Undecidable Boundedness Problem for Datalog Programs

Gerd G. Hillebrand[1]
Paris C. Kanellakis[2]
Harry G. Mairson[3]
Moshe y. Vardi[4]

Technical Report No. CS-92-20

April 1992

[1]Brown University, Department of Computer Science Box 1910, Providence, RI 02912
[2]Brown University, Department of Computer Science Box 1910, Providence, RI 02912
[3]Brandeis University,Waltham, MA 02254
[4]IBM Research Almaden Research Center K53-802, 650 Harry Rd., San Jose, CA 95120-6099

Undecidable Boundedness Problems for Datalog Programs*

Gerd G. Hillebrand† Paris C. Kanellakis‡ Harry G. Mairson§ Moshe Y. Vardi¶

April 10, 1992

Abstract

A given Datalog program is bounded if its depth of recursion is independent of the input database. Deciding boundedness is a basic task for the analysis of database logic programs. The undecidability of Datalog boundedness was first demonstrated by Gaifman et al. We introduce new techniques for proving the undecidability of various kinds of boundedness, which allow us to considerably strengthen the results of Gaifman et al. In particular: (1) We use a new generic reduction technique to show that program boundedness is undecidable for arity 2 predicates, even with linear rules. (2) We use the mortality problem of Turing machines to show that uniform boundedness is undecidable for arity 3 predicates and for arity 1 predicates when $\neq$ is also allowed. (3) We use a single rule polymorphically to show that program (resp., predicate) boundedness is undecidable for two linear rules (resp., one rule and a projection), one initialization rule and small arity predicates.

1 Introduction

It has been realized for some time that first-order database query languages are lacking in expressive power. This has led to the study of *Datalog* programs [K90, U89], which combine positive existential first-order formulas with recursion—see [CH85]. Analyzing the depth of recursion of these database logic programs has emerged as a key problem, e.g., for parallel evaluation [CK86, K88, UV88] or for optimization [N89b, S88].

Datalog boundedness (i.e., whether the depth of recursion of a given program is a constant independent of the input database) is interesting because it is the simplest case of

*The results presented here appeared in preliminary form in [V88] (Section 2 of this paper) and [HKMV91] (Sections 3–6 and the appendix of this paper).

†Dept. of Computer Science, Brown University, Providence, RI 02912. Supported by ONR grant N00014-83-K-0146 ARPA Order No. 6320-1.

‡Dept. of Computer Science, Brown University, Providence, RI 02912. Supported by NSF grant IRI-8617344 and ONR grant N00014-83-K-0146 ARPA Order No. 6320-1.

§Dept. of Computer Science, Brandeis University, Waltham, MA 02254. Supported by NSF Grant CCR-9017125 and by Texas Instruments. This author also thanks the CS Dept. at UC Santa Barbara, the Music Academy of the West, and the Cate School of Carpenteria for their generous hospitality.

¶IBM Research Almaden Research Center K53-802, 650 Harry Rd., San Jose, CA 95120-6099.

recursion analysis. Boundedness is also a semantic property (i.e., it is preserved by program equivalence): a Datalog program is bounded iff it is equivalent to a positive existential first-order formula [NS91] iff it is equivalent, over finite structures, to a first-order formula [AG89]. Let us describe the problem, its status (see also [KA89]), and our contributions.

1.1 Basic Definitions

Datalog Syntax: *A (Datalog) program P is a finite set of rules.*

Here, a *rule* is a statement of the form $p(\mathbf{X}) :\!-\!\!-\, \varphi$. Its *head* $p(\mathbf{X})$ is a *predicate atom*, that is, p is a predicate symbol of arity $a \geq 0$ and $\mathbf{X}$ is a list of a variable symbols, not necessarily distinct. Its *body* φ is a finite nonempty list of predicate atoms.

In a program, any predicate symbol occurring in the head of a rule is called an *intensional database predicate* (IDB); all others are called *extensional database predicates* (EDBs). A rule is *recursive* if its body contains at least one IDB occurrence; otherwise it is an *initialization* rule. A program is *linear* if in the body of each rule there can be at most one IDB occurrence. A program has arity a if the maximum arity of any IDB in the program is a; note that EDBs can have any arity.

There are three conventions for *input* and *output* symbols: (1) The *predicate* I/O convention is that EDBs are the input and a designated predicate symbol the output. (2) The *program* I/O convention is that EDBs are the input and all predicate symbols the output. (3) The *uniform* I/O convention is that all predicate symbols are both input and output. □

Datalog Semantics: *Let P be a program and D a database over its input predicate symbols. Then $P^{\infty}(D)$ is the projection on the output predicate symbol(s) of the finite set of ground atoms, which have derivation trees from D and P.*

Here, a *ground atom* (or *fact*) is a predicate atom with constant symbols substituted for the variables. A *database* over a set of predicate symbols is a finite collection of ground atoms with predicate symbols from this set.

Note that the I/O convention of P determines the possible input databases D (as finite sets of ground atoms over the input predicate symbols) and the output databases (as the finite sets of ground atoms over the output predicate symbols that have derivation trees).

A *derivation tree* from a database D and a program P for a ground atom t is a tree where: (1) each node of the tree is labeled by a ground atom, (2) the root is labeled by t, (3) each leaf is labeled by a ground atom of D, and (4) for each internal node, there is an instantiation of the variables of a rule in P with constants occurring in D so that the head is the label of that node and the body is the list of labels of its children.

The *height* of a derivation tree is the length of its longest path (intuitively, it is the depth of recursion used in this derivation of the root). We use $P^{i}(D)$ for the subset of $P^{\infty}(D)$ with derivation trees of height at most i. □

Datalog Boundedness: *Program P is bounded if $P^{\infty}(D) = P^{c}(D)$ for some constant c independent of D.*

We use the terms *predicate*, *program* and *uniform* boundedness depending on the I/O convention. □

For the same set of rules, *uniform boundedness* $\Rightarrow$ *program boundedness* $\Rightarrow$ *predicate boundedness*, but the converses need not hold. From the definitions, one can see that undecidability of uniform boundedness automatically translates into undecidability for the other kinds; and that decidability of predicate boundedness implies decidability of the other kinds.

Uniform boundedness (also known as *strong* boundedness) is the simplest Datalog problem involving recursion. Note that, the corresponding uniform equivalence of programs is decidable [S88] (first shown in [CK86] for one IDB), unlike the forms of equivalence corresponding to the other two I/O conventions [Sh87].

Example: To illustrate Datalog and boundedness consider the following canonical example [N89a]:

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

This example is program bounded since $buys(Z,Y)$ can be changed to $likes(Z,Y)$ to yield an equivalent recursion-free program. (It is also uniformly bounded.) On the other hand the program:

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

is inherently recursive (i.e., is not equivalent to any recursion-free program). $\Box$

1.2 Previous Work

An early definition and use of bounded recursion appeared in the context of universal relations [MUV84]. The first results about boundedness were positive: PTIME graph-theoretic decision procedures for subclasses of linear programs were proposed in [I86, N89a] and this focused attention on the problem. Unfortunately, as first shown in [GMSV87], boundedness is undecidable in general; more importantly, undecidability of boundedness entails undecidability for many other questions concerning recursion.

Uniform boundedness is Σ_1^0-complete [GMSV87]; the previous undecidability results with the smallest arity involved linear, 5-ary programs [GMSV87]. With respect to the number of rules, there is some fixed linear program P with one IDB (but considerably more than two recursive rules) such that uniform boundedness is undecidable for $P \cup \{\rho\}$ for a variable initialization rule ρ [GMSV87].

Program boundedness is also Σ_1^0-complete [GMSV87] and the known undecidability results with the smallest arity involve linear, 4-ary programs. The previous undecidability results with the smallest number of rules involved one nonlinear recursive rule and two initialization rules [A89].

Predicate boundedness is even harder; it is Σ_2^0-complete [CGKV88] because of the final projection. Predicate boundedness (and thus program and uniform boundedness as well) is shown to be decidable for monadic programs in [CGKV88], which also contains a partial analysis of the complexity of this decision problem (see also [M90]). Decision procedures are possible in some other special cases: e.g., [V88] shows the NP-completeness of the boundedness problem for arity 2 programs with one linear rule; also for chain rules and some of their generalizations we have decidability through context-free language finiteness tests [G90]. Other decidable cases appear: in [S85] for uniform boundedness of "typed template dependency" rules, in [NS91] which generalizes [I86, N89a], and in [HKMV91], which presents a decidability result for certain single-rule programs.

1.3 Our Contributions

In this paper we concentrate on program arity and number of rules. We improve the state-of-the-art on undecidable boundedness problems for Datalog programs and introduce a number of new techniques. In particular, we present the following results—where the item numbers (plus one) correspond to the section numbers in the paper:

1. We show that program boundedness is undecidable for binary programs with linear rules, thereby extending the results of [GMSV87] from arity 4 to arity 2 programs. The proof technique is related to [GMSV87], but instead of simulating computations we just check whether a computation encoded in the database is a valid terminating computation. This result is optimal with respect to arity and linearity, since predicate (and thus program) boundedness is decidable for monadic programs [CGKV88].

2. We show that uniform boundedness is undecidable for arity 1 linear programs with $\neq$. This resolves an open question of [GMSV87], where undecidability is shown for program boundedness. The proof is a reduction from the *mortality problem for Turing machines* [H66]. It is a qualitatively different proof from the techniques used in [GMSV87]. (The mortality problem has recently been used to prove the undecidability of the semi-unification problem [KTU90].) Given the decidability results of [CGKV88] for monadic programs, this tight result illustrates the power of $\neq$.

3. We show that uniform boundedness is undecidable for arity 3 programs. This improves the arity from 5 to 3 (but leaves open arity 2). The proof, which uses nonlinear rules, is a refined version of the previous reduction from the mortality problem for Turing machines. In the appendix we give a different proof of this result along the lines of [GMSV87]. Using Turing machines instead of counter machines we simplify the proof in [GMSV87] and decrease the arity. Unlike the proof in [GMSV87], our proof uses nonlinear rules. We hope that some further refinement of either of the two proofs might be useful for settling the case of arity 2.

4. We show that program boundedness is undecidable for two linear recursive rules and one initialization rule. This improves the results of [A89] with respect to linearity and initialization, but adds one recursive "flood-halt rule". The proof is based on the flooding technique of [GMSV87] and a *polymorphic* encoding of many rules in one. The arity used is 6.

