PTIME Queries Revisited

Alan Nash[1], Jeff Remmel[2], and Victor Vianu[3]

[1] Mathematics and CSE Departments, UC San Diego, La Jolla, CA 92093, USA
[2] Mathematics Department, UC San Diego, La Jolla, CA 92093, USA
[3] CSE Department, UC San Diego, La Jolla, CA 92093, USA

Abstract. The existence of a language expressing precisely the PTIME queries on arbitrary structures remains the central open problem in the theory of database query languages. As it turns out, two variants of this question have been formulated. Surprisingly, despite the importance of the problem, the relationship between these variants has not been systematically explored. A first contribution of the present paper is to revisit the basic definitions and clarify the connection between these two variants. We then investigate two relaxations to the original problem that appear as tempting alternatives in the absence of a language for the PTIME queries. The first consists in trying to express the PTIME queries using a richer language that can also express queries beyond PTIME, but for which there exists a query processor evaluating all PTIME queries in PTIME. The second approach, studied by many researchers, is to focus on PTIME properties on restricted sets of graphs. Our results are mostly negative, and point out limitations to both approaches. Finally, we turn to a natural class of languages that we call finitely generated, whose syntax is obtained by applying a fixed set of constructors to a given set of building blocks. We identify a broad class of such languages that cannot express all the PTIME queries.

1 Introduction

The existence of a language expressing precisely the PTIME queries on arbitrary structures remains the most tantalizing open problem in the theory of database query languages. This question was first raised by Chandra and Harel [3] and later reformulated by Gurevich [9] who also stated the conjecture (now widely accepted) that no such language exists.

To reason about the existence of a language for the PTIME queries, one has to first come up with a very broad definition of query language (or logic), then define what it means for a logic to express the PTIME queries. It turns out that two such definitions have been proposed. To our knowledge, despite the importance of the problem, the relationship between these variants has not been systematically explored. We show that these two variants are different and may conceivably have distinct answers.

It is generally accepted that a query language specifies queries using expressions consisting of strings of symbols over some alphabet. We call these the

programs of the language. A first requirement is that a language should have effective syntax, meaning that its syntactically correct programs can be effectively enumerated. The semantics of a language L associates to each program in L a particular query. For simplicity, and since arbitrary structures can be efficiently represented as graphs [9], we focus in this paper on queries that are properties of graphs. Thus, we consider languages whose semantics associates to each program in the language a property of graphs. We say that L *expresses* the set of PTIME properties of graphs, denoted by P_G, if the set of graph properties associated to programs in L by its semantics is precisely P_G.

It is clear that simply having a language L expressing P_G is not satisfactory. At a minimum, we would like to be able to effectively and uniformly evaluate the programs in L. In other words, we would like to have a Turing machine E that, given as input a program p in L and a graph G, decides whether G satisfies the property defined by p. We call E an *evaluator* for L. Intuitively, an evaluator corresponds to a query processor for L. If such an evaluator exists, we call the language L *computable*. Since we are targeting the PTIME properties, we would further like E to uniformly evaluate every fixed program p in L in polynomial-time with respect to G. If such E exists, we call L *P-bounded*. The first formulation of the problem of the existence of a language for PTIME, by Chandra and Harel [3], asks whether there exists a P-bounded language expressing the PTIME queries. Most other definitions (e.g. [9, 5]) further require that an explicit polynomial bound for the number of steps of each program in L as evaluated by the evaluator be effectively computable. In this case, we call L *effectively P-bounded*. The above notions extend naturally to properties of graphs: we call a set of PTIME properties of graphs computable, P-bounded, or effectively P-bounded iff there exists such a language expressing it.

Our first set of results shows that these two notions are distinct. In terms of languages, we show that:

(i) there exists a computable language for P_G that is not P-bounded and
(ii) if P_G is P-bounded, then there exists a P-bounded language for P_G that is not effectively P-bounded.

We also show that (i) and (ii) above hold for any computable subset of P_G that includes all finite properties.

It is legitimate to wonder whether P-bounded languages that are not effectively P-bounded are mere curiosities that can be avoided: given a P-bounded language, is it always possible to find an effectively P-bounded language for the same set of properties? We answer this question (and the corresponding one for computable vs. P-bounded) in the negative by showing the following:

(iii) there exist sets of PTIME graph properties that are computable, but not P-bounded and
(iv) there exist sets of PTIME graph properties that are P-bounded, but not effectively P-bounded.

In the special case of P_G, it remains open whether the existence of a P-bounded language for P_G implies the existence of an effectively P-bounded one.

In the absence of a language for the PTIME properties, various relaxations to the problem appear to offer tempting alternatives. We examine two natural approaches. The first consists in trying to capture P_G using a richer language allowing to express properties that likely lie beyond PTIME. Suppose we have a language L that expresses all of P_G, and possibly more. For example, such a language is Existential Second-Order logic ($\exists$SO), that is known to express the NP properties [6]. Assuming that P $\neq$ NP, some of the formulas in $\exists$SO express polynomial-time properties, while others do not. Furthermore, under the same assumption, it is easily shown, using Trakhtenbrot's theorem, that it is undecidable whether a given formula expresses a PTIME property. However, it is conceivable that $\exists$SO has an evaluator E that happens to evaluate every polynomial-time property in polynomial time. This would mean that a user could not only express all polynomial-time properties using $\exists$SO, but such properties could actually be evaluated uniformly in polynomial time. Short of an actual language for P_G, this would seem like a good alternative. Unfortunately, this solution is not a real alternative to a language for P_G. Indeed, we show that, if $\exists$SO (or any language that can express all of P_G) has an evaluator that computes all P_G properties in polynomial time, then there exists a P-bounded language expressing *exactly* P_G. Thus, the alternative formulation is no easier than the original problem of finding a language for P_G.

