Finite Queries do not Have Effective Syntax[*]

Alexei P. Stolboushkin[†]
Fourth Dimension Software
555 Twin Dolphin Dr.
Redwood City, CA 94404
and UCLA Mathematics Department
Los Angeles, CA 90095
aps@4ds.com

Michael A. Taitslin[‡]
Department of Computer Science
Tver State University
Tver, Russia 170000
Michael.Taitslin@tversu.ru

May 30, 2003

Abstract

A relational query is called *finite,* or sometimes *safe,* iff it yields a finite answer in every database state.

The set of finite queries of relational calculus is known to be unsolvable. However, in many cases it is possible to impose syntactical restrictions on the class of queries that guarantee finiteness and do not reduce the expressive power of the calculus.

We show that unfortunately this is not always the case, as we construct a recursive domain with decidable theory where any solvable (or enumerable, for that matter) subclass of queries either contains an infinite query, or misses a finite one. We show that although any domain can always be extended to a domain with an effective syntax for finite

[*]A preliminary version of this paper appeared in the Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, San Jose, CA, May 22-25, 1995.

[†]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.

queries, such extensions of our domain necessarily have undecidable theories. The reason decidability of domain matters is, this property is in effect equivalent to the ability to answer queries effectively.

Using the same example, we further show undecidability of the problem of *relative finiteness,* which is, given a query and a database state, to decide upon finiteness of the query in this state.

This settles two long-standing open problems in the theory of relational databases.

1 Introduction

In the relational model of databases introduced by E.F. Codd [Cod70, Cod72] a database state is thought of as a finite collection of relations between elements. For example, a father-son relation can be represented in the form of one binary relation (or a two-column table). Names of the relations and their arities (numbers of argument places) are fixed and called a database scheme. Particular information stored in the relations of a given scheme is called a database state.

For instance, as we acquire more and more information about fathers and sons, the database states change, but the scheme (one binary relation) does not.

Database relations (tables) are always going to be finite.

For many applications it is convenient to pre-fix an infinite *domain* where the finite relations are going to be defined. It may be the set of natural numbers, or strings in an alphabet, or alike. Often, we also fix a set of domain functions and/or relations. In the case of natural numbers it may be $+$, $<$, etc. These functions/relations are infinite by their nature. When we refer to a domain, we mean the domain together with the set of domain functions and relations that we consider.

In formulating queries to our database, we use a *query language.* The basic query language is the language of first-order logic (see [End72]). It uses domain functions/relations as well as the relations from the database scheme.

For example, consider the above database about fathers and sons. This database can be organized over the infinite domain of strings, and we throw in the equality $=$. Let F be the father-son relation. Then the formula $M(x)$:

$$\exists y, z(y \neq z \wedge F(x,y) \wedge F(x,z))$$

results in the unary relation (one-column table) that consists of those x's who have more than one son. While $G(x, z)$:

$$\exists y(F(x, y) \wedge F(y, z))$$

produces the table of "grandfathers-grandsons".

Now we want the resulting relations (the answers to our queries) to be finite relations. The trouble with this is that some first-order formulas give infinite answers. Obviously, $\neg F(x, y)$ is such a formula. But worse than that, $M(x) \vee G(x, z)$ may give an infinite answer too, because M(x) does not bound z at all. There are some other cases when we get infinite answers.

Such formulas (that *may* give an infinite answer) are called *infinite,* or *unsafe,* as opposed to *finite,* or *safe*, formulas that always produce finite answers[1].

A closely related to safety is the notion of *domain-independence.* A formula is called domain-independent if it always yields an answer within the "active domain" of the query, the latter being the set of all constants used in the querying formula and/or elements contained in the database relations. The term "safe" is sometimes used to mean "domain-independent". However, within this paper we always "safe" to mean "finite".

Both these (as well as some other) definitions of safety have advantages and disadvantages, and have been extensively studied. We refer the reader to [Kif88] where M. Kifer gives a comparative analysis of these classes of formulas[2].

The situation was well understood in [Ull82] where J.D. Ullman raised the question of whether it is possible to tell safe formulas from unsafe. This has become known as the *safety problem.* This question was answered negatively by R.A. Di Paola [Di 69], M.Y. Vardi [Var81], and independently by A.K. Ailamazian et al. [AGSS86]. The answer is negative for any infinite domain as long as the database scheme contains at least one binary relation (or a relation of a higher arity). Also, the negative answer holds for whatever definition of safety is used.

Taking into account this result that the class of safe formulas is not solvable (and not even recursively enumerable), we will consider the three ways for dealing with the undecidability of the class of safe queries:

1. finitely-representable relations

2. relative safety

[1] Observe that the formula $M(x) \vee G(x, z)$ only gives an infinite answer if there is a person who parented two or more sons.

[2] It should be noted that, for some domains, these classes of formulas coincide.

3. effective syntax

But before we describe what these ways were and what progress has been made in each of these ways, we are going to spend a little time discussing domains.

1.1 Decidable domains

While we are concerned with practical issues, we need to restrict the class of domains that we consider.

¿From now on we assume that the equality $=$ always is available. The first natural requirement is that we only consider *recursive* domains, that is that all domain functions and predicates are recursive.

The second requirement is decidability of the first-order theory of the domain. In other words, we need to be able to effectively determine, given a first-order formula in the pure domain without free variables, whether this formula is true.

If we have constants for all the elements of the domain (which is a natural assumption for strings or natural numbers, for example), then the second requirement implies the first. But in general, they are independent. For instance, the theory of natural numbers with $+$ and $\times$ (aka Arithmetic) is undecidable, but recursive. On the other hand, real numbers with the standard set of operations is decidable, but not recursive[3].