5. We show that predicate boundedness is undecidable for one linear recursive rule, one projection and one initialization rule. This exchanges the "flood-halt rule" for a projection. The arity used is 7. This is an undecidability proof for a very simple set of *connected* rules—see [GMSV87] for a definition of connectivity and for a technique of proving undecidability under this restriction.

2 Undecidability of Boundedness for Binary Programs

In [GMSV87], boundedness was proven undecidable by reduction from the halting problem for 2-counter machines. The idea was that the database encodes a prefix of the natural numbers, and the program simulates the computation of the machine. Here we take a different approach. First, we use Turing machines instead of 2-counter machines. Secondly, we let the database encode a prefix of a computation of the machine, and we let the program check that the computation is legal.

Let M be a Turing machine with an alphabet Γ and set S of states. The set $\Delta = \Gamma \cup (S \times \Gamma)$ is called the *extended* alphabet of M. It is well-known that configurations of M can be described by words in Δ^*. Let $\Delta' = \Delta \cup \{\#\}$, where # is a new symbol. We can encode a computation of M as a word in $(\Delta')^*$, where # symbols mark the beginning of new configurations. More precisely, a *k-prefix* of M is a string $C \in (\Delta')^{(k+1)(k+2)/2}$ of the form $\#C_1\#\ldots\#C_k\#$, where $C_1 \in \Delta$ encodes the initial configuration of M, and $C_i \in \Delta^i$ encodes a configuration of M that succeeds the configuration C_{i-1} for $1 < i \leq k$.

For the purposes of this proof, we make a number of assumptions about the Turing machine M. First, we assume that M is started on a tape semi-infinite to the right with the head positioned on the leftmost cell. Thus, the initial configuration C_1 can be described by a single letter $(s, \sqcup) \in \Delta$, where s is the start state of M and $\sqcup$ is the blank symbol. Second, we assume that M never moves off the left edge of this semi-infinite tape. Third, we describe the configuration C_i by specifying the contents of the i leftmost cells. This is without loss of generality since at most i cells can be visited by the head in i moves. This also justifies the fact that $C_i \in \Delta^i$.

The idea of the reduction below is to let the database encode a k-prefix of M. Every element in the database encodes a letter in this k-prefix. We have a unary EDB predicate q_a for every letter $a \in \Delta'$; an element x encodes the letter a precisely when $x \in q_a$ holds. We have a unary relation *first* that encodes the first letter, and a binary relation *succ* that encodes the adjacency relation between letters. Thus, to encode the word *abc*, the database needs to contain elements x, y, z such that $q_a = \{x\}$, $q_b = \{y\}$, $q_c = \{z\}$, $\mathit{first} = \{x\}$, and $\mathit{succ} = \{(x, y), (y, z)\}$. Of course, not all databases would indeed constitute a meaningful encoding. This is a problem we will have to deal with.

We now construct a Datalog program P that simulates M, such that M halts on the empty tape if and only if P is bounded. Since the halting problem for Turing machines is undecidable, so is boundedness of Datalog programs. The program P will have one binary IDB predicate *FING*. The idea is that to check that a word encodes a legal computation of M it suffices to check triples of letters in corresponding positions in successive configurations. Thus, if C_i and C_{i+1} are two successive configurations and $\#C_i\#C_{i+1}\# = a_0a_1\ldots a_ia_{i+1}a_{i+2}\ldots a_{2i+1}a_{2i+2}a_{2i+3}$, where $a_0 = a_{i+1} = a_{2i+3} = \#$, one would check all pairs of triples $\langle a_ja_{j+1}a_{j+2}, a_{j+i+1}a_{j+i+2}a_{j+i+3}\rangle$ for $0 \leq j \leq i$. It is well-known that there is a relation $R_M \subseteq (\Delta')^6$ such that two configurations are successive if and only if for every pair of corresponding triples *abc* and *def* we have $(a, b, c, d, e, f) \in R_M$. The relation *FING* is supposed to contain elements in corresponding positions in successive configurations. One could think of pairs in *FING* as pairs of fingers pointing to corresponding positions.

The program P has five types of rules: *encoding rules* that check that no element of the database encodes more than one letter, *halting rules* that check whether the computation reaches a halting state, *error detecting rules* that check whether the computation encoded by the database is legal, a *finger pointing rule* that initializes the pointing fingers, and *finger moving rules* that move the fingers to the next pair of corresponding positions. The only way the fingers can keep being moved is along a legal computation. Thus, the program will be unbounded if and only if there are arbitrarily long legal computations. But that is possible precisely when M diverges on the empty tape. We now see the construction in detail.

Encoding. For every pair of distinct letters $a, b \in \Delta'$, $a \neq b$, we have a rule:

$$FING(U,V) \quad :- \quad q_a(X), q_b(X).$$

Halting. Let $h \in S$ be the halting state. For every letter $a \in \Gamma$ we have a rule:

$$FING(U,V) \quad :- \quad q_{(h,a)}(X).$$

Error Detecting. For all letters $a, b, c, d, e, f \in \Delta'$ such that $(a, b, c, d, e, f) \notin R_M$ we have a rule:

$$\begin{aligned} FING(U,V) \quad :- \quad & FING(X_2,Y_2), \\ & succ(X_1,X_2), succ(X_2,X_3), succ(Y_1,Y_2), succ(Y_2,Y_3), \\ & q_a(X_1), q_b(X_2), q_c(X_3), q_d(Y_1), q_e(Y_2), q_f(Y_3). \end{aligned}$$

Finger Pointing. Let $s \in S$ be the starting state of M, and let $\sqcup \in \Gamma$ be the blank symbol. We have the rule:

$$\begin{aligned} FING(X_1,X_3) \quad :- \quad & first(X_1), succ(X_1,X_2), succ(X_2,X_3), \\ & q_\#(X_1), q_{(s,\sqcup)}(X_2), q_\#(X_3). \end{aligned}$$

Finger Moving. These rules move the fingers to pairs of corresponding positions. For all letters $a, b, c, d \in \Delta$ we have the rules:

$$\begin{aligned} FING(X_2,Y_2) \quad :- \quad & FING(X_1,Y_1), \\ & succ(X_1,X_2), succ(Y_1,Y_2), \\ & q_\#(X_1), q_a(X_2), q_\#(Y_1), q_b(Y_2). \end{aligned}$$

$$\begin{aligned} FING(X_2,Y_2) \quad :- \quad & FING(X_1,Y_1), \\ & succ(X_1,X_2), succ(Y_1,Y_2), \\ & q_a(X_1), q_c(X_2), q_b(Y_1), q_d(Y_2). \end{aligned}$$

$$\begin{aligned} FING(X_2,Y_3) \quad :- \quad & FING(X_1,Y_1), \\ & succ(X_1,X_2), succ(Y_1,Y_2), succ(Y_2,Y_3), \\ & q_a(X_1), q_\#(X_2), q_b(Y_1), q_c(Y_2), q_\#(Y_3). \end{aligned}$$

Note that if any rule of the first three types is ever used then the recursion terminates immediately, since *FING* is "flooded" by all pairs of elements. Such rules are called *flood-halt rules*.

Lemma 2.1 *If M diverges on the empty tape, then for any constant k there exists a database D such that $P^k(D) \neq P^\infty(D)$.*

Proof: Let $C_1, \ldots, C_k$ be the first k configurations in an infinite computation of M over the empty tape, where $C_i \in \Delta^i$ for $1 \leq i \leq k$. Let C be the string $\#C_1\#\ldots\#C_k\#$. Note that $C \in (\Delta')^m$, where $m = (k+1)(k+2)/2$, i.e., $C = a_1, a_2, \ldots, a_m$, where $a_i \in \Delta'$, for $1 \leq i \leq m$. Let the database D encode C. That is, D consists of the elements $\{1, \ldots, m\}$ with the following facts: $\mathit{first}(0)$, $\mathit{succ}(i, i+1)$ for $1 \leq i \leq m-1$, and $q_a(i)$ iff $a = a_i$ for $1 \leq i \leq m$. It is easy to see that the encoding, halting, and error detecting rules are never applied. The finger pointing rule generates the fact $\mathit{FING}(1,3)$. After that the finger moving rules are applied until the fact $\mathit{FING}(m-k-1, m)$ is generated. Note that every application of a finger moving rule increases the left argument of FING precisely by 1. Thus the last fact is generated after $m-k-1 = k(k+1)/2 > k$ rule applications. It follows that $P^k(D) \neq P^\infty(D)$. □

Lemma 2.2 *If M halts on the empty tape in k steps, then for any database D we have that $P^\infty(D) = P^{(k+2)(k+3)/2}(D)$.*

Proof: Suppose that D is a database such that $P^\infty(D) \neq P^{(k+2)(k+3)/2}(D)$. Clearly, that means neither the encoding rule nor the halting rule were applied, since if either of these were applied then we would have that $P^\infty(D) = P^1(D)$. Similarly, if an error detecting rule was applied, then it must have been the last rule to be applied. Thus, the first $(k+2)(k+3)/2$ rules that were applied must have been finger pointing or finger moving rules. It follows that the database contains elements $c_1, \ldots, c_m$, where $m = (k+2)(k+3)/2$, such that $\mathit{first}(c_1)$, $\mathit{succ}(c_i, c_{i+1})$ for $1 \leq i < m$, and for each i, there exists a unique $a_i \in \Delta'$ such that $q_{a_i}(c_i)$ for $1 \leq i \leq m$.

Let D' be the subset of D that consists only of the elements $c_1, \ldots, c_m$ and the above mentioned facts. We can think of D' as the encoding of a string $C = a_1 \ldots a_m \in (\Delta')^m$. Notice that none of the a_i's encode a halting state, since the halting rule was not applied. It follows that C cannot be a $(k+1)$-prefix of M, since M halts in k steps. Consequently C must contain an "error." In other words, if P is applied to D', then an error detecting rule would be applied within m steps. It follows that $P^\infty(D) = P^m(D)$—contradiction. □

From the above two lemmas we know that M halts on the empty tape if and only if P is bounded.

Theorem 2.3 *Program boundedness is undecidable for linear binary Datalog programs with a single IDB.*

3 Uniform Boundedness in the Presence of Negation

In this section, we consider the language Datalog$^{\neq}$, which has a "$\neq$" predicate with the obvious semantics (we assume that "$\neq$" is used only in the bodies of rules). We show that the presence of "$\neq$" is sufficient to make uniform boundedness undecidable even for monadic programs.