The second alternative to finding a language for P_G is to focus on interesting subsets of graphs rather than all graphs. For example, a beautiful result by Grohe shows that the PTIME properties of planar graphs can be expressed by a P-bounded language, specifically FO+LFP augmented with counting [8]. Such results raise the hope that the PTIME properties on larger and larger subsets of graphs can be captured and perhaps that, once a certain threshold is overcome, this might be extended to any set of PTIME properties. However, we prove a result that suggests there is no such threshold. It states that, for every PTIME-recognizable class $\mathcal{G}$ of graphs with infinite complement there exists a set of PTIME properties of graphs that includes all the PTIME properties of graphs in $\mathcal{G}$ and for which there is a computable, yet not P-bounded language. We also show an analogous result for effectively P-bounded languages. Of course, this does not invalidate the program of finding increasingly large sets of graphs whose PTIME properties have an (effectively) P-bounded language.

The notion of language used above is extremely general and may allow for very artificial constructs, not acceptable in real query languages. Given the difficulty in settling the question of the existence of a language for the PTIME queries in this general setting, it is tempting to wonder if additional criteria of naturalness may render the problem easier. Motivated by this, we consider here *finitely generated languages* (FGLs). These capture a wide array of languages in which queries are defined from finitely many "building blocks" using a fixed finite set of constructors. The classical example of an FGL is FO (the constructors implement $\exists, \forall, \vee, \wedge$, and $\neg$). However, our notion of FGL is much more powerful, since it allows for the individual building blocks and constructors to perform arbitrary PTIME computations. In fact, building blocks are formalized

as polynomial-time properties, and constructors as polynomial-time Turing machines with oracle calls to other constructors or building blocks. The restricted structure of FGLs immediately removes some of the issues discussed above: all FGLs are effectively P-bounded. One might naturally wonder if the additional structure of FGLs allows to prove that there is no such language expressing exactly the PTIME properties of arbitrary graphs. This question remains open. However, we exhibit a broad class of FGLs, called *Set* FGLs (SFGLs), for which this can be proven. Informally, SFGLs are FGLs restricted in the way the constructors and building blocks in a program exchange information. Calls to oracles are made on hereditarily finite sets. The information exchanged does not break automorphisms of the input and is subject to restrictions on size and depth of nesting. Hereditarily finite sets can easily represent complex values used in many concrete database query languages [1]. SFGLs capture a natural programming paradigm, shared by many languages. One way to view SFGL's is as a generalization of FO with finitely many polynomially-computable Lindström quantifiers (see [5]). It is known that fixpoint logics with finitely many polynomially-computable Lindström quantifiers cannot express P_G (see [4, 5]).

The paper is organized as follows. In Section 2 we formalize the notions of language, evaluator, and (effectively) P-bounded language and property. Section 3 presents our results comparing these notions. In Section 4 we discuss the two alternatives to obtaining a language for the PTIME properties: considering richer languages, and focusing on restricted sets of graphs. Finally, Section 5 presents the results on SFGLs.

2 Preliminaries

In this section we review some of the basic concepts related to query languages and their complexity, and introduce notation used throughout the paper.

We assume familiarity with Turing machines. We also assume a fixed effective enumeration of all Turing machines and denote by M_e the e-th Turing machine. We also assume knowledge of usual query languages such as first-order logic (FO), and FO extended with a least fixpoint operator, denoted FO+LFP (e.g., see [1, 5]). For a positive integer k, FO^k denotes the FO sentences using at most k variables, and similarly for $(\mathrm{FO+LFP})^k$.

Properties and their complexity. For simplicity, we focus here on PTIME *properties* rather than output-producing queries. A relational signature is a finite set of relation symbols together with associated arities. A finite structure over a given signature consists of a finite domain D and interpretations of the relation symbols in the signature as finite relations of appropriate arities over D. A property of structures over some signature is a set of finite structures over that signature, closed under isomorphism. We denote properties by $Q, R, S...$ and sets of properties by $\mathcal{Q}, \mathcal{R}, \mathcal{S}...$. Since structures over an arbitrary signature can be efficiently encoded as graphs (e.g., see [9, 5]), we will only consider in the sequel the relational signature consisting of a single binary relation representing

the edges of a directed graph whose nodes are the elements of the domain. We denote this signature by γ, and the set of all finite graphs (finite structures over γ) by $\mathcal{G}$.

The *complexity* of a property is defined using classical complexity classes. To do this, we need to talk about the resources used by a Turing Machine "implementing" an algorithm for checking that a structure has the desired property. Since Turing Machines do not take structures as inputs, we need to use instead encodings of structures as strings. We use the following simple encoding for structures over signature γ. Suppose the structure represents a graph G whose set of nodes is D of size n. Let λ be a one-to-one mapping from D onto $\{1, \ldots, n\}$, and let $\chi_G : \{1, \ldots, n\}^2 \to \{0, 1\}$ be the characteristic function of the set of edges in G via λ (so $\chi_G(\lambda(u), \lambda(v)) = 1$ iff (u,v) is an edge). The encoding of G is a string over alphabet $\{0, 1\}$ consisting of all $\chi_G(i, j)$ listed in lexicographic order of the pairs (i, j). This encoding clearly depends on the labeling λ and is denoted by $enc_\lambda(G)$. The length of $enc_\lambda(G)$ is denoted by $|enc_\lambda(G)|$, and note that $|enc_\lambda(G)| = n^2$, where n is the number of nodes in the graph. As a shorthand, we also denote $|enc_\lambda(G)|$ by $|G|$.

