Safe Stratified Datalog With Integer Order Does not Have Syntax

Alexei P. Stolboushkin*
Department of Mathematics
UCLA
Los Angeles, CA 90024-1555
aps@math.ucla.edu

Michael A. Taitslin†
Department of Computer Science
Tver State University
Tver, Russia 170000
p000101@tversu.ru

September 14, 2006

Abstract

Stratified Datalog with integer (gap)-order (or $Datalog^{\neg,<_z}$) is considered. A $Datalog^{\neg,<_z}$-program is said to be safe iff its bottom-up processing terminates on all valid inputs. We prove that safe $Datalog^{\neg,<_z}$-programs do not have effective syntax in the sense that there is no recursively enumerable set S of safe $Datalog^{\neg,<_z}$-programs such that every safe $Datalog^{\neg,<_z}$-program is equivalent to a program in S.

Introduction

We consider databases organized over the domain $\mathbf{Z}$ of integer numbers together with the standard linear order. Even if the initial database state is finite, answers to very simple first-order queries may be infinite. For domains that admit elimination of quantifiers in first-order formulas, Kanellakis et al. [5] suggested using *finitely representable* (f.r.) database states, where extensional database predicates are defined as quantifier-free first-order formulas. Because of the elimination of quantifiers, answers to first-order queries in f.r. states are, again, f.r.

*This work has been partially supported by NSF Grant CCR 9403809.

†A part of this research was carried out while the author was visiting UCLA and supported in part by NSF Grant CCR 9403809.

Although the domain **Z** does not admit elimination of quantifiers, its extension with the so-called *gap-orders* $<_g$ for all natural numbers g already admits elimination of quantifiers. $x <_g y$ means $x + g < y$. Clearly, this extension of **Z** is conservative.

Thus, this extended domain admits effective *bottom-up evaluation* of first-order queries in *closed form* w.r.t. f.r. states. "Closed form" means that whenever you start from an f.r. state, you end up having an f.r. answer that can therefore be stored in the database and used in future queries as an extensional predicate. "Bottom-up evaluation" refers to the process of evaluating queries according to their structure, from inside-out, by constructing for each sub-formula a finite representation of its value. This process is much more efficient than the tuple-based evaluation.

However, the expressive power of first-order queries in this domain is severely limited. This motivated research into using constraint logic programs (see [2, 3]) for querying finitely representable databases over the integer order. Logic programs without negation, when they terminate, result in f.r. answers too. This means that the result of one program, or its negation, can be used as input for another program. This leads to the notion of Datalog with *stratified negation,* or $Datalog^{\neg,<_z}$, where negations are allowed, but only w.r.t. the intensional predicates whose computation already terminated (cf. [1]).

This machinery only works well, however, if the Datalog program terminates. If it does not, the construction collapses. One remedy is to consider only those $Datalog^{\neg,<_z}$-programs whose termination is guaranteed for all inputs. Such programs often are called *safe.* Notice that this definition is semantical in nature.[1]

Revesz [6] introduced a syntactical notion of safety for $Datalog^{\neg,<_z}$-programs, which guarantees semantical safety. The syntax is remarkably powerful—queries expressible in this syntax may have non-elementary complexity—and yet easy (=PTime) to check. As a matter of fact, it was not clear what kind of (semantically) safe queries, if any, could not be expressed in this syntax.

In this paper, we ultimately settle this problem by showing that no syntax exists for *all* semantically safe queries of $Datalog^{\neg,<_z}$. In particular, the syntax introduced in [6] is incomplete. Formally, we show that any recursively enumerable set of $Datalog^{\neg,<_z}$-programs either contains infinitely

[1] To be sure, the notion of safety only makes sense when a specific query evaluation algorithm is fixed. Within this paper, we concentrate on the bottom-up evaluation algorithm.

many unsafe programs, or does not contain any program for infinitely many safe $Datalog^{\neg,<_z}$-definable queries.