The proof is by reduction from the *Turing machine mortality problem*, which was shown to be undecidable in [H66]. This problem is defined as follows.

Consider a deterministic Turing machine M operating on a two-way infinite tape. Each stage in a computation of M can be described by a quadruple (l, s, r, q), where q is the current state of the finite state control, s is the symbol currently scanned, and l and r are *infinite* strings of symbols specifying the contents of the tape to the left and right of the head. Such quadruples are called ∞*-configurations* of M, and each ∞-configuration uniquely determines the entire computation of M starting in that ∞-configuration. Now, call M *mortal* if it has no infinite computations, i.e., if for every ∞-configuration (l, s, r, q), the computation of M starting at (l, s, r, q) must eventually terminate. (This is a stronger condition than saying that M halts on every input, since the computation may be started in an arbitrary state of M and the tape may contain an infinite number of non-blank symbols.) The *mortality problem* is the problem of deciding whether a given Turing machine is mortal.

Theorem 3.1 (Hooper 1966) *The mortality problem is undecidable.*

For technical reasons, we will be interested in the following seemingly stronger version of mortality: Call a Turing machine M *uniformly mortal* if there exists a constant ℓ such that M halts after at most ℓ steps when started in an arbitrary ∞-configuration. Obviously, every uniformly mortal Turing machine is mortal, but it turns out that the reverse implication is also true.

Theorem 3.2 *A Turing machine M is mortal if and only if it is uniformly mortal.*

Proof: Let $\delta_1, \ldots, \delta_n$ be the possible transitions of M. We call a sequence $\delta_{i_1}, \ldots, \delta_{i_k}$ of transitions *consistent* if it reflects a computation of M, i.e., if there exists a ∞-configuration (l, s, r, q) from which M will execute that sequence.

Now arrange all consistent transition sequences in a (possibly infinite) tree, with the empty sequence at the root and each node extending the sequence at its parent by one transition. This tree is of bounded degree. Also: (1) M is mortal iff there is no infinite path in the tree, and (2) M is uniformly mortal iff there are no arbitrarily long paths in the tree. By König's Lemma, these two conditions are equivalent. $\Box$

Using Theorem 3.2, one can easily see that mortality is recursively enumerable: It suffices to guess the constant c and simulate M on all c-length segments of tape, starting from all possible states.

Let us now see how the theorem can be applied to uniform Datalog boundedness. The undecidability proof works as follows: We construct, for any Turing machine M, a Datalog$^{\neq}$ program P that is uniformly bounded if and only if M is uniformly mortal. The program simulates M in such a way that long derivations of P correspond to long computations of M and vice versa. Hence, if M is uniformly mortal, the length of any derivation of P is bounded by a fixed constant, whereas if M is immortal, P will admit arbitrarily long derivations.

The simulation is essentially an exercise in *list processing* in a relational style. We use a ternary EDB predicate $cons(X, Y, Z)$ to express the fact that Z represents a list whose *car* is represented by X and whose *cdr* is represented by Y. Of course, X, Y, and Z will be single database constants—the *cons* fact merely makes Z behave as a list. To encode the machine states and tape symbols, say N in total, we use a set of unary EDB predicates $int_1(X), \ldots, int_N(X)$, with $int_i(X)$ stating that X represents the integer i. Given these predicates, we can encode machine configurations using a *monadic* IDB predicate *CONF*: a fact $CONF(C)$ states that C represents a five element list (t, l, s, r, q) called a *time-configuration*, where q is an integer encoding the state, s is an integer encoding the current symbol, l and r are lists of integers encoding the tape contents to the left and right of the head, and t is a "timestamp"—that is to say, a list whose length is incremented at each step. By including this timestamp, we are guaranteed to produce a new configuration every step even in case the simulated program is looping.

Of course there is no reason to assume that the *cons* and *int* predicates will be available or will behave as expected, because the input to the program is completely arbitrary. The program therefore includes, besides the rules simulating the transitions of M, a number of "checking rules" that ensure the consistency of the data used for the simulation. If a checking rule detects any "non-standard" behavior on the part of the database, the simulation will be immediately terminated by "flooding" the IDB predicate, i.e. deriving every possible fact within a single step.

We now describe the rules of P. Let M be given by its state space $Q = \{q_1, \ldots, q_n\}$, its tape alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$, and its transition relation $\Delta \subseteq Q \times \Sigma \times \{Left, Stay, Right\} \times Q \times \Sigma$. We assume that no two quintuples in Δ begin with the same two symbols and that M halts if it encounters a state/symbol combination for which no transition is defined. In a rule, we use $C = (T, L, S, R, Q)$ as a shorthand for $cons(T, U, C), cons(L, V, U), cons(S, W, V), cons(R, Q, W)$ with variables U, V, W that appear nowhere else in the program P.

The rule corresponding to a transition $(q_i, \sigma_k, Right, q_j, \sigma_l)$ is:

$$\begin{aligned} CONF(C') \;\;:\!\!-\;\; & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L', S', R', Q'), \\ & cons(S'', L, L'), cons(S', R', R), \\ & int_i(Q), int_j(Q'), int_k(S), int_l(S''), \\ & cons(Y, T, T'). \end{aligned}$$

Similarly, the rule corresponding to a transition $(q_i, \sigma_k, Left, q_j, \sigma_l)$ is:

$$\begin{aligned} CONF(C') \;\;:\!\!-\;\; & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L', S', R', Q'), \\ & cons(S'', R, R'), cons(S', L', L), \\ & int_i(Q), int_j(Q'), int_k(S), int_l(S''), \\ & cons(Y, T, T'). \end{aligned}$$

A transition $(q_i, \sigma_k, Stay, q_j, \sigma_l)$ is encoded as:

$$\begin{aligned} CONF(C') \;\;:\!\!-\;\; & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L, S'', R, Q'), \\ & int_i(Q), int_j(Q'), int_k(S), int_l(S''), \\ & cons(Y, T, T'). \end{aligned}$$

As mentioned above, it is also necessary to check the consistency of the database. This is achieved by the following additional rules, which make sure that no element of the database codes more than one integer and that each list has a unique *car* and *cdr* part:

$$\begin{aligned} \mathit{CONF}(C) &\ :\!-\ \ \mathit{int}_i(X), \mathit{int}_j(X). \quad (\text{for } 1 \leq i < j \leq N) \\ \mathit{CONF}(C) &\ :\!-\ \ \mathit{cons}(X, Y, Z), \mathit{cons}(X', Y', Z), X \neq X'. \\ \mathit{CONF}(C) &\ :\!-\ \ \mathit{cons}(X, Y, Z), \mathit{cons}(X', Y', Z), Y \neq Y'. \end{aligned}$$

Note that the variable C does not occur in the body of these rules. Therefore, if a checking rule fires, every possible *CONF* fact can be derived in a single step and the program will obviously be bounded.

We now claim that P is uniformly bounded if and only if M is uniformly mortal. One direction of this claim is easy to see:

Lemma 3.3 *If M is not uniformly mortal, then P is not uniformly bounded.*

Proof: Let D be the infinite database (infinite set of ground atoms over the EDB predicates) whose infinite set of constants is $U :=$ the closure of the set $\{1, \ldots, N\}$ under the operation of taking ordered pairs. D contains the following facts: $\mathit{int}_i(i)$ for $1 \leq i \leq N$, $\mathit{cons}(x, y, \langle x, y \rangle)$ for all $x, y \in U$, and $\mathit{CONF}(\langle 1, x \rangle)$ for all $x \in U$. Note that the definition of $P^k(D)$ and $P^\infty(D)$ apply even when D is infinite.

Clearly, none of the checking rules applies to D. All finite computations of M can be simulated within D, because there are sufficient integers to encode tape symbols and machine states, sufficient *cons* facts to encode all finitely nested lists of these integers, and sufficient *CONF* facts to encode all time-configurations of M with a "timestamp" value of 1. Since a new "timestamp" value is generated at each simulation step, the *CONF* facts generated during a simulation are all different. Thus, the immortality of M implies that $P^k(D) \neq P^\infty(D)$ for all $k \geq 1$. By choosing only the EDB facts occurring in a single derivation of height $k + 1$, plus the initial *CONF* fact, one can then obtain, for any integer $k \geq 1$, a finite subset D' of D such that $P^k(D') \neq P^\infty(D')$. □

For the reverse direction, we have to prove that, no matter what the input database contains, every derivation of P mirrors some computation of M (unless a checking rule is triggered). We introduce some notation first.

Let $\delta_1, \ldots, \delta_n$ be the transitions of M and $\rho_1, \ldots, \rho_n$ be the corresponding rules of P. Recall that a sequence $\delta_{i_1}, \ldots, \delta_{i_k}$ of transitions is *consistent* if there exists a ∞-configuration (l, s, r, q) from which M will execute that sequence. By linearity of the program P, every derivation consists of a sequence of rule applications $\rho_{i_1}, \ldots, \rho_{i_k}$. We call such a sequence of rule applications *standard* if the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$ is consistent.

Lemma 3.4 *If M is uniformly mortal, then the length of standard derivations of P is bounded by a constant ℓ independent of the input database.*

Proof: No standard derivation can be longer than the longest computation of M. □

Note that any derivation, standard or non-standard, defines a unique sequence of head movements on the tape via the $\{Left, Stay, Right\}$ components in its corresponding sequence of transitions. We want to establish that whenever during the course of a derivation the head goes back to a cell where it has already been before, the symbol representing its contents is the same as the symbol used when the cell was written—unless a checking rule fires. This is done in Lemma 3.6, for which the following lemma is a preliminary.

Lemma 3.5 *Let $\rho_{i_1}, \ldots, \rho_{i_k}$ be a derivation such that in the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$, the head scans the same tape cell c at the beginning of the first and last transitions and does not visit any cell to the left (resp., right) of c in between. If the checking rules of P do not apply, then the database constants used to instantiate the L (resp., R) variables in the first and last rule applications are the same.*

Proof: Assume that the checking rules of P do not apply. Let us denote by θ_j the database constant used to instantiate some variable Θ in the jth rule application. Thus, l_1 and l_k (resp., r_1 and r_k) are the database constants used to instantiate the L (resp., R) variables in the first and last rule applications. We prove the claim for l_1 and l_k, the claim for r_1 and r_k then follows by symmetry.