The justification for decidability is simple. If the domain theory is not decidable, then the answers, whether finite or infinite, are not computable. Now consider a countable domain with decidable theory. Suppose we have constants for all elements of the domain. We now show that finite answers are computable. Indeed, suppose we know somehow that $F(\vec{x})$ gives a finite answer in the given database state. Since we have constants, and the state is a finite collection of finite relations, the formula $F(\vec{x})$ can be translated into a pure domain formula $F'(\vec{x})$ (this technique was used in [AGSS86, GSSS86]). For example, if a binary database relation R consists of the pairs

$$\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \ldots, \langle a_r, b_r \rangle,$$

we can replace each occurrence of $R(x, y)$ with

$$((x = a_1 \wedge y = b_1) \vee (x = a_2 \wedge y = b_2) \vee \ldots \vee (x = a_r \wedge y = b_r)).$$

[3]By the way, in this case some applications may justify dropping the recursiveness requirement—see [KKR90].

Now let us order all tuples of elements of the domain of the size of $\vec{x}$: $\vec{a_1}, \vec{a_2}, \ldots$.

Consider the formula $\exists \vec{x}(F'(\vec{x}))$. If it's false, then the answer is the empty relation. Suppose it is not false. Then by checking $F(\vec{a_1}), F(\vec{a_2}), \ldots$, one at a time, we find the first $\vec{a_k}$ that makes the formula $F(\vec{a_k})$ true. This is going to be the first row in the resulting relation.

Now take the formula $\exists \vec{x}(\vec{x} \neq \vec{a_k} \wedge (F'(\vec{x})))$. If it's false, we are done. Otherwise, by continuing checking tuples, we find the second tuple to include into the resulting relation.

And so forth.

Thus, we just described an algorithm (as inefficient as it is) of answering queries. Note that for at least safe queries, this algorithm always stops.

For a particular domain with decidable theory, we may or may not be able to come up with a more efficient algorithm of answering safe queries, but what we want to stress now is that decidability is definitely required for certain.

We now consider the three ways referred to previously for dealing with the undecidability of the class of safe queries.

1.2 Infinite relations

One way of handling the situation is to accept infinite relations that may result in answering infinite queries. Note that although infinite, these relations are finitely representable. Of course we can not actually generate the infinite relations (kiss "good-bye" the idea of printing results). But still, the database remains capable of answering questions of whether a certain tuple belongs to a relation, finite or infinite, or whether a certain fact holds. This approach was mentioned in [AGSS86, GSSS86], and developed into a nice theory by P.C. Kanellakis et al. [KKR90].

Since this approach is not dealt with in our paper, we will not discuss details here.

1.3 Relative safety

This approach also was introduced in [AGSS86, GSSS86]. The idea is as follows. Although the formula that you use may be infinite, in a given state you may be lucky and the answer may be finite. In this case, it would be desirable to get this finite answer. If you were not lucky and the answer is infinite, it would be desirable to get this information, that the answer is infinite, from the database. The driving idea is that unsafety of a query may be due to a rather unlikely situation that happens rarely or never at all.

The question is, is it possible to implement this approach? It is easy to see that, technically, the problem is, is it possible, for a *fixed* database state, to tell formulas with finite answers from those with infinite? This has become known as the *relative safety* problem [AGSS86, GSSS86][4]. By its very formulation this problem is domain-specific[5]. [AGSS86, GSSS86] and [AH91] showed that, unlike the safety problem, the relative safety problem is decidable for many domains.

Since [AGSS86, GSSS86], it had remained unknown whether the relative safety problem is decidable for each domain with decidable theory (or even with undecidable theory, for that matter). Our paper resolves this question for the first time by constructing a decidable domain for which relative finiteness is unsolvable.

1.4 Effective syntax

This approach is due to M.Y. Vardi [Var81] and J.D. Ullman [Ull82], and since then it has been developed in many publications, including [Ull88], [Van91], and [Hir91].

The idea is that although the set of safe formulas is unsolvable (and not even enumerable), it may be possible to impose certain syntactical restrictions on the class of formulas that we are going to use such that all the safe *queries* and only those can be formulated with these restrictions.

In other words, the problem can be put as follows. Does there exist a recursive subclass of safe formulas such that every safe formula is equivalent to one in this subclass?

We will call such a subclass a recursive *syntax.* One may consider recursively enumerable (r.e.) subclasses as well. As usual, for this kind of problems, existence of an r.e. syntax implies existence of a recursive syntax, so we are going to henceforth concentrate on the existence of a recursive syntax. J.D. Ullman in [Ull82] (and somewhat more clearly in [Ull88]) shows that a recursive syntax for domain-independent queries exists. A. Van Gelder and R. Topor [Van91] address the issue of efficiency of syntax. These papers dealt with the domain that has the equality predicate only.

For some primitive domains, for instance, for the domain with only the equality predicate, the classes of finite and domain-independent queries co-

[4] Originally, [AGSS86, GSSS86] used the term "state finiteness".

[5] As well as the safety problem, it is just that for the safety problem the negative answer holds for any domain.

incide, so the syntax actually work for both the classes. For some more developed domains, these classes differ, however, it is not hard to develop a syntax for finite queries for most of the domains considered in the literature. We consider the key examples in the next section.

This syntactical approach has definite advantages over the relative safety one, especially as in more and more cases the actual queries to databases are formulated by software rather than people. Thus, naturality of query languages becomes perhaps less important, while stability in certain sense (for example, finiteness of the query answer for every query) becomes more important.

