard.* Note that the x_t need not be distinct. There can be also additional *succ* facts over the x_t: the chain need not contain all EDB relations over its elements.

LEMMA 2.1. *If $k \geq t, h$ and if M does not halt in time $t \geq 0$, then (t, s, c_1, c_2) is a configuration of M iff $CN(t, s, c_1, c_2) \in P^{t+1}(D_k) - P^t(D_k)$.*

PROOF. The proof is by induction on t. P simulates exactly the execution of M, since the database is standard. The inequalities $k \geq t, h$ guarantee that we have enough integers in the database for the time points and for encoding the states of M. □

LEMMA 2.2. *If (t, s, c_1, c_2) is a configuration of M and D has a zero-based chain of length $k + 1$, where $k \geq t, h$, then $CN(t, s, c_1, c_2) \in P^{t+1}(D)$.*

PROOF. The proof is by induction on t. We simulate the computation of M using the members $x_0, \ldots, x_k$ of a *zero*-based chain in the role of integers. The fact that we can have repetitions (i.e., the same member playing the role of more than one integer), or additional nonstandard facts, does not matter as far as deriving $CN(t, s, c_1, c_2)$ is concerned. (Nonstandard chains may, of course, enable the derivation of additional configurations not realized by M.) □

LEMMA 2.3. *If D does not contain any zero-based chain of length $k + 1$, then $P^\infty(D) = P^{k+1}(D)$.*

PROOF. All bottom up derivations of P must start by using the initialization rule. Then, as long as the halting rule is not applied, each step consists of an application of some transition rule. By checking the rules, one easily sees that in a sequence of $n + 1$ steps (including the initialization) we derive a sequence of *CN*-facts in which the "time points" form a *zero*-based chain of length

$n + 1$. By our assumption on D, we must have $n < k$. Hence, either we can derive a halting configuration in $< k + 1$ steps and then every CN-fact is derivable in one additional step, or $P^{k+1}(D) = P^k(D)$. In either case, $P^{k+1}(D)$ constitutes the fixpoint. □

LEMMA 2.4. *M halts iff P is bounded.*

PROOF. For the "if" direction, assume that M diverges. Then by Lemma 2.1, for all standard databases D_k with $k \geq h$, the sequence

$$P^0(D_k), P^1(D_k), \ldots, P^k(D_k)$$

is strictly ascending. Hence, P is unbounded.

For the "only if" direction, assume that M halts in, say t steps and let $m = max(t, h + 1)$. Let D be any database. If D contains a *zero*-based chain of length m, then by Lemma 2.2, $P^{t+1}(D)$ contains a fact of the form $CN(t, h, c_1, c_2)$ and, via the halting rule, $P^{t+2}(D)$ contains all CN-facts over the elements of D. If D contains no *zero*-based chain of length m, then by Lemma 2.3, $P^m(D)$ is already the fixpoint. Thus, for all databases, $P^k(D)$ is the fixpoint, where $k = max(t + 2, h + 1)$ □

As a corollary, we get:

THEOREM 2.5. *Boundedness is r.e.-complete.*

PROOF. The reduction above shows that boundedness is r.e.-hard. We now argue that boundedness is recursively enumerable. It is not difficult to show that, for any IDB predicate $R(X_1, \ldots, X_n)$, there is a recursive sequence

$$\varphi_1^R, \varphi_2^R, \ldots, \varphi_m^R, \ldots$$

of existential positive first order formulas in the EDB predicates, having $X_1, \ldots, X_n$ as free variables, such that the following holds:

—φ_m^R logically implies φ_{m+1}^R;
—For every database D, we have $R(a_1, \ldots, a_n) \in P^\infty(D)$ iff, for some m, $\varphi_m^R(a_1, \ldots, a_n)$ holds in D;
—For every database D, if, for every IDB predicate R, φ_m^R and φ_{m+1}^R define in D the same relation, then $P^\infty(D) = P^m(D)$.

This implies that P is bounded iff there exist an m such that, for every IDB predicate R, we have that φ_{m+1}^R implies φ_m^R in all finite structures. Observe that whether one existential formula implies another in all finite structures is decidable [14]. Hence, boundedness is r.e.-complete. □

Remarks

(1) The programs used in the above reduction are *linear*, that is, at most one IDB atom occurs on the right-hand side of each rule. Linear programs are in general more benign than arbitrary programs. For example, query evaluation for arbitrary programs is complete for PTIME [40], while it is in NC for linear programs [39].
(2) In the halting rule, the head variables do not occur in the body. Such rules are considered to be undesirable in real database programs (cf. [38]). We can include these variables in the body by insisting that they appear in some EDB fact of the database. This means that we replace the halting

rule by the finitely many rules obtained by adding, for each head variable X not occurring in the body, atoms such as $zero(X)$, $succ(X, X')$, or $succ(X', X)$ where X' is a new variable different from all the other variables in the clause. For example, one of these rules will be

$$CN(A, B, C, D) :- CN(T, S, C_1, C_2), S = h, succ(A, A'), succ(B', B), succ(C', C), succ(D, D').$$

Evidently, for every given database (i.e., set of EDB facts), these rules will have together the effect of the halting rule. Such rules, however, combine pieces of unrelated data; they correspond to the algebraic operation of *cross product*, an unnatural operation in a database context. It is reasonable to require that all rules be *connected*, where this is defined as follows: Construct a graph over all the variables in the rule by joining every two variables occurring in the same atom in the body by an edge, then this graph is connected. Clearly, the halting rule in our reduction (as well as the alternative rules suggested above) is not connected.

We can make our rules connected by increasing the arity of the IDB predicate. See Appendix A. We do not know if such an increase is necessary.

(3) Whether some fixed universal Turing machine halts on a given input is obviously r.e.-complete. For the corresponding 2CM, we see that its halting problem, given an initialization of one of its counters, is r.e.-complete. The counter initialization can be simulated by an initialization rule, that is to say, a rule whose body contains only EDB predicates. Hence, we get a fixed linear program P with one IDB predicate such that the problem whether, given an initialization rule r, $P \cup \{r\}$ is bounded is r.e.-complete.

(4) Our undecidability result (first given in [15]) is for linear programs with a one 4-ary IDB predicate. It can be sharpened to linear programs with one binary IDB predicate [41]. The proof in [41] uses the same idea to be used in Section 4: the database is supposed to encode a computation and the program "checks" its correctness. But it uses Turing machines instead of 2CMs. As far as arity is concerned, 2 is the best possible, in view of the result that for monadic IDB predicates boundedness is decidable [11].

(5) The class of Datalog programs for which the decision problem is posed can be restricted by imposing a bound on the number of recursive rules. Remark 3 shows that we get undecidability within some fixed bound. The bound, however, is large: We have to simulate a universal 2CM and a rough estimate yields on the order of several hundred rules. This bound can be significantly sharpened. In particular, Abiteboul [1] has shown that boundedness is undecidable for the class of *sirups*, that is, programs with a single recursive rule, all other rules being initialization rules (cf. [12]). On the other hand, Vardi [41] has shown that for linear sirups with a binary IDB predicate boundedness is decidable (and in fact NP-complete). Furthermore, it can be shown that that boundedness is undecidable for linear programs with two recursive rules [18].

(6) Although boundedness is Σ_1^0 complete, it can be shown, using similar techniques, that for programs containing more than one IDB predicate the boundedness of the program on the goal predicate is Σ_2^0 complete.

3. *Undecidability of Strong Boundedness*

So far, we had a clear separation between EDB predicates, viewed as input, and IDB predicates, viewed as output. Although this distinction is appropriate when we consider Datalog programs as queries, it is not quite appropriate when we consider Datalog programs as the intensional part of a *deductive database* [16]. In this application, the input may contain explicit facts about the IDB predicates as well; the division is therefore between extensional facts (those given as input) and intensional facts (those derived by the program). Also, if we want to do *local* optimization, that is, we want to consider programs that are part of bigger programs, then IDB-facts constitute possible inputs. The division between the two kinds of predicates is, however, retained in that EDB predicates cannot occur in the heads of rules.

This motivates the following framework: An input to a program P is a pair (D, I), where D is a database (finite set of EDB facts over a set of constant symbols) and I is a finite set of IDB facts over the elements of D. I is the set of *initial IDB facts*, or the *initial assignment*. $P^n(D, I)$ is defined in the obvious way as $P^n(D \cup I)$. A program P is *strongly bounded* if, for some fixed k, $P^k(D, I) = P^\infty(D, I)$, for all inputs (D, I). Strong boundedness implies boundedness, but not vice versa. The two decidability questions are therefore incomparable. There is, however, an easy reduction of strong boundedness to boundedness [32].