Let $\mathcal{Q}$ be a property of graphs. We say that a Turing machine M decides $\mathcal{Q}$ iff for every graph G and labeling λ of its nodes, M halts on input $enc_\lambda(G)$ and accepts iff G has property $\mathcal{Q}$. Note that there is no requirement on inputs that are not correct encodings of graphs. Also observe that, since $\mathcal{Q}$ is closed under isomorphism, acceptance by M must be independent of the particular labeling λ. That is, for all labellings λ_1, λ_2, M accepts $enc_{\lambda_1}(G)$ iff M accepts $enc_{\lambda_2}(G)$.

We can now relate properties and complexity. We say that a property $\mathcal{Q}$ of graphs is a PTIME property iff there exists a Turing machine M deciding the property, and $k \in \mathbb{N}$, such that M halts on input $enc_\lambda(G)$ in at most $|enc_\lambda(G)|^k$ steps. We denote the set of PTIME properties of graphs by P_G.

Languages and evaluators. To reason about the existence of a language for the PTIME properties, we need a very broad definition of query language (or logic). It is generally accepted that a query language specifies queries using expressions consisting of strings of symbols over some alphabet, which we call its *programs*. Moreover, the language should have effective syntax, meaning that its syntactically correct programs can be effectively enumerated. As a useful side effect, this allows us to ignore the specific syntax of a language, and simply refer to its programs by their index in the enumeration (1st program, 2nd program, etc). Since we will only be interested in data complexity and not query complexity, the cost of the translation between an index and the corresponding program is irrelevant. Thus, we can simply assume that the programs of the language are the indexes themselves, consisting of all strings in $\{0, 1\}^*$. Whenever needed, we interpret such strings as positive natural numbers as follows: the string w corresponds to the natural number whose binary representation is $1w$ (this eliminates the problem of leading zeros and renders the mapping bijective). We denote the set of all such strings in $\{0, 1\}^*$ by $\mathcal{E}$.

Given that the syntax of languages consists of the expressions in $\mathcal{E}$ and can be assumed fixed, we can define a language by the semantics associated to the

expressions in $\mathcal{E}$. Thus, a language L for graph properties is a mapping associating to each expression $e \in \mathcal{E}$ a property $L(e)$ of graphs. We write $[L]$ for the set of properties defined by L. Of course, two different languages may express the same set of properties.

Observe that the semantics of a language is an abstract mapping, independent of any notion of computability or complexity. To capture the latter, we consider the notion of *evaluator* of a language. Intuitively, an evaluator corresponds to a query processor: it takes as input a program in the language together with a graph, and evaluates the program on the graph. More formally, an evaluator for a language L is a Turing machine E that takes as input a program e and the encoding of a graph G and evaluates e on G. To make this more precise, let us first fix a PTIME-computable pairing function $\langle -, - \rangle$ for $\mathbb{N}$, that is, a bijection $\langle -, - \rangle : \mathbb{N}^2 \to \mathbb{N}$ such that both $\langle -, - \rangle$ and π_1, π_2 satisfying $\pi_1(\langle x, y \rangle) = x$ and $\pi_2(\langle x, y \rangle) = y$ are PTIME computable (e.g., such a pairing function is provided in [10]). The tape alphabet of E is $\{0, 1\}$ and e and G are encoded as the binary representation of the integer $\langle e, enc_\lambda(G) \rangle$ for some labeling λ of the nodes of G. On any input of the form $\langle e, enc_\lambda(G) \rangle$, E halts and outputs 1 if G has property $L(e)$ and 0 otherwise. Note that a given language can have many different evaluators.

Languages and complexity. What does it mean to have a language for the PTIME properties? We consider several notions that relate languages to properties of a given complexity, most of which have been proposed before. One of the contributions of the paper is to clarify the relationship between the different notions in a systematic way.

Consider a language L, defining a set of properties $[L]$. A first attempt at relating L to the polynomial-time properties is to look at the connection between $[L]$ and P_G. We say that L expresses P_G iff $[L] = \mathrm{P}_G$. However, it is clear that this alone is not satisfactory. At a minimum, we would like to be able to effectively evaluate the queries in L. In other words, we would like to have, at the very least, an evaluator for L. If such is the case, we call the language L *computable*. We would also like to actually evaluate the queries of L in polynomial time. This is formalized as follows. We say that L has a P-bounded evaluator if it has some evaluator E that, for every *fixed* program e, runs in polynomial time on input $\langle e, enc_\lambda(G) \rangle$. The fact that we fix e means that our definition captures data rather than query complexity. Of course, a language that has a P-bounded evaluator only expresses polynomial-time properties.

Next, suppose we are given a P-bounded evaluator E for a language. The evaluator runs in polynomial time, but we do not necessarily know ahead of time the bounding polynomial. However, for many specific languages, such as FO+LFP, we are able to infer an explicit polynomial bound from the syntax. This is a nice property to have. We call an evaluator E effectively P-bounded if there exists a computable total mapping $B : \mathcal{E} \to \mathbb{N}$ that produces, for every program e, a number k such that E runs in time $|G|^k$ on input $\langle e, enc_\lambda(G) \rangle$.

We say that a language is (effectively) *P-bounded* if it has an (effectively) P-bounded evaluator. Similarly, a set of properties $\mathcal{P}$ is (effectively) P-bounded if there exists some (effectively) P-bounded language defining $\mathcal{P}$.