We proceed by induction on k. The claim is trivially true if $k = 1$.

If $k = 2$, ρ_{i_1} must have been a "stay" transition and the database must contain the following facts:

$$cons(t_1', u_1', c_1'), cons(l_1, v_1', u_1'), cons(s_1'', w_1', v_1'), cons(r_1, q_1', w_1'),$$
$$cons(t_2, u_2, c_2), cons(l_2, v_2, u_2), cons(s_2, w_2, v_2), cons(r_2, q_2, w_2),$$

where the u's, v's, and w's instantiate the variables hidden in the $C = (T, L, S, R, Q)$ notation.

Since the output of a rule is the input of the next rule, $c_1' = c_2$. Thus, by the third checking rule, $u_1' = u_2$, which, by the second checking rule, implies that $l_1 = l_2$.

If $k > 2$ and the head does not scan cell c at the beginning of any of the transitions $\delta_{i_2}, \ldots, \delta_{i_{k-1}}$, we can apply the induction hypothesis to $\rho_{i_2}, \ldots, \rho_{i_{k-1}}$ and infer that $l_2 = l_{k-1}$. Similar to the $k = 2$ case, we also have that $l_1' = l_2$ and hence $l_1' = l_{k-1}$. Now, since ρ_{i_1} and $\rho_{i_{k-1}}$ must have been "right" and "left" transitions, respectively, the database must contain $cons(s_1'', l_1, l_1')$ and $cons(s_{k-1}', l_{k-1}', l_{k-1})$ facts. By the third checking rule, then, $l_1 = l_{k-1}'$. Again similar to the $k = 2$ case, we infer that $l_{k-1}' = l_k$ and hence $l_1 = l_k$.

If $k > 2$ and there is some $j \in \{2, \ldots, k-1\}$ such that the head scans cell c at the beginning of transition δ_{i_j}, we can apply the induction hypothesis to $\rho_{i_1}, \ldots, \rho_{i_j}$ and $\rho_{i_j}, \ldots, \rho_{i_k}$ and infer that $l_1 = l_j = l_k$. □

Lemma 3.6 *Let $\rho_{i_1}, \ldots, \rho_{i_k}$ ($k \geq 2$) be a derivation such that in the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$, the head scans the same tape cell c at the beginning of the first and last transitions and does not scan this cell at the beginning of any other transition. If the checking rules of P do not apply, then the database constant used to instantiate the S'' variable in the first rule application is the same as the database constant used to instantiate the S variable in the last rule application.*

Proof: If $k = 2$, then δ_{i_1} must have been a "stay" transition and the claim follows from an argument similar to the $k = 2$ case of Lemma 3.5.

If $k > 2$, then ρ_{i_1} and $\rho_{i_{k-1}}$ must have been "right" and "left" transitions, respectively, (or vice versa, in which case the proof is symmetric) and Lemma 3.5 can be applied to $\rho_{i_2}, \ldots, \rho_{i_{k-1}}$. This shows that $l_2 = l_{k-1}$, unless a checking rule applies. We also have, similar to the $k = 2$ case of Lemma 3.5, that $l'_1 = l_2$ and hence $l'_1 = l_{k-1}$, if no checking rule applies. Because of the first and $(k-1)$th transition, the database must contain facts $cons(s''_1, l_1, l'_1)$ and $cons(s'_{k-1}, l'_{k-1}, l_{k-1})$. This implies, by the second checking rule, that $s''_1 = s'_{k-1}$. Using again the $k = 2$ argument of Lemma 3.5, it follows that $s'_{k-1} = s_k$ and therefore finally $s''_1 = s_k$, unless a checking rule applies. □

Lemma 3.7 *Let $\rho_{i_1}, \ldots, \rho_{i_k}$ be a derivation. Then either a checking rule applies or for each $j \in \{1, \ldots, k-1\}$, the database constant used to instantiate the Q' variable in ρ_{i_j} is equal to the database constant used to instantiate the Q variable in $\rho_{i_{j+1}}$.*

Proof: Similar to the case $k = 2$ of Lemma 3.5. □

Lemma 3.8 *If the checking rules of P do not apply, every derivation is standard.*

Proof: Let $\rho_{i_1}, \ldots, \rho_{i_k}$ be a derivation and $\delta_{i_1}, \ldots, \delta_{i_k}$ be the corresponding sequence of transitions. Assume that the checking rules of P do not apply. Then by Lemma 3.7 and the first checking rule (which ensures that no database constant can encode two different integers), it cannot happen that a transition δ_{i_j} conflicts with the state of M after transition $\delta_{i_{j-1}}$. By Lemma 3.6 and the first checking rule, it is also impossible that a transition δ_{i_j} where the head scans tape cell c conflicts with the value that was last written into c. Therefore, the sequence $\delta_{i_1}, \ldots, \delta_{i_k}$ must be consistent and thus $\rho_{i_1}, \ldots, \rho_{i_k}$ is standard. □

Lemma 3.9 *If M is uniformly mortal, then P is uniformly bounded.*

Proof: By Lemma 3.4 and Lemma 3.8. □

This completes the proof that the reduction works. Applying Theorems 3.1 and 3.2, we conclude:

Theorem 3.10 *Uniform boundedness is undecidable for linear monadic Datalog$^{\neq}$ programs with a single IDB predicate.*

4 Uniform Boundedness for Ternary Datalog

Our goal here is to refine the simulation of the preceding section so that the $\neq$ predicate is no longer needed and thus an undecidability result for plain Datalog is obtained. This can be done by increasing the arity of the *CONF* predicate to three.

The idea is as follows: Since $\neq$ is not available anymore, we cannot guarantee that *cons* will be a one-to-one function. In other words, a database constant representing a list may have several different database constants representing its *car* or *cdr* part. Therefore, when we *cons* the T', L', S', R', Q' values at the end of one rule application and then split the result into T, L, S, R, Q in the subsequent rule application, we may not see the same values again. However, our simulation will still be faithful as long as all *car*'s and *cdr*'s of the same constant "look the same" with respect to the int_i predicates, since that is the only information we ever use of a constant.

Let us call two database constants x and x' *equivalent* (denoted as $x \sim x'$) if either: (1) x equals x', or (2) x and x' occur as the *car* parts of two equivalent elements z and z', or (3) x and x' occur as the *cdr* parts of two equivalent elements z and z', or (4) x and x' are both equivalent to some element y.

What we need, then, for our simulation to work is that two equivalent database constants cannot represent two different integers. In fact, this needs to be true only for those database constants actually used in a simulation. The way we enforce this condition is by using two additional arguments of $CONF$ to compute all relevant pairs of equivalent values "on the fly" and fail the simulation if equivalent values are found that correspond to different integers.

The new simulation program P uses the same predicates as the old one, except that we now use a ternary predicate $CONF(C, X_1, X_2)$, where C encodes a time-configuration of M and X_1, X_2 is a pair of values proven to be equivalent.

The rules of P are of three kinds. First, there is a rule to initialize an equivalence computation:

$$CONF(C, X, X) \quad :\!\!- \quad CONF(C, X_1, X_2).$$

Then, for each pair $1 \leq i < j \leq N$, there is a checking rule

$$CONF(U, V, W) \quad :\!\!- \quad CONF(C, X_1, X_2), int_i(X_1), int_j(X_2).$$

Finally there are the transition rules. Each one comes in three flavors: one to do the transition and propagate the equivalence along *car* parts, one to do the transition and propagate the equivalence along *cdr* parts, and one to do the transition and propagate the equivalence transitively. For a typical transition $(q_i, \sigma_k, Right, q_j, \sigma_l)$, the three rules are:

$$\begin{aligned} CONF(C', X_1', X_2') \quad :\!\!- \quad & CONF(C, X_1, X_2), \\ & cons(X_1', Y_1, X_1), cons(X_2', Y_2, X_2), \\ & /\!*\ \text{Clauses for } C,\ C' \text{ as in Section 3 } *\!/ \end{aligned}$$

$$\begin{aligned} CONF(C', X_1', X_2') \quad :\!\!- \quad & CONF(C, X_1, X_2), \\ & cons(Y_1, X_1', X_1), cons(Y_2, X_2', X_2), \\ & /\!*\ \text{Clauses for } C,\ C' \text{ as in Section 3 } *\!/ \end{aligned}$$

$$\begin{aligned} CONF(C', X_1, X_3) \quad :\!\!- \quad & CONF(C, X_1, X_2), CONF(C, X_2, X_3), \\ & /\!*\ \text{Clauses for } C,\ C' \text{ as in Section 3 } *\!/ \end{aligned}$$

Note that there is no interaction between the checking variables and the configuration variable. This makes it possible to "piggyback" any desired equivalence computation onto a sufficiently long derivation path.

We now follow a proof outline similar to that of the previous section. Due to the non-linear rules, however, we will reason about paths in derivation trees. If δ_i is a transition of M, we number the corresponding three rules of P by $\rho_{i,1}, \rho_{i,2}, \rho_{i,3}$. A path in a derivation tree involving a sequence of rule applications $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ is called *standard* if the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$ is consistent. Note that *any* path defines a sequence of head movements on the tape, via the $\{Left, Stay, Right\}$ components in its corresponding sequence of transitions.

Lemma 4.1 *If M is not uniformly mortal, then P is not uniformly bounded.*

Proof: Same as for Lemma 3.3, except that the $CONF$ facts in D will now be of the form $CONF(\langle 1, x\rangle, y, y)$, $x, y \in U$. □

Lemma 4.2 *If M is uniformly mortal, then the length of any standard path in a derivation of P is bounded by a constant ℓ independent of the input database.*

Proof: No standard path can be longer than the longest computation of M. □

Lemma 4.3 *Let $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ be a path in a derivation tree such that in the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$, the head scans the same tape cell c at the beginning of the first and last transitions and does not visit any cell to the left (resp., right) of c in between. Then the database constants used to instantiate the L (resp., R) variables in the first and last rule applications are equivalent.*

Proof: The proof is very similar to the induction in Lemma 3.5, with "equivalence" substituted for "equality." The only difference is that where we used the non-applicability of the checking rules to propagate equality to *car* and *cdr* parts, we now use the definition of equivalence instead.