Of course, our result implies undecidability of safety for $Datalog^{\neg,<_z}$ as a whole, i.e., that one cannot decide for a program R whether it is safe. However, our result hits deeper in that we show impossibility of *any* syntactical safety restriction on the $Datalog^{\neg,<_z}$-programs that would not simultaneously be semantical. As a matter of fact, oftener than not an effective syntax for an undecidable class of programs does exist. By way of example, consider the ever popular class PTIME. Again, one cannot generally say whether a given program is in PTIME. However, it is easy to come up with a syntactical class of programs that consists of PTIME programs and covers the whole class PTIME function-wise.[2]

On the technical side, one of the main results of this paper is that, under the bottom-up semantics, for any Turing machine one can effectively construct a $Datalog^{\neg,<_z}$-program that computes the same function and is safe whenever the machine is total (Theorem 2.2). Although by appearance, the result looks similar to the one by P. Revesz (Proposition 2.3 in [7]) that any Turing-computable function is *expressible* by a query of $Datalog^{\neg,<_z}$, a closer look reveals that the the two results are altogether different. To emphasize only one distinction, the programs that *express* (total) Turing-computable functions in [7] need not terminate under the bottom-up semantics, hence, they may not be safe.

1 Definitions

Throughout this paper, we deal with the domain $\mathbf{Z}$ of integer numbers together with the relations $=$ of equality, $<$ of the integer linear order, and $<_g$ of the integer gap order, for all natural numbers g. $x <_g y$ means that $x + g < y$. The gap orders are first-order expressible using usual order but the reason to include them is that the resulting first-order language admits quantifier elimination. Atomic formulas that use equality, order or gap order only are called pure domain formulas, or *atoms.* Truth of atoms is defined the usual way.

$Datalog^{\neg,<_z}$-programs also use extra (not pure domain) relation names, each of a fixed arity. The extra names are of two types. The names of the first type are *extensional,* or *input,* names. They are thought of as the input

[2]Say, take only the programs that track their own execution time and terminate when a target polynomial is reached.

database.

The names of the second type are *intensional,* for their meaning is going to be computed by our program. Further, some of the intensional names are called *output* names, and the remaining intensional names are called *internal.* The idea is, although we are interested in computing output relations only, our computation itself may require generating intermediate results that are temporarily stored in internal names and discarded afterwards.

Then, we want to consider the stratified negation. That is to say, intensional names may be used under negation, but not sooner than their calculation terminates. Formally, intensional names are ranked by consecutive positive integer numbers with the smallest rank 1, and an intensional name may be used under negation only in defining an intensional name of a higher rank.

The syntax of $Datalog^{\neg,<_z}$ is traditional. A $Datalog^{\neg,<_z}$-program, which we sometimes also refer to as a *stratified program,* is a finite set of rules. Each rule has a head and a body. The body can be either empty or be a sequence of formulas. The head of a rule is an atomic formula of the form $P(x_1, x_2, \ldots, x_n)$ where $x_1, x_2, \ldots, x_n$ is a list of pairwise different variables and P is an intensional name of arity n. The rank of the rule is defined as the rank of P. Each formula in a body must be of one of the following form:

- an atom or its negation
- an atomic formula of the form $P(x_1, x_2, \ldots, x_k)$, where $x_1, x_2, \ldots, x_k$ is a list of variables and P is an extensional name of arity k
- an atomic formula of the form $P(x_1, x_2, \ldots, x_k)$, where $x_1, x_2, \ldots, x_k$ is a list of variables and P is a k-ary intensional name *whose rank is less than or equal to the rank of the rule*
- a formula of the form $\neg P(x_1, x_2, \ldots, x_k)$ where $x_1, x_2, \ldots, x_k$ is a list of variables and P is a k-ary intensional name *whose rank is less than the rank of the rule*

So a rule has the form

$$P(x_1, x_2, \ldots, x_n) \longleftarrow \varphi_1, \varphi_2, \ldots, \varphi_m,$$

where $\varphi_1, \varphi_2, \ldots, \varphi_m$ are formulas used in its body. Note that we do not require the variables occurring at the right side to occur at the left side; in

case they do not, the interpretation is existential—see a formal definition below.

A *state* for a $Datalog^{\neg,<_z}$-program R is an assignment of a set of integer number tuples of proper arity to every extensional name in R. We assume that all the sets are represented finitely by quantifier-free pure domain formulas. For a state for R, a program R is a mapping which assigns a set of integer number tuples $R(P)$ of proper arity for each intensional name P in R. $R(P)$ is defined step-by-step. Steps are enumerated by pairs of natural numbers. In steps (i, j), the mapping adds new tuples to $R(P)$ for intensional names P of the rank i.

