Program schemes, queues, the recursive spectrum and zero-one laws[*]

Iain A. Stewart[†]
Department of Mathematics and Computer Science,
University of Leicester, Leicester LE1 7RH, U.K.

Abstract

We prove that a very basic class of program schemes augmented with access to a queue and an additional numeric universe within which counting is permitted, so that the resulting class is denoted $\text{NPSQ}_{+}(1)$, is such that the class of problems accepted by these program schemes is exactly the class of recursively enumerable problems. The class of problems accepted by the program schemes of the class NPSQ(1) where only access to a queue, and not the additional numeric universe, is allowed is exactly the class of recursively enumerable problems that are closed under extensions. We define an infinite hierarchy of classes of program schemes for which NPSQ(1) is the first class and the union of the classes of which is the class NPSQ. We show that the class of problems accepted by the program schemes of NPSQ is the union of the classes of problems defined by the sentences of all vectorized Lindström logics formed using operators whose corresponding problems are recursively enumerable and closed under extensions; and, as a result, has a zero-one law. Moreover, we also show that this class of problems can be realized as the class of problems defined by the sentences of a particular vectorized Lindström logic. Finally, we show how our results can be applied to yield logical characterizations of complexity classes and provide logical analogues to a number of inequalities and hypotheses from computational complexity theory involving (non-deterministic) complexity classes ranging from **NP** through to **ELEMENTARY**.

[*]An extended abstract of this paper appeared as a paper of the same title in the *Proceedings of the 7th Annual International Computing and Combinatorics Conference* (*COCOON* 2001), Lecture Notes in Computer Science, Springer-Verlag (2001) 39–48.

[†]Supported by EPSRC Grant GR/M 12933.

1 Introduction

Descriptive complexity theory is the study of the relationship between the computational complexity of a problem and the logical definability of the problem (as such, descriptive complexity theory forms a substantial research area within finite model theory). The subject essentially originated with Fagin's Theorem [15] which states that a problem (more precisely, an encoding of a problem) can be accepted by a non-deterministic Turing machine running in polynomial-time if, and only if, it can be defined by a sentence of existential second-order logic; and since then the relative relationships of a whole range of complexity classes and logics have been examined (see, for example, [14, 26], which we also use as our references for finite model-theoretic concepts and notions not given in this paper; whereas we refer the reader to [17] for definitions of the standard complexity classes considered in this paper). Having logical equivalents of hypotheses in complexity theory (for example, **NP** = **co-NP** if, and only if, existential second-order logic and universal second-order logic define the same class of problems) enables different lines of attack on these hypotheses using (inexpressibility) tools from finite model theory, such as Ehrenfeucht-Fraïssé games and zero-one laws, which are not available in the traditional Turing machine-based setting.

The connection between a complexity class and a logic need not be as tight as it is in Fagin's Theorem. For example, it is known that a problem is in **P** if, and only if, it can be defined by a sentence of inductive fixed-point logic, and that a problem is in **PSPACE** if, and only if, it can be defined by a sentence of partial fixed-point logic, but in both cases under the assumption that every finite structure comes with a linear order of its elements (in the form of a binary relation describing a successor relation), *i.e.*, when we deal with ordered finite structures [24, 38]. It turns out that on the class of all finite structures there are problems in **P** (resp. **PSPACE**) which are not definable by any sentence of inductive (resp. partial) fixed-point logic. Furthermore, these problems can by trivial in a complexity-theoretic sense: one such is PARITY, the problem consisting of those finite structures, over some fixed signature, of even size (PARITY is definable in neither inductive nor partial fixed-point logic). Whilst having logical characterizations of complexity classes on the class of ordered finite structures is very useful on a number of counts, having logical characterizations on the class of all finite structures is much more preferable. For instance: ordered structures are much more difficult to work with when applying the inexpressibility tools of finite model theory; in applications of descriptive complexity theory, to database theory for example, it is almost always the case that finite structures are not ordered; and it is often undecidable as to whether a formula is well-formed when we restrict ourselves to the class of ordered finite structures (we shall return to this remark later when we discuss what it means for a logic to be a logic!).