In considering the existence of a language L for the polynomial-time properties, two alternative requirements for such a language have been proposed. (1) requires L to express precisely P_G, and have a P-bounded evaluator [3]. (2) additionally requires L to have an *effectively* P-bounded evaluator [9, 5]. That is, (1) requires P_G to be P-bounded and (2) requires P_G to be effectively P-bounded.

3 Computable, P-bounded, and Effectively P-bounded Languages

What is the connection between the notions of computable, P-bounded, and effectively P-bounded language? We consider this question next. As we shall see, these notions are generally distinct. This says that there are different flavors of the question of the existence of a language for PTIME and that the answers may be distinct for different flavors.

Obviously, every effectively P-bounded language is P-bounded and every P-bounded language is computable. Consider now the converse inclusions. Of course, a computable language L may express properties that are not in P_G, in which case it cannot be P-bounded. However, suppose L expresses only properties in P_G. Is it the case that L must also be P-bounded? We next show this is not the case. In fact, we exhibit a computable language expressing *precisely* the properties in P_G, that has no P-bounded evaluator.

Before we state the result, note that it is easy to find a computable language for P_G. We recall such a language, defined in slightly different form by Andreas Blass and Yuri Gurevich [9], that we denote L_Y. The syntax of L_Y consists of all FO+LFP sentences φ over signature $\gamma \cup \{\leq\}$. Recall that an FO+LFP sentence φ over this signature is order-invariant on a graph H iff its value on H and an ordering $\leq$ of the nodes of H is independent of the choice of $\leq$. Furthermore, φ is order invariant iff it is order invariant on all graphs. The semantics of L_Y is defined next. Although we are considering sentences φ using $\leq$ in addition to γ, we define $L_Y(\varphi)$ as a property of graphs alone, as follows. Let φ be a sentence and G a graph. If φ (viewed as a usual FO+LFP sentence) is order-invariant for all graphs H of size at most that of G, then G has property $L_Y(\varphi)$ iff φ evaluated as an FO+LFP sentence on G with some arbitrarily chosen ordering $\leq$ is true. Otherwise, G does not have property $L_Y(\varphi)$. Note that, if φ is order invariant on all graphs, then $L_Y(\varphi)$ defines the same property as φ, so is a property in P_G. If φ is not order invariant, then $L_Y(\varphi)$ contains only finitely many graphs, so it is again in P_G. Finally, since order-invariant FO+LFP sentences express all P_G properties [11, 12], it follows that L_Y expresses precisely the P_G properties. Clearly, L_Y has an evaluator, so it is a computable language for P_G. That is,

Remark 1. P_G is computable.

Remark 2. Note that the language L_Y is coNP-bounded. One might naturally wonder if it can be proven that L_Y has no P-bounded evaluator. Clearly, such a result must be conditional upon assumptions such as P $\neq$ NP. However, we are not aware of any proof that L_Y has no P-bounded evaluator even under such complexity-theoretic assumptions. Thus, L_Y remains, for the time being, a candidate language for P_G.

As an intriguing aside, we mention a connection to another problem that appears to be similarly open:

(†) Input: A non-deterministic Turing machine M and a string 1^n.
Question: Does M accept ϵ (the empty string) in at most n steps?

It can be shown that L_Y is P-bounded iff there exists some algorithm solving (†) in $\mathrm{TIME}(n^{f(M)})$ for some arbitrary function f. In other words, the problem can be solved by a (uniform) algorithm that is polynomial in n for fixed M (note that the non-uniform version of the problem is trivial: for each fixed M there exists an algorithm that is polynomial in n and solves (†)). Interestingly, the (non)-existence of such an algorithm for (†) appears to be open, and does not immediately follow from usual complexity-theoretic assumptions.

Theorem 1. *Every computable set of properties $\mathcal{P}$ that includes all finite properties has a computable language L which is not P-bounded.*

Proof. Since $\mathcal{P}$ is computable, it has a computable language L_C; we use L_C to build L. The semantics of L is defined as follows. We view the expressions in $\mathcal{E}$ as natural numbers. Let $L(2n+1) = L_C(n)$. Next, let $L(2n)$ be defined as follows. Let G be a graph and $\bar{G}$ the complete graph with the same nodes as G. Run the n-th Turing machine M_n on input $\langle 2n, enc_\lambda(\bar{G})\rangle$ for some arbitrary λ (note that the encoding of $\bar{G}$ is independent of λ). If M_n does not stop in $2^{|\bar{G}|}$ steps, then $G \notin L(2n)$. If $|\bar{G}|$ is the smallest size for which M_n stops in $2^{|\bar{G}|}$ steps, then $G \in L(2n)$ iff M_n rejects $\bar{G}$. If $|\bar{G}|$ is not the smallest such size, then $G \notin L(2n)$. Note that $L(2n)$ contains only finitely many graphs, so is in P_G.

Next, suppose L has a P-bounded evaluator E, and suppose E is M_e. Since E is P-bounded, M_e runs in polynomial time with respect to $|\bar{G}|$ on every input of the form $\langle f, enc_\lambda(G)\rangle$ for fixed f. In particular, M_e runs in polynomial time with respect to $|\bar{G}|$ on input $\langle 2e, enc_\lambda(\bar{G})\rangle$. It follows that there exists some $\bar{G}$ such that M_e stops in at most $2^{|\bar{G}|}$ steps. By definition of $L(2e)$, the smallest such $\bar{G}$ has property $L(2e)$ iff M_e rejects. This contradicts the assumption that E is an evaluator for L.

