Undecidable Boundedness Problems for Datalog Programs*

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

October 11, 1994

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) By encoding all possible transitions of a two-counter machine in a single rule, we 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 (6 or 7).

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 fundamental problem, e.g., for parallel evaluation [CK86, K88, UV88] or for optimization [N89b].

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* (IDB) predicate; all others are called *extensional database* (EDB) predicates. 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 predicate in the program is a; note that EDB predicates can have any arity.

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

The various I/O conventions correspond to various views of Datalog programs. The predicate I/O convention views a Datalog program as a definition of a single output predicate. The program I/O convention views a Datalog program as a simultaneous definition of multiple output predicates. The uniform I/O convention considers a Datalog program as a piece of a larger Datalog programs, so all predicates are both input and output. By adding new EDB predicates, one can always separate the roles of input and output predicates; thus, the uniform I/O convention can be viewed as a special case of the program I/O convention.

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. $\square$

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. (Uniform boundedness is called *strong* boundedness in [GMSV87].) $\square$

For the same set of rules, *uniform boundedness* $\Rightarrow$ *program boundedness* $\Rightarrow$ *predicate boundedness*, but the converses need not hold. Clearly, program boundedness means predicate boundedness for all IDB predicates. Thus, decidability of predicate boundedness implies decidability of program boundedness. As we observed earlier, the uniform I/O convention can be viewed as a special case of the program I/O convention, so decidability of program boundedness implies decidability of uniform boundedness. Conversely, undecidability of uniform boundedness implies undecidability of program boundedness, and undecidability of program boundedness implies undecidability of predicate boundedness.

While predicate boundedness could in principle be harder than uniform boundedness, we do not know of any class of Datalog programs for which predicate boundedness is undecidable while uniform boundedness is decidable. In contrast, uniform equivalence of Datalog programs is decidable [S88] (first shown in [CK86] for one IDB predicate), while predicate equivalence is undecidable [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). $\square$

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 predicate (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:

$$\mathit{FING}(U,V) \quad :\!\!- \quad q_a(X), q_b(X).$$

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

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

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:

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

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:

$$\mathit{FING}(X_1,X_3) \quad :\!\!- \quad \mathit{First}(X_1), \mathit{Succ}(X_1,X_2), \mathit{Succ}(X_2,X_3),$$

$$q_\#(X_1), q_{(s,\sqcup)}(X_2), q_\#(X_3).$$

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:

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

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

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

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

$First(c_1)$, $Succ(c_i, c_{i+1})$ 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 predicate.*

3 Uniform Boundedness in the Presence of Negation

We now study the language Datalog$^{\neq}$, which is Datalog augmented with a "$\neq$"-predicate denoting inequality between constants (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]. The 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 under the read/write head, and l and r are *infinite* strings of symbols specifying the contents of the tape to the left and right of the head. We call such a quadruple a *configuration* of M. Observe that each configuration uniquely determines the entire computation of M starting in that configuration. Now, call M *mortal* if for every configuration (l, s, r, q), the computation of M starting at (l, s, r, q) must eventually reach a halting state. (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. Recall König's Lemma, which says that in a tree of bounded degree, there is an infinite path if and only if there are paths of unbounded length. Thus, (1) and (2) are equivalent. $\square$

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), where q is an integer encoding the state, s is an integer encoding the symbol under the read/write head, l and r are lists of integers encoding the tape contents to the left and right of the head and t is a list of integers acting as a "timestamp" whose rôle we will explain soon. Here, C will again be a single database constant—it merely behaves like the indicated five-element list with respect to the $Cons$ and Int_i predicates.