Whilst it remains unknown as to whether there is a logic capturing **P** (or indeed any complexity class for which the expectation is that it is properly contained in **NP**) on the class of all finite structures, Abiteboul and Vianu proved the following result [2, 3]: **P** = **PSPACE** if, and only if, the classes of problems, on the class of all finite structures, defined by the sentences of inductive fixed-point logic and partial fixed-point logic, respectively, are one and the same (we reiterate that there are trivial problems, in a complexity-theoretic sense, not definable in inductive and partial fixed-point logic). It is particularly interesting that a complexity-theoretic statement like **P** = **PSPACE** has an equivalent formulation in logic where there is a such a mismatch between the complexity class and the class of problems defined in the respective logic. Abiteboul, Vardi and Vianu [1] and Dawar [12] subsequently obtained similar results involving a range of complexity classes including **P**, **NP**, the classes of the Polynomial Hierarchy **PH**, **PSPACE** and **EXPTIME**. One thing all of the logics involved in these results have in common is that they can be realised as fragments of bounded-variable infinitary logic $\mathcal{L}^{\omega}_{\infty\omega}$ (one property of which is that it has a zero-one law on the class of all finite structures; and so there is no chance of a tight relationship between the respective complexity classes and logics).

In obtaining their result, Abiteboul and Vianu developed a model of computation known as a loosely coupled generic machine which was renamed a relational machine in [1]. Essentially, a relational machine is a Turing machine augmented with a relational store and the ability to perform relational operations on this store. Also, the input to a relational machine is a finite structure and not the encoding of such as a string of symbols (encoding an input as a string of symbols, as is the case with a standard Turing machine, means essentially providing an order on the elements of the underlying finite structure). Relational machines can actually be regarded as an effective fragment of $\mathcal{L}^{\omega}_{\infty\omega}$. However, Abiteboul and Vianu are not the only researchers in recent times to work with models of computation taking finite structures as their inputs. Perhaps the best known computational model taking finite structures as inputs is the abstract state machine (formerly called evolving algebra) due to Gurevich [22].

Continuing this theme of developing computational models taking finite structures, and not the encodings of such, as their inputs, the expressive power of different classes of program schemes has recently been considered. Program schemes are similar in flavour to relational machines and abstract state machines. They form a model of computation that is closer to the notion of a high-level program than a logical formula is; yet they remain amenable to logical manipulation. Program schemes (of various sorts) originated in the 70s (for example, see [8, 9, 16, 31]), without much regard being paid to an analysis of resources, before a closer complexity analysis was undertaken in, mainly, the 80s (for example, see [23, 27, 29, 37]: though the analysis tended to be on ordered finite structures). In [18, 32, 34, 35, 36], the computational power of different classes of program schemes, on the class of all finite structures, is compared with the expressive power of logics previously considered in descriptive complexity theory (and also with the computational power of resource-bounded classes of Turing machines).

A crucial difference between program schemes and the traditional logics of descriptive complexity theory is that program schemes can be augmented with 'computational' constructs not immediately available in logic, such as stacks and arrays; although it is sometimes the case that the resulting classes of problems have arisen previously in a different guise. For example: in [5], it was shown that the program schemes of the class NPS accept exactly the class of problems defined by the sentences of Immerman's transitive closure logic [25], and that the program schemes of the class NPSS (essentially, the program schemes of NPS augmented with access to a stack) accept exactly the class of problems defined by the sentences of path system logic [20, 28, 33]; in [34], the program schemes of the class NPSA (essentially, the program schemes of NPS augmented with access to arrays) were shown to have a zero-one law and to accept problems not definable in $\mathcal{L}^{\omega}_{\infty\omega}$; and in [18], the class of problems accepted by the program schemes of the class RFDPS (obtained by incorporating for-loops as opposed to while-loops into program schemes) was shown to be a proper sub-class of the polynomial-time solvable problems, to contain every problem defined by the sentences of inductive fixed-point logic, and to contain problems not in the class of problems known as Choiceless Polynomial-Time, due to Blass, Gurevich and Shelah [6].