Since P_G is computable,

Corollary 1. *P_G has a computable language that is not P-bounded.*

We next consider the connection between the notions of P-bounded language and effectively P-bounded language.

Theorem 2. *Every P-bounded set of properties $\mathcal{P}$ that includes all finite properties has a P-bounded language L which is not effectively P-bounded.*

Proof. Let K be some P-bounded language defining $\mathcal{P}$. We define a language L as follows. First, $L(2n+1) = K(n)$. This ensures that L expresses all properties expressed by K. Next, we define $L(2n)$ as follows. Suppose $n = \langle e, b\rangle$. Intuitively, we define $L(2n)$ so that M_e cannot be an evaluator for L with bounding function M_b. To this end, let G be a graph. To determine if $G \in L(2n)$, proceed as follows. First, run M_b on input $2n$ for $|G|$ steps. If M_b does not halt in $\leq |G|$ steps, then $G \notin L(2n)$. Otherwise, suppose that $|G| = t^2$. Then if $M_b(2n)$ halts in $\leq (t-1)^2$ steps, then $G \notin L(2n)$. Finally, if $M_b(2n)$ halts in s steps where $(t-1)^2 < s \leq t^2$, then let k be the output of $M_b(2n)$. Next, run M_e on input $\langle 2n, enc_\lambda(\bar{G})\rangle$ for $|G|^k$ steps. If M_e halts, then $G \in L(2n)$ iff M_e rejects. Otherwise, $G \notin L(2n)$. Note that $L(2n)$ is a finite property, so it is already expressed by K. Clearly, L expresses precisely $\mathcal{P}$ and is P-bounded.

Now suppose L is effectively P-bounded. Then L has an evaluator E with bounding function B. Let $E = M_e$ and $B = M_b$. Let $n = \langle e, b\rangle$ and consider $L(2n)$. Since M_b halts on input $2n$, there exists a graph G such that M_b halts on $2n$ in at most $|G|$ steps. Consider the smallest such G. Let $k = M_b(2n)$. Since M_b computes the bounding function for M_e, it follows that M_e stops on input $\langle 2n, enc_\lambda(\bar{G})\rangle$ in at most $|G|^k$ steps. However, by the definition of $L(2n)$, $G \in L(2n)$ iff M_e rejects on input $\langle 2n, enc_\lambda(\bar{G})\rangle$. This contradicts the assumption that M_e is an evaluator for L.

Example 1. Consider the fixpoint queries defined by the FO+LFP sentences. The language FO+LFP is effectively P-bounded, and the properties it defines includes all finite properties. By Theorem 2, there exists some other language defining the fixpoint queries, that is P-bounded but not effectively P-bounded.

Corollary 2. *If P_G is P-bounded,[4] then it has a P-bounded language that is not effectively P-bounded.*

Remark 3. Theorem 2 states *the existence* of P-bounded languages that are not effectively P-bounded, for all P-bounded sets of properties that include the finite ones. A natural question is whether there are P-bounded sets of properties that do not have *any* effectively P-bounded language. The answer is affirmative: Theorem 5 in the next section shows the existence of many such classes of properties.

Clearly, it would be of interest to know if the existence of a P-bounded language expressing P_G implies the existence of an effectively P-bounded one. This remains open.

4 PTIME from Above and from Below

In the absence of a language expressing precisely the polynomial-time queries, various relaxations to the problem of capturing P_G can be useful. We describe here two natural approaches. The first consists in trying to capture the PTIME

[4] Recall that it is not known whether P_G is P-bounded.

queries using a richer language allowing to express queries possibly not in PTIME, but that has an evaluator that evaluates every PTIME query in PTIME. The second approach, studied by many researchers, is to focus on PTIME properties on restricted sets of graphs. Our results are mostly negative and point out limitations to both approaches.

4.1 P-Faithful Evaluators

Suppose we have a language L that expresses all of P_G and possibly more. For example, such a language is Existential Second-Order logic ($\exists$SO), that is known to express the NP properties [6]. Assuming that P $\neq$ NP, some of the formulas in $\exists$SO express polynomial-time properties, while others do not. Furthermore, under the same assumption, it is easily shown using Trakhtenbrot's theorem that it is undecidable whether a given formula expresses a PTIME property. However, it is conceivable that $\exists$SO has an evaluator E that happens to evaluate every polynomial-time property in polynomial time. This means that a user can not only express all polynomial-time properties using $\exists$SO, but such properties can actually be evaluated in polynomial time. Short of an actual language for P_G, this would seem like a tempting alternative.

Unfortunately, this solution is not a real alternative to a language for P_G. Indeed, we show that, if $\exists$SO (or any language that can express all of P_G) has an evaluator that computes all P_G properties in polynomial time, then there exists a P-bounded language expressing *exactly* P_G. Thus, the alternative formulation is no easier than the original problem of finding a language for P_G.

We first formalize the above notions.

Definition 1. *Let L be a language expressing all properties in P_G. An evaluator E for L is* P-faithful *iff $E(\langle e, enc_\lambda(G)\rangle)$ runs in polynomial time with respect to G for every fixed e such that $L(e) \in P_G$. Furthermore, E is* effectively P-faithful *iff there exists a computable mapping $B : \mathcal{E} \to \mathbb{N}$ that produces, for every e for which $L(e) \in P_G$, a number k such that $E(\langle e, enc_\lambda(G)\rangle)$ runs in time $|G|^k$.*

We can now show the following.