On the other hand, it may be that unsafety of a formula is due to a rather rare situation, and then it may be useful to be able to use this formula for as long as the actual state-unsafety does not happen.

Again, since [AGSS86, GSSS86] it had remained unknown whether a recursive syntax for finite formulas exists for every domain. Our paper resolves this question for the first time by constructing a decidable domain without any recursive syntax for finite formulas.

In this paper we show that there exists a recursive domain with decidable theory in which (1) there is no recursive syntax for finite queries, and in which (2) the relative safety problem is undecidable.

The structure of the remainder of the paper is as follows. Section 1 briefly reviews some classifiable and solvable cases. Section 2 introduces a recursive domain with decidable theory which we call *theory of traces,* and shows (1) and (2) for this domain. The only technically hard part of the paper is the proof of decidability of the theory, and it is given in the Appendix.

2 Some Positive Cases

The examples considered in this section are intended to show that for many well-known domains with decidable or undecidable theories effective syntax is easy to construct. Ordered natural numbers were first considered in [AGSS86, GSSS86] and in [Hir91, AH91].

The simplest possible example to start with is an infinite domain with the only domain relation of equality. In this case, as we mentioned already, every finite formula is domain independent. To see why, suppose that a formula yields an answer that contains an element not in the active domain. Since this element is "loose", the formula will remain true if we substitute any other element in its place. Therefore, the answer is actually infinite.

Since the active domain can be captured with a simple formula, the easiest effective syntax for this case consists of restricting answers for all formulas to the active domain.

It is also easy to see that the relative safety problem, that is the problem, given a database state and a formula, to decide whether this formula yields a finite answer in *this state,* is decidable. Indeed, it suffices to fix an arbitrary element not in the active domain and check whether any tuple that only includes this element and active domain elements satisfies the formula.

2.1 Natural numbers with linear order

This is a somewhat more interesting case. Let us call this domain $N^<$. Of special interest is the fact that the results presented here remain true for **extensions** of the domain, that is even if new domain functions/relations are thrown in without changing the domain elements.

Fact 2.1

There is a finite formula not equivalent to any domain-independent formula, in any extension of the domain $N^<$.

HINT: It is known that the active domain is definable in the relational calculus. Let $\Delta(x)$ be the formula that defines it.

Then, obviously, the formula

$$(\forall y)(\Delta(y) \longrightarrow x > y) \wedge (\forall y)(y < x \longrightarrow (\exists z)(\Delta(z) \wedge z \geq y))$$

is finite, but not domain-independent. Indeed, this formula defines the smallest integer greater than all the active domain elements. This element is different for different active domains, hence the query is not domain-independent. However, the resulting relation of this query is always one-element, and therefore finite. *Q.E.D.*

However, it is not hard to come up with a recursive syntax for finite formulas for this domain[6].

Theorem 2.2 *There exists a recursive syntax for the set of finite queries for any extension of the domain $N^<$.*

PROOF: Let $\varphi(x_1, x_2, \ldots, x_k)$ be an arbitrary formula. The formula

$$\varphi^F(x_1, x_2, \ldots, x_k)$$

[6]The one presented below is, although the simplest, but not very efficient.

is called a finitization of $\varphi(x_1, x_2, \ldots, x_k)$ and is as follows:

$$\varphi(x_1, x_2, \ldots, x_k) \wedge (\exists m)(\forall x_1, x_2, \ldots, x_k)$$
$$(\varphi(x_1, x_2, \ldots, x_k) \longrightarrow \wedge_{i=1}^{k} x_i < m).$$

The second part of this formula says that there exists an element that is greater than any element in the answer yielded by $\varphi(x_1, x_2, \ldots, x_k)$.

It is easy to see that, first, the finitization of any formula is finite, and, second, the finitization of a finite formula is equivalent to this finite formula.

Therefore, the set of the finitizations of all formulas is a recursive syntax for finite queries. *Q.E.D.*

This simple trick works for a great deal of domains, including

1. natural numbers with $<$, $+$, and $-$ (aka Presburger Arithmetic),
2. natural numbers with $<$, $+$, and $\times$ (aka Arithmetic),

etc. Of course, Arithmetic is a bad domain in the sense that its theory is not decidable, and the only reason we bring this up at all is to show that the existence of a recursive syntax is, somewhat surprisingly, not related to decidability or recursivity:

Corollary 2.3 *There exists a domain with undecidable theory with a recursive syntax for finite queries. Such a domain can be chosen to be recursive or not recursive.*

For example, Arithmetic with an additional Arithmetical Truth predicate is a non-recursive domain, but according to this corollary, there exists an effective syntax for finite queries over this domain.

Here is another surprising corollary of our observations (we remind the reader that we only consider countable domains):

Corollary 2.4 *For any domain D there exists its extension D' with a recursive syntax for finite queries. If D is recursive, a recursive D' can be chosen.*

HINT: Take D' to be an extension of both D and $N^{<}$. *Q.E.D.*

As for the relative safety, we can formulate the following

Theorem 2.5 *The relative safety problem is decidable for any decidable extension of $N^{<}$.*

HINT: In a given state, a formula yields a finite answer iff it is equivalent to its finitization.

Now, in a fixed database state, any formula can be translated into pure domain formula.

However, the equivalence problem for pure domain formulas is, by the condition of the theorem, decidable. *Q.E.D.*

Essentially the same technique can be used for many other domains of interest that are not extensions of $N^<$. For example, integers with $<$ can be handled similarly after a minor modification of the finitization procedure.