Let us again denote by θ_j the database constant used to instantiate some variable Θ in the jth rule application. Thus, l_1 and l_k (resp., r_1 and r_k) are the database constants used to instantiate the L (resp., R) variables in the first and last rule applications. We prove the claim for l_1 and l_k, the claim for r_1 and r_k then follows by symmetry.

We proceed by induction on k. The claim is trivially true if $k = 1$.

If $k = 2$, ρ_{i_1,j_1} must have been a "stay" transition and the database must contain the following facts:

$$cons(t_1', u_1', c_1'), cons(l_1, v_1', u_1'), cons(s_1'', w_1', v_1'), cons(r_1, q_1', w_1'),$$
$$cons(t_2, u_2, c_2), cons(l_2, v_2, u_2), cons(s_2, w_2, v_2), cons(r_2, q_2, w_2),$$

where the u's, v's, and w's instantiate the variables hidden in the $C = (T, L, S, R, Q)$ notation.

Since the output of a rule is the input of the next rule, $c_1' = c_2$. By rule (3) in the definition of "$\sim$", then, $u_1' \sim u_2$ and by rule (2), $l_1 \sim l_2$.

If $k > 2$ and the head does not scan cell c at the beginning of the transitions $\delta_{i_2}, \ldots, \delta_{i_{k-1}}$, we can apply the induction hypothesis to $\rho_{i_2,j_2}, \ldots, \rho_{i_{k-1},j_{k-1}}$ and infer that $l_2 \sim l_{k-1}$. Similar to the $k = 2$ case, we also have that $l_1' \sim l_2$. Thus, by equivalence rule (4), $l_1' \sim l_{k-1}$. Now, since δ_{i_1} and $\delta_{i_{k-1}}$ must have been "right" and "left" transitions, respectively, the database must contain $cons(s_1'', l_1, l_1')$ and $cons(s_{k-1}', l_{k-1}', l_{k-1})$ facts. Hence $l_1 \sim l_{k-1}'$, by equivalence rule (3). An argument similar to the $k = 2$ case above gives $l_{k-1}' \sim l_k$ and applying equivalence rule (4) again we conclude that $l_1 \sim l_k$.

If $k > 2$ and there is some $m \in \{2, \ldots, k-1\}$ such that the head scans cell c at the beginning of transition δ_{i_m}, we can apply the induction hypothesis to $\rho_{i_1,j_1}, \ldots, \rho_{i_m,j_m}$ and $\rho_{i_m,j_m}, \ldots, \rho_{i_k,j_k}$ and infer that $l_1 \sim l_m$ and $l_m \sim l_k$. By equivalence rule (4), $l_1 \sim l_k$. $\Box$

Lemma 4.4 *Let $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ $(k > 2)$ be a path in a derivation tree such that in the corresponding sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_k}$, the head scans the same tape cell c at the beginning of the first and last transitions and does not scan this cell at the beginning of any other transition. Then the database constant used to instantiate the S'' variable in the first rule application is equivalent to the database constant used to instantiate the S variable in the last rule application.*

Proof: If $k = 2$, then δ_{i_1} must have been a "stay" transition and the claim follows from an argument similar to the $k = 2$ case of Lemma 4.3.

If $k > 2$, then δ_{i_1} and $\delta_{i_{k-1}}$ must have been "right" and "left" transitions, respectively, (or vice versa, in which case the proof is symmetric) and Lemma 4.3 can be applied to $\rho_{i_2,j_2}, \ldots, \rho_{i_{k-1},j_{k-1}}$. This shows that $l_2 \sim l_{k-1}$. We also have, similar to the $k = 2$ case of Lemma 4.3, that $l'_1 \sim l_2$ and hence $l'_1 \sim l_{k-1}$. Because of the first and $(k-1)$th transition, the database must contain facts $cons(s''_1, l_1, l'_1)$ and $cons(s'_{k-1}, l'_{k-1}, l_{k-1})$. Applying equivalence rule (2), we obtain $s''_1 \sim s'_{k-1}$. We also have, again similar to the $k = 2$ case of Lemma 4.3, that $s'_{k-1} \sim s_k$. By equivalence rule (4), then, $s''_1 \sim s_k$. $\Box$

Lemma 4.5 *Let $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ be a path in a derivation tree. Then for $1 \leq m < k$, the database constant used to instantiate the Q' variable in ρ_{i_m,j_m} is equivalent to the database constant used to instantiate the Q variable in $\rho_{i_{m+1},j_{m+1}}$.*

Proof: Similar to the case $k = 2$ of Lemma 4.3. $\Box$

Lemma 4.6 *Let D be a database and let $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ be a path in a derivation tree of P on D. If this path is nonstandard, then there are two elements $x_1, x_2 \in D$ such that $x_1 \sim x_2$ and $int_i(x_1)$, $int_j(x_2)$ with $i \neq j$.*

Proof: Let $\delta_{i_1}, \ldots, \delta_{i_k}$ be the sequence of transitions corresponding to $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$. Assume that for any pair $x_1, x_2 \in D$ with $x_1 \sim x_2$, we have $int_i(x_1)$, $int_j(x_2)$ only if $i = j$. Then, by Lemma 4.5, it cannot happen that a transition δ_{i_j} conflicts with the state of M after transition $\delta_{i_{j-1}}$, and, by Lemma 4.4, it is also impossible that a transition δ_{i_j}, where the head scans tape cell c, conflicts with the value that was last written into c. Therefore, the sequence $\delta_{i_1}, \ldots, \delta_{i_k}$ must be consistent and thus $\rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ is standard. $\Box$

Lemma 4.7 *If M is uniformly mortal, then P is uniformly bounded.*

Proof: Let ℓ be an upper bound on the length of computations of M and let D be any database. Consider a path $\pi := \rho_{i_1,j_1}, \ldots, \rho_{i_k,j_k}$ of length $k > \ell$ in a derivation tree of P on D. This path must contain an inconsistency among its first $\ell + 1$ transitions. Thus, by Lemma 4.6, there are two elements $x_1, x_2 \in D$ such that $x_1 \sim x_2$ and $int_i(x_1)$, $int_j(x_2)$ with $i \neq j$. Let ρ_{i_m,j_m} be the rule application where x_1 occurs and ρ_{i_n,j_n} $(m < n \leq \ell + 1)$ be the rule application where x_2 occurs.

As can be seen from the induction in Lemma 4.3, the number of equivalence rule applications needed to prove the equivalence of x_1 and x_2 depends only on $n - m$, but not on the input database. Since $n - m \leq \ell$, there is a constant C_ℓ depending only on ℓ such that if the length of the path π is at least C_ℓ, the proof of $x_1 \sim x_2$ can be "piggybacked" onto π, which will produce a fact $\mathit{CONF}(u, x_1, x_2)$ for some time-configuration u. Since $\mathit{int}_i(x_1), \mathit{int}_j(x_2)$ with $i \neq j$, a checking rule will trigger, so that the recursion depth of P on D is at most $C_\ell + 1$. If there is no path longer than $C_\ell - 1$, the inconsistency might go undetected, but then clearly $P^\infty(D) = P^{C_\ell - 1}(D)$. Hence, on any database the recursion depth of P is at most $C_\ell + 1$. □

Theorem 4.8 *Uniform boundedness is undecidable for ternary Datalog programs with a single IDB.*

5 Queries with two linear recursions

This and the next section focus on programs with a small number of rules. As our first result, we show that program boundedness is undecidable for query programs having two linear recursive rules and one initialization rule.

The reduction is from the halting problem for 2-counter machines (2CM), and follows the basic outline of the proof in [GMSV87]. The technical improvement is the polymorphic encoding of the entire transition function of the 2CM by a single linear recursion.

Given a 2CM M, we construct a Datalog program P simulating M, such that M halts iff P is bounded. The program has three rules: one *initialization rule* that simulates the initial configuration of M, one *transition rule* simulating state transitions, and one *halting rule* which guarantees a bounded fixpoint in the case of an accepting computation. We construct P so that every IDB fact appearing in the fixpoint will have a proof involving the initialization rule, zero or more applications of the transition rule, and a possible final use of the halting rule.

We simulate configurations of M by a single IDB relation $\mathit{ID}(H, T, F, S, C_1, C_2)$, which should be read informally as "at time H, with T and F coding *true* and *false*, M was in state S, with C_1 and C_2 the values of the counters." Since IDB facts in the fixpoint can be associated with derivation trees, the variable H can be thought of as encoding the height of a tree; this more liberal interpretation will be used further in the next section. The variables T and F are included because the simulation of M will be founded on a database representation of Boolean logic. The EDB relations in the database can be divided into three groups.

Logic relations: We include EDB relations $\mathit{not}(X, Y)$, $\mathit{and}(X, Y, Z)$, and $\mathit{or}(X, Y, Z)$, to be read informally as "the negation of X is Y," "the conjunction of X and Y is Z," and "the disjunction of X and Y is Z."

State relation: We have an EDB $\mathit{state}(S, B_1, \ldots, B_k)$ encoding the states of M, where S is the "name" of the state, and the B_i code it as a binary string, using F and T as 0 and 1. The states of M are $\{0, 1, \ldots, h = 2^k - 1\}$, where 0 is the initial state and h the halting state. We typically write the initial state as $\mathit{state}(S_0, F, F, \ldots F)$ and the halting state as $\mathit{state}(S_h, T, T, \ldots T)$.

Arithmetic relations: Finally, we allow two EDB relations for *counting* over a database representation of integers. To encode "X is zero," we write $zero(T, X)$, and to encode "X is nonzero," we write $zero(F, X)$. For successor and predecessor, we write $order(T, X, Y)$ or $order(F, Y, X)$ to mean "Y is the successor of X." The importance of this expressiveness is that we will require a *syntactically uniform* way of encoding whether counter i of M is incremented or decremented in its transition from C_i to C_i'. By writing $order(B, C_1, C_1')$, we then need only to compute a Boolean value B: if B is true, the counter increases, otherwise it decreases. The arithmetic relations are called *signed predicates*, since their meaning is encoded by the "sign" of the Boolean prefix.

Observe that we have casually referred to database constants and relations as being "true," "conjunction," and so on, when in fact there is no *prima facie* reason why there should be any fidelity on the part of the database to our name calling. It is entirely possible for $and(X, Y, Z)$ to encode any ternary relation. Part of the role of the query program P, then, is to enforce the fidelity we desire. We shall refer to a computation as *standard* when it conforms to our designated functionality for the EDB relations, and refer otherwise to it as *non-standard*.