Theorem 3. *If there is an (effectively) P-faithful language L for P_G, then there is an (effectively) P-bounded language K for P_G.*

Proof. The syntax of K consists of pairs (e, φ) where $e \in \mathcal{E}$ is interpreted with the semantics of L and φ is in L_Y (recall L_Y, the computable language expressing P_G, from Section 3). Suppose L has a P-faithful evaluator E_L, and let E_Y be an evaluator for L_Y. Let us define an evaluator E_K for K as follows. E_K on input (e, φ) and G does the following. First, start computing $E_L(e, H)$ and $E_Y(\varphi, H)$ on all graphs H smaller than G for $|G|$ steps. If in this number of steps $E_L(e, H)$ and $E_Y(\varphi, H)$ both halt for some H and one accepts while the other rejects, then reject G (so G does not have property $K(e, \varphi)$). Otherwise, run E_L on input (e, G) and accept iff E_L accepts. Note that, if $L(e)$ and $L_Y(\varphi)$ define different properties, then $K(e, \varphi)$ is finite (and is evaluated in polynomial time by the

evaluator E_K). Otherwise, K is evaluated on input (e, φ) and G in polynomial time with respect to G, using the evaluator E_L applied to e and G, which takes polynomial time with respect to G because $L(e)$ is in P_G and E_L is P-faithful. An analogous argument shows that if E_L is effectively P-faithful then K is an effectively P-bounded language for P_G.

4.2 PTIME properties with no (effectively) P-bounded language

A productive alternative approach to the problem of finding a language for the PTIME queries has been to focus on interesting subsets of graphs rather than all graphs. We briefly mention two results that provide some insight into this approach. The results, proven by diagonalization, show that every "well-behaved" class of graphs can be extended to a class of graphs whose PTIME properties do not have an (effectively) P-bounded language (we omit the details).

Theorem 4. *For every PTIME-recognizable set of graphs $\mathcal{G}_0$ with infinite complement there exists a computable set of graph properties $\mathcal{H} \subset P_G$ that is not P-bounded and includes all PTIME properties of $\mathcal{G}_0$.*

Theorem 5. *For every PTIME-recognizable set of graphs $\mathcal{G}_0$ with infinite complement such that its set of PTIME properties is P-bounded, there exists a P-bounded set of graph properties $\mathcal{H} \subset P_G$ that is not effectively P-bounded and includes all PTIME properties of $\mathcal{G}_0$.*

5 Finitely generated languages

In this section we turn to *finitely generated languages* (FGLs). These capture a wide array of languages in which queries are defined from finitely many "building blocks" using a finite set of constructors. The classical example of an FGL is FO. However, our notion of FGLs is much more powerful.

Since we will be focusing on languages expressing PTIME queries, we require each of the building blocks and each constructor to be computable in polynomial time. We formalize this as follows. The syntax of an FGL L is given by all terms that can be built by using a finite set C of constant symbols and a finite set F of functions symbols with associated finite arities. The semantics of L is as follows:

- to each $c \in C$ we associate a property K_c (a "building block") defined by a polynomial-time Turing machine M_c and
- to each $f \in F$ of arity k, we associate a polynomial-time Turing machine M_f (a "constructor") with access to k oracles.

The evaluator E for L is defined recursively as follows:

- If $t \in C$, then E on input $\langle t, enc_\lambda(G)\rangle$ runs M_c on input $enc_\lambda(G)$.
- If $t = f(t_1, \ldots, t_k)$ then E on input $\langle t, enc_\lambda(G)\rangle$ runs M_f on input $enc_\lambda(G)$ with oracles $E(t_1, -), \ldots, E(t_k, -)$.

Clearly, FGLs can be viewed as languages according to our general definition, since there exists an effectively computable bijection between the terms providing the syntax of FGLs and the set of strings $\mathcal{E}$ used for arbitrary languages. The following is immediate from the definition of FGLs.

Remark 4. Every FGL is effectively P-bounded.

One might naturally wonder if the additional structure of FGLs allows to prove that there is no such language expressing exactly the PTIME properties of graphs. This question remains open, even for ordered structures. To gain some intuition into the difficulties involved in settling this question, let us consider FGLs on ordered structures. Let FO+LFP^r consist of FO+LFP sentences using second-order variables (inductively defined relations) of arity at most r. We can show the following using an extension of the standard simulation of PTIME Turing machines on ordered structures by FO+LFP:

Lemma 1. *On ordered structures:*

(i) Each FGL is included [5] *in FO+LFP^r for some r.*
(ii) For every r, FO+LFP^r is included in some FGL.

It is known that on ordered structures, (a) if FO+LFP^r = PTIME for some r then PTIME $\neq$ PSPACE, and (b) if FO+LFP$^r \neq$ PTIME for some $r > 1$ then LOGSPACE $\neq$ PTIME [5, 7]. This together with Lemma 1 implies the following:

Theorem 6. *(i) If there exists an FGL expressing PTIME on ordered structures then PTIME $\neq$ PSPACE.*
(ii) If no FGL expresses PTIME on ordered structures then LOGSPACE $\neq$ PTIME.

Theorem 6 shows that settling the question of whether an FGL can express PTIME on ordered structures would resolve long-standing open problems in complexity theory. The question remains open for arbitrary structures. This leads us to consider a restriction of FGLs for which this question can be settled. We introduce *set FGLs (SFGLs)*, which are FGLs that operate on hereditarily finite sets under some restrictions. Before we do this, we introduce some terminology related to sets.