In this paper, we augment the basic class of program schemes NPS(1) (the first class of program schemes in the infinite hierarchy which defines NPS) with a queue and an extra 'numeric' universe. We prove that when both augmentations are present, the class of problems accepted by the resulting class of program schemes $\text{NPSQ}_{+}(1)$ is exactly the class of recursively enumerable problems, *i.e.*, the class of problems $\mathcal{RE}$; and when only the queue is added, the class of problems accepted by the resulting class of program schemes NPSQ(1) is exactly the class of recursively enumerable problems that are closed under extensions, *i.e.*, the class of problems $\mathcal{RE} \cap \mathcal{EXT}$. Intuitively, the class of program schemes NPSQ(1) caters for first-order existential quantification through its ability to guess. We proceed to introduce universal quantification by constructing an infinite hierarchy of classes of program schemes obtained by interleaving the application of universal quantification with the basic constructs of the program schemes of NPSQ(1). We denote the union of the classes of the resulting hierarchy by NPSQ. It turns out that the class of problems accepted by the program schemes of NPSQ has an alternative 'semantic' characterization as the class of problems defined by the sentences of any vectorized Lindström logic for which the problem corresponding to the operator is recursively enumerable and closed under extensions. It follows as a corollary that this class of problems has a zero-one law. We remark that by a result of [34], there are problems in **NP** that are accepted by program schemes of NPSQ(1) but which are not definable in $\mathcal{L}^{\omega}_{\infty\omega}$, which is an encompassing logic for many of the logics studied in finite model theory (as well as for the relational machines of Abiteboul and Vianu). Furthermore, we show that NPSQ can actually be realized as a vectorized Lindström logic (and not just as the union of such) where the operator defining the logic corresponds to a problem complete for NPSQ(1) (resp. $\mathcal{RE}$) via quantifier-free first-order translations (resp. with a numeric universe).

The fact that NPSQ$_+$(1) accepts exactly the recursively enumerable problems enables us to restrict the resources (time and space) used by these program schemes so that we can logically capture many complexity classes ranging from **NP** through to **ELEMENTARY**. (By 'logically capture' we mean equate the problems in a complexity class with the problems defined by the sentences of a logic, where by 'logic' we mean according to Gurevich's definition [21], which we shall detail later . Harking back to an earlier remark on logical characterizations of complexity classes on the class of ordered finite structures, such characterizations are not always 'logical' in the sense of Gurevich.). By looking at the respective classes of program schemes where access to the numeric universe is removed, we move from the complexity class to its fragment of problems closed under extensions. In doing so, we show that many complexity-theoretic inequalities, *e.g.*, **NEXPTIME** $\neq$ **2-NEXPTIME**, and hypotheses, *e.g.*, **NEXPTIME** $\neq$ **EXPSPACE**, are equivalent to these inequalities and hypotheses for the fragments, *e.g.*, **NEXPTIME**$\cap\mathcal{EXT}$ $\neq$ **2-NEXPTIME**$\cap\mathcal{EXT}$ and **NEXPTIME**$\cap\mathcal{EXT}$ $\neq$ **EXPSPACE**$\cap\mathcal{EXT}$; and that these fragments can be logically captured (in the sense of Gurevich: it is not immediately obvious that a semantic restriction such as **NEXPTIME**$\cap\mathcal{EXT}$ can be captured syntactically by a logic). Hence, even though a conjecture such as **NEXPTIME** $\neq$ **EXPSPACE** might appear to live solely within the realm of Turing machines, it has equivalent formulations in logic: one involving resource-bounded classes of program schemes involving a numeric universe which capture exactly the complexity classes **NEXPTIME** and **EXPSPACE**, and another involving resource-bounded classes of program schemes without access to a numeric universe which capture the classes of problems **NEXPTIME**$\cap\mathcal{EXT}$ and **EXPSPACE**$\cap\mathcal{EXT}$. This is interesting for the same reason that the equivalence results of Abiteboul *at al.* are interesting, *i.e.*, given that complexity-theoretic trivial problems, like PARITY, can not be accepted by any of the program schemes in any of the fragments such as **NEXPTIME**$\cap\mathcal{EXT}$ and **EXPSPACE**$\cap\mathcal{EXT}$.

Whilst our equivalence results are similar to those of Abiteboul *et al.*, they involve different logics: recall, we mentioned earlier that there are problems in **NP** that are accepted by program schemes of NPSQ(1) but which are not definable in $\mathcal{L}^{\omega}_{\infty\omega}$. Also, whereas the logics involved in the results due to Abiteboul *el al.* have a zero-one law (because they are fragments of $\mathcal{L}^{\omega}_{\infty\omega}$), the classes of program schemes involved in our equivalence results define problems closed under extensions and so have a 'one-law' (apart from the program scheme, over some fixed signature, which accepts no finite structures).