The transitions of M are encoded by rules of the form $CONF(C') \text{ :— } CONF(C), \varphi$, where φ is a list of clauses that specify how the various components of the lists represented by C and C' are related for a particular transition of M. A computation of M is simulated by a series of applications of these rules, where each application uses a known fact $CONF(C)$ corresponding to the current configuration of M to derive a fact $CONF(C')$ corresponding to the subsequent configuration of M. Thus, each computation of M leads to a derivation of P of the same length and vice versa. Unfortunately, if M enters an infinite loop, the corresponding derivation might just reprove the same $CONF$ facts over and over again, and so P could be bounded even if M has nonterminating computations. To eliminate this possibility, we use the "timestamp" component t of a configuration (t, l, s, r, q), which is a list of integers. Each transition rule simply specifies that the length of the t component of the output configuration must be one larger than the length of the t component of the input configuration. In this way, if the same machine configuration is encountered twice during a computation, the database constants representing the two occurrences must be different, because their timestamp fields will be different.

Of course, the correctness of the simulation hinges on the fact that the *Cons* and Int_i predicates behave as expected. For example, the *Car* and *Cdr* parts of a list should be unique and a database constant should not represent two different integers. However, since the input database is completely arbitrary, this is not necessarily the case. The program therefore includes, besides the rules simulating the transitions of M, a number of "checking rules" that ensure the consistency of the *Cons* and Int_i predicates. 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 the expression $C = (T, L, S, R, Q)$ as an abbreviation for the sequence of clauses $Cons(T, U, C)$, $Cons(L, V, U)$, $Cons(S, W, V)$, $Cons(R, Q, W)$, where the variables U, V, W appear nowhere else. Thus, $C = (T, L, S, R, Q)$ says that C represents a five-element list with components T, L, S, R, Q.

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

$$\begin{aligned} CONF(C') \quad :\!\!- \quad & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L', S', R', Q'), \\ & Cons(S'', L, L'), Cons(S', R', R), \\ & Int_i(Q), Int_j(Q'), Int_k(S), Int_l(S''), \\ & Cons(Y, T, T'). \end{aligned}$$

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

$$\begin{aligned} CONF(C') \quad :\!\!- \quad & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L', S', R', Q'), \\ & Cons(S'', R, R'), Cons(S', L', L), \\ & Int_i(Q), Int_j(Q'), Int_k(S), Int_l(S''), \\ & Cons(Y, T, T'). \end{aligned}$$

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

$$\begin{aligned} CONF(C') \quad :\!\!- \quad & CONF(C), \\ & C = (T, L, S, R, Q), C' = (T', L, S'', R, Q'), \\ & Int_i(Q), Int_j(Q'), Int_k(S), Int_l(S''), \\ & Cons(Y, T, T'). \end{aligned}$$

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:

$$CONF(C) \;:\!-\; Int_i(X), Int_j(X). \qquad (\text{for } 1 \le i < j \le N)$$

$$CONF(C) \;:\!-\; Cons(X,Y,Z), Cons(X',Y',Z), X \neq X'.$$

$$CONF(C) \;:\!-\; Cons(X,Y,Z), Cons(X',Y',Z), Y \neq Y'.$$

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. This is established in the following two lemmas.

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

Proof: Let k be an arbitrary integer. We have to construct a database D such that $P^k(D) \neq P^\infty(D)$, i.e. there are some facts in the output of P on D that cannot be derived by derivation trees of height k or less.

Let $\mathcal{C} = (l_0, s_0, r_0, q_0) \stackrel{\delta_{i_1}}{\to} (l_1, s_1, r_1, q_1) \stackrel{\delta_{i_2}}{\to} \cdots \stackrel{\delta_{i_{k+1}}}{\to} (l_{k+1}, s_{k+1}, r_{k+1}, q_{k+1})$ be a computation of M of length $k+1$. Such a computation exists, because M is not uniformly bounded. We construct D so that it contains just the facts necessary for a faithful simulation of $\mathcal{C}$ and no checking rule applies.