We do not want to make any attempt for completeness, however, we are going to present another example just to show that the phenomenon of syntax does not completely rely on the discrete ordering.

2.2 Natural numbers with successor and =

We are going to consider *unordered* natural numbers with the successor function $'$, such that $x' = x + 1$. Let us call this domain N'. The reason we consider it is to make a technical point, that we do not necessarily need $<$ for effective syntax.

It is well-known that in N' ordering of numbers is not expressible.

A proof is: any linearly ordered structure has no loop as a substructure; but any FO formula over N' has a model with a loop as a substructure.

In contrast to the ordered case, simple tricks do not seem to work here, and an accurate proof will need to follow the procedure of quantifier elimination for this domain (see [Mal61, Mal62] and surveys in [ELTT65, Rab77]).

Observe that any formula is equivalent to a disjunction of the formulas of the form:

$$(\exists x)\Phi$$

or their negations, where Φ is a conjunction of formulas of the forms

$$x = y^{(n)}, \quad x^{(n)} = y, \quad x \neq y^{(n)}, \quad x^{(n)} \neq y.$$

We denote $x^{(0)} = x$ and $x^{(n+1)} = (x^{(n)})'$. If $x^{(n)} = y$, we write $x = y^{(-n)}$. For integer n and m, $(x^{(n)})^{(m)}$ is $x^{(n+m)}$.

If Φ contains inequalities only, the formula $(\exists x)\Phi$ is true. Otherwise Φ contains an equality. If it contains $x = y^{(n)}$, we simply replace all occurrences of x with $y^{(n)}$ and thus eliminate the quantifier. If the equality is of the form $x = y^{(-n)}$, we replace x with $y^{(-n)}$, and additionally add the conjunction:

$$y \neq 0 \wedge \ldots \wedge y \neq (n-1).$$

As a result, every (pure domain) formula in this domain is equivalent to a quantifier-free formula in the same domain. However, given a quantifier-free formula, it is easy to decide upon the finiteness of the answer it yields.

This takes care of the decidability of the relative safety problem.

Now to establish the existence of a recursive syntax in this case we would need to analyze the quantifier elimination even closer, and to notice that, if the quantifier depth of the formula is q, the new constants introduced under the quantifier-elimination procedure are within the distance 2^q from the constants in the original formula (and 0). The important fact is that this is not going to depend on the size of the formula otherwise. The proof is the same as in the previous case.

Let $\Delta_q^+(x)$ be the formula that describes the active domain *plus* the elements that are within the specified range 2^q of it. This formula is easy to write.

Then the effective syntax is the same as for domain-independent formulas, except it uses the extended active domain in place of the active domain.

In other words, a formula $\varphi(\vec{x})$ of quantifier depth q is finite iff the answer it yields only contains elements from the extended active domain $\Delta_q^+(x)$.

Hence,

Theorem 2.6 *The relative safety problem is decidable for the domain N'.*

Theorem 2.7 *There exists a recursive syntax for the set of finite queries for the domain N'.*

Although this technique is very domain-specific, the same ideas can be carried out for many other domains, say, for strings (words in a finite alphabet) with lexicographical ordering.

3 The Negative Case

The goal of this section is to present a recursive domain with decidable theory where there is no recursive syntax for finite queries, and where the relative safety problem is undecidable.

In quest for a decidable theory with this property the authors thoroughly reviewed many theories considered in [Rab77] and [ELTT65], however, were unable to find an appropriate theory.

Hence, we invented a new domain specially coined to serve our needs. The naturality of this domain is admittedly open to discussion.

A reader with a background in Recursion Theory may notice in our design certain similarities to Kleene's predicate. However, throwing in the full Kleene's predicate would make the theory undecidable. So what we actually are doing, is we are weakening Kleene's predicate to the extent that the first-order theory becomes decidable.

Now finiteness of a query implicitly involves a second-order property, and we manage to use this peculiar second-order property in conjunction with our predicate to express totality of recursive functions.

So much for the underlying informal ideas.

Now, formally, let us define the domain as follows.

The domain is the set of all possible words (or strings) in the alphabet $\{1, *, \#, -\}$. The signature contains the only ternary predicate symbol P, as well as all the constant words in the alphabet. Also, we consider the language with equality $=$.

Let us consider the standard single-tape Turing machines in the alphabet $\{1, -\}$. As usual, the machines use $-$ as a white-space marker. Initially, an input word $w \in \{1, -\}^*$ written on the tape is surrounded by infinitely many $-$ from both sides, and machines always start from reading the leftmost character of the word w. Internally, Turing machines use the two-character alphabet $\{1, -\}$ and throughout the computation, modify characters written on the tape. A machine can run forever, but if it stops, it leaves a finite word written on the tape. If at this moment the tape only contains $-$ in all positions, the result of the computation is defined as the empty word ϵ. Otherwise, the result is the leftmost word in the alphabet $\{1\}$ written on the tape and surrounded by $-$.

The Turing machines themselves can be represented as strings in the alphabet $\{1, -, \#\}$ with $\#$ being a delimiter (we require that every machine contains at least one $\#$). The details of a particular representation are not otherwise important.

Let a word $w \in \{1, -\}^*$ and a Turing machine $M \in \{1, -, \#\}^+$ be given. We now define a *trace* of M in w as a sequence of "snapshots" of a partial computation of M in w separated with $*$. A trace starts with $M*$ and then, for each step in the computation, contains the internal state of the machine M, the tape (the minimal part of it that covers all non-$-$ characters), and the position of the head in the tape, all separated with $*$. For instance, the first snapshot always is $1 * w *$, indicating that the machine is in the internal state 1, the tape contains the word w surrounded with $-$, and the position of the head within w is 0 (empty word in the unary system).