PROPOSITION 3.1. *For any program P, let $\tilde{P}$ be obtained by adding, for every IDB predicate R, a distinct new EDB predicate, $\tilde{R}$, of the same arity and the rule*

$$R(X, Y, \dots) :- \tilde{R}(X, Y, \dots).$$

Then P is strongly bounded iff $\tilde{P}$ is bounded.

PROOF. Observe that each input (D, I) for P determines a unique database D' for the EDB vocabulary of $\tilde{P}$, obtained by substituting $\tilde{R}$ for R in $D \cup I$. The correspondence is one-to-one and onto, so that for any IDB predicate R, we have

$$P^n(D, I) \cap R \subseteq \tilde{P}^{n+1}(D') \cap R \subseteq P^{n+1}(D, I) \cap R. \quad \square$$

The proposition shows that deciding boundedness is at least as difficult as deciding strong boundedness. The undecidability of strong boundedness is therefore a stronger result. Indeed, the proof of this last result uses a variant of the previous construction, with an additional complication, which increases the arity of the recursive predicate.

As before, given a 2CM M, we want to construct a program P, such that P is strongly bounded iff M halts. Our previous construction fails here for the following reason:

Let M be a 2CM that halts when started in its initial configuration, but diverges when started from an inaccessible state s, looping in state s while increasing both counters at each transition. When we code M as a Datalog program, it is clearly bounded for any database of EDB facts. Suppose in addition that no *zero*-based chain is long enough to simulate the entire computation. Consider, however, what happens when the database contains an IDB fact $CN(x_0, \tilde{s}, x_0, x_0)$ where $\tilde{s}$ codes s (using a *zero*-based chain long

enough to represent all the machine states), and $\{succ(x_1, x_{i+1}) | 0 \leq i \leq m\} \subseteq D$ forms a chain that is *not zero*-based. We can then simulate M for m steps; barring additional EDB facts about the x_i, any proof of $CN(x_m, \tilde{s}, x_m, x_m)$ will require a proof tree of height m. Since m depends on the database, the Datalog program is not bounded, even though M halts when started from the initial configuration.

This example naturally suggests that P should be constructed so that every chain in the database can be used to simulate the standard computation of M, where we allow *any* x to play the role of "zero". However, once we have chosen a "zero," we must keep it; else, we shall not be able to tell when a counter is empty. Hence, we add an additional argument to mark the chosen "zero", We thus try a 5-ary predicate $CN(Z, T, S, C_1, C_2)$, which we are going to read as:

> With Z, the zero-point, the machine is at time T in state S with the counters C_1 and C_2.

We no longer need the predicate *zero*, since the zero-point can be used to define the states; $S = j$ ("the state is j") is now an abbreviation of

$$succ(Z, A_1), succ(A_1, A_2), \ldots, succ(A_{j-1}, S),$$

where Z is the variable occupying the zero-argument throughout the rule. That the counter C is empty is expressible by "$C = Z$"; we do not have explicit equality, but we get the same effect simply by substituting Z for C everywhere in the rule.

There remains however the problem of expressing the *nonemptiness* of a counter C. Our previous coding $succ(X, C)$ ("C is a successor of something"), will no longer work: being a successor does not imply being greater than our chosen zero-point. Our chosen "zero" can be a successor, even in a standard database. The difficulty here concerns correct simulations in standard databases. If $succ(X, C)$ is used to express the nonemptiness of C, then we might also generate "incorrect configurations". These additional configurations might lead to a halting configuration, when in fact M diverges.

In order to express the condition that a counter is greater than the zero-point correctly, we generate recursively all the "greater-than-zero" elements. This requires an increase in the arity of CN. The additional variable ranges over all members that are greater than our chosen zero, that is, are reachable from it by a chain of length greater than 1. To say that the counter is not empty, we simply equate it with that variable. Since we have to state the nonemptiness conditions for two counters, we need to additional variables, one for each counter. We thus end up with a 7-ary predicate:

$$CN(Z, G_1, G_2, T, S, C_1, C_2)$$

to be read as

> When Z is the zero-point, G_1 and G_2 are strictly positive and, at time T, the machine is in state S and the counters C_1 and C_2.

The new rules are

Initialization

$$CN(Z, G, G, Z, Z, Z, Z) :- succ(Z, G).$$

Greater-than-zero Rules. These rules are needed to generate and move freely among the "greater than zero" elements. For each counter, we have an *increase rule* and a *reset rule*. Here are the rules for the first counter:
G_1-*increase*

$$CN(Z, G_1', G_2, T', S, C_1, C_2) :- CN(Z, G_1, G_2, T, S, C_1, C_2), \\ succ(G_1, G_1'), succ(T, T').$$

G_1-*reset*

$$CN(Z, G_1, G_2, T', S, C_1, C_2) :- CN(Z, G_1', G_2, T, S, C_1, C_2), \\ succ(Z, G_1), succ(T, T').$$

The *increase* rule enables us to "increase" the variable in question, while keeping the other variables fixed. The *reset* rule enables us to reset it to the successor of the zero-point. Obviously, these rules make it possible to derive from $CN(z, g_1, g_2, t, s, c_1, c_2)$ another fact $CN(z, g_1', g_2', t', s, c_1, c_2)$ where g_1' and g_2' are reachable from z by successor chains of length greater than 1 (observe, however, that the "time" variable cannot be so easily controlled). The variables Z, G_1, and G_2 appear in the following transition rules as fixed parameters.

Transitions. These rules are constructed as before, except that the conditions for the emptiness and nonemptiness of a counter are expressed by substituting, respectively, Z or the "greater than zero" variable for the counter variable.

For example, the transition $\delta(j, >, =) = (j', pop, push)$, has the corresponding rule:

$$CN(Z, G_1, G_2, T', S', C_1, C_2) :- CN(Z, G_1, G_2, T, S, G_1, Z), \\ succ(T, T'), \\ S = j, S' = j', \\ succ(C_1, G_1), succ(Z, C_2).$$

and the transition $\delta(j, >, >) = (j', push, push)$ has the corresponding rule:

$$CN(Z, G_1, G_2, T', S', C_1, C_2) :- CN(Z, G_1, G_2, T, S, G_1, G_2), \\ succ(T, T') \\ S = j, S' = j', \\ succ(G_1, C_1), succ(G_2, G_2).$$

Note that *two* new variables (i.e., G_1 and G_2) are needed in order to express the fact that both counters are nonzero.

Halting

$$CN(A, B, C, D, E, F, G) :- CN(Z, G_1, G_2, T, S, C_1, C_2), S = h.$$

In what follows, D_k is, as before, the standard database with $k + 1$ members and $0, 1, \ldots, h$ are all the states of M.

LEMMA 3.2. *For each nonnegative t, there exists a number $n_t > t$ such that if $k \geq n_t, h$ and M does not halt in less than t steps, then (t, s, c_1, c_2) is a configuration in the standard computation of M iff $CN(0, 1, 1, n_t - 1, s, c_1, c_2) \in P^{n_t}(D_k) - P^{n_t - 1}(D_k)$.*

PROOF. Since the database is standard, the simulation of the computation of M yields the correct configurations. There are $t + 1$ different configurations in the computation up to (t, s, c_1, c_2). In order to derive a configuration, all preceding ones must be derived. Since $k \geq n_t, h$, such a derivation exists. Note that n_t is the smallest number of derivation steps and the minimal height of a proof tree, because the program is linear. Observe that, in general, we can have $n_t > t + 1$ because we may need to apply the *greater-than-zero* rules that have no analogue in the computation of M. While in the proof of Section 2, each application of a rule generated a new machine configuration, the *G_t-increase* and *G_t-reset* rules in this proof do not correspond to any machine state transition. As a consequence, the relation between *time* and *proof height* does not hold linearly, though as we will see, it does hold quadratically. This latter relation is sufficient to imply undecidability of boundedness. □

LEMMA 3.3. *Let (t, s, c_1, c_2) be a configuration of M's standard computation. Then, if $x_0, x_1, \ldots, x_k$ is a chain in the database D such that $k \geq t, h$, then $CN(x_0, x_1, x_1, t, s, c_1, c_2) \in P^{n_t}(D)$, where n_t is the number from Lemma 3.2.*

PROOF. We use the chain, with x_0 as the zero-point in order to simulate M's standard computation. At most, n_t simulation steps will be needed to get the tth configuration (if D is nonstandard we might do with less). □

LEMMA 3.4. *If the database D does not contain a chain of length $k + 1$, then for all initial assignments I, $P^{\infty}(D \cup I) = P^k(D \cup I)$.*

PROOF. Observe simply that all rules except the initialization rule increase the time variable at every step. □

LEMMA 3.5. *M halts iff P is strongly bounded.*

PROOF. Exactly like the proof of the analogous Lemma 2.4. □

From this lemma, we derive:

THEOREM 3.6. *Strong boundedness is r.e.-complete.*

PROOF. That strong boundedness is r.e.-hard follows in the obvious way from Lemma 3.5. That it is r.e. follows from Theorem 2.5 and Proposition 3.1. □

It is interesting to note here that for *typed* rules with a single predicate, strong boundedness is decidable [34].[2]

[2] In such rules, different columns of each predicate range over disjoint domains. For example, the rule

$$p(X, Y) :- p(X, Z), p(Z, Y)$$

is not typed, while the rule

$$p(X, Y, Z) :- p(X, Y, Z'), p(X, Y', Z)$$

is typed.