The details are as follows. We can assume that the stayte space and tape alphabet of M is a subset of the integers $1 \ldots N$. Furthermore, since M can access at most $k+1$ cells during $\mathcal{C}$, we can assume that there are at most $k+1$ nonblank symbols on the tape in any configuration of $\mathcal{C}$. Thus, a configuration (l_i, s_i, r_i, q_i) can be represented by two integers s'_i, q'_i and two lists of integers l'_i, r'_i of length $\le k+1$.

We define the constants of D to be the following: (1) the integers $1 \ldots N$; (2) for $0 \le i \le k+1$, the lists l'_i, r'_i, and 1_i, where 1_i is a list containg i 1's (these will serve as timestamps); (3) for $0 \le i \le k+1$, the lists $(1_i, l'_i, s'_i, r'_i, q'_i)$, (l'_i, s'_i, r'_i, q'_i), (s'_i, r'_i, q'_i), and (r'_i, q'_i).

D contains the following facts: (1) $Int_i(i)$ for $1 \le i \le N$; (2) $Cons(x,y,z)$ whenever x, y, z are constants of D such that z is a list with *Car* x and *Cdr* y; (3) $CONF((1_0, l'_0, s'_0, r'_0, q'_0))$.

It is easy to verify that the checking rules do not apply to D and that P can derive $CONF((1_{k+1}, l'_{k+1}, s'_{k+1}, r'_{k+1}, q'_{k+1}))$ from D by a sequence of $k+1$ rule applications mimicking the $k+1$ steps of $\mathcal{C}$. Furthermore, this fact cannot be derived in less than $k+1$ rule applications, because the checking rules do not apply and each transition rule can only increase the length of the timestamp field by 1 at each step. Therefore, $P^k(D) \neq P^\infty(D)$. □

For the reverse direction, we have to prove that, no matter what the input database contains, either a checking rule applies or every derivation of P mirrors some computation of M. Let us call an input database D *standard* if the checking rules of P do not apply, i.e., if for all x, x', y, y', z, $Cons(x,y,z) \wedge Cons(x',y',z) \Rightarrow x = x' \wedge y = y'$ and $Int_i(x) \wedge Int_j(x) \Rightarrow i = j$. Otherwise D is called *nonstandard*.

Lemma 3.4 *If M is uniformly mortal, then P is uniformly bounded.*

Proof: Clearly P is uniformly bounded on nonstandard databases, because every *CONF* fact can be derived by a single application of a checking rule. It suffices therefore to study the behavior of P on a standard input database D.

Let $\delta_1, \delta_2, \ldots, \delta_k$ be the possible transitions of M and $\rho_1, \rho_2, \ldots, \rho_k$ be the corresponding rules of P. Consider a derivation $\pi = \mathit{CONF}(c_0) \stackrel{\rho_{i_1}}{\rightarrow} \mathit{CONF}(c_1) \stackrel{\rho_{i_2}}{\rightarrow} \cdots \stackrel{\rho_{i_m}}{\rightarrow} \mathit{CONF}(c_m)$ of P on D and the corresponding sequence of transitions $\delta_{i_1}, \delta_{i_2}, \ldots, \delta_{i_m}$ of M. If we can show that $\delta_{i_1}, \delta_{i_2}, \ldots, \delta_{i_m}$ is a legal computation of M, i.e. if there exists a configuration (l, s, r, q) from which M will execute this sequence, then we are done, because there is a uniform bound on the length of computations of M.

It is easy to see that $\delta_{i_1}, \delta_{i_2}, \ldots, \delta_{i_m}$ is a legal computation if and only if it satisfies two conditions: (1) Each transition δ_{i_j} must be compatible with the state of M after transition $\delta_{i_{j-1}}$; (2) If at the beginning of transition δ_{i_j}, the head is scanning a cell that was last written into during transition δ_{i_k} (where $k < j$), then δ_{i_j} must be compatible with the symbol written during δ_{i_k}.

In terms of the derivation π, condition (1) is equivalent to saying that the database constants q and q' used to instantiate the variable Q in rule application ρ_{i_j} and Q' in rule application $\rho_{i_{j-1}}$ must represent the same integer, and condition (2) is equivalent to saying that the database constants s and s'' used to instantiate the variable S in rule application ρ_{i_j} and S'' in rule application ρ_{i_k} must represent the same integer.