Thus, if M does not stop in w, there are infinitely many different traces of M in w. However, if it does stop in w, then the number of different traces is finite.

A trace therefore is a word in the alphabet $\{1, *, \#, -\}$.

Note that the machines, the input words, and the traces, being all written in different alphabets, do not intersect. Also, there are words that are of neither of these three types, which we will call "other words".

Now the only (ternary) signature predicate P is defined as follows.

$P(M, w, p)$ iff M, w, p are a machine, an input word, and a trace, respectively, and p is a trace of M in w. The equality is also allowed. Let us denote this domain $\mathcal{T}$.

By the *Theory of Traces* we mean the set of true first-order pure domain sentences of $\mathcal{T}$. In the Appendix we establish decidability of the Theory of Traces (Corollary A.4).

Theorem 3.1 *The set of finite formulas of the domain $\mathcal{T}$ does not have an effective syntax.*

PROOF: Consider a database scheme that consists of one constant symbol[7] c.

Given a Turing machine M, consider the following formula $\psi_M(x)$:

$$P(M, c, x).$$

Observe that the formula $\psi_M(x)$ is finite iff M is total. Indeed, if M is total, then, for any c, only finitely many x's may satisfy $P(M, c, x)$.

If, on the other hand, M is not total, there exists c such that infinitely many traces x satisfy $P(M, c, x)$. Then, obviously, $\psi_M(x)$ is infinite.

Now suppose that the theorem does not hold. Then there exists a recursive enumeration $\phi_1(x), \phi_2(x), \ldots$ of finite formulas (that use c in addition to the domain constants and to the predicate P) with one free variable such that any finite formula with one free variable is equivalent to one in this list.

Without loss of generality, we may assume that a variable, say z, is not used in the formulas of this list.

Consider a recursive enumeration of all, total or not, Turing machines, $M_1, M_2, \ldots$.

Given a machine M_k and a formula $\phi_r(x)$, consider the formula

$$(\forall z)(\forall x)\left(\psi_{M_k}(x)\left[\frac{z}{c}\right] \longleftrightarrow \phi_r(x)\left[\frac{z}{c}\right]\right),$$

[7] Of course this formally is not a database scheme, but this technicality will be taken care of later.

where $[\frac{z}{c}]$ is the operation of substituting the variable z for the constant symbol c in a formula.

This last formula is therefore a pure domain formula, and because the decidability of the theory (see Corollary A.4), we can check whether it is true or not.

Now if it happens to be true, we know, that M_k is a total machine, because the truth of this sentence implies that $\psi_{M_k}(x)$ is finite.

On the other hand, if M_k is total, then ψ_{M_k} is finite, and, therefore, for some r, the above sentence is going to be true, for $\phi_1(x), \phi_2(x), \ldots$ include all finite queries with one variable.

Hence, by continuously analyzing all pairs of k and r, we can establish a recursive enumeration of *all* total Turing machines.

But this is known to be impossible. A straightforward proof of this fact can be obtained by a simple diagonalization.

Hence, a contradiction.

Finally, notice that we do not need to stick with the constant c. A database scheme may contain, say, one unary relation R instead of the constant symbol, and then we will define the totality formula $\psi_M(x)$ as follows:

$$(\forall x, y)(R(x) \wedge R(y) \longrightarrow x = y) \wedge (\exists y)(R(y) \wedge P(M, y, x)).$$

The same proof can be carried out here with minor adjustments. *Q.E.D.*

By Corollary 2.4, we know that for some recursive extension $\mathcal{T}'$ of $\mathcal{T}$, an effective syntax for all finite queries does exist. Unfortunately, we are now going to show that the theory of such a $\mathcal{T}'$ is necessarily undecidable.

Corollary 3.2 *For no extension $\mathcal{T}'$ of $\mathcal{T}$ that has a decidable theory, a recursive syntax exists for finite queries.*

PROOF: The proof of Theorem 3.1 continues to work. *Q.E.D.*

The situation is no better with relative safety:

Theorem 3.3 *The relative safety problem is undecidable for $\mathcal{T}$.*

PROOF: In the notation of Theorem 3.1, notice that $\psi_M(x)$ is finite in the state c iff M stops starting from the value of c. However, it is undecidable to determine whether a Turing machine stops on an input. *Q.E.D.*

4 Conclusion

We introduced a domain $\mathcal{T}$ which encodes all possible computations. This domain is arguably a natural choice in several applications related to storing results of computations, for example in databases of computational experiments. This domain also satisfies the two natural criteria for a practical database domain that we discussed, namely, recursivity and decidability of the theory. Together, these two guarantee that the answer to any query, if finite, can be effectively computed.

This *"if finite"* part is the center of our study. We showed that, unfortunately, in this domain $\mathcal{T}$, it is undecidable to say whether a given query in a given database state yields a finite answer (Theorem 3.3).

An alternative approach is to invent an effective syntax that would cover all the finite queries. We showed that this approach fails for the domain $\mathcal{T}$ as well (Theorem 3.1). In Corollary 2.4, we showed that any recursive domain can be extended to a recursive domain with a recursive syntax for finite queries. In view of this, we also investigated the possibility that the reason for the effective syntax to not exist is in the weakness of $\mathcal{T}$, rather than in its excessive strength. However, we showed that it gets about as bad as it could be, and that no *decidable* extension of $\mathcal{T}$ provides for an effective syntax for finite formulas (Corollary 3.2).