In Section 2, we give the basic definitions concerning the program schemes encountered in this paper, before explaining how to relate classes of finite structures and sets of strings of symbols in Section 3. In Section 4, we prove our basic results concerning the classes of program schemes NPSQ$_+$(1) and NPSQ(1); and we extend the class of program schemes NPSQ(1) to an infinite hierarchy NPSQ in Section 5. Our applications are given in Section 6, and Section 7 contains our conclusions and some directions for further research.

2 Program schemes with queues

Throughout, all our signatures consist of a finite tuple of relation symbols (of various arities) and constant symbols; and we assume that every signature has at least one relation symbol. A *finite structure* $\mathcal{A}$ over the signature σ consists of: a finite *universe* (or *domain*), denoted $|\mathcal{A}|$; a relation $R^{\mathcal{A}} \subseteq |\mathcal{A}|^a$ for every relation symbol R of σ of arity a; and a constant $C^{\mathcal{A}} \in |\mathcal{A}|$ for every constant symbol C of σ (henceforth, we do not always differentiate between relations and relation symbols and between constants and constant symbols: this causes no confusion). If the universe $|\mathcal{A}|$ of the finite structure $\mathcal{A}$ has size n then we say that $\mathcal{A}$ has *size n*, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ too (again, this causes no confusion). Throughout, all our structures are finite and of size at least 2. A *problem* is an isomorphism-closed class of finite structures over some fixed signature. If $\mathcal{A}$ and $\mathcal{B}$ are both finite σ-structures such that $|\mathcal{A}| \subseteq |\mathcal{B}|$ and $\mathcal{B}$ restricted to $|\mathcal{A}|$ is $\mathcal{A}$ (and so, in particular, $C^{\mathcal{A}} = C^{\mathcal{B}} \in |\mathcal{A}|$ for every constant symbol C of σ) then $\mathcal{B}$ is an *extension* of $\mathcal{A}$ and we write $\mathcal{A} \subseteq \mathcal{B}$. If Ω is a problem, over the signature σ, for which: $\mathcal{A}$ and $\mathcal{B}$ are finite σ-structures; $\mathcal{A} \subseteq \mathcal{B}$; and $\mathcal{A} \in \Omega$, imply that $\mathcal{B} \in \Omega$, then Ω is said to be *closed under extensions*. We denote the class of problems that are closed under extensions by $\mathcal{EXT}$.

Program schemes are more 'computational' means for defining classes of problems than are logical formulae. A *program scheme* $\rho \in$ NPS(1) involves a finite set $\{x_1, x_2, \ldots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature σ. It consists of a finite sequence of *instructions* where the first instruction is '`input(`x_1`,`x_2`,...,`x_m`)`', for some $m \leq k$, and the variables of

$\{x_1, x_2, \ldots, x_m\}$ are known as the *input-output variables*, with the variables of $\{x_{m+1}, x_{m+2}, \ldots, x_k\}$ being the *free variables*. The remaining instructions are one of the following:

- an *assignment instruction* of the form 'x_i `:=` y', where $i \in \{1, 2, \ldots, m\}$ and where y is a variable from $\{x_1, x_2, \ldots, x_k\}$ or a constant symbol of σ;
- a *guess instruction* of the form '`guess` x_i', where $i \in \{1, 2, \ldots, m\}$;
- a *while instruction* of the form '`while` φ `do` $\alpha_1; \alpha_2; \ldots; \alpha_q$ `od`', where φ is a quantifier-free first-order formula over σ, involving variables from $\{x_1, x_2, \ldots, x_k\}$, and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ is another instruction of one of the forms given here (note that there may be nested while instructions); or
- an *accept instruction* `accept` or a *reject instruction* `reject`.

A program scheme $\rho \in \mathrm{NPS}(1)$ over σ with $k-m$ free variables takes expansions $\mathcal{A}'$ of σ-structures with $k-m$ constants as input (one constant for each free variable) and computes on $\mathcal{A}'$ in the obvious way except that: execution of an instruction '`guess` x_i' non-deterministically assigns an element of $|\mathcal{A}'|$ to the variable x_i; initially, every input-output variable is non-deterministically assigned a value from $|\mathcal{A}'|$; and if a computation encounters an accept or a reject instruction then the computation halts. Note that the value of a free variable never changes throughout a computation: the free variables appear as if they were constant symbols. The structure $\mathcal{A}'$ is *accepted* by ρ, and we write $\mathcal{A}' \models \rho$, if, and only if, there exists a computation of ρ such that an accept instruction is reached (note that some computations might not be terminating). We can easily build the usual 'if' and 'if-then-else' instructions using while instructions (see, for example, [32]): henceforth, we shall assume that these instructions are at our disposal. Note that the class of structures accepted by a program scheme of NPS(1) is a problem, *i.e.*, closed under isomorphisms.