For a step (i, j), $P(a_1, a_2, \ldots, a_n)$ is true iff either the tuple $a_1, a_2, \ldots, a_n$ was included in P before this step, or P is an extensional name and the tuple is contained in the set assigned to P by the state. $\neg P(a_1, a_2, \ldots, a_n)$ is true iff the rank of the intensional name P is less than i and $P(a_1, a_2, \ldots, a_n)$ is false. The first step is $(1, 0)$. Before the first step $R(P)$ are empty for all the intensional names.

A *rule instantiation* is defined as a substitution for each variable in the rule of an integer number. In the step (i, j), if there is no intensional name of the rank i, the program R stops. Otherwise, let

$$P(a_1, a_2, \ldots, a_n) \longleftarrow A_1, A_2, \ldots, A_m$$

be a rule instantiation for a rule in R, P of the rank i, and let $A_1, A_2, \ldots, A_m$ be true. Then the tuple $a_1, a_2, \ldots, a_n$ is added to $R(P)$ if the tuple is not contained in $R(P)$ yet. If no tuple is added at step (i, j) we proceed to the step $(i + 1, 0)$. Otherwise we proceed to the step $(i, j + 1)$.

A state is said to be *finite* for a program R iff the program stops in it. A program is finite or *safe* iff all states are finite for it. It can be observed that every $Datalog^{\neg,<_z}$-program without stratification (that is, a program with intensional symbols of rank 1 only) is safe.

Two safe programs R_1 and R_2 with the same extensional names are equivalent in a state iff they have the same output intensional names of the same arities, and for each output intensional name P, $R_1(P)$=$R_2(P)$ in the state. Two safe programs are equivalent iff they are equivalent in any state.

2 Impossibility of safe syntax

The goal of this section is to prove that there is no effective syntax for safe programs. Let us outline the idea of the proof. Consider Turing machines

in the alphabet $\{0, 1\}$, where 0 is used as the blank symbol. A one-way infinite input tape for such a machine contains finitely many 1's in the first few positions, which can be interpreted as a natural number in the unary notation, and all other cells contain 0. If, in an input, such a machine stops, it leaves finitely many 1's on the tape. We may consider the first uninterrupted string of 1's left on the tape to be the output natural number in the unary notation. Then, every machine defines a partial function on natural numbers.

We want to show that, for any such Turing machine, there exists a stratified program that computes the same function. First problem is, programs do not work with tapes, they work with database states. Then, these states are finitely representable, but not necessarily finite. However, we can develop a coding scheme that will represent any natural number in the unary notation in the form of a *finite* database state. Perhaps the simplest such coding is by a unary predicate N as follows.

- if N is assigned a set of a cardinality 0 or > 2, it does not represent a number
- N assigned a set of cardinality 1 or 2 represents the natural number

$$\max(N) - \min(N)$$

For a natural number n, let $\hat{n}$ denote its representation in the form of unary predicate. Consider a stratified program R whose signature includes a single extensional predicate $INPUT$ and a single output intensional predicate $OUTPUT$. If, for a number n, R terminates when $\hat{n}$ is assigned to $INPUT$, $OUTPUT$ is assigned a certain set. If this set is $\hat{m}$ for some m, we say that $R(n) = m$. Otherwise we say that $R(n) = 0$. This way, every such program R defines a partial function on natural numbers.

Henceforth, we consider programs whose only extensional predicate is a unary $INPUT$, and only output intensional predicate is a unary $OUTPUT$. However, these programs may have other internal intensional predicates. The idea of our proof is to show that total Turing machines and *safe* stratified programs are effectively translatable to each other in the way that preserves the functions they define on natural numbers. While it is known that total Turing machines do not have an effective syntax.