The predicate CN used in the reduction of the halting problem is 7-ary. This result can be sharpened by showing how the arity can be reduced to 5.

THEOREM 3.7. *Strong boundedness for programs with 5-ary IDB predicates is r.e.-complete.*

PROOF. See Appendix B. □

For monadic IDB predicates, strong boundedness is decidable. This follows from the decidability of monadic boundedness [12] by Lemma 3.1. In a recent paper [19], strong boundedness has been further shown to be undecidable for arity 3 allowing binary rules, and arity 5 with linear rules. The question of strong boundedness for arity 2, as well as arity 3 with linear rules, remains open.

4. *Unary Recursion*

As remarked above, boundedness for programs with only monadic IDB predicates is decidable [11]. However, if we include among the EDB predicates the *inequality* predicate, $\neq$ (to be interpreted in the standard way), then even in the monadic case boundedness is undecidable. This will be shown in this section. In fact, we show undecidability for linear programs with one monadic IDB predicate and one recursive rule.

We call the extended language, obtained by adding $\neq$, Datalog$^{\neq}$. In Datalog$^{\neq}$, we allow for example rules such as

$$A(X,Y) :- B(X,Y), X \neq Y.$$

Note that adding equality ($=$) to the EDB predicates does not add expressive power, since every rule in which $X = Y$ occurs in the body can be rewritten in equivalent form without $=$, by substituting everywhere the same variable for X and Y.

It is known that Datalog$^{\neq}$ is strictly more expressive that Datalog [36]. From a practical point of view, however, Datalog$^{\neq}$ queries do not seem to be any harder to evaluate than Datalog programs. The reader is referred to [25] for a study of the expressive power of Datalog$^{\neq}$.

As before, we reduce the halting of a 2CM to boundedness, but we use the database in a different way. The idea of the previous reduction was to represent by the database a prefix of the natural numbers and by the IDB predicate—a computation of the 2CM. In the present reduction, we represent by the database both a prefix of the natural numbers and a computation of the 2CM. The unary IDB predicate is used only to check that the prefix and the computation are "correct". When either an "error" in the representation is discovered, or when the computation halts, the IDB predicate becomes "saturated". The IDB predicate is also used as a counter for time, so that a nonhalting 2CM will give rise to an unbounded program.

Our previous $CN(T, S, C_1, C_2)$ is now an EDB predicate. The other two EDB predicates are our previous $zero(X)$, $succ(X,Y)$. There is one unary IDB predicate $B(X)$.

The first three rules check the *zero* and *succ* predicates.

Check Zero

$$B(U) :- succ(X,Y), zero(Y)$$

(*i.e., if a zero-member is a successor than* $B(U)$ *holds for all values of* U.)

Check Successor

$$B(U) :- succ(X,Y), succ(X,Z), Y \neq Z$$
$$B(U) :- succ(Y,X), succ(Z,X), Y \neq Z$$

We now need rules to check the CN predicate.

Check Initialization. Here we check the correctness of the initial configuration: The state should not be different from the initial time point (recall that 0 is the initial state) and the same goes for each counter:

$$B(U) :- CN(T,S,C_1,C_2), zero(T), S \neq T$$
$$B(U) :- CN(T,S,C_1,C_2), zero(T), C_1 \neq T$$
$$B(U) :- CN(T,S,C_1,C_2), zero(T), C_2 \neq T$$

Check Transitions. Consider for example the transition $\delta(j, >, =) = (j', pop, push)$. Corresponding to it there are rules such as

$$\begin{aligned} B(U) :- & CN(T,S,C_1,C_2), CN(T',S',C_1',C_2'), \\ & succ(T,T'), \\ & S = j, zero(C_2), \\ & succ(X,C_1), \\ & S'' = j', S' \neq S'' \end{aligned}$$

The above rule checks whether the change of state is represented correctly by CN, that is, it "saturates" B if the resulting state (here S') differs from the correct one (here S''). Similar rules check the correct setting of the counters.

The next rule checks whether the halting state has been reached.

Halting

$$B(U) :- CN(T,S,C_1,C_2), S = h.$$

Finally, we have two rules that prevent the computation from being bounded as long as a halting state has not been reached. We simply increase B with every computation step. Thus, B acts like a counter for time.

Unboundedness

$$B(T) :- CN(T,S,C_1,C_2), zero(T)$$
$$B(T') :- B(T), CN(T,S,C_1,C_2), CN(T',S',C_1',C_2'), succ(T,T').$$

Note that the above program is linear and contains only *one* recursive rule, which is somewhat surprising in view of the indications that unboundedness for the class of Datalog programs with a single, linear recursive rule might be decidable (all the positive results in [21], [22], [29], [32], and [41] are for this class).

LEMMA 4.1. *If M diverges, then, for every sufficiently large k, there exists a database D and a fact f such that $f \in P^{k+1}(D) - P^k(D)$.*

PROOF. Let the database D include the constants $\{0, 1, \ldots, k\}$ with the standard interpretation:

—*zero*(0);
—For $0 \leq i < k$, *succ*$(i, i+1)$;
—For $0 \leq i \leq k$, the fact $CN(i, s_i, c_{1,i}, c_{2,i})$ that describes the correct configuration of M at time i.

Since M diverges, the halting rule is never used. Since *zero* and *succ* are standard and the facts for CN include only correct configurations, the rules for checking zero, successor, initialization, or transitions are never used. Clearly, $B(k) \in P^{k+1}(D) - P^k(D)$ □

LEMMA 4.2. *Assume that M halts in t steps. Let s be the number of states in M and let* $\tau = max(t, s)$. *Then for every database D, we have that* $P^\infty(D) = P^{\tau+1}(D)$.

PROOF. If the halting rule or one of the rules that check zero, successor, initialization, or transitions can be applied to D, then we can get every B-fact in one step. So assume that none of these rules is applicable. If $P^\infty(D) = P^\tau(D)$, then we are done. So consider, for contradiction, a fact f such that $f \in P^{\tau+1}(D) - P^\tau(D)$. The derivation of f only uses the initialization and unboundedness rules. It is easily seen that such a derivation, starting with the initialization rule: $B(T) :- CN(T, S, C_1, C_2)$, *zero*$(T)$, generates a sequence of facts $B(a_0), \ldots, B(a_\tau)$, where *zero*$(a_0)$, *succ*$(a_i, a_{i+1})$, for $(0 \leq i \leq \tau)$ are EDB facts, and $B(a_{i+1})$ is derived from $B(a_i)$ via the rule

$$B(T') :- B(T), CN(T, S, C_1, C_2), CN(T', S', C_1', C_2'), succ(T, T').$$

This means that, for each $0 < i \leq \tau$, there is a CN fact that is used, together with another CN fact representing a previous configuration, in deriving $B(a_i)$.

The sequence of these EDB facts must represent a correct computation of M. For if $CN(a_0, s_0, c_{1,0}, c_{2,0})$ does not represent the true initial configuration, then the check initialization rule would apply. Also, the sequence $a_0, \ldots, a_\tau$ must be standard, else a check-successor rule would apply. Finally, all the transitions between succeeding configurations must be correct. To see this, note that the chain $a_0, \ldots, a_\tau$ is long enough to encode all states of M; it then follows that if $CN(a_i, s, c_1, c_2)$ and $CN(a', s', c_1', c_2')$ are EDB facts, where $i < \tau$ and *succ*(a_i, a') is an EDB fact, and if $CN(a_i, s, c_1, c_2)$ represents a nonhalting configuration of M (where $a_0, \ldots, a_\tau$ represent the numbers), then $CN(a', s', c_1', c_2')$ must represent the true succeeding configuration, else a check-transition rule would apply.