Sets. Given x and y, the pairing of x and y is $\{x, y\}$ and the union of x and y is $x \cup y$. For any finite set A, the *set of hereditarily finite sets over A*, HF(A), is the smallest set containing all elements in A (*atoms*), the empty set, and closed under the operations of pairing and binary union. Consider $x \in$ HF(A). The *transitive closure* of x, tc(x), is the smallest set y satisfying $x \subseteq y$ and $\forall u, v(u \in v \in y \rightarrow u \in y)$. We write $\|x\|$ for $|\mathrm{tc}(x)|$. We set atoms(x) := tc(x) $\cap$ A and say that x is *atomless* if atoms(x) = $\emptyset$. We can think of x as a directed acyclic graph with $|\mathrm{tc}(x)| + 1$ nodes. Given an order of the atoms A, we can x as a string of

[5] Inclusion refers to the sets of properties expressed by each language.

length $\|x\|^2$. The *rank* of an atom is 0, the rank of the empty set is 0, and the rank of any other set x, $\mathrm{rank}(x)$, is $\max(1 + \mathrm{rank}(y) : y \in x)$. We encode the ordered pair $\langle x, y\rangle$ in the standard way as $\{\{x\}, \{x, y\}\}$ and we encode tuples inductively by $\langle x, y, z\rangle = \langle\langle x, y\rangle, z\rangle$. As an aside, note that hereditarily finite sets can represent the complex values common in databases, obtained by nested application of set and tuple constructors (e.g., see [1]).

Every permutation σ of A induces an automorphism of $\mathrm{HF}(A)$ (which we also call σ) in the obvious way. We say that $S \subseteq A$ *A-supports* x if every permutation σ of A which fixes S pointwise fixes x. We set $\mathrm{supp}_A(x) := S$ where $S \subseteq A$ is the smallest set S which A-supports x if there is such S satisfying $|S| < |A|/2$. Otherwise, we set $\mathrm{supp}_A(x) := A$. It is not obvious, but $\mathrm{supp}_A(x)$ is well-defined. We set $\mathrm{supp}(x) := \mathrm{supp}_{\mathrm{atoms}(x)}(x)$. Notice that $\mathrm{supp}(x) = \mathrm{supp}_A(x) \cap \mathrm{atoms}(x)$.

We say that $x, y \in \mathrm{HF}(A)$ are isomorphic, $x \cong y$, if there is a bijection $\sigma : \mathrm{atoms}(x) \to \mathrm{atoms}(y)$ such that $\sigma(x) = y$.

A set FGLs (SFGL) is an FGL for which all inputs are (encodings of) sets $x \in \mathrm{HF}(\mathrm{atoms}(x))$ and for which there is a number m and a function g such that for each term $t = f(t_1, \ldots, t_k)$, every input q to an oracle call made by the constructor M_f in the evaluation of t on input x satisfies:

1. $\mathrm{atoms}(q) \subseteq \mathrm{atoms}(x)$
2. $\|q\| \in O(\mathrm{atoms}(x)^m)$, and
3. $\mathrm{rank}(q) \leq g(t, \mathrm{rank}(x))$.

In addition, for each oracle t_i, the set $Q_t^i(x)$ consisting of all inputs q to calls to t_i made by M_f in the evaluation of t on input x is independent of the encoding of x (so is well defined). This requirement implies that $Q_t^i(x)$ is fixed by all automorphisms of x, a fact that is critical to the proof of Lemma 2 below. Finally, we require closure under isomorphism. That is, if $x \cong y$, then for all terms t, $x \models t$ iff $y \models t$.[6]

SFGLs are powerful enough to simulate FO with finitely many Lindström quantifiers $\mathbf{Q}$ [4, 5]. We briefly outline the simulation on a structure $\mathcal{A}$. We need

- one constant c_R for every relation symbol of $\mathcal{A}$,
- function symbols $f_\neg$ of arity 2 and $f_\vee$, $f_\wedge$ of arity 3,
- one function symbol f_Q of arity 2 for every Lindström quantifier Q,
- a constant $c_\emptyset$ corresponding to $\emptyset$, and
- function symbols f_p and f_u of arity 2 corresponding to pairing and union.

The term t_ϕ providing the simulation mimics the structure of $\phi \in \mathrm{FO}(\mathbf{Q})$: each logical operator corresponds to a constructor, which makes calls to its oracles on inputs $\mathcal{A}\bar{a}$ consisting of $\mathcal{A}$ extended with a tuple $\bar{a}$ providing a valuation for a subset $\bar{z}$ of the variables. There is one subtlety: the constructors calling oracles corresponding to sub-formulas must decide what components of $\bar{a}$ to pass to each sub-formula, which is determined by its free variables. This information is specified by an additional oracle defined by a term using $c_\emptyset$, f_p, and f_u and

[6] We write $x \models t$ if t accepts x.

accepting precisely one atomless set that encodes the needed information. We write $t_{\bar{v}}$ for the term that accepts precisely the set representing $\bar{v}$. We define:

- If ϕ is an atomic formula $R\bar{x}$, then $t_\phi := c_R$.
- If $\phi(\bar{z})$ is $\alpha(\bar{x}) \wedge \beta(\bar{y})$, then $t_\phi := f_\wedge(t_\alpha, t_\beta, t_{\langle \bar{x},\bar{y}\rangle})$ (similarly for $\vee$ and $\neg$).
- If $\phi(\bar{z})$ is $Q\bar{x}\alpha(\bar{x}\bar{z})$, then $t_\phi := t_{f_Q}(t_\alpha, t_{\langle \bar{x}\rangle})$.