We now describe how the computation of M is simulated by the three-rule query program.

Initialization rule: This rule encodes the initial state of M:

$$\begin{aligned}
&ID(Z, T, F, S_0, Z, Z) :\!\!-\\
&\quad not(T, F), not(F, T),\\
&\quad and(T, T, T), and(T, F, F), and(F, T, F), and(F, F, F),\\
&\quad or(T, T, T), or(T, F, T), or(F, T, T), or(F, F, F),\\
&\quad state(S_0, F, F, \ldots, F), \ldots, state(S_h, T, T, \ldots, T),\\
&\quad zero(T, Z).
\end{aligned}$$

The body of the rule ensures that the Boolean logic and the states of M are "all there" in the database—otherwise, computation cannot begin, and P is bounded. The rule further ensures that Z is zero, and that S_0 is a "tag" naming the initial state.

Transition rule: We have one linear recursion encoding *all* the transitions of M:

$$\begin{aligned}
&ID(H', T, F, S', C_1', C_2') :\!\!-\\
&\quad ID(H, T, F, S, C_1, C_2),\\
&\quad zero(F, H'), order(T, H, H'), order(F, H', H),\\
&\quad zero(Z_1, C_1), zero(Z_2, C_2), state(S, B_1, \ldots, B_k),\\
&\quad order(P_1, C_1, C_1'), order(P_2, C_2, C_2'), state(S', B_1', \ldots, B_k'),\\
&\quad \Phi[B_1, \ldots, B_k, Z_1, Z_2] \mapsto [B_1', \ldots, B_k', P_1, P_2].
\end{aligned}$$

The old state of M is encoded as $ID(H, T, F, S, C_1, C_2)$, and the next state is encoded as $ID(H', T, F, S', C_1', C_2')$. The first three lines of the rule body represent "inputs" to the computation; the fourth line (and the rule head) represents the "output" from the computation.

Observe that both IDB predicates in the rule head and body share T and F as the encoding of true and false. In any repeated use of the transition rule, then, T and F are *persistent* and serve as *constants*. The choice of T and F is determined by the initialization rule, which ensures that the requisite state and logic encodings exist in the initial database.

The EDB subgoals $zero(F, H')$, $order(T, H, H')$, and $order(F, H', H)$ serve to perform a sort of transitive closure on the "integers" in the database, where this transitive closure is *oblivious* to the computed configurations, serving to "make more successors." The computation can only continue if a long enough successor chain is found. Notice that this chain is "doubly linked," since $H' = H + 1$ is encoded as $order(T, H, H')$ ("the successor of H is H'"), and $order(F, H', H)$ ("the predecessor of H' is H").

In the third line of the body, we see variables $Z_1, Z_2, B_1, \ldots, B_k$ which do not appear in either IDB predicate. As such, they are instantiated "existentially" from the database. In a standard computation, they are instantiated either to the bindings for T or F; as such, Z_i is T iff counter i is zero, and the B_i encode the current state in binary using T and F.

The fourth line of the body sets up the outputs, where in a standard computation, P_i is T if counter i is increased and F if counter i is decreased, and the B'_i and S' represent the new state with a tag and binary encoding.

The last line represents a *Boolean function* defining the transition function of M, from the *input variables* $B_1, B_2, \ldots, B_k, Z_1, Z_2$ to the *output variables* $B'_1, B'_2, \ldots, B'_k, P_1, P_2$. The value of each Boolean output variable is merely a Boolean function of the input variables.

For example, suppose that M shifts into state 3 precisely when it was in state 4 and the first counter was zero, or in state 5 when the second counter was nonzero. Suppose as well that X_i is a Boolean value indicating whether or not M is in state i, and X'_i codes whether M is in state i after one transition. We then write X'_3 as the function $(X_4 \wedge Z_1) \vee (X_5 \wedge \neg Z_2)$. Of course, we realize this logical formula in *relational style* using the logic that has been built into the query program, by adding the following subgoals to the transition rule:

$$and(X_4, Z_1, G_1), not(Z_2, G_2), and(X_5, G_2, G_3), or(G_1, G_3, X'_3).$$

In this coding, the G_i are new logic variables that we think of as being existentially quantified from constants of the EDB. Each G_i represents the output of a particular *logic gate* realized via the Boolean relations. If we need to realize a circuit with *no gates* (for instance, if currently in state 6, shift into state 7), we do so by *double negation*, adding the subgoals $not(X_6, G), not(G, X'_7)$.

Left unexplained in this example is how to proceed from a *binary* encoding of the state (using the B_i) to a *unary* encoding via the X_j. Of course, this too is mere circuitry, realized in the same relational style. Hardware designers use such circuitry (and its reverse, from unary to binary coding) as the building block of multiplexors and demultiplexors.

Halting rule:

$$ID(U, V, W, X, Y, Z) \quad :\!- \quad ID(H, T, F, S_h, C_1, C_2), state(S_h, T, T, \ldots, T).$$

This rule floods the fixpoint if an IDB fact in the database encodes a halting state. It is the only rule in P that is not connected.

We now show that if M diverges, then for any integer $k \geq 0$, there exists a database D_k such that $P^{k-1}(D_k) \neq P^k(D_k)$, so that P is not bounded.

Definition 5.1 *The* standard database D_k *has constants* $\{t, f, s_0, s_1, \ldots, s_h, 0, 1, 2, \ldots, k\}$ *and the following EDB facts:*

$$\begin{aligned}
&not(t, f), not(f, t),\\
&\text{and}(t, t, t), \text{and}(t, f, f), \text{and}(f, t, f), \text{and}(f, f, f),\\
&or(t, t, t), or(t, f, t), or(f, t, t), or(f, f, f),\\
&state(s_0, f, \ldots, f, f), state(s_1, f, \ldots, f, t), \ldots, state(s_h, t, \ldots, t, t),\\
&zero(t, 0), zero(f, 1), \ldots, zero(f, k),\\
&order(t, 0, 1), order(t, 1, 2), \ldots, order(t, k-1, k),\\
&order(f, 1, 0), order(f, 2, 1), \ldots, order(f, k, k-1).
\end{aligned}$$

Lemma 5.2 *If M diverges, then for any constants s, c_1, c_2, every proof of $ID(k, t, f, s, c_1, c_2)$ over D_k has height k.*

Proof: The database forces the computation to be standard, and the halting rule cannot be used. Hence the height variable in any proof is initialized to 0, and can only increase by one at each step. □

Lemma 5.3 *If M halts in h steps, then $P^{h+2}(D_h) = P^{\infty}(D_h)$.*

Proof: By the halting rule. □

Lemma 5.4 *If M halts in h steps, then $P^{h+2}(D) = P^{\infty}(D)$ for* any *database D of EDB facts.*

Proof: Suppose there exists an IDB fact I, where I has a proof Π of height greater than $h+2$; we show another proof exists of height at most $h+2$. The proof Π can only use the halting rule as the last step, otherwise a shorter proof can be easily found, since the head of the halting rule can be instantiated to I. Therefore, Π begins by using the initialization rule, and follows with at least h uses of the transition rule.

Let $d_0, d_1, \ldots, d_h$ be the database constants used to instantiate H in Π (with possible repetitions), and c_t, c_f be the constants used to instantiate T and F. Observe that under the mapping $t \mapsto c_t$, $f \mapsto c_f$, and $i \mapsto d_i$ $(0 \leq i \leq h)$, we have an embedding ϕ of D_h in D. Then for any IDB fact I', from $I' \in P^{h+1}(D_h)$ it follows that $\phi(I') \in P^{h+1}(D)$. Hence, $ID(d_h, c_t, c_f, s, a, b) \in P^{h+1}(D)$ where $state(s, c_t, \ldots, c_t) \in D$. Because of the use of the halting rule, $I \in P^{h+2}(D)$. □

We remark that the embedding ϕ described in the above proof is in fact a *containment mapping* (see [U89]) between conjunctive queries: any $(h+2)$-fold *unwinding* of P into a conjunctive query can be mapped into any t-fold unwinding, for $t \geq h+2$. The construction of the containment mapping is essentially given in the above proof: it identifies T, F, and state names in both queries, and appropriately maps variables denoting *counter values* and proof height in the $(h+2)$-fold unwinding to variables of the (H-defined) chain in the t-fold unwinding.

Theorem 5.5 *Program boundedness is undecidable for query programs having two linear recursive rules and one initialization rule.*

Proof: By Lemma 5.2 and 5.4. □

6 Queries with one linear recursion and a projection

We now refine the proof of the previous section, exchanging one linear recursion—the halting rule—for a (connected) projection.

In order to eliminate the halting rule, we add a variable Z to the predicate ID, encoding the constant zero, and we increase the arity of the signed predicate *order*. Using the constants T and F to sign the predicate, we adopt the following "standard" interpretation:

$order(T, T, X, Y)$	"Y is the successor of X"
$order(T, F, X, Y)$	"Y is the predecessor of X"
$order(F, T, X, Y)$	"Y is a zero *reachable* from X"
$order(F, F, X, Y)$	"X is a zero *reachable* from Y"

Before plunging into detail, we broadly sketch how the new reduction will work, using the above modifications. Every constant in the database used to instantiate a counter variable will be linked to its predecessor as before, but *also* to zero via the relations $order(F, T, -, -)$ and $order(F, F, -, -)$.

We simulate M *as if* its halting state h has the following unusual property: if M enters state h for the first time, it simultaneously reduces both counters to zero. In addition, M may then *reenter* state h, and reset its counters to *any* two values reached by *either* counter in the course of the computation. It should be clear that the essential nature of the halting problem is not changed: either M reaches the designated state h, or diverges with unbounded counters. The simulation uses the new definition of *order* to realize this zeroing and resetting of counters.

In a standard computation, if M has not reached state h, P simulates the counters of M using $order(T, -, -, -)$, a *ternary* relation interpreted exactly like the definition of *order* in Section 5, which codes the relations between successive integers. To enter state h, zero the counters, and arbitrarily reset them, P simulates M using the ternary relation $order(F, -, -, -)$, which codes the relations between all the integers and zero. We now present the Datalog simulation of M, underlining the changes from the previous version.