Hence, $CN(a_\tau, s_\tau, c_{1,\tau}, c_{2,\tau})$ represents the τth configuration. But this must be the halting configuration, contradicting the assumption that the halting rule is not applicable. □

THEOREM 4.3. *Boundedness is r.e.-complete for unary linear Datalog*$^\neq$ *programs.*

PROOF. Lemmas 4.1 and 4.2 show that boundedness is r.e.-hard. Showing that boundedness is recursively enumerable is the same as in Theorem 2.5. □

In [18], it is further shown that strong boundedness is undecidable for unary Datalog$^\neq$ programs.

5. *A General Undecidability Result*

In the previous sections, we have focused on the boundedness problem. There are several other properties of Datalog programs that are relevant to their evaluation, and we now mention some of them.

Let P be a program with a goal predicate R, and let D be a database. Denote by *answer*(P, D) the answer of P on D, that is, the set of all R-facts in $P^\infty(D)$. A program P is *first order* if it is equivalent to a first-order query, that is, there is a first-order query Q, such that *answer*$(P, D) = Q(D)$ for every database D. First-order programs can be evaluated in LOGSPACE. A program P is *essentially linear* if it is equivalent to a linear program. Essentially linear programs can be evaluated in NC.

Of course, the complexity of evaluation is also a property of a program. A program P is in LOGSPACE if the set $\{(f, D) : f \in P(D)\}$ is in LOGSPACE. Similarly, a program P is in NC if the set $\{(f, D) : f \in P(D)\}$ is in NC.

We now prove a Rice-style theorem (cf. [33]) for Datalog programs. As a result, it follows that none of the above properties is decidable (assuming that LOGSPACE and NC are different from PTIME). We begin by motivating some definitions that do not all have precise analogues in the proof of Rice's Theorem, but are required for the proof of its Datalog equivalent.

Intuitively, whether a program has a given property depends on the class of databases we have in mind. For example, let P be a program that simulates a 2CM M, as is done in Section 2, except that the halting rule is omitted (the rule that puts every tuple in *CN*, once the halting state is reached). If we consider only standard interpretation of *succ* and *zero*, then P is bounded if and only if M halts. However, if *succ* and *zero* get a nonstandard interpretation, then P might be unbounded even if M halts. So if M halts, then P is bounded with respect to the class of databases in which *succ* and *zero* get a standard interpretation, but it might be unbounded with respect to the class of all databases. This example motivates the following set of definitions:

Definition 5.1. Let $\mathscr{P}$ be the class of programs and let $\mathscr{D}$ be the class of all databases. A *property* Π is a subset of $\mathscr{P} \times 2^{\mathscr{D}}$. If $(P, \Psi) \in \Pi$, then we say that program P has the property Π in, or with respect to, the class Ψ of databases.

Boundedness with respect to a given class of databases means that the program is bounded in the given class. We use *bounded* without qualification, to mean *bounded in the class of all databases*.

Definition 5.2. A property Π *contains boundedness* if every bounded program has the property Π with respect to the class of all databases.

Our undecidability theorem will depend essentially on the assumption that the property in question of Datalog programs contains boundedness. Consider the property of *emptiness*; a program P is empty with respect a class Ψ of databases if no goal fact is derived by P on any database in Ψ. This property satisfies all the conditions for our undecidability proof with the exception of containing boundedness: Emptiness clearly implies boundedness, but not the converse. Nevertheless, emptiness with respect to the class of all databases is decidable in linear time [35].

Definition 5.3. A property is *trivial with respect to a class* Ψ *of databases* if, with respect to Ψ, either all programs have the property, or none have the property.

Definition 5.4. A property Π is *semantic* if, in each class of databases, it is preserved under equivalence: if P and P' are equivalent in the class Ψ (i.e., compute the same goal relation in each member of Ψ), then P has the property Π with respect to Ψ iff P' does.

Observe that the properties of boundedness, first-order equivalence, and essential linearity mentioned above are semantic.

Definition 5.5. Consider the family of databases whose EDB-vocabulary includes three specific predicates: *zero*, *succ*, and *max* (and possibly others). A database of this family is *ordered* if these predicates get the standard interpretation. It means that in an ordered database, *succ* is interpreted by some successor relation on the domain of the database,[3] *zero* is interpreted by the minimal element of *succ*, and *max* is interpreted by the maximal element of *succ*.

Definition 5.6. A property Π is *stable* if each program that has the property with respect to the class of all databases also has the property with respect to the class of ordered databases.

For example, boundedness is a stable property (a bounded program is bounded with respect to every class and, in particular, in the class of ordered databases). However, the opposite property of unboundedness is not stable. It is easy to see that the properties of Datalog programs mentioned above are stable.

Definition 5.7. A property Π is *strongly nontrivial* if it is nontrivial with respect to the class of ordered databases.

The property of being first order is known to be strongly nontrivial [13]. All the other properties mentioned above are strongly nontrivial, assuming that LOGSPACE and NC are different from PTIME. We note that, for essential linearity, this follows from the fact that there are nonlinear programs that express PTIME-complete problems [40], whereas linear programs are in NC [39].

THEOREM 5.8. *Let* Π *be a property that is semantic, stable, strongly nontrivial, and contains boundedness. Then it is undecidable whether a query program has property* Π *with respect to the class of all databases.*

PROOF. Let P be a program that does not have the property Π with respect to the class of ordered databases. There is such a P, since our property is strongly nontrivial.

The idea of the proof is to construct a new program Q that combines P with a program that simulates a 2CM M. The construction is carried out by adding to each IDB predicate of P four arguments and adding to the body of each

[3]The *domain* of a database is the set of all constants appearing in EDB relations.

rule of P conjuncts that force a simulation of M in the four extra arguments. This is done so as to make the following two claims true:

(1) If M halts, then the new program Q is bounded; hence, it has the property Π, with respect the class of all databases.
(2) If M does not halt, then Q and P are equivalent on all ordered databases.

Now assume that M does not halt. Since Π is semantic, claim (2) implies that Q does not have property Π with respect to ordered databases. Hence, by stability, Q does not have the property with respect to the class of all databases. Together with claim (1), this shows that M halts iff Q has the property Π in the class of all databases. This gives the reduction that implies the theorem.

We now describe the construction and prove (1) and (2). First, consider a program Q obtained as follows:

Program Q has the same EDB predicates as P. For each IDB predicate p in P, program Q has a corresponding predicate p' with four additional arguments; these are used to simulate a 2CM M as in Section 2. Q has as well a goal predicate g, which is the same as that of P. For each rule of P, Q has one or more associated rules. An execution of any associated rule executes on the old arguments the original rule of P, while it simulates on the new arguments a transition of M. In addition, Q has rules that allow it to simulate on the new arguments the transitions of M, while leaving the old arguments fixed.

The construction of Q ensures that it can simulate P as long as it can simultaneously simulate a computation of M. Q has as well a rule that allows the derivation from any g'-fact a corresponding g-fact, simply by deleting the four additional arguments. Thus, Q generates the same g-facts as P, as long as the computation of M goes on. Finally, a set of *halting rules* guarantee that once a halting state is reached, all predicates in Q are "saturated" (i.e., include all tuples). The halting rules guarantee that if M halts, then Q is bounded, the argument being the same as that used in Section 2. Thus, we establish our first claim.

If M does not halt, then on an ordered database of size n, Q can simulate the first n steps of M's computation, provided that $n \geq s$ (where s is the number of states). Assuming this inequality, it follows that Q can also simulate any proof of a P-fact having height at most n. Therefore, Q will produce any g-fact obtainable by at most n iterations of P. This is not sufficient for the second claim, since there might be g-facts that need more than n iterations. However, the number of iterations needed to obtain the fixpoint of P is no more than n^k where k is the sum of all the arities of P's IDB predicates. This bound is established by observing that there are n^k possible IDB facts altogether and that the fixpoint is reached when an iteration does not produce a new IDB fact. So the remedy to our problem is to encode the time points not by the members of the ordered database but by the k-tuples, where the tuples are ordered lexicograpically. This means that instead of a single argument for encoding time, we have k new arguments. Similarly, we have k arguments for each of the counters.

We also have to take care of the restriction that $n \geq s$. This restriction excludes only finitely many databases. We can effectively determine (by running P) the g-facts produced in each of them. We add to Q a set of initialization rules, to be called *small database rules*, which derive the necessary

additional g-facts. Namely, for each database of size less than s, use distinct variables to represent the database elements and add any rule whose body represents the set of EDB facts and whose head represents a derivable g-fact.