To sum this up, in $\mathcal{T}$ finite queries are effectively answerable, but in $\mathcal{T}$ or in any extension of $\mathcal{T}$ where finite queries continue to be effectively answerable, they do not have an effective syntax.

Acknowledgments

We are grateful to Moshe Vardi and the late Paris Kanellakis for helpful comments.

References

[AGSS86] A.K. Ailamazian, M.M. Gilula, A.P. Stolboushkin, and G.F. Schwarz. Reduction of the relational model with infinite domains to the case of finite domains. *Doklady Akademii nauk SSSR*, 286(2):308–311, 1986. Russian; Translation in *S*oviet Physics Doklady, **3**1, 11–13.

[AH91] A. Avron and J. Hirshfeld. On first order database query languages. In *Proc. 6th IEEE Symp. on Logic in Computer Science*, pages 226–231, 1991.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[Cod72] E.F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 33–64. Prentice-Hall, 1972.

[Di 69] R.A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *Journal of the ACM*, 16(2):324–327, 1969.

[ELTT65] Yu.L. Ershov, I.A. Lavrov, A.D. Taimanov, and M.A. Taitslin. Elementary theories. *Russian Mathematical Surveys*, 20(4):35–105, 1965.

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

[GSSS86] M.M. Gilula, A.P. Stolboushkin, G.F. Schwarz, and K.V. Shvachko. Some algorithmic problems in the theory of relational databases. In A.K. Ailamazian, editor, *Problem-Oriented Computational Systems*, volume 125 of *Problems in Cybernetics*, pages 81–91. USSR Academy of Sciences, Moscow, USSR, 1986. Russian.

[Hir91] J. Hirshfeld. Safe queries in relational databases with functions. In *CSL '91: 5th Workshop on Computer Science Logic*, LNCS, pages 173–183. Springer-Verlag, 1991.

[Kif88] M. Kifer. On safety, domain independence, and capturability of database queries. In *Proc. Third International Conference on Data and Knowledge Bases*, 1988.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 299–313, 1990.

[Mal61] A.I. Mal'cev. On elementary theories of locally free universal algebras. *Doklady Akademii nauk SSSR*, 138:1009–1012, 1961. Russian.

[Mal62] A.I. Mal'cev. Axiomatizable classes of locally free algebras of some types. *Sibirsky Mathematical journal*, 3:729–743, 1962. Russian.

[Rab77] M.O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 3. North-Holland, Amsterdam, New York, Oxford, 1977.

[Ull82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 2 edition, 1982.

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

[Van91] Van Gelder, A. and R.W. Topor. Safety and translation of relational calculus queries. *ACM Trans. on Database Systems*, 16(2):235–278, June 1991.

[Var81] M.Y. Vardi. The decision problem for database dependencies. *Information Processing Letters*, pages 251–254, 1981.

A Appendix: Decidability of the Theory of Traces

In this appendix we establish decidability of the Theory of Traces introduced in this paper. By itself, this is a somewhat surprising result, since the model that we consider contains full information about all possible computations.

The Theory of Traces and its domain $\mathcal{T}$ are defined in Section 3.

Let us start with a trivial observation.

Fact A.1 *Domain $\mathcal{T}$ is recursive.*

For recursive models, one of the most common techniques of proving decidability of theories is elimination of quantifiers. In other words, suppose we are able to come up with an effective procedure of constructing, given an arbitrary sentence, an equivalent quantifier-free sentence. Then the theory is decidable, because the model is recursive.

Such theories are said to admit elimination of quantifiers.

However, the Theory of Traces does not admit elimination of quantifiers by itself. The usual technique is then to try to extend the signature by throwing in new recursive functions and/or predicates that would allow the theory to admit elimination of quantifiers.

This technique is indeed successful here, as we will see, and actually all the new functions and predicates are first order expressible in the old theory.

Now the new signature symbols. We introduce four new unary predicates M, W, T, and O that separate, respectively, Turing machines, input words, traces, and "other words".

We introduce infinitely many new unary symbols $\{B_w\}_{w \in W}$ defined as follows. $B_w(x)$ iff the input word x starts with w, or $w \in x\{-\}^*$.

We also introduce infinitely many new binary symbols $\{D_i\}_{i=1}^{\infty}$ defined as follows. $D_i(M, w)$ iff the machine M has at least i different traces in w.

Finally, we introduce two unary functions w and m such that

$$w(x) = \begin{cases} v & \text{if } x \text{ is a trace of some machine} \\ & \text{in } v, \\ \epsilon & \text{otherwise.} \end{cases}$$

$$m(x) = \begin{cases} M & \text{if } x \text{ is a trace of the machine } M \\ & \text{in some word,} \\ \epsilon & \text{otherwise.} \end{cases}$$

ϵ here (and everywhere else) is empty word.

It is easy to see that (1) all these functions and predicates are recursive, (2) they all are expressible by first order formulas of the original signature (the meaning of that for functions is that formulas with these functions can be rewritten without), and (3) the predicate P of the Theory of Traces is first order expressible using the new signature.

The only nontrivial case is expressibility of B_w. In this case, the formula in the original signature will assert that a constant Turing machine that reads w and then goes into an infinite loop (and that however stops if the attempt to read w fails), has at least $|w|$ different traces.

Let us call the new theory the *Reach Theory of Traces.*

We also introduce the predicates $\{E_i\}_{i=1}^{\infty}$ that are dual to $\{D_i\}_{i=1}^{\infty}$: $E_i(M, w)$ iff the machine M has exactly i different traces in w. Obviously, they are quantifier-free expressible via $\{D_i\}_{i=1}^{\infty}$.