To see why these conditions are fulfilled in a standard database, let us visualize the database as a directed graph, where each fact $\mathit{Cons}(x, y, z)$ is represented by 4 edges: an edge labeled *Car* from z to x, an edge labeled *Cdr* from z to y, an edge labeled Car^{-1} from x to z, and an edge labeled Cdr^{-1} from y to z. Observe that if there is a Car^{-1} edge from x to z and a *Car* edge from z to x', then x and x' must be the same node, because the database is standard; the same goes for *Cdr* edges. More generally, we have the following property: If there is a path from x to x' such that the sequence $l_1, \ldots, l_n$ of its edge labels can be transformed into the empty sequence by successively deleting adjacent labels of the form $\mathit{Car}^{-1}\,\mathit{Car}$ or $\mathit{Cdr}^{-1}\,\mathit{Cdr}$, then x and x' must be the same node. Intuitively, if we "pack" a constant into a list and then "unpack" it again later, we get the same constant back.

Consider now again the case of two successive rule applications $\mathit{CONF}(c_{j-2}) \stackrel{\rho_{i_{j-1}}}{\rightarrow} \mathit{CONF}(c_{j-1}) \stackrel{\rho_{i_j}}{\rightarrow} \mathit{CONF}(c_j)$. Let t, l, s, r, q be the constants used to instantiate variables T, L, S, R, Q of ρ_{i_j} and t', l', s', r', q' be the constants used to instantiate variables T', L', S', R', Q' of $\rho_{i_{j-1}}$ (in general, we use uppercase letters for variables and lowercase letters for constants). We have $c_{j-1} = (t', l', s', r', q')$ and $c_{j-1} = (t, l, s, r, q)$. Recall that the notation $C = (T, L, S, R, Q)$ is a shorthand for $\mathit{Cons}(T, U, C)$, $\mathit{Cons}(L, V, U)$, $\mathit{Cons}(S, W, V)$, $\mathit{Cons}(R, Q, W)$, where the variables U, V, W occur nowhere else. Thus, there is a path labeled $\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}$ from q' to c_{j-1} and a path labeled $\mathit{Cdr}\,\mathit{Cdr}\,\mathit{Cdr}\,\mathit{Cdr}$ from c_{j-1} to q. It follows that there is a path labeled $\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}\,\mathit{Cdr}^{-1}\,\mathit{Cdr}\,\mathit{Cdr}\,\mathit{Cdr}\,\mathit{Cdr}$ from q' to q and therefore q and q' must be the same constant. Since this constant cannot represent two different integers in a standard database, condition (1) is true.

The argument for condition (2) is similar. Let s and s'' be the constants used to instantiate variable S in rule application ρ_{i_j} and S'' in rule application ρ_{i_k}, where at the beginning of transition δ_{i_j}, M's head is scanning a cell that was last written into during transition δ_{i_k} $(k < j)$. Assume w.l.o.g. that δ_{i_k} was a "right" transition (and $\delta_{i_{j-1}}$ therefore a "left" transition). Then there is a path from s'' to s whose label sequence is of the form

$$Car^{-1}\, wx_{k+1}wx_{k+2}w \ldots wx_{j-2}w\; Car\; v,$$

where $w = Car^{-1}\, Cdr^{-1}\, Cdr\; Car$, $v = Car^{-1}\, Cdr^{-1}\, Cdr^{-1}\, Cdr\; Cdr\; Car$, and for $k < l < j-1$, x_l is either Cdr, Cdr^{-1}, or empty, depending on whether δ_{i_l} is a "left", "right", or "stay" transition. This path arises in the following way. The initial Car^{-1} leads from s'' to the constant instantiating L' in ρ_{i_k}. The sequence w connects the constant instantiating L' in one rule application to the constant instantiating L in the next rule application. x_l connects the constants instantiating L and L' in ρ_{i_l}, the Car after the last w connects the constants instantiating L and S' in $\delta_{i_{j-1}}$, and the trailing v connects the constant instantiating S' in $\rho_{i_{j-1}}$ to s. Since there must be an equal number of "left" and "right" transitions between δ_{i_k} and $\delta_{i_{j-1}}$, the labels on the path "cancel out" and it follows that s and s'' are the same constant. Again, this constant cannot represent two different integers, so condition (2) is also true. □