Initialization rule:

$$
\begin{aligned}
&ID(Z, T, F, \underline{Z}, S_0, Z, Z) :\!-\\
&\quad \underline{order(F, T, Z, Z), order(F, F, Z, Z)},\\
&\quad not(T, F), not(F, T),\\
&\quad and(T, T, T), and(T, F, F), and(F, T, F), and(F, F, F),\\
&\quad or(T, T, T), or(T, F, T), or(F, T, T), or(F, F, F),\\
&\quad state(S_0, F, F, \ldots, F), \ldots, state(S_h, T, T, \ldots, T),\\
&\quad zero(T, Z).
\end{aligned}
$$

Transition rule:

$$
\begin{aligned}
&ID(H', T, F, \underline{Z}, S', C_1', C_2') :\!-\\
&\quad ID(H, T, F, \underline{Z}, S, C_1, C_2),
\end{aligned}
$$

$$\begin{aligned}
&zero(F,H'), order(T,T,H,H'), order(T,F,H',H),\\
&\underline{order(F,T,H',Z), order(F,F,Z,H')},\\
&\underline{order(F,T,C_1',Z), order(F,F,Z,C_1')},\\
&\underline{order(F,T,C_2',Z), order(F,F,Z,C_2')},\\
&zero(Z_1,C_1), zero(Z_2,C_2), state(S,B_1,\ldots,B_k),\\
&order(P_1,\underline{Q_1},C_1,C_1'), order(P_2,\underline{Q_2},C_2,C_2'), state(S',B_1',\ldots,B_k'),\\
&\underline{\Psi}[B_1,\ldots,B_k,Z_1,Z_2] \mapsto [B_1',\ldots,B_k',P_1,P_2,\underline{Q_1,Q_2}].
\end{aligned}$$

Since M may in state h reset the counters to many possible values, the transition map is no longer a *function*. Nevertheless, the computation of output variables P_i and Q_i is still entirely functional.

For instance, given the definition of *order*, P_1 should (in a standard computation) be false when M is about to enter state h, resetting the counters to zero, *or reentering* state h, setting the counters to any earlier value:

$$P_1 = \neg(B_1' \wedge B_2' \cdots \wedge B_k').$$

In addition, Q_1 is similarly false when P_1 is false and M is reentering state h, *or* P_1 is true, and M is incrementing the first counter. We abbreviate the logic formula of the latter incrementing as *push*:

$$Q_1 = \neg((\neg P_1 \wedge B_1 \wedge \cdots \wedge B_k) \vee (P_1 \wedge push)).$$

Again, we code the P_i and Q_i as subgoals in relational style, always introducing new logic variables to represent the output of Boolean logic gates.

Finally, instead of a halting rule, we have the following

Projection rule:

$$I(C_1,C_2) \quad :\!\!- \quad ID(H,T,F,Z,S,C_1,C_2).$$

Note that the resulting program is the first program in this paper with more than one IDB predicate. We take here the predicate I/O convention, where I is the output predicate. This means that $P^k(D)$ is the projection on I of the set of atoms that have derivation trees of height at most k. Because not all IDB predicates are output predicates, the undecidability result that we prove here applies only to predicate boundedness.

Let D_k now denote the standard database as in the previous section, with the modification of *order* to:

$$\begin{aligned}
&order(t,t,0,1), order(t,t,1,2),\ldots, order(t,t,k-1,k),\\
&order(t,f,1,0), order(t,f,2,1),\ldots, order(t,f,k,k-1),\\
&order(f,t,0,0), order(f,t,1,0),\ldots, order(f,t,k,0),\\
&order(f,f,0,0), order(f,f,0,1),\ldots, order(f,f,0,k).
\end{aligned}$$

We assume without loss of generality that M diverges iff it diverges with unbounded counters. This caveat is not truly restrictive, since M can be simulated by another 2CM having this property.

Lemma 6.1 *If M diverges, then for every standard database D_k, there exists an $\ell \geq k+2$ and an I-fact f such that $f \in P^{\ell}(D_k) - P^{\ell-1}(D_k)$.*

Proof: The importance of this lemma is that when M diverges, P is not bounded; in particular, any putative "bound" on the fixpoint can be contradicted by an EDB input and an IDB fact, where the fact enters the fixpoint only when the number of iterations of P on the given EDB exceeds the bound.

Observe that the computation with EDB D_k is standard, and the divergence of M assures that arbitrary resetting of counters (via the halting state) cannot occur. Since M diverges with unbounded counters, either $I(k, c)$ or $I(c, k)$ is in the fixpoint, for some database constant $c \in \{0, 1, \ldots, k\}$; without loss of generality, let it be the former. Any proof of $I(k, c)$ requires at least $k+2$ steps, since the counters must be initialized to zero, can only increase by one at each step, and a final projection is required. Thus if ℓ is the smallest integer such that $I(k, c) \in P^{\ell}(D_k)$, we know $\ell \geq k+2$. □

Lemma 6.2 *Suppose that M halts after h steps. Then for any database D, we have that $P^{h+3}(D) = P^{\infty}(D)$.*

Proof: Suppose $I(c', c'') \in P^{\infty}(D)$ is an I-fact ($c', c'' \in D$) having a proof Π of height greater than $h+3$. We show that $I(c', c'')$ also has a proof of height no more than $h+3$.

The proof Π ends with a projection, and thus contains a subproof for $ID(h', t, f, z, s, c', c'')$ of height $h+2$. As in the proof of Lemma 5.4, this implies the existence of constants $d_0 = z, d_1, \ldots, h' = d_{h+1}, t, f$ in D, where the EDB contains all the Boolean logic, the "linking" relations for the d_i, i.e., successor, predecessor, as well as reachability to and from zero. M can then be simulated on these constants, deriving a proof of $ID(d_{h-1}, t, f, z, s, d', d'') \in P^{h}(D)$ for some constants $s, d', d'' \in D$, where this ID is the configuration just prior to halting. Since $order(f, t, d', z)$ and $order(f, t, d'', z)$ must be in the EDB, we infer $ID(d_h, t, f, z, s_h, z, z) \in P^{h+1}(D)$ where s_h names the halting state, and since the EDB must contain $order(f, f, z, c')$ and $order(f, f, z, c'')$, we know $ID(d_{h+1}, t, f, z, s_h, c', c'') \in P^{h+2}(D)$. Taking the final projection, we derive $I(c', c'') \in P^{h+3}(D)$. □

Combining these two lemmas, we conclude:

Theorem 6.3 *Predicate boundedness is undecidable for programs having one linear recursion, one initialization, and one projection.*

7 Conclusions and Open Problems

We have presented three new techniques for proving undecidability of Datalog boundedness. (1) The "finger" technique enabled us to get a tight undecidability result for program boundedness with respect to arity and linearity. (2) For uniform boundedness, reduction from the mortality problem for Turing machines seems to be a promising new approach—it allowed us to get a tight undecidability result for Datalog$^{\neq}$ and to strengthen the known results for Datalog. (3) Our "polymorphic" encoding for undecidability of one linear recursive rule and a projection might be useful for understanding the precise effect of connectivity on small arities, which is still open. Some other interesting open questions are:

- To complete the arity classification: *Is uniform boundedness decidable for arity 2 programs?* Additional linearity and connectivity restrictions (even for arity 4) give us more challenging open problems. Tight complexity bounds are still open for the monadic case (see [CGKV88, M90]).

- To complete the number of rules classification: (1) *Is uniform boundedness decidable for linear single rule programs?* Note that, linear, arity 4 is NP-hard [K88]; non-linear is also open. (2) *Is program boundedness decidable for one linear rule, any initialization rules and any arity predicates?* Note that, linear, arity 2 is NP-complete [V88]; some sufficient conditions for decidability appear in [HKMV91]; without linearity the problem is undecidable [A89].

Appendix: Ternary Datalog via the Halting Problem

In this appendix, we give a proof of the undecidability of uniform boundedness for ternary Datalog programs along the lines of [GMSV87]. By using Turing machines instead of counter machines, the argument can be considerably simplified and the arity gets down to three. Our proof, however, uses nonlinear rules, unlike the construction in [GMSV87]. We hope that a refinement of either of our two proofs will eventually resolve the undecidability of uniform boundedness for the binary case.

The reduction we are going to present is from the halting problem for Turing machines. Our program P will simulate the computation of some given Turing machine M on the empty tape. If the computation terminates, the IDB predicate will be flooded, thereby making the program bounded. Let us first give an informal overview.

Configurations of M are encoded as sets of facts $TAPE(\mathit{origin}, \mathit{time}, \mathit{cell}, \mathit{symbol}, \mathit{state})$, which should be read as "the content of tape cell *cell* at time *time*—both counted from *origin*—is *symbol*; and iff *state* is nonzero, the head is currently scanning this cell with the finite control being in state *state*." The behavior of tape cells over time is described by a set of rules expressing $TAPE(\mathit{origin}, \mathit{time}+1, \mathit{cell}, \mathit{newsymbol}, \mathit{newstate})$ in terms of $TAPE(\mathit{origin}, \mathit{time}, \mathit{cell}-1, \mathit{symbol}_L, \mathit{state}_L)$, $TAPE(\mathit{origin}, \mathit{time}, \mathit{cell}, \mathit{symbol}, \mathit{state})$, and $TAPE(\mathit{origin}, \mathit{time}, \mathit{cell}+1, \mathit{symbol}_R, \mathit{state}_R)$. It is easy to see that any transition of M can be encoded by a suitable set of rules of this form.

To encode the *time*, *cell*, *symbol*, and *state* information, we assume that the input database contains a set of constants that can serve as integers, and a $succ(x, y)$ predicate linking each "integer" to its successor. The program will ensure that long derivations can exist only if the database contains long successor chains. Hence, if the simulated machine halts, either all derivations are short, or there exists a derivation long enough to guarantee the existence of a sufficient supply of "integers" to simulate a complete computation of M and flood the IDB predicate upon reaching the halting state. If M does not halt, we can easily manufacture "standard" databases on which P will run arbitrarily long.

A closer look at the $TAPE$ predicate reveals that it is possible to get rid of some of its arguments. First, the *symbol* and *state* fields can be merged, since both range over a finite domain. The resulting symbol/state combination can then be piggybacked onto the cell number using a technique described in the appendix of [GMSV87]: We assume that every "integer" in the database comes in as many "flavors" as there are symbol/state combinations. By putting the right "flavor" of *cell* into a $\mathit{TAPE}(\mathit{origin}, \mathit{time}, \mathit{cell})$ fact, we can then encode its symbol and state content.