We shall now describe Q in more detail. We start with the first variant in which the time and the counter values are represented by single arguments. The final variant that uses an encoding by tuples will be easily obtained from this.

We couple every initialization rule of P with the initialization rule that simulates the initial configuration of M. Let the initialization rule of P be

$$p(X_1, \ldots, X_n) :- \mathscr{A}_1, \ldots, \mathscr{A}_k, \tag{1}$$

where $\mathscr{A}_1, \ldots, \mathscr{A}_k$ are EDB atoms (i.e., with EDB predicates). The corresponding rule of Q is

$$p'(X_1, \ldots, X_n, Z, Z, Z, Z) :- \mathscr{A}_1, \ldots, \mathscr{A}_k, zero(Z), \tag{2}$$

where p' is the IDB predicate corresponding to p. The last four arguments of p' are the ones added, the first is the time argument, the second represents the state and the last two represent the counters.

Now consider a noninitialization rule

$$p(X_1, \ldots, X_n) :- \mathscr{A}_1, \ldots, \mathscr{A}_j, \ldots, \mathscr{A}_k, \tag{3}$$

where $\mathscr{A}_1, \ldots, \mathscr{A}_j$ are IDB atoms and the rest are EDB atoms. We couple this rule with each of the rules that simulate the transitions of M. The result will be of the form:

$$p'(X_1, \ldots, X_n, T', S', C_1', C_2') :- \mathscr{A}_1', \ldots, \mathscr{A}_j', \mathscr{A}_{j+1}, \ldots, \mathscr{A}_k, \mathscr{C}, \tag{4}$$

where each $\mathscr{A}_i'$ $(i = 1, \ldots, j)$ is obtained from $\mathscr{A}_i$ by replacing the IDB predicate, say q_i, by its corresponding q_i', keeping the same variables in the old positions and filling the new four arguments with T, S, C_1, C_2. The same variables appear in the new four positions in all the $\mathscr{A}_i'$ and none of them coincides with an old variable. $\mathscr{C}$ is the body of the transition rule, constructed as in Section 2. For example, if the transition is $\delta(j, >, =) = (j', pop, push)$, then $\mathscr{C}$ is the following body:

$$\begin{aligned} &succ(T, T') \\ &S = j, S' = j', \\ &succ(X, C_1), zero(C_2), \\ &succ(C_1', C_1), succ(C_2, C_2'). \end{aligned} \tag{5}$$

The old configuration is described by T, S, C_1, C_2; the new configuration is represented by the variables T', S', C_1', C_2' that appear in the head of rule (4).

The rules constructed so far are such that if, on some given database, Q generates the fact $p'(x_1, \ldots, x_n, t', s', c_1', c_2')$, then P generates the fact $p(x_1, \ldots, x_n)$. This is easily verified and will continue to hold with respect to the full set of rules, to be presently defined.

Now, in the case of a nonhalting M, we want also to have a converse inclusion for sufficiently large-ordered databases: If P generates in i iterations $p(x_1, \ldots, x_n)$ and the database size enables to simulate i steps in M's computa-

tion, then some fact $p'(x_1, \ldots, x_n, t', s', c'_1, c'_2)$ is generated by Q. The obvious way to produce this fact is to couple the generation of $p(x_1, \ldots, x_n)$ in P with a simulation of M. The rules included so far enable us to do so if P is linear, but not in general. The reason for this is that the last four coordinates are the same throughout the body of each rule of Q. Now, each step of Q couples an execution step of P with a transition of M, hence the last four arguments represent the lth configuration of M, where l is also the height of the proof via P. If in P a fact f is directly derived from some previously generated facts, say f_1, f_2, whose proofs have unequal heights, we will not be able to simulate this step by using the corresponding facts f'_1, f'_2, because they will differ on the time argument. In order to overcome this difficulty, we include rules that enable the computation of M to proceed, while the other arguments are kept fixed. This enables us to bring at each stage the last four arguments in all the needed IDB facts to the same maximal level.

We therefore add, for each IDB predicate p of P and for each transition of M the rule:

$$p'(X_1, \ldots, X_n, T', S', C'_1, C'_2) :- p'(X_1, \ldots, X_n\ T, S, C_1, C_2), \mathscr{C},$$

where, as before, $\mathscr{C}$ is the body of the transition rule.

In order to get via Q the "correct" facts in the goal without the auxiliary four arguments, we include a projection rule for g', where g is the common goal predicate:

$$g(X_1, \ldots, X_m) :- g'(X_1, \ldots, X_m, T, S, C_1, C_2).$$

Finally, we include halting rules which "saturate" all the predicates of Q, once a halting state is reached

$$q(X_1, \ldots, X_m) :- p'(Y_1, \ldots, Y_n, T, S, C_1, C_2), zero(Z), S = h.$$

Here, all variables are distinct, q ranges all over predicates of Q and p over all predicates of P.

To get the final variant, where time and counters are encoded by k-tuples (for some fixed sufficiently large k), we replace each of the time and the counter arguments by k arguments. The rules are obtained by replacing each variable occupying these arguments by the corresponding k variables and by replacing conditions of the form $zero(X)$, or $succ(X, Y)$, by conditions that define the zero and successor in the ordered set of k tuples. Now the successor in the lexicographically ordered k-tuples is defined by cases, depending on which members of the tuple are maximal (similar to the definition of successor in decimal, or n-ary notation). We can split each of our old rules into several rules that correspond to these cases. A neater way of achieving the desired result is to introduce a k-ary IDB *nonrecursive* predicate $zero_k$ and a $2k$-ary *nonrecursive* predicate $succ_k$, along with rules that define them in terms of *zero* and *succ*. For example, the rules for $k = 2$ are as follows (recall that *max* is used for the largest member of the ordered database):

$$\begin{aligned} zero_2(X, X) &:- zero(X). \\ succ_2(X, Y, X, Y') &:- succ(Y, Y'). \\ succ_2(X, Y, X', Y') &:- succ(X, X'), zero(Y'), max(Y). \end{aligned}$$

The new rules are now obtained simply by using $zero_k$ and $succ_k$ for the k-tuples that represent time and counters, in all cases where *zero* and *succ* have been used for the single time and counter variables.

The arguments given above are sufficient to show that if M does not halt, then Q is equivalent to P on all ordered databases. On the other hand, arguing much in the same way as in Section 2, we show that, if M halts, then Q is bounded on all databases. We recapitulate the argument briefly. Say that M halts in t steps. If, on a given database D, no initialization rule of P applies, or if D does not contain a zero-element, then the fixpoint of Q is obtained by at most one iteration (needed for defining the nonrecursive predicates $zero_k$ and $succ_k$). Otherwise, if the database contains sufficiently long successor chains for simulating the t steps in M's computation, then the fixpoint is obtained by at most $t + 2$ iterations; one for establishing M's initial configuration, t for the computation, and one for applying the halting rules. Recall that these chains need not be standard; they may contain repeating elements. The remaining case means that the length i of the maximal chain is either less than s, where s is the number of states, or that $i^k < t + 1$; in either case, the fixpoint is obtained by i^k iterations and this is bounded by $max(s^k, t)$. The presence of small database rules makes no difference, since they are initialization rules that do not affect the recursion. □

Note that we can limit ourselves to one "universal" 2CM, provided that we allow arbitrary initializations of one counter (cf. Remark 3 in Section 2). Consequently, a bound on the number of recursive rules, or the number of recursive predicates or the recursive arity, in the program P, will imply a bound (not the same) on the corresponding parameter of Q. To be specific, let us introduce the following notation:

$rr(P)$ = the number of recursive rules in P;
$rp(P)$ = the number of recursive predicates in P;
$ra(P)$ = the recursive arity of P = the maximal arity of recursive predicates.

Then, checking the proof, we see that $rr(Q) \leq A \cdot rr(P)$, $rp(Q) = rp(P)$, $ra(Q) \leq B \cdot rp(P) \cdot ra(P)$, where A and B are constants depending on M. Also the linearity of P implies that of Q. Recall also that P can be any program that does not have the property Π with respect to ordered databases. Hence, we get the following strengthening of the theorem:

COROLLARY 5.9. *Let Π be a semantic, stable, and strongly nontrivial property of programs, which contains boundedness. If there is a program P that does not have the property Π with respect to ordered databases, then it is undecidable whether a given P' has the property Π with respect to all databases, where P' ranges over all programs with $\leq A \cdot rr(P)$ recursive rules, $\leq rp(P)$ recursive predicates and $\leq B \cdot rp(P) \cdot ra(P)$ recursive arity. Here A and B are some fixed constants that do not depend on Π or on P. Moreover, if P is linear, then P' can be also restricted to range over linear programs only.*

Another corollary of the theorem concerns some particular properties mentioned in this paper.