Theorem 2.1 *For any stratified program R there exists—and can be effectively constructed—a Turing machine M that defines the same partial function on natural numbers. If R is safe, the construction gives a total M.*

PROOF: The target program works as follows. Given a number n, it assigns $\hat{n}$ to $INPUT$ and then interprets the computation by R step-by-step, according to the stepwise definition of semantics of a stratified program. At each step (i, j), our machine stores the values of all intensional predicates in the form of first-order pure domain formulas.[3] This computation may never end, and then the result in n is undefined. However, if R terminates in $\hat{n}$, our interpretation terminates too, and as a result, we have a value for $OUTPUT$ in the form of a first-order pure domain formula (note that, since the pure domain theory admits elimination of quantifiers, this representation can be translated into a finite representation, although we do not need this).

Since the pure domain theory is decidable, we can effectively determine whether this value for $OUTPUT$ represents a number. If it does, we can determine which number, and then write this number in the unary notation down to the tape and stop, otherwise, we write 0 in the unary notation to the tape and stop.

This computation can be carried out by a Turing machine, although explicitly writing such a machine would be a long boring exercise. Finally, if R is safe, it always terminates, and particularly it terminates when working on representations for natural numbers. Hence, for a safe R, the machine is total. *Q.E.D.*

The other direction is slightly more technical:

Theorem 2.2 *For any Turing machine M there exists—and can be effectively constructed—a stratified program R that defines the same partial function on natural numbers. If M is total, the construction gives a safe R.*

PROOF: We want to concentrate on the case when the value assigned to $INPUT$ does represent a number. However, since we are going to construct a safe R, the case when it does not represent a number shall also be considered. It turns out that the program can decide whether $INPUT$ represents a number from the outset, and then if it does not, simply terminate right away with no matter which value for $OUTPUT$—notice that as far as the numerical function goes it does not matter.

For example, the program may use nullary intensional predicates BAD

[3] i.e., such formulas that only use constants and the integer (gap)-orders.

and $GOOD$ to make this determination as follows:

$$\begin{aligned} GOOD &\longleftarrow INPUT(x), \neg BAD \\ BAD &\longleftarrow INPUT(x), INPUT(y), INPUT(z), \\ &\quad x \neq y, x \neq z, y \neq z \end{aligned}$$

If $GOOD$ is true, it indicates that the database state does indeed represent a number. So all the other rules in our program may start with $GOOD$, and this guarantees termination for non-numerical inputs right away. We will omit $GOOD$ from the rules below, just to simplify notation.

Further, we need to select some number to serve as 0. For a numerical state, we can pick up the minimal element in $INPUT$ as 0. Formally, this can be done by a stratified program that defines a unary predicate $ZERO$ to include this minimal number only, however, to simplify notation, we will simply use 0 as a constant. Similarly, we will use a constant max for the maximal number in $INPUT$, and constants $1, 2, \ldots, |Q|$, where Q is the set of the internal states of our Turing machine. Clearly, using any of these constants in the rules is simply an abbreviation for a long routine list of formulas.

We will also use a binary successor relation S, $S(x, y) \Longleftrightarrow x + 1 = y$. This relation is definable using the gap orders as follows:

$$S(x, y) \longleftarrow x < y, \neg(x <_1 y)$$

To simulate computation by a Turing machine, we will use the following list of internal intensional predicates:

- ternary $TAPE$. $TAPE(i, j, k)$ indicates that in the step i of our computation the cell number j contains symbol k (0 or 1)
- binary $CELL$. $CELL(i, j)$ indicates that in the step i of our computation the cell number j is the current position of the machine
- binary $STATE$. $STATE(i, j)$ indicates that in the step i of our computation the internal state is j

The initial configuration of the Turing machine M can be explained by the

following rules (we assume that the initial state is always 0):

$$\begin{aligned} TAPE(i,j,k) &\longleftarrow i = 0, j \geqslant \max, k = 0 \\ TAPE(i,j,k) &\longleftarrow i = 0, j < \max, j \geqslant 0, k = 1 \end{aligned}$$

$$CELL(i,j) \longleftarrow i = 0, j = 0$$

$$STATE(i,j) \longleftarrow i = 0, j = 0$$

We can also include the following rule asserting that the cells which are different from the current position of the machine do not change:

$$TAPE(i,j,k) \longleftarrow S(\ell,i), TAPE(\ell,j,k), CELL(\ell,c), c \neq j$$

Further simulation of the Turing machine is done according to the rules of this machine. Generally, a rule is of the form:

$$(q,k) \longrightarrow (s,m,n,a),$$

and it indicates that when the machine sees the symbol k in the internal state q, it replaces k in the current cell with s, moves according to $m \in \{\text{left}, \text{right}, \text{stay}\}$, and, generally, goes into the internal state n. If, however, the movement prescribed by the rule is left, but the current cell is the leftmost and no left movement is possible, the machine goes into the internal state a.