We now describe the details of P. Its predicates are:

- An EDB predicate $\mathit{succ}(X, Y)$, stating that Y is the successor of X.
- A group of EDBs $\mathit{copy}_1(X, Y), \ldots, \mathit{copy}_N(X, Y)$, where N is the number of different symbol/state combinations. $\mathit{copy}_i(X, Y)$ states that Y is a copy of the integer X in flavor i. We use $\mathit{copied}(X)$ as a shorthand for $\mathit{copy}_1(X, Y_1), \ldots, \mathit{copy}_N(X, Y_N)$ with fresh variables $Y_1, \ldots, Y_N$; i.e., $\mathit{copied}(X)$ says that X is available in all flavors.
- An IDB predicate $\mathit{TAPE}(Z, T, C)$, stating that at time $T - Z$, the symbol/state content of cell C is given by i, where i is the flavor of C.

The rules of P are of three kinds: *starting rules*, *transition rules*, and a *halting rule*. We assume that M operates on a semi-infinite tape, that it never attempts to read past the left edge of the tape, and that it halts by writing a blank and entering a designated halting state h.

Starting rules: These rules position the head on cell 0 and recursively clear the tape. This recursion proceeds *in parallel* with the simulation.

$$\begin{aligned}
\mathit{TAPE}(Z, Z, C) &\;:\!\!-\;\; \mathit{copy}_s(Z, C). \\
\mathit{TAPE}(Z, Z, C) &\;:\!\!-\;\; \mathit{succ}(Z, X), \mathit{copy}_b(X, C). \\
\mathit{TAPE}(Z, Z, C') &\;:\!\!-\;\; \mathit{TAPE}(Z, Z, C), \\
&\qquad\;\; \mathit{copy}_b(X, C), \mathit{succ}(X, Y), \mathit{copied}(Y), \mathit{copy}_b(Y, C').
\end{aligned}$$

Here s is an integer representing the combination of a blank with the start state, and b is an integer representing the combination of a blank and state 0 (which means that the head is not on this cell).

Transition rules:

$$\begin{aligned}
\mathit{TAPE}(Z, T', C') \;:\!\!-\;\; & \mathit{TAPE}(Z, T, C_L), \mathit{TAPE}(Z, T, C), \mathit{TAPE}(Z, T, C_R), \\
& \mathit{copy}_i(X_L, C_L), \mathit{copy}_j(X, C), \mathit{copy}_k(X_R, C_R), \\
& \mathit{succ}(X_L, X), \mathit{succ}(X, X_R), \\
& \mathit{copy}_{\delta(i,j,k)}(X, C'), \\
& \mathit{succ}(T, T'), \mathit{copied}(T').
\end{aligned}$$

where $\delta(i, j, k)$ is the function that computes the new symbol/state content of a cell from the previous values of the cell and its two neighbors. There are a few additional rules of the form:

$$\begin{aligned}
\mathit{TAPE}(Z, T', C') \;:\!\!-\;\; & \mathit{TAPE}(Z, T, C), \mathit{TAPE}(Z, T, C_R), \\
& \mathit{copy}_i(Z, C), \mathit{copy}_j(X, C_R), \\
& \mathit{succ}(Z, X), \\
& \mathit{copy}_{\delta'(i,j)}(Z, C'), \\
& \mathit{succ}(T, T'), \mathit{copied}(T').
\end{aligned}$$

to deal with the leftmost cell, because it has no left neighbor.

Halting rule: We flood the $TAPE$ predicate as soon as the halting configuration is reached:

$$TAPE(U,V,W) \quad \text{:—} \quad TAPE(Z,T,C), copy_h(X,C).$$

Here h is an integer representing the combination of a blank with the halting state.

Lemma 7.1 *If M does not halt, then P is unbounded.*

Proof: Let D be the infinite database whose universe is $\mathbb{N} \cup (\mathbb{N} \times \{1,\ldots,N\})$ and which contains the facts $succ(k,k+1)$ for all $k \in \mathbb{N}$ and $copy_i(k,\langle k,i\rangle)$ for all $\langle k,i\rangle \in \mathbb{N} \times \{1,\ldots,N\}$. It is easy to see that $P^k(D) \neq P^\infty(D)$ for all $k > 0$. By choosing only the EDB facts occurring in a single derivation of height $k+1$, one can then obtain, for any integer $k \geq 1$, a finite subset D' of D such that $P^k(D') \neq P^\infty(D')$. □

Lemma 7.2 *If M terminates, then P is uniformly bounded.*

Proof: Assume M terminates after k steps. Let D_{2k} be the database whose universe is $\{0,\ldots,2k\} \cup \{0,\ldots,2k\} \times \{1,\ldots,N\}$ and which contains the facts $succ(k,k+1)$ for $k \in \{0,\ldots,2k-1\}$ and $copy_i(k,\langle k,i\rangle)$ for $\langle k,i\rangle \in \{0,\ldots,2k\} \times \{1,\ldots,N\}$. It is easy to see that the entire computation of M can be simulated within D_{2k}. (The $2k$ stems from the fact that each simulation step "loses" the rightmost tape cell for lack of a right neighbor, so that $2k$ cells are needed in the beginning to have k cells left in the end.)

Now if D is any database which admits a derivation of height $2k$, we can obtain an *embedding* of D_{2k} into D by mapping the integers $0,\ldots,2k$ to the elements used to instantiate the T variable, and mapping each pair (i,j) to the $copy_j$ counterpart of the image of i (whose existence is guaranteed by the $copied(T')$ goal in the body of every recursive rule). The halting rule then guarantees that every $TAPE$ fact can be proved in at most $2k+1$ steps. □

Combining the two lemmas, we have:

Theorem 7.3 *Uniform boundedness is undecidable for ternary Datalog programs with a single IDB.*

References

[A89] S. Abiteboul, *Boundedness of Single Rule Programs is Undecidable*, Information Processing Letters, **32** (1989), 281–289.

[AG89] M. Ajtai, Y. Gurevich, *Datalog versus First Order*, Proc. 30th IEEE Symp. on Foundations of Computer Science (1989), 142–148.

[CH85] A.K. Chandra, D. Harel, *Horn Clause Queries and Generalizations*, J. Logic Programming, **2** (1985), 1–15.

[CGKV88] S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis, M.Y. Vardi, *Decidable Optimization Problems for Database Logic Programs*, Proc. 20th ACM Symp. on Theory of Computing (1988), 477-490.

[CK86] S.S. Cosmadakis, P.C. Kanellakis, *Parallel Evaluation of Recursive Rule Queries*, Proc. 5th ACM Symp. on Principles of Database Systems (1986), 280–293.

[GMSV87] H. Gaifman, H. Mairson, Y. Sagiv, M.Y. Vardi, *Undecidable Optimization Problems for Database Logic Programs*, Proc. 2nd IEEE Symp. on Logic in Computer Science (1987), 106–115. Full version in IBM Research Report RJ7386; also to appear in J. ACM, 1992.

[G90] I. Guessarian, *Deciding Boundedness for Uniformly Connected Datalog Programs*, Proc. 3rd International Conf. on Database Theory (1990), Lecture Notes in Computer Science 470, Springer-Verlag, 395–409.

[H66] P.K. Hooper, *The Undecidability of the Turing Machine Immortality Problem*, Journal of Symbolic Logic, **31** (1966), 219–234.

[HKMV91] G. Hillebrand, P.C. Kanellakis, H.G. Mairson, M.Y. Vardi *Tools for Datalog Boundedness*, Proc. 10th ACM Symp. on Principles of Database Systems (1991), 1–12

[I86] Y.E. Ioannidis, *A Time Bound on the Materialization of Some Recursively Defined Views*, Algorithmica, **1** (1986), 361–385.

[K88] P.C. Kanellakis, *Logic Programming and Parallel Complexity*, Foundations of Deductive Databases and Logic Programming, J. Minker ed., Morgan-Kaufmann 1988, 547–586.

[K90] P.C. Kanellakis, *Elements of Relational Database Theory*, Handbook of Theoretical Computer Science, Vol. B, Chapter 17, J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin ed., North-Holland 1990, 1073–1157.

[KA89] P.C. Kanellakis, S. Abiteboul, *Database Theory Column*, SIGACT News, **20** (1989), 17–23.

[KTU90] A.J. Kfoury, J. Tiuryn, P. Urzyczyn, *The Undecidability of the Semi-Unification Problem*, Proc. 22nd ACM Symp. on Theory of Computing (1990), 468–476.

[MUV84] D. Maier, J.D. Ullman, M.Y. Vardi, *On the Foundations of the Universal Relation Model*, ACM Trans. on Database Systems, **9** (1984), 283–308.

[M90] R. van der Meyden, *Predicate Boundedness of Linear Monadic Datalog is in PSPACE*, unpublished manuscript, Sept. 1990.

[N89a] J.F. Naughton, *Data Independent Recursion in Deductive Databases*, J. Computer and System Sciences, **38** (1989), 259–289.

[N89b] J.F. Naughton, *Minimizing Function-free Recursive Definitions*, J. ACM, **36** (1989), 69–91.

[NS91] J.F. Naughton, Y. Sagiv, Naughton, J.F., Sagiv, Y.: *A simple characterization of uniform boundedness for a class of recursions*, J. Logic Programming, **10** (1991), 233–254.

[S85] Y. Sagiv, *On Computing Restricted Projections of Representative Instances*, SIAM J. of Computing, **17** (1988), 1–22

[S88] Y. Sagiv, *Optimizing Datalog Programs*, Foundations of Deductive Databases and Logic Programming, J. Minker ed., Morgan-Kaufmann 1988, 659–698.

[Sh87] O. Shmueli, *Decidability and Expressiveness Aspects of Logic Queries*, Proc. 6th ACM Symp. on Principles of Database Systems (1987), 237–249.

[U89] J.D. Ullman, *Principles of Database and Knowledge Base Systems: Volumes I,II*, Computer Science Press 1989.

[UV88] J.D. Ullman, A. Van Gelder, *Parallel Complexity of Logical Query Programs*, Algorithmica, **3** (1988), 5–42.

[V88] M.Y. Vardi, *Decidability and Undecidability Results for Boundedness of Linear Recursive Queries*, Proc. 7th ACM Symp. on Principles of Database Systems (1988), 341–351.