Although it may seem that the Theory of Traces is weak, it can actually be used for expressing very involved properties of computations by Turing machines. For instance, the following lemma deals with the problem of existence of a Turing machine that would satisfy certain conditions on its initial behavior.

Lemma A.2

Let

$$v_1, v_2, \ldots, v_k, \quad u_1, u_2, \ldots, u_l$$

be constant input words, and let

$$i_1, i_2, \ldots, i_k, \quad j_1, j_2, \ldots, j_l$$

be natural numbers. Let all $v_1, v_2, \ldots, v_k$*,* $u_1, u_2, \ldots, u_l$ *be of length greater than the maximal of* $i_1, i_2, \ldots, i_k$*,* $j_1, j_2, \ldots, j_l$*.*

Consider the following formula:

$$(\exists x)(D_{i_1}(x, v_1) \wedge D_{i_2}(x, v_2) \wedge \ldots \wedge D_{i_k}(x, v_k) \wedge$$
$$E_{j_1}(x, u_1) \wedge E_{j_2}(x, u_2) \wedge \ldots \wedge E_{j_l}(x, u_l)).$$

The formula is true in the Reach Theory of Traces iff for no pair r, q *of natural numbers any of the following holds:*

1. $i_r > j_q$ *and the prefixes of* v_r *and* u_q *of length* j_q *coincide;*
2. $j_r > j_q$ *and the prefixes of* u_r *and* u_q *of length* j_q *coincide.*

PROOF: If the formula is true, then the condition is trivially satisfied.

Suppose the condition holds. We can explicitly construct the Turing machine that would witness the quantifier. This machine (that can actually be written as a finite automaton) stops at exactly the specified words in the specified numbers of steps. *Q.E.D.*

Theorem A.3 (Quantifier Elimination Theorem)

Reach Theory of Traces admits elimination of quantifiers.

PROOF: As usual, we only need to consider the case when ψ is a quantifier-free formula, and to show how to eliminate the existential quantifier from $\exists x(\psi)$.

Further, because the existential quantifier can be distributed to a disjunction, we may assume that ψ is a conjunction of atomic formulas and their negations, and each of the atomic sub-formulas contains x.

Further, since we have a partition of the domain into four disjoint sets M, W, T, and O, we may assume that the quantifier is specialized by one of these predicates.

Moreover, because of the definition of the only two functions, any nested term always equals ϵ. So we may assume that all terms are not nested.

As well as that, we may assume that ψ does not contain any equality of the form $x = t$, for if it does contain $x = t$ then we can simply substitute t for x and thus eliminate the quantifier.

Let us consider all the four cases separately.

Case M: $(\exists x \in M)(\psi)$. Since x is said to be in M, any $B_v(x)$ is false no matter what v is. Then, $w(x) = m(x) = \epsilon$.

Since negations of D's and E's are equivalent to conjunctions of E's and D's, respectively, we may assume that ψ is of the following form:

$$\begin{aligned} x \neq t_1 \wedge x \neq t_2 \wedge \ldots \wedge x \neq t_n \wedge \\ D_{i_1}(x, v_1) \wedge D_{i_2}(x, v_2) \wedge \ldots \wedge D_{i_k}(x, v_k) \wedge \\ E_{j_1}(x, u_1) \wedge E_{j_2}(x, u_2) \wedge \ldots \wedge E_{j_l}(x, u_l), \end{aligned} \tag{1}$$

where $v_1, v_2, \ldots, v_k, u_1, u_2, \ldots, u_l$ are terms with no x's.

Using B_v for all input words whose length does not exceed the maximum of $i_1, i_2, \ldots, i_k, j_1, j_2, \ldots, j_l$, we can further reduce ψ to a finite (although big) disjunction of formulas where second arguments of D's and E's are constant words.

For instance, $E_2(x, y)$ is equivalent to

$$\begin{aligned} B_{--}(y) \wedge E_2(x, --) \vee \\ B_{-1}(y) \wedge E_2(x, -1) \vee \\ B_{1-}(y) \wedge E_2(x, 1-) \vee \\ B_{11}(y) \wedge E_2(x, 11). \end{aligned}$$

To sum up, we may assume that in the formula 1 all

$$v_1, v_2, \ldots, v_k, \quad u_1, u_2, \ldots, u_l$$

are constant words.

By Lemma A.2, we can find out whether

$$\begin{aligned} D_{i_1}(x, v_1) \wedge D_{i_2}(x, v_2) \wedge \ldots \wedge D_{i_k}(x, v_k) \wedge \\ E_{j_1}(x, u_1) \wedge E_{j_2}(x, u_2) \wedge \ldots \wedge E_{j_l}(x, u_l) \end{aligned}$$

is satisfiable. But if it is satisfiable, it is satisfiable for infinitely many different machines x, and we can also satisfy the system of inequalities.

Case W: $(\exists x \in W)(\psi)$. We start from observing that $\neg B_s(x)$ is translatable into conjunction of some $B_{s_1}(x) \wedge \ldots \wedge B_{s_r}(x)$. Then, any conjunction $B_{s_1}(x) \wedge \ldots \wedge B_{s_r}(x)$ is either equivalent to one of its members, or is false (if some two indices s_{r_1}, s_{r_1} are not prefixes of one another).