To illustrate, consider the simulation of a conjunction $\alpha(\bar{x}) \wedge \beta(\bar{y})$. On input $\mathcal{A}\bar{a}$, $M_{f_\wedge}$ first queries its last oracle on atomless sets in their canonical order until some set s is accepted. If s does not encode appropriate tuples of variables, the constructor rejects. Otherwise, $M_{f_\wedge}$ uses $\bar{x}$ and $\bar{z}$ to obtain from $\bar{a}$ the tuples on which to issue queries to t_α and t_β: Notice that in the simulation of FO(**Q**), requirements (2) and (3) in the definition of SFGL are satisfied: constructors call oracles on inputs of the form $\mathcal{A}\bar{a}$ where $\bar{a}$ is a tuple of variables whose rank increases by at most a constant at each call. Thus, (2) and (3) can be viewed as generalizing this mode of computation.

Theorem 7. *There is no SFGL that expresses all PTIME properties of graphs.*

Proof. (outline) By Proposition 1 below, there is some b so that we can decide $x \models t$ in time $O(\|x\|^b)$ for x satisfying $x = \text{atoms}(x)$ (i.e. a "naked" set). In this case we can set $r = 1$ and $s = 0$ and we have $\|x\| = |\text{atoms}(x)|$. The result follows by a straightforward adaptation of the Time Hierarchy theorem [10].

Proposition 1. *If every building block of an SFGL runs in time $O(n^{t_c})$ and every constructor runs in time in time $O(n^{t_f})$ and has arity at most k, then for every term t, for every fixed r, s, m and for every x satisfying*

$$|\text{supp}(x)| \leq s, \quad \|x\| \in O(|\text{atoms}(x)|^m), \text{ and } \text{rank}(x) \leq r,$$

we can decide $x \models t$ in time $O(|\text{atoms}(x)|^b)$, where $b := m \cdot \max(t_c, t_f, 4)$.

Proof. (outline) Assume we have t_c, t_f, k, r, s and m satisfying the hypotheses. We show by induction on term depth d that the statement holds for each x satisfying the hypotheses. This is clear for $d = 0$; for the inductive step we use the following simulation to evaluate term $t = f(t_1, \ldots, t_j)$. We compute as M_f does on input x, except for each query q to oracle i we first look for q' isomorphic to q within an internal table T_i (initially empty). If such q' is found, we do not issue the query and instead use the answer we obtained for q'. Otherwise, we issue the query and add q and the result of the query to the table T_i. We can divide the running time of this simulation into three parts: time spent in (1) the body of M_f, (2) table lookup, (3) queries. We set $n_x = \|x\|$, $a_x = |\text{atoms}(x)|$, $r_x = \text{rank}(x)$, $s_x = |\text{supp}(x)|$, $s_q := s + b$, and $r_q := g(t, r)$.

1. The time spent in the body of M_f is $O(n_x^{t_f}) \subseteq O(a_x^{mt_f}) \subseteq O(a_x^b)$.
2. To do the table lookup for a query q, we first compute its support, which we can do in time $O(a_x^2 n_x^2)$. To check for isomorphism against q' in the table, we try all possible bijections $\sigma : \text{supp}(q) \to \text{supp}(q')$. By Lemma 2 we know that

$|\mathrm{supp}(q)| \leq s_q$ for large enough x, so this adds a factor of $s_q!$. Finally, also by Lemma 2 we know that the number of isomorphism classes of q depends only on r_q and s_q. Since $\|x\| \in O(|\mathrm{atoms}(x)|^m)$, we can do the table lookup in time $O(a_x^{2+2m}) \subseteq O(a_x^b)$.

3. We know from above that the total number of queries we need to make depends on k, r_q, and s_q, but not on n_x. We can show that $|\mathrm{atoms}(q)| \leq s_q$ or $|\mathrm{atoms}(q)| \geq a_x - s_q$. If the former holds, $\|q\|$ is bounded by a constant depending only on r_q and s_q If the latter holds we have $O(a_x^m) = O(|\mathrm{atoms}(q)^m)$. Either way, $\|q\| \in O(|\mathrm{atoms}(q)|^m)$ so we can apply the induction hypothesis using r_q and s_q in place of r and s. Therefore, we can answer all queries in time $O(|\mathrm{atoms}(q)|^b) \subseteq O(a_x^b)$.

Lemma 2. *If every constructor runs in time $O(n_x^{b/m})$, then for fixed s and large enough x satisfying 1, 2, and 3 of Proposition 1 every query $q \in Q_x^t$ must satisfy $|\mathrm{supp}_A(q)| \leq s+b$ where $A = \mathrm{atoms}(x)$ and therefore also $|\mathrm{supp}(q)| \leq s+b$. The number of isomorphism classes in Q_x^t depends only on $g(t,r)$ and $s+b$.*

Lemma 2 is an extension of Theorem 24 in [2]. The (difficult!) proof is omitted.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison Wesley, 1995.
2. A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.
3. A. Chandra and D. Harel. Structure and complexity of relational queries. *J. Comput. Syst. Sci.*, 25(1):99–128, 1982.
4. A. Dawar and L. Hella. The expressive power of finitely many generalized quantifiers. *Inf. Comput.*, 123(2):172–184, 1995.
5. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 2nd edition, 1999.
6. R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS Proceedings, 1974.
7. M. Grohe. *The structure of fixed-point logics.* PhD thesis, Albert-Ludwigs Universität Freiburg, 1994.
8. M. Grohe. Fixed-point logics on planar graphs. In *Proc. Symp. on Logic in Computer Science*, 1998.
9. Y. Gurevich. Logic and the challenge of computer science. In E. Borger, editor, *Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
10. J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Reading, MA: Addison Wesley, 1979.
11. N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.
12. M. Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.