Remark 1 The class of program schemes NPS(1) was defined slightly differently in [5] in that two 'built-in' constants were always assumed to be available and acceptance was determined by the values of the input-output variables at an appropriate point in the computation. This is of no significance to us in that the classes of problems defined by our class of program schemes NPS(1) and the identically-named class of program schemes from [5] are one and the same. Also, the parameter '1' in NPS(1) reflects the fact that this class of program schemes is the first class in an infinite hierarchy of classes of program schemes. We shall return to such hierarchies later. $\square$

Whereas a stack and arrays were incorporated into the program schemes of NPS(1) in [5, 34, 35, 36], let us now augment these program schemes with access to a queue, *i.e.*, a first-in, first-out store. In addition to the above instructions, we include the instructions '`push` x_i' and 'x_i `:= pop`' although in this latter instruction we insist that x_i must be an input-output variable, whereas in the former it can also be a free variable. The first instruction takes the current value of the variable x_i and pushes this value onto the end of the queue (the value of x_i does not change and the length of the queue increases by 1), and the second takes the current value from the head of the queue and sets x_i to have this value (with the length of the queue decreasing by 1). Initially, the queue is empty and if ever we attempt to pop from an empty queue then that particular computation is deemed not to be accepting. We denote the program schemes of NPS(1) augmented with a queue by NPSQ(1).

We can also augment our program schemes with a numeric universe. That is, we can assume that the variables of a program scheme are of one of two types: variables of the first type, the *element type*, take values from the domain of the input structure (as they have done so far); and variables of the second type, the *numeric type*, take values from the *numeric universe*, namely $\{0, 1, \ldots, n-1\}$, where the input structure has size n (we assume that the universe of the input structure and the numeric universe are disjoint). We insist that all variables of numeric type must be input-output variables and that they are initialized to 0. Additionally, there are two constant symbols 0 and max which are always interpreted as the numbers 0 and $n-1$. The instructions available to the variables of numeric type, $t_1, t_2, \ldots, t_q$, say, are assignments of the form 't_i `:=` t_j', 't_i `:=` 0' and 't_i `:=` max', and assignments adding one to the value of a variable, of the form 't_i `:=` t_j `+ 1`' (if the variable t_j has value $n-1$ then execution of this instruction causes that particular computation to reject the input); and 't_i `=` t_j', 't_i `=` 0', 't_i `=` max' and their negations can appear as atoms and negated atoms in quantifier-free first-order formulae used as tests in while instructions (these quantifier-free first-order formulae might be combinations of atoms involving variables of both element and numeric type). Note that we do not allow numeric values to be pushed onto the stack (if we did then we would have to deal with the possibility of a type mismatch when popping values off the stack). The class of program schemes NPSQ(1) augmented with a numeric universe, as above, is denoted NPSQ$_+$(1). Again, the classes of structures accepted by the program schemes of NPSQ$_+$(1) are problems.

3 Finite structures and Turing machines

Finite structures are abstract objects and, as such, there is generally no canonical representation of a finite structure as a string over $\{0, 1\}$, *i.e.*, as a bit-string. Nevertheless, we can still arrange for finite structures to be input to Turing machines (throughout this paper all our Turing machines are non-deterministic).