COROLLARY 5.10. *The property of being first order is undecidable. The property of being in LOGSPACE is undecidable, provided LOGSPACE is different*

from PTIME. The properties of being essentially linear and of being in NC are undecidable, provided NC is different from PTIME.

Papadimitriou and Van Gelder (private communication) and Ullman and Van Gelder [39] observed that the undecidability of the NC property (as well as that of the polynomial fringe property, which is a syntactic property and, hence, not covered by our theorem) can be derived easily from Greibach's Theorem [19, Theorem 8.14]. Their argument establishes the undecidability for the class of programs with an unbounded number of recursive predicates; in this sense, the derived result is weaker. On the other hand, the recursive arity that is needed there is only 2.

One might be tempted to conjecture that Theorem 5.8 can be strengthened so as to parallel directly Rice's Theorem; namely, that every nontrivial semantic property is undecidable. In this formulation, we consider properties only with respect to the class of all databases. The discussion following Definition 5.2 makes clear that the assumption of containing boundedness is essential to the truth of the undecidability theorem. The conditions of stability and strong nontriviality, on the other hand, seem necessary as a consequence of the proof technique, as they ensure that a 2CM simulation can indeed be carried out.

6. *Conclusions*

We have shown that it is undecidable whether a Datalog program computes the fixpoint of any database in a bounded number of steps. This undecidability result is shown in a "weak" form where the database contains EDB facts only, and in a "strong" form where IDB facts are also allowed. The proof technique is generic simulation of 2-counter machines, where the Datalog program codes the machine, and the database codes a (potentially nonstandard) interpretation of an initial segment of integers. The weak version of the theorem is also shown for an extension of Datalog with inequality, where the database codes a machine computation, and the query program checks that the coding is legitimate. We presented as well a general undecidabilty result patterned after Rice's Theorem.

A variety of open questions remain to be resolved. As in the case of many undecidability proofs, it remains to fix precisely the syntactic constraints (in this case, on query programs) that imply undecidability. The important characteristics include the number of recursive rules, the arity of the IDB predicates, and the degree of recursion (i.e., linear, binary, etc.) in the recursive rules). (A more detailed review of many open questions relating to boundedness appears in [24].) Some progress has been made along these lines in [11] and [18]. However, the decidability of boundedness is still open for query programs with one linear recursive rule and one or more initialization rules, as is the "linear sirup" strong boundedness question with one linear recursive rule and no initialization rules. It is possible that our analogue of Rice's Theorem may be sharpened as well, by eliminating some of the conditions on Datalog programs that facilitated proof of the generalized undecidability theorem.

Appendix A. Connected Rules

To make our rules connected, we increase the arity of *CN* to 5. The fifth argument of *CN* in all its occurrences will be a new variable *U*. We use an

additional EDB predicate *connect* to connect all variables to U. That is, if $V_1, \ldots, V_k$ are the variables in one of our rules, then we add the atoms $connect(V_1, U), \ldots, connect(V_k, U)$ to the body of the rule. For example, the initialization rule becomes

$$CN(Z, Z, Z, Z, U) :- zero(Z), connect(Z, U)$$

and the halting rule becomes

$$\begin{aligned} CN(A, B, C, D, U) :- & \; CN(T, S, C_1, C_2, U), \\ & zero(Z), succ(Z, A_1), \\ & succ(A_1, A_2), \ldots, succ(A_{h-1}, S), \\ & connect(Z, U), connect(A_1, U), \ldots, \\ & connect(A_{h-1}, U), \\ & connect(A, U), connect(B, U), \\ & connect(C, U), \\ & connect(D, U), connect(T, U), \\ & connect(S, U), \\ & connect(C_1, U), connect(C_2, U). \end{aligned}$$

Observe that, in any deduction of an IDB fact, the rules in the query program are instantiated such that the variable U is always mapped to some fixed database constant. We may think of the constant as being informally *chosen* by the initialization rule, and *maintained* by the transition rules and halting rule. Thus, a *persistent variable* U allows Datalog expressibility of constants.

Given a database D of EDB facts, we may decompose D as $\bigcup_u D_{(u)}$, where u is a database constant appearing in D, and $D_{(u)}$ contains the EDB facts over constants c such that $connect(c, u) \in D$. The decomposition is not necessarily disjoint, but each $D_{(u)}$ is a database; furthermore, if P is bounded, then there exists an integer $k \geq 0$, where $P^\infty(D_{(u)}) = P^k(D_{(u)})$. Every IDB fact in $P^k(D)$ appears uniquely in some $P^k(D_{(u)})$, so that the analysis of Section 2 "factors" nicely. The standard database D_t over constants $\{0, 1, \ldots, t\}$ then has added EDB predicates $connect(j, 0)$ for $0 \leq j \leq t$, and an unbounded proof (in the sense of Lemma 2.4) is over constants occurring in some fixed $D_{(u)}$, so a halting computation may be simulated in $P^k(D_{(u)})$. We leave it to the reader to check that the remaining details in the proof of Lemma 2.4 go through with the obvious modifications. A virtually identical modification can be made to the strong boundedness proof of Section 3.

Appendix B. Reduction of the Arity from 7 to 5

First, we reduce the arity to 6 by deleting the time argument in CN. The idea is, roughly speaking, to obtain in the simulation only the *machine-configurations* (i.e., without a time indicator) produced during the computation.

We shall use a 6-ary predicate CN, obtained from the previous one by the deletion of the time argument. An IDB-fact of the form $CN(z, g_1, g_2, s, c_1, c_2)$ represents a machine configuration (s, c_1, c_2) where, as before, c_1 and c_2 are used also in the role of integers; namely, the lengths of the successor chains

from z to c_1 and c_2. If the database is standard, then indeed c_1 and c_2 determine unique integers. (in nonstandard databases, c_i may determine more than one integer, or no integer); hence, if s represents a state of M, then the IDB fact represents a true machine configuration of M.

Consider the set of rules, P, obtained from the ones just given, simply by deleting everywhere the time argument, leaving all the rest exactly as before.

For standard databases, we need at least c simulation steps of the program in order to increase a counter from zero (i.e., z) to c. Hence, we get the following analogue of Lemma 3.2.

LEMMA B1. *If M does not halt and (s, c_1, c_2) is a machine configuration produced in the computation, then, for all standard databases, at least $max(c_1, c_2)$ iterations are needed in order to derive a CN-fact representating this configuration.*

This shows that if M does not halt and, moreover, the values of the counters are unbounded, than P is not bounded, and, a fortiori, not strongly bounded. It leaves unsettled the case where M does not halt but *cycles*, where M is said to *cycle* if its standard computation (i.e., the one starting in the initial state, with both counters initialized to 0) does not halt and contains finitely many machine configurations.

Obviously, M cycles just when the standard computation consists of a finite sequence of machine configurations, followed by an infinitely repeating cycle. (The problem of a cycling machine does not arise in the case of the previous 7-ary *CN*, because such a machine goes through infinitely many configurations.) We shall now show why it suffices to consider only noncycling machines. We shall actually show more; it suffices to consider only *completely noncycling* machines, where this is defined by:

> A 2CM is *completely noncycling* if, for every machine configuration *mc*, the computation that starts with *mc* is noncycling.

The restriction to noncycling machines is imposed in order to make the reduction (of halting to strong boundedness) work for the case in which M does not halt. We shall need the restriction to completely noncycling machines for the case where M halts. (Note that a halting M need not be completely noncycling; it may cycle when started on some configuration that does not occur in the standard computation.) The argument is based on the following claim:

> *There is an effective procedure that transforms any given M into a completely noncycling M' such that, M' halts iff M either halts or cycles.*

(Roughly speaking, the idea is to construct a machine, $\hat{M}$ with additional counters, that imitates M on the first two counters and codes, in one of the others, the list of machine configurations of M's computation. After carrying out a move of M and listing the new configuration, it checks whether it appears previously in the list. If it does, $\hat{M}$ halts; otherwise, it proceeds with the next move of M. By well-known techniques, we can then construct a two-counter M' that simulates $\hat{M}$. It is easy to see that M' does not cycle when started on the initial configuration. To ensure that it does not cycle when started on any configuration one has to go into some technical niceties concerning the encoding of configuration lists and the simulation of more than two counters by two counters. The main point is to encode lists in such a way

that, even when the auxiliary counters are given arbitrary initial values, the updating and decoding of lists will be "correct" from a certain point on, thus ensuring that a cycle will be eventually detected. The construction is easier for Turing machines, where the configuration list consists of the sequence of configuration codes, separated by a special punctuation symbol. A construction for 2CMs is obtainable by going through Turing machines and translating back into a 2CM, using a suitable simulation, for example, one obtained along the lines described in [19].