Similarly to the case M, we assume that ψ is of the form

$$\begin{aligned} x \neq t_1 \wedge x \neq t_2 \wedge \ldots \wedge x \neq t_n \wedge \\ B_s(x) \wedge \end{aligned} \tag{2}$$

$$\begin{aligned} D_{i_1}(v_1, x) \wedge D_{i_2}(v_2, x) \wedge \ldots \wedge D_{i_k}(v_k, x) \wedge \\ E_{j_1}(u_1, x) \wedge E_{j_2}(u_2, x) \wedge \ldots \wedge E_{j_l}(u_l, x), \end{aligned} \tag{3}$$

where $v_1, v_2, \ldots, v_k, u_1, u_2, \ldots, u_l$ are terms with no x's.

Now if such an input word x exists, then there exists also a short x (of length less than or equal to the maximum of $i_1, i_2, \ldots, i_k, j_1, j_2, \ldots, j_l$ and the length of s. Therefore, the formula is translatable into a finite disjunction of ψ's with constants substituted for x.

Case T: $(\exists x \in T)(\psi)$. This is perhaps the messiest case of all the four. Not much can be assumed here for both input words, and machines, can be obtained from x by means of the signature functions.

As before, the second arguments of all D_i and E_i predicates may be assumed to be constant input words (see Case M for an explanation). Further, we may assume that in ψ there is a unique, and positive, occurrence of $B_s(x)$ for a certain s (see Case W).

So generally ψ is a conjunction of the form

$$[m(x) = t\wedge][w(x) = v\wedge] \tag{4}$$

$$x \neq p_1 \wedge \ldots \wedge x \neq p_n \wedge \tag{5}$$

$$m(x) \neq t_1 \wedge \ldots \wedge m(x) \neq t_r \wedge \tag{6}$$

$$B_s(w(x)) \wedge w(x) \neq y_1 \wedge \ldots \wedge w(x) \neq y_q \wedge \tag{7}$$

$$D_{i_1}(m(x), v_1) \wedge D_{i_2}(m(x), v_2) \wedge \ldots \wedge D_{i_k}(m(x), v_k) \wedge \tag{8}$$

$$E_{j_1}(m(x), u_1) \wedge E_{j_2}(m(x), u_2) \wedge \ldots \wedge E_{j_l}(m(x), u_l), \tag{9}$$

where $v_1, v_2, \ldots, v_k, u_1, u_2, \ldots, u_l$ are constants.

Square brackets in 4 indicate that either one of the two equalities, or even both of them together, may be missing. It is clear that different equalities of this form can be eliminated.

We will separately treat the situations when the equalities in 4 are present or absent, which makes four cases.

Case $T - 1$: both equalities in 4 are missing. In this case it is sufficient to be able to satisfy the system 8—9 of assertions, for if it is satisfiable, the entire formula is satisfiable. Indeed, there are infinitely many behaviorally equivalent but syntactically different machines. On the other hand, we can always find an input word that satisfies 7 and form, together with a chosen machine, a trace different from $p_1, p_2, \ldots, p_n$.

Now the satisfiability of the system of assertions can be decided upon by Lemma A.2.

Case $T-2$: in 4, only $m(x) = t$ is present.

Then, in the sought trace, the machine is going to be concrete, and we can simply substitute t for $m(x)$ everywhere in the formula.

That leaves the conjunction of 4, 5, and 7. Which conjunction is obviously always satisfiable[8].

Case $T-3$: in 4, only $w(x) = v$ is present.

Similarly to the previous case, 7 then disappears.

Now, similarly to the case $T-1$, satisfiability of the formula is reduced to satisfiability of the system 8—9 of assertions.

Case $T-4$: both equalities in 4 are present.

As in the cases $T-2$ and $T-3$, 6—9 disappear.

So the remaining part of the formula is as follows:

$$m(x) = t \wedge w(x) = v \wedge \tag{10}$$

$$x \neq p_1 \wedge \ldots \wedge x \neq p_n. \tag{11}$$

Some of the traces $p_1, p_2, \ldots, p_n$ may have the same machine t, and some may have different ones. We can assert that, for instance, p_1 has the machine t, by the formula $m(p_1) = t$. By considering all possible combinations of true/false assertions about the machines of $p_1, p_2, \ldots, p_n$ we will get a finite disjunction of formulas.

Now if for a particular disjunct we know that, say, the machine of p_1 is not t, then the inequality $x \neq p_1$ is true.

Hence, in the above formula we may start from the assumption that all p_i have the same machine t.

Similarly, we may (after doing the same procedure with inputs of traces $p_1, p_2, \ldots, p_n$) assume that all p_i have the same input v.

Further, by doing the same disjunction trick with equalities/inequalities between $p_1, p_2, \ldots, p_n$, we can justify the assumption that all $p_1, p_2, \ldots, p_n$ are pairwise different.

The formula that we will end up with is as follows:

$$m(x) = t \wedge w(x) = v \wedge$$
$$x \neq p_1 \wedge \ldots \wedge x \neq p_n \wedge$$

[8]Note here that the formula itself may or may not be satisfiable depending on the x-free part obtained from 6, 8, and 9

$$
\begin{aligned}
\wedge_{i \neq j, i,j=1}^{n} p_i \neq p_j \wedge \\
\wedge_{i=1}^{n} w(p_i) = v \\
\wedge_{i=1}^{n} m(p_i) = t.
\end{aligned}
$$

It is now obvious that existence of x satisfying the above formula asserts exactly existence of at least $n+1$ different traces of computations of t in v, and is therefore equivalent to $D_{n+1}(t, v)$.

Case O: $(\exists x \in O)(\psi)$.

This is a trivial case. Since no functions or relations can reasonably deal with this x, ψ essentially is a system of inequalities, and then a satisfying x always exists.

Q.E.D.

Corollary A.4 *Reach Theory of Traces, and Theory of Traces, are decidable.*