Consider the signature σ consisting of the relation symbols $R_1, R_2, \ldots, R_r$, of arities $a_1, a_2, \ldots, a_r$, respectively, and the constant symbols $C_1, C_2, \ldots, C_c$. Let $\mathcal{A}$ be a σ-structure of size n. Encode $\mathcal{A}$ in the following way. First, choose a linear ordering $u_0, u_1, \ldots, u_{n-1}$ of the elements of $|\mathcal{A}|$. Next, encode each relation $R_i^{\mathcal{A}}$ as the bit-string of length n^{a_i} where the jth bit is 1 (resp. 0) if the tuple $(u_{k_1}, u_{k_2}, \ldots, u_{k_{a_i}})$, where $(k_1, k_2, \ldots, k_{a_i})$ is the jth tuple in the lexicographic ordering of the elements of $\{0, 1, \ldots, n-1\}^{a_i}$, is such that $R_i(u_{k_1}, u_{k_2}, \ldots, u_{k_{a_i}})$ holds (resp. does not hold) in $\mathcal{A}$. Next, encode the constant $C_i^{\mathcal{A}}$ as the $(\lceil \log_2(n) \rceil)$-bit binary representation of the integer k for which the constant $C_i^{\mathcal{A}}$ has the value u_k. Finally, take the encoding $e_\sigma(\mathcal{A})$ of $\mathcal{A}$ to be the concatenation of the bit-strings encoding $R_1, R_2, \ldots, R_r, C_1, C_2, \ldots$, and C_c. Of course, our encoding scheme is non-deterministic in that we chose one of the $n!$ linear orderings of the elements of $|\mathcal{A}|$ upon which to base our encoding of $\mathcal{A}$.

We say that a problem Ω over σ is *accepted* by a Turing machine M if M accepts exactly those bit-strings encoding structures in Ω (with respect to any one of the above encoding schemes: our Turing machine need not halt on inputs not accepted); and so, in particular, as to whether an encoding $e_\sigma(\mathcal{A})$ of any σ-structure $\mathcal{A}$ is accepted by M is independent of the particular linear order chosen when encoding $\mathcal{A}$. Note that the length of $e_\sigma(\mathcal{A})$ is also independent of the particular linear ordering chosen; and if $|\mathcal{A}| = n$ then we denote this length by $e_\sigma(n)$. Define $\mathcal{RE}$ as the class of problems accepted by some Turing machine. Note that not every Turing machine accepts a problem and that $\mathcal{RE}$ is not the class of recursively enumerable languages but the class of recursively enumerable problems[1]. However, any (resp. recursively enumerable) language can be realized as a problem over the signature σ consisting of the binary relation symbol L and the unary relation symbol B (resp. accepted by a Turing machine). A string over $\{0, 1\}$ can be considered as a σ-structure $\mathcal{A}$ (in fact, as a class of σ-structures) whose binary relation $L^{\mathcal{A}}$ describes a *successor relation*, *i.e.*, a relation $\{(u_0, u_1), (u_1, u_0), \ldots, (u_{n-2}, u_{n-1})\}$, where $|\mathcal{A}| = \{u_0, u_1, \ldots, u_{n-1}\}$, which details the bit positions of the string, and whose unary relation $B^{\mathcal{A}}$ details which bits are 0 and which bits are 1 (with

[1]In the extended abstract of this paper, referenced earlier, we called the class of problems accepted by some Turing machine the class of recursive problems and denoted it $\mathcal{REC}$. This was misleading and we have altered our definition and notation accordingly in this paper.

respect to this successor relation).

4 Recursively enumerable problems

We begin by proving that the class of program schemes NPSQ$_+(1)$ consists of the recursively enumerable problems. Throughout, we identify a class of program schemes (resp. a logic, a class of Turing machines) with the class of problems accepted by those program schemes (resp. defined by the sentences of the logic, accepted by those Turing machines).

Theorem 2 *A problem is accepted by a program scheme of NPSQ$_+(1)$ if, and only if, it can be accepted by a Turing machine; that is, NPSQ$_+(1) = \mathcal{RE}$.*

Proof Suppose that the problem Ω is accepted by some (non-deterministic) Turing machine M. We shall construct a program scheme ρ of NPSQ$_+(1)$ which simulates M. Without loss of generality, we can assume that: the Turing machine M has a two-way infinite work-tape and a read-only input-tape; the work-tape alphabet is $\{0, 1, b\}$; and the input-tape has a left-hand marker symbol l and a right-hand marker symbol r which delimit the input string.