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

Theorem 3.5 *Uniform boundedness is undecidable for linear monadic Datalog$^{\neq}$ programs with a single IDB predicate.*

4 Uniform Boundedness for Ternary Datalog

We now refine the reduction of the previous section to eliminate the use of $\neq$ and obtain an undecidability result for plain Datalog. This can be done by increasing the arity of the *CONF* predicate to three.

Let us again visualize an input database as a directed graph, where each fact $Cons(x, y, z)$ is represented by a Car edge from z to x, a Cdr edge from z to y, a Car^{-1} edge from x to z, and a Cdr^{-1} edge from y to z. The crucial argument for the correctness of the simulation was that in a standard database, the traversal of a path whose edge labels "cancel out" leads back to the starting node. In other words, "unpacking" some complicated list structure yields the same constants that were "packed in" earlier.

Clearly, this property need not hold if we cannot enforce the uniqueness of Car's and Cdr's via the "$\neq$" predicate. Observe, however, that the proof of Lemma 3.4 actually required a weaker property, namely: if there is a path from x to x' whose edge labels "cancel out", and if $Int_i(x)$ and $Int_j(x')$, then $i = j$. Whether x and x' are equal does not matter for the correctness of the simulation, since P looks only at the integers represented by constants, not the constants themselves.

Let us call two constants x and x' *equivalent* (denoted by $x \sim x'$) if there is a path from x to x' such that its edge labels cancel, i.e. the sequence of labels can be transformed into the empty sequence by deleting adjacent labels of the form $Car^{-1}\, Car$ or $Cdr^{-1}\, Cdr$. We say that a database is *weakly standard* if there are no constants x, x' such that $x \sim x'$, $Int_i(x)$, $Int_j(x')$, and $i \neq j$. Clearly, any weakly standard database is standard, but the converse need not hold. However, since a weakly standard database has exactly the property required to make the proof of Lemma 3.4 go through, we can conclude that if M is uniformly mortal, then P is uniformly bounded on all weakly standard databases.

The reason we are interested in weakly standard databases is that this property can be checked without using the "$\neq$" predicate. Essentially, we have to generate all pairs of equivalent constants and see whether there is one whose members represent different integers. This can be done as follows. Observe that if two constants x and x' are equivalent, then 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. Thus, we can compute all pairs of equivalent constants by the following Datalog program:

$$\begin{aligned}
EQUIV(X,X) &\ \text{:—}\\
EQUIV(X,X') &\ \text{:—}\quad Cons(X,Y,Z), Cons(X',Y',Z'), EQUIV(Z,Z')\\
EQUIV(X,X') &\ \text{:—}\quad Cons(Y,X,Z), Cons(Y',X',Z'), EQUIV(Z,Z')\\
EQUIV(X,X') &\ \text{:—}\quad EQUIV(X,Y), EQUIV(Y,X')
\end{aligned}$$

For each pair $1 \leq i < j \leq N$, we would then have a checking rule

$$CONF(U) \ \text{:—}\ EQUIV(X,X'), Int_i(X), Int_j(X').$$

With these rules, we can detect any database that is not weakly standard and flush the simulation. But unfortunately, these rules are *themselves* unbounded, because the recursion depth of the *EQUIV* program depends on the list nesting depth in the input database. To control the *EQUIV* recursion, we have to *couple* it with the simulation, so that on a weakly standard database, the length of the *EQUIV* computation is bounded by the length of the simulated computation, whereas in a non-weakly-standard database, we will hopefully find the inconsistency fairly soon.