The claim implies that the problem of whether or not a given 2CM does halt or cycle is reducible to the problem of whether or not a given completely noncycling 2CM halts. Since the first problem is r.e.-complete (this can be easily shown by standard methods), the second problem is undecidable. Hence, it suffices to reduce halting to strong boundedness only for completely noncycling machines.

The argument given above shows that the reduction works if M does not halt. For the case where M halts, we use the following lemma, which parallels Lemma 3.3.

LEMMA B2. *Let (t, s, c_1, c_2) be a configuration of M's standard computation. Then, for some number n_t and for all sufficiently large k, if $x_0, x_1, \ldots, x_k$ is a chain in the database D, then $CN(x_0, x_1, x_1, s, c_1, c_2) \in P^{n_t}(D)$.*

PROOF. This claim is easily seen to be true by observing that an IDB fact representing the machine configuration (s, c_1, c_2) is derivable, provided that we have a long enough successor chain for encoding the states of M and all the counter values in the computation. Moreover, all counter values are obviously $\leq t$. □

Assuming that M halts, if the database has a sufficiently long chain, we shall be able to apply the halting rule and "saturate" the IDB predicate; therefore, P is strongly bounded on all databases with sufficiently long chains, where "sufficiently long" is determined by the halting computation and the number of states in M.

There remains the case in which M halts, but the database does not have a sufficiently long chain needed to simulate the halting computation. In the case of a 7-ary CN, a bound on the length of chains implies a bound on the number of iterations by which new IDB facts are obtained (Lemma 3.4). In the present case, this is no longer true. The reason for that is that, in a nonstandard database, a member c may be reachable from the "zero" z by chains of different lengths. Thus, configurations not belonging to the standard computation might arise in the simulation. Moreover, when a counter value, say n, is decreased by 1, we can move to any predecessor of the element representing n. This predecessor may lie on another chain and it may represent there a value $< n - 1$. On this new chain, our program P can simulate more than one increase of counter before reaching n. We might thus "simulate" by repeated back-and-forth moves an unbounded number of steps of some nonstandard diverging noncycling computation. The problem does not arise in the 7-ary case, because every move increases the time argument, consequently, as the proof of Lemma 3.4 shows, a bound on chain-length restricts uniformly the lengths of the derivations.

We note that the problem will not arise if the database satisfies the following condition:

For every x, y, every two successor chains from x to y have the same length.

Call such a database *semi-standard* and let $d(x, y)$ be the length of every successor chain from x to y ($d(x, y) = \infty$ if there are no chains from x to y).

Assuming that the database is semi-standard, consider a derivation (i.e., a sequence of IDB facts, where each fact, except the first, is derived from its predecessor). Note that the "zero point", say z, must be the same throughout the derivation. By checking the rules one easily sees, that, for every IDB fact, $CN(z, g_1, g_2, s, c_1, c_2)$, occurring in the derivation, except possibly the last, c_1 and c_2 must be on successor chains that start at z (including the possibilities $c_i = z$). Therefore, this IDB fact represents the machine configuration $(d(z, s), d(z, c_1), d(z, c_2))$. Since increase and decrease moves are represented correctly by increases and decreases in the values of $d(\ ,\ )$, the derivation, except possibly the last step, simulates correctly some computation of M (not necessarily the standard one). Any bound on the length of chains implies a bound on the counter values in the simulated computation. Since M is completely noncycling, this implies also a bound on the number of computation steps. Hence, for semi-standard databases, we get the following analog of Lemma 3.4:

LEMMA B3. *For a completely noncycling M, if D is a semi-standard database containing no chain length $k + 1$, then, for every initial assignments I, $P^{\infty}(D \cup I) = P^{m_k}(D \cup I)$, where m_k depends on k only.*

Consequently, if the completely noncycling M halts, then P is strongly bounded in the class of semi-standard databases. In order to take care of databases that are not semi-standard, we augment P by a set of *semi-standard rules* designed to check whether there are two successor chains of different length from the same starting point to the same endpoint. When such chains are found the IDB predicate is "saturated" as in the case of halting. The checking uses the same IDB predicate CN. We add a new "state", not belonging to the states of M, in which the checking is to be carried out. We assume that the states of M are $0, 2, 3, \ldots, h$, where, as before, 0 is the initial state. 1, which is not among the states of M, will play the role of the "checking state". The checking consists in constructing, in the second and third arguments, successor chains of unequal length starting at the same point. The "saturation" rule is applicable if the endpoints of the chains coincide. As before, the first argument is used for the "zero" and the fourth hosts the state parameter, that is, 1. The last two arguments are not used.

The *semi-standardness rules*:

$$
\begin{aligned}
CN(Z, Z, G, S, X, X) &:\!- succ(Z, G), S = 1;\\
CN(Z, Z, G_2', S, X, X) &:\!- CN(Z, Z, G_2, S, X, X),\\
&\qquad succ(G_2, G_2');\\
CN(Z, G_1', G_2', S, X, X) &:\!- CN(Z, G_1, G_2, S, X, X),\\
&\qquad succ(G_1, G_1') succ(G_2, G_2');\\
CN(A, B, C, D, E, F) &:\!- CN(Z, G, G, S, X, X).
\end{aligned}
$$

The first rule switches to "state 1", initializes the second argument to "zero" and starts a chain of positive length on the third argument; the second rule increases this chain by 1; the third rule increases both the second-argument and the third-argument chains by 1 and the last rule puts all 6-tuples into CN, once the two chains meet at the same endpoint.

It is obvious that, in every derivation in which a semi-standardness rule is used, the state must be 1 throughout the derivation. The last semi-standardness rule can be used iff the database is not semi-standard. If all chains are of length $\leq k$ and the database is not semi-standard, then after at most $k-1$ applications of the semi-standardness rules, we shall derive an IDB fact of the form $CN(z, g, g, s, x, x)$, enabling the application of the last rule. This, together with the previously established facts, implies that if the completely noncycling M halts, then P is strongly bounded in the class of all databases.

This completes the reduction of the arity to 6. A further reduction of arity is obtained by getting rid of the state argument. The basic observation here is that we can put some fixed bound on the number of states of the 2CMs to be simulated, provided that we allow an arbitrary initialization of one of the counters. This derives from the observation, made in Remark 3 of the previous section, that, by using a "universal 2CM", we get one particular 2CM such that the problem whether it halts, given a nonzero initialization of one of its counters, is r.e.-complete. The same is true if the problem is whether it halts-or-cycles and, as we indicated above, we can transform this 2CM into a completely noncycling machine.

Let the number of states be k. As we did above, we add an extra state, 1, for checking semi-standardness. Altogether, the states are $0, 1, 2, \ldots, k$.

Now let the database encode, an initial segment of the natural numbers and $k+1$ disjoint isomorphic copies of it, marked by EDB predicates st_j, $j = 0, \ldots, k$. The mappings of the initial segment onto the copies is given by an EDB predicate $copy(X, Y)$ which is supposed to read: "Y is a member corresponding to X in one of the copies". Here, X is supposed to range over the "prototype" initial segment and Y—over all the copies. The machine configuration (s, c_1, c_2) is now encoded by the pair (c_1', c_2') where c_1', c_2' represent c_1 and c_2 in the s-copy. Let $copy_j(X, Y)$ stand for:

$$copy(X, Y), st_j(Y),$$

and let $copied(X)$ stand for:

$$copy_0(X, Y_0), \ldots, copy_k(X, Y_k).$$

We use this in order to express the fact that X has corresponding members in all copies (we always assume that the Y_j are additional variables distinct from all other clause variables).

When simulating a computation, the variables in the first three arguments are supposed to range over the "prototype chain"; the counter variables range over the copies and thus they encode also the state.