Essentially, the queue of our program scheme ρ will hold a description of the tapes of the Turing machine. We start with the work-tape. Let u and v be distinct elements of an input structure (we can guess these elements as the first act of our program scheme). In our description of the work-tape, we encode the work-tape symbol 0 as the triple (u, u, u), the work-tape symbol 1 as the triple (v, v, v) and the blank symbol b as the triple (u, u, v); and we use the triples (v, u, u) and (v, v, u) as delimiters. Let the work-tape at some instantaneous description (ID) of M be of the form:

$$w_1, w_2, \ldots, w_i, \ldots, w_m,$$

reading from left to right where w_1 (resp. w_m) is the first blank symbol to the left (resp. right) of the work-tape head beyond which the work-tape is entirely blank. Furthermore, suppose that the work-tape head is scanning the symbol w_i. The queue will consist of: the triple encoding w_1; the triple encoding w_2; $\ldots$; the triple (v, u, u) (to denote that the head is scanning the next symbol); the triple encoding w_i; $\ldots$; the triple encoding w_m; and the triple (v, v, u) (to denote that we have reached the end of our description of the work-tape). In particular, by popping and pushing symbols from and onto the queue until we find the triple (v, u, u), we can ascertain the symbol the work-tape head is currently scanning; and having the description of the ID appear on the queue 'cyclically shifted' does not result in any information loss.

We shall describe the state of the Turing machine M in any ID using a constant number of variables of our program scheme ρ. For example, if M has q states then: we could encode the first state by setting the tuple of (input-output) variables $(x_1, x_2, \ldots, x_q)$ of ρ to $(u, v, \ldots, v)$; we could encode the second state by setting the tuple of variables $(x_1, x_2, \ldots, x_q)$ to $(v, u, v, \ldots, v)$; and so on. Also, should our program scheme ρ know which move of the Turing machine M to simulate and the effect upon the work-tape and the work-tape head, we could easily simulate this move by making appropriate manipulations of the queue, *i.e.*, repeated popping and pushing.

Consequently, we are left with the problem of dealing with the input tape. This is more complicated as initially our program scheme is not given an encoding of the input structure $\mathcal{A}$, which has size n, say: it simply has access to the raw finite structure itself. However, we can build our own encoding of $\mathcal{A}$. Prior to any simulation of the Turing machine M, our program scheme guesses a linear ordering on $|\mathcal{A}|$ and stores this linear ordering on the queue (protected by delimiters). Note that because we have access to the (disjoint) domain $\{0, 1, \ldots, n-1\}$, the capacity to count using the elements of this domain and the integers 0 and $n-1$ (through the constants 0 and max), we can ensure that our guessed linear order contains every element of $|\mathcal{A}|$ exactly once (this is the only place in ρ where we use the numeric universe and the add-one operation with its associated constants).

Suppose that our guessed linear order is $u_0, u_1, \ldots, u_{n-1}$. This linear order defines a concrete encoding of our input structure (see the discussion at the beginning of this section) and it is this encoding which we assume is presented to the Turing machine M (note that because of our stipulation that M accepts either every encoding of $\mathcal{A}$ or no encoding of $\mathcal{A}$, we can work with any encoding we choose to). We use our linear order to write a description of the input-tape of M on our queue. In more detail, suppose that we wished to describe the encoding of a binary relation $E^{\mathcal{A}}$ on our queue. Within ρ, we would have two variables in which to store the particular bit of $E^{\mathcal{A}}$ we were dealing with (starting with the bit indexed by (u_0, u_0)). Using while-loops, our linear order and repeated poppings and pushings, we would register the bits of $E^{\mathcal{A}}$ indexed by the pairs:

$$(u_0, u_0), (u_0, u_1), \ldots, (u_0, u_{n-1}), (u_1, u_0), (u_1, u_1), \ldots, (u_{n-1}, u_{n-1})$$

on the queue. We encode constants similarly; although note that we use the linear order, and not the numeric universe and the add-one operation, in our calculation of the binary representation of the index of our constant (this will be of relevance in a later proof). As with the work-tape, the input-tape head position is registered using delimiters.

It is now straightforward, given descriptions of the input-tape and the work-tape on the queue, to simulate the computation of M. The converse, that any problem in NPSQ$_+(1)$ is in $\mathcal{RE}$, is trivial. $\Box$

Our strategy to simulate a Turing machine in the proof of Theorem 2 depended upon us being able to guess a linear order and also to check that this was a *bona fide* linear order by counting in our numeric universe. The question remains as to what sort of problems can be accepted by program schemes of NPSQ(1).

Theorem 3 *A problem is accepted by a program scheme of NPSQ*(1) *if, and only if, it can be accepted by a Turing machine and is closed under extensions*; *that is, NPSQ*$(1) = \mathcal{RE} \cap \mathcal{EXT}$.