We can accomplish this coupling by increasing the arity of the *CONF* predicate to three. The two additional arguments are used to compute pairs of equivalent constants exactly as in the *EQUIV* program above, but in lockstep with the simulated computation. The rules of the new simulation program P are of three kinds. First, there is a rule to initialize an equivalence computation:

$$CONF(C,X,X) \ \text{:—}\ CONF(C,X_1,X_2).$$

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

$$CONF(U,V,W) \ \text{:—}\ CONF(C,X_1,X_2), Int_i(X_1), Int_j(X_2).$$

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:

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

$$
\begin{aligned}
CONF(C', X_1', X_2') \quad &:\!\!- \quad CONF(C, X_1, X_2), \\
&\qquad Cons(Y_1, X_1', X_1), Cons(Y_2, X_2', X_2), \\
&\qquad /\!*\ \text{Clauses for } C,\ C' \text{ as in Section 3 } *\!/ \\
CONF(C', X_1, X_3) \quad &:\!\!- \quad CONF(C, X_1, X_2), CONF(C, X_2, X_3), \\
&\qquad /\!*\ \text{Clauses for } C,\ C' \text{ as in Section 3 } *\!/
\end{aligned}
$$

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.

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

Proof: Same as for Lemma 3.3, except that as initial $CONF$ fact we use $CONF((1_0, l_0', s_0', r_0', q_0'), 1, 1)$. □

Lemma 4.2 *If M is uniformly mortal, then P is uniformly bounded.*

Proof: As we saw above, P will be uniformly bounded on all weakly standard databases, so it suffices to consider an input database D that is not weakly standard. Let ℓ be an upper bound on the length of computations of M and assume that there is a path $\pi := \rho_{i_1}, \ldots, \rho_{i_m}$ of length $m > \ell$ in a derivation tree of P on D. (If no such path exists, we are done.) Let $c_0, c_1, \ldots, c_m$ be the values of the first argument of the $CONF$ facts along π. The sequence of transitions $\delta_{i_1}, \ldots, \delta_{i_m}$ of M corresponding to π cannot be a valid computation, in fact, even the sequence $\delta_{i_1}, \ldots, \delta_{i_{\ell+1}}$ cannot. As seen in the proof of Lemma 3.4, there must therefore exist two equivalent constants x, x' representing different integers such that these constants appear in the derivation $\rho_{i_1}, \ldots, \rho_{i_{\ell+1}}$ as values of Q and Q' in adjacent rule applications or as values of S and S'' in rule applications where the head scans the same cell. Moreover, the length of the path linking these constants is bounded by some constant C_ℓ depending only on ℓ. (We may choose $C_\ell \geq \ell$.) It is easy to see that the equivalence of x and x' can then be derived by the $EQUIV$ program above in recursion depth at most C_ℓ. If $m \geq \ell$, this derivation can be turned into a derivation of $CONF(c_\ell, x, x')$ by changing $EQUIV$ to $CONF$ and by instantiating the first arguments of the $CONF$ facts in each path of the derivation tree to $c_0, \ldots, c_\ell$. Since x and x' represent different integers, the checking rule applies to $CONF(c_\ell, x, x')$, and all possible $CONF$ facts can therefore be derived in $C_\ell + 1$ steps. If there is no path π of length $\geq \ell$, the inconsistency of the database might go undetected, but then the recursion depth of P is less than $C_\ell + 1$ anyway. Thus, P is uniformly bounded. □

Theorem 4.3 *Uniform boundedness is undecidable for ternary Datalog programs with a single IDB predicate.*

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.

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

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

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* and each of the counters is set to zero. 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, 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 [?]. Hence the halting problem for 2CM's is undecidable.