To express the transition $\delta(j, >, =) = (j', pop, push)$, we shall use the

following clause (where *CN* is now of arity 5):

$$\begin{aligned} CN(Z, G_1, G_2, C_1', C_2') :- \; & CN(Z, G_1, G_2, C_1, C_2), \\ & copy_J(G_1, C_1), copy_J(Z, C_2), \\ & succ(X_1, G_1), succ(Z, X_2) \\ & copy_{J'}(X_1, C_1'), copy_{J'}(X_2, C_2'), \\ & copied(G_1), copied(G_2), copied(Z), \\ & copied(X_1), copied(X_2). \end{aligned}$$

The last part in the body guarantees that all the points used in the initial segment have corresponding members in all copies. Consequently, if we can simulate a computation by using some chain, then this chain will have duplicates in each of the copies that correspond to the different states. This ensures that, with every nonstandard computation that is simulated by the program on a given database, the standard computation can be simulated using the full length of the chain. It is now easy to see how the rest of the rules for simulating the 2CM are to be rewritten in terms of the new vocabulary.

The first semi-standardness rule is now rewritten as:

$$CN(Z, Z, G, X, X) :- succ(Z, G), copy_1(X).$$

The others remain unchanged, except that the state argument is omitted.

ACKNOWLEDGMENTS. We are grateful for the participants of the Standard NAIL! Seminar for fruitful discussions. We'd also like to thank Ron Fagin and Phokion Kolaitis for their helpful comments on a previous draft of this paper.

REFERENCES

1. ABITEBOUL, S. Boundedness is undecidable for Datalog programs with a single recursive rule. *Inf. Proc. Lett. 32* (1989) 281–288.
2. AHO, A. V., SAGIV, Y., AND ULLMAN, J. D. Efficient optimization of a class of relational expression. *ACM Trans. Datab. Syst. 4*, 4 (Dec. 1979), 435–454.
3. AHO, A. V., AND ULLMAN, J. D. Universality of data retrieval languages. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages* (San Antonio, Tex., Jan. 29–31). ACM, New York, 1979, pp. 110–117.
4. BANCILHON, F., AND RAMAKRISHNAN, R. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM Conference on Management of Data* (Washington, D.C., May 28–30), ACM, New York, 1986, pp. 16–52.
5. CHAKRAVARTHY, U. S., MINKER, J., AND GRANT, J. Semantic query optimization—additional constraints and control strategies. In *Proceedings of the 2nd International Conference on Expert Database Systems* (Charleston, S.C.). Benjamin Cummings, Menlo Park, Calif., 1986, pp. 259–169.
6. CHANDRA, A. K. Programming primitives for database languages. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26–28). ACM, New York, 1981, pp. 50–62.
7. CHANDRA, A. K., AND HAREL, D. Computable queries for relational databases. *J. Comput. Syst. Sci. 21* (1980), 156–178.
8. CHANDRA, A. K., AND HAREL, D. Structure and complexity of relational queries. *J. Comput. Syst. Sci. 25* (1982), 99–128.
9. CHANDRA, A. K., AND HAREL, D. Horn-clause queries and generalizations. *J. Logic Prog. 1* (1985), 1–15.

10. CHANDRA, A. K., AND MERLIN, P. M. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing* (Boulder, Colo., May 2–4). ACM, New York, 1977, pp. 77–90.
11. COSMADAKIS, S. S., GAIFMAN, H., KANELLAKIS, P. C., AND VARDI, M. Y. Decidable optimization problems for database logic programs. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (Chicago, Ill., May 2–4). ACM, New York, 1988, pp. 477–490.
12. COSMADAKIS, S. S., KANELLAKIS, P. Parallel evaluation of recursive rule queries. In *Proceedings of the 5th Annual ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Cambridge, Mass., Mar. 24–26). ACM, New York, 1986, pp. 280–293.
13. DE ROUGEMONT, M. Uniform definability on finite structures with successor. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing* (Washington, D.C., Apr. 30–May 2). ACM, New York, 1984, pp. 409–417.
14. DREBEN, D., AND GOLDFARB, W. D. *The Decision Problem: Solvable Classes of Quantificational Formulas*. Addison-Wesley, Reading, Mass., 1979.
15. GAIFMAN, H., MAIRSON, H., SAGIV, Y., AND VARDI, M. Y. Undecidable optimization problems for database logic programs. In *Proceedings of the 2nd Annual IEEE Symposium on Logic in Computer Science* (Ithaca, N.Y.). IEEE, New York, 1987, pp. 106–115.
16. GALLAIRE, H., AND MINKER, J. *Logic and Databases*. Plenum Press, New York, 1978.
17. HENSCHEN, L. J., AND NAQVI, S. A. On compiling queries in recursive first-order databases. *J. ACM 31*, 1 (Jan. 1984), 47–85.
18. HILLEBRAND, G., KANELLAKIS, P., MAIRSON, H., AND VARDI, M. Tools for Datalog boundedness. In *Proceedings of the 10 Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Denver, Colo., May 29–31). ACM, New York, 1991, pp. 1–12.
19. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.
20. IMMERMAN, N. Relational queries computable in polynomial time. *Inf. Cont. 68* (1986), 86–104.
21. IOANNIDIS, Y. E. A time bound on the materialization of some recursively defined views. In *Proceedings of the 11th International Conference on Very Large Data Bases* (Stockholm, Sweden). Morgan-Kaufmann, San Mateo, Calif., 1985, pp. 219–226.
22. IOANNIDIS, Y. E. Bounded recursion in deductive databases. *Algorithmica 1* (1986), pp. 361–385.
23. JARKE, M., CLIFFORD, J., AND VASSILIOU, Y. An optimizing Prolog front-end to a relational query system. In *Proceedings of the ACM Conference on Management of Data* (Boston, Mass., June 18–21). ACM, New York, 1984, pp. 296–306.
24. KANELLAKIS, P., AND ABITEBOUL, S. Deciding bounded recursion in database logic programs. *SIGACT News 20* 4 (Fall 1989), 17–23.
25. KOLAITIS, P. G., AND VARDI, M. Y. On the expressive power of Datalog—tools and a case study. *J. Comput. Syst. Sci.*, to appear.
26. LLOYD, J. W. *Foundations of Logic Programming* Springer-Verlag, New York, 1987.
27. MAIER, D., ULLMAN, J. D., AND VARDI, M. Y. On the foundations of the universal relation model. *ACM Trans. Datab. Syst. 9* (1984), 283–308.
28. MOSCHOVAKIS, Y. N. *Elementary Induction on Abstract Structures*. North Holland, Amsterdam, The Netherlands, 1974.
29. NAUGHTON, J. F. Data independent recursion in deductive databases. *J. Comput. Syst. Sci. 38* (1989), 259–289.
30. NAUGHTON, J. F. Minimizing function-free recursive inference rules. *J. ACM 36* 1 (Jan. 1989), 69–91.
31. NAUGHTON, J. F., AND SAGIV, Y. Minimizing expansions of recursions. In *Resolution of Equations in Algebraic Structures*, vol. 1. H. Ait-Kaci and M. Nivat, Eds., Academic Press, Orlando, Fla., 1989, pp. 321–349.
32. NAUGHTON, J. F., AND SAGIV, Y. A simple characterization of uniform boundedness for a class of recursions. *J. Logic Prog. 10* (1991), 233–254.
33. ROGERS, H. *Theory of Recursive Functions and Effective Computability* McGraw-Hill, New York, 1967.
34. SAGIV, Y. On bounded database schemes and bounded Horn-clause programs. *SIAM J. Comput. 17* (1988), 1–22.
35. SAGIV, Y., AND VARDI, M. Y. Safety of Datalog queries over infinite databases. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pa., Mar. 29–31). ACM, New York, 1989, pp. 160–171.

36. SHMUELI, O. Decidability and expressiveness aspects of logic queries. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Cal., Mar. 23–25). ACM, New York, 1987, pp. 237–249.
37. ULLMAN, J. D. Implementation of logical query languages for databases. *ACM Trans. Datab. Syst. 10* 3 (Sept. 1985), 289–321.
38. ULLMAN, J. D. *Principles of Database and Knowledge-Base Systems*, vol. I. Computer Science Press, Rockville, Md., 1989.
39. ULLMAN, J. D., AND VAN GELDER, A. Parallel complexity of logical query programs. *Algorithmica 3* (1988), pp. 5–42.
40. VARDI, M. Y. The complexity of relational query languages. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif, May 5–7). ACM, New York, 1982, pp. 137–146.
41. VARDI, M. Y. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., Mar. 21–23). ACM, New York, 1988, pp. 341–351.
42. ZLOOF, M. Query-by-example: Operations on the transitive closure. IBM res. rep. RC5526. IBM Watson Research Center, Yorktown Heights, N.Y., 1976.

RECEIVED FEBRUARY 1990; REVISED 1991; ACCEPTED AUGUST 1991