For each such Turing machine rule, we include a set of rules into our program. For example, let:

$$(3,1) \longrightarrow (0, \text{left}, 2, 7)$$

be a rule in our Turing machine. It causes inclusion of the following set of rules into our program:

$$
\begin{array}{rcl}
TAPE(i,j,k) & \longleftarrow & S(\ell,i), TAPE(\ell,j,1), CELL(\ell,j), \\
 & & STATE(\ell,3), k=0 \\
 & & \\
CELL(i,j) & \longleftarrow & S(\ell,i), TAPE(\ell,j,1), CELL(\ell,j), \\
 & & STATE(\ell,3), j=0 \\
CELL(i,j) & \longleftarrow & S(\ell,i), TAPE(\ell,j,1), CELL(\ell,x), \\
 & & STATE(\ell,3), S(j,x), x \neq 0 \\
 & & \\
STATE(i,j) & \longleftarrow & S(\ell,i), TAPE(\ell,p,1), CELL(\ell,p), \\
 & & STATE(\ell,3), p=0, j=7 \\
STATE(i,j) & \longleftarrow & S(\ell,i), TAPE(\ell,p,1), CELL(\ell,p), \\
 & & STATE(\ell,3), p \neq 0, j=2
\end{array}
$$

Let us now define unary predicates $LAST$ and MAX as follows:

$$
\begin{array}{rcl}
LAST(\ell) & \longleftarrow & STATE(\ell,x), S(\ell,p), \\
 & & \neg STATE(p,0), \neg STATE(p,1), \ldots, \\
 & & \neg STATE(p,|Q|) \\
MAX(m) & \longleftarrow & LAST(\ell), n<m, n \geqslant 0, TAPE(\ell,n,0)
\end{array}
$$

Clearly, $LAST$ defines the last step in the computation by the Turing machine, if it stops. MAX defines the set of positive integer numbers m such that on the resulting tape left by our computation, there is at least one cell that is numbered lower than m and contains a 0. In particular, if the tape left by our Turing machine contains only 0's, the predicate is going to contain all positive integers.

We can finally define $OUTPUT$ as a predicate of the next rank as follows:

$$
\begin{array}{rcl}
OUTPUT(m) & \longleftarrow & m=0 \\
OUTPUT(m) & \longleftarrow & m>0, \neg MAX(m), S(m,x), MAX(x)
\end{array}
$$

Clearly, the value of $OUTPUT$ is going to be exactly the result of the computation by our Turing machine in our input, of course if this computation terminates. Since total Turing machines always terminate, our program R for a total Turing machine M is safe: in an $\hat{n}$, it computes $\hat{m}$ such that

$m = M(n)$, in a state that does not represent a natural number, it terminates right away. *Q.E.D.*

Corollary 2.3 *There is no effective syntax for safe programs.*

PROOF: Indeed, existence of such an effective syntax would, by Theorem 2.1, yield a recursive enumeration of total Turing machines such that every Turing computable total function is computed by a machine in this enumeration—here we use Theorem 2.2 to assure that every Turing computable total function appears in the enumeration. Such enumeration is known to be impossible (see [8]). *Q.E.D.*

References

[1] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[2] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[3] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[4] P.C. Kanellakis and D.Q. Goldin. Constraint programming and database query languages. In *Proc. International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, pages 96–120, 1994.

[5] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, August 1995.

[6] Peter Z. Revesz. Safe stratified Datalog with integer order programs. In *Principles and practice of constraint programming—CP '95 (Cassis, 1995)*, pages 154–169. Springer, Berlin, 1995.

[7] P.Z. Revesz. A closed form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[8] H. Rogers. *Theory of recursive functions and effective computability.* McGaw-Hill, 1967.