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 $ID(H, T, F, S, C_1, C_2)$, 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 $Not(X, Y)$, $And(X, Y, Z)$, and $Or(X, Y, Z)$, 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 relation $State(S, B_1, \ldots, B_k)$ 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 $State(S_0, F, F, \ldots, F)$ and the halting state as $State(S_h, T, T, \ldots, T)$. The behavior of a 2CM clearly does not depend on the particular *names* we choose for the states; our conventions merely facilitate the coding.

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:

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

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:*

$$\begin{aligned}
&Not(t, f), Not(f, t),\\
&And(t, t, t), And(t, f, f), And(f, t, f), And(f, f, f),\\
&Or(t, t, t), Or(t, f, t), Or(f, t, t), Or(f, f, f),\\
&State(s_0, f, \ldots, f, f), State(s_1, f, \ldots, f, t), \ldots, State(s_h, t, \ldots, t, t),\\
&Zero(t, 0), Zero(f, 1), \ldots, Zero(f, k),\\
&Order(t, 0, 1), Order(t, 1, 2), \ldots, Order(t, k-1, k),\\
&Order(f, 1, 0), Order(f, 2, 1), \ldots, Order(f, k, k-1).
\end{aligned}$$

Lemma 5.2 *If M diverges, then for any constants s, c_1, c_2, each 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: The key idea is to mimic the computation in a standard database D_k *within* the arbitrary database D, and then use the argument of the previous lemma. 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 $h+2$. Let Π be a proof of I having minimum height among all proofs of I with height at least $h+2$. 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. Since the rule head of the halting rule does not share any variables with the rule body, it is redundant to use it more than once in a proof. As all rules are linear, it would appear several times along the "spine" of the proof. Given that the head can be instantiated to *anything* as long as the rule body is satisfied, a shorter proof results from taking the *lowest* occurrence of its use along the spine, choosing the instantiation of the head given by the *highest* use along the spine, and omitting the parts of the proof that occur between these uses. 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. Now repeat the argument of Lemma 5.3, using the embedded isomorphic copy of D_k 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.

The whole point of the halting rule was that given a proof of a halting configuration, the halting rule provided suffient power to prove *anything* in one more rule application. When a 2CM computation halted, then, it allowed the deduction of any fact in the fixpoint, with a proof of essentially the same height as the number of steps in the halting computation. Assume that a 2CM diverges only with unbounded counters (that is, it cannot "cycle" over a fixed number of machine IDs): we might then consider projecting on the counter variables to derive a new IDB relation. In the case of divergence, we are guaranteed that the query program is not bounded, using an argument virtually identical to that in the previous section. (While the "time" variable has been projected out, the unbounded counters serve to generate new IDB facts.) However, in the case of a halting configuration, how is the "prove anything in one step" power of the (now omitted) halting rule to be simulated?

Here is the essential idea: imagine modifying the definition of a 2CM so that when a transition takes place to the halting state, the counters are both magically reset to zero, and any transition *from* the halting state can reset the counters to *any* arbitrary value. The Datalog simulation of the previous section links simulated a counter values via connections to its (simulated) successor and predecessor, because the counter values only change by 1 at each transition step. To simulate the modified machines where special transtions are allowed to and from the halting state, each simulated counter value must be directly linked to the simulated value for 0, so we can jump directly to 0, and from 0 to any value.

The following syntactic modifications are made to the query program in order to implement the above argument. 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:

$$
\begin{aligned}
&ID(Z, T, F, \underline{Z}, S_0, Z, Z) \text{ :— } \\
&\quad \underline{Order(F, T, Z, Z), Order(F, F, Z, Z)}, \\
&\quad Not(T, F), Not(F, T), \\
&\quad And(T, T, T), And(T, F, F), And(F, T, F), And(F, F, F), \\
&\quad Or(T, T, T), Or(T, F, T), Or(F, T, T), Or(F, F, F), \\
&\quad State(S_0, F, F, \ldots, F), \ldots, State(S_h, T, T, \ldots, T), \\
&\quad Zero(T, Z).
\end{aligned}
$$

Transition rule:

$$
\begin{aligned}
&ID(H', T, F, \underline{Z}, S', C_1', C_2') \text{ :— } \\
&\quad ID(H, T, F, \underline{Z}, S, C_1, C_2), \\
&\quad Zero(F, H'), Order(T, T, H, H'), Order(T, F, H', H),
\end{aligned}
$$

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

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.[1]

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 proof of $ID(h',t,f,z,s,c',c'')$ 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.*

[1] If the 2CM had an extra counter, divergence with unbounded counters only would be easy: we would always increment the extra counter at every transition. However, a 2CM can simulate a counter machine with an arbitrary, fixed number of counters. If counter i ($1 \leq i \leq k$) contains c_i, code this information in one of the two counters of the 2CM using prime factors, e.g., $2^{c_1}3^{c_2}\cdots p_j^{c_j}\cdots p_k^{c_k}$ where p_j is the jth prime number. Then adding and subtracting 1 from a counter value is simulated by multiplying or dividing by a fixed prime; testing for zero counter is simulated by testing for zero remainder after division by a fixed prime number. Two counters easily suffice for these elementary arithmetic operations.

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 $TAPE(origin, time, cell, symbol, state)$, 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 $TAPE(origin, time+1, cell, newsymbol, newstate)$ in terms of $TAPE(origin, time, cell-1, symbol_L, state_L)$, $TAPE(origin, time, cell, symbol, state)$, and $TAPE(origin, time, cell+1, symbol_R, state_R)$. 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 *TAPE*(*origin*, *time*, *cell*) fact, we can then encode its symbol and state content.

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

- An EDB predicate $Succ(X,Y)$, stating that Y is the successor of X.
- A group of EDB predicates $Copy_1(X,Y), \ldots, Copy_N(X,Y)$, where N is the number of different symbol/state combinations. $Copy_i(X,Y)$ states that Y is a copy of the integer X in flavor i. We use $Copied(X)$ as a shorthand for $Copy_1(X,Y_1), \ldots, Copy_N(X,Y_N)$ with fresh variables $Y_1, \ldots, Y_N$; i.e., $Copied(X)$ says that X is available in all flavors.
- An IDB predicate $TAPE(Z,T,C)$, 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.

$$
\begin{aligned}
TAPE(Z,Z,C) &\;:\!\!-\;\; Copy_s(Z,C).\\
TAPE(Z,Z,C) &\;:\!\!-\;\; Succ(Z,X), Copy_b(X,C).\\
TAPE(Z,Z,C') &\;:\!\!-\;\; TAPE(Z,Z,C),\\
&\qquad\;\; Copy_b(X,C), Succ(X,Y), Copied(Y), Copy_b(Y,C').
\end{aligned}
$$

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:

$$
\begin{aligned}
TAPE(Z,T',C') &\;:\!\!-\;\; TAPE(Z,T,C_L), TAPE(Z,T,C), TAPE(Z,T,C_R),\\
&\qquad\;\; Copy_i(X_L,C_L), Copy_j(X,C), Copy_k(X_R,C_R),
\end{aligned}
$$

$$\begin{aligned}&Succ(X_L, X), Succ(X, X_R),\\&Copy_{\delta(i,j,k)}(X, C'),\\&Succ(T, T'), Copied(T').\end{aligned}$$

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:

$$\begin{aligned}TAPE(Z, T', C') \quad :\!-\quad & TAPE(Z, T, C), TAPE(Z, T, C_R),\\ & Copy_i(Z, C), Copy_j(X, C_R),\\ & Succ(Z, X),\\ & Copy_{\delta'(i,j)}(Z, C'),\\ & Succ(T, T'), Copied(T').\end{aligned}$$

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

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

$$TAPE(U, V, W) \quad :\!-\quad TAPE(Z, T, C), Copy_h(X, C).$$

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 predicate.*

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.