Proof Let $\mathcal{A}$ and $\mathcal{B}$ be σ-structures such that $\mathcal{A} \subseteq \mathcal{B}$. If $\mathcal{A}$ is accepted by some program scheme ρ of NPSQ(1) then we can mirror an accepting computation in a computation of ρ on $\mathcal{B}$ (essentially, because our tests in while instructions are quantifier-free). Hence, NPSQ(1) $\subseteq \mathcal{RE} \cap \mathcal{EXT}$.

Conversely, suppose that Ω is a problem, over the signature σ, in $\mathcal{RE} \cap \mathcal{EXT}$. In particular, Ω is accepted by some Turing machine M. Construct a program scheme ρ' of NPSQ(1) as in the proof of Theorem 2 except that when the linear order is guessed, the checks made are that every element appears in the linear order at most once and that every constant appears in the linear order. Note that when the input-tape is described on the queue of ρ', the parameter n used in the description is not the size of the input structure but the number of elements in the domain of the linear order. Also, whenever ρ' makes a guess, include code to ensure that the guess is always an element in the domain of the linear order (essentially, 'keep guessing until a good guess is made').

Suppose that the σ-structure $\mathcal{A}$ is in Ω; that is, every encoding of $\mathcal{A}$ is accepted by M. Then there is a computation of ρ' on input $\mathcal{A}$ which guesses a linear order whose domain is the whole of $|\mathcal{A}|$; and as a result, this computation of ρ' on input $\mathcal{A}$ simulates that of M on the respective encoding of $\mathcal{A}$. Thus, $\mathcal{A}$ is accepted by ρ'.

Suppose that the σ-structure $\mathcal{A}$ is accepted by ρ'. Then there is an accepting computation where this computation guesses a linear order whose elements come from the subset B of $|\mathcal{A}|$ (and B contains all constants of $\mathcal{A}$). Let $\mathcal{B}$ be the restriction of $\mathcal{A}$ to B (and so $\mathcal{A}$ is an extension of $\mathcal{B}$). The accepting computation of ρ' on input $\mathcal{A}$ simulates an accepting computation of M on input the respective encoding of $\mathcal{B}$ (because any guess results in an element of B). Hence, this encoding of $\mathcal{B}$ is accepted by M and thus $\mathcal{B} \in \Omega$. But Ω is closed under extensions and so $\mathcal{A} \in \Omega$. The result follows. $\Box$

5 Hierarchies and Lindström logics

We can extend our class of program schemes NPSQ(1) to a hierarchy of classes of program schemes essentially by interleaving applications of universal quantifiers with the basic constructs of the program schemes of NPSQ(1). In more detail, assume that we have defined a class of program schemes NPSQ($2m-1$), for some $m \geq 1$, and that any program scheme has associated with it: a set of input-output variables; a set of free variables; and a set of bound variables (this is certainly the case when $m = 1$, with the associated set of bound variables being empty).

Definition 4 Let the program scheme $\rho \in$ NPSQ($2m-1$) be over the signature σ. Suppose that ρ has: input-output variables $x_1, x_2, \ldots, x_k$; free variables $x_{k+1}, x_{k+2}, \ldots, x_{k+s}$; and bound variables $x_{k+s+1}, x_{k+s+2}, \ldots, x_{k+s+t}$ (note that if $m = 1$ then ρ has no bound variables: this may not be the case if $m > 1$). Let $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ be free variables of ρ, for some p (and so $k+1 \leq i_1 < i_2 < \ldots < i_p \leq k+s$). Then:

$$\forall x_{i_1} \forall x_{i_2} \ldots \forall x_{i_p} \rho$$

is a program scheme of NPSQ($2m$), which we denote by ρ', with: no input-output variables; free variables those of $\{x_{k+1}, x_{k+2}, \ldots, x_{k+s}\} \setminus \{x_{i_1}, x_{i_2}, \ldots, x_{i_p}\}$; and the remaining variables of $\{x_1, x_2, \ldots, x_{k+s+t}\}$ as its bound variables.

A program scheme such as ρ' takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining $s-p$ constants as input (one for each free variable), and ρ' accepts such an expansion $\mathcal{A}'$ if, and only if, for every expansion $\mathcal{A}''$ of $\mathcal{A}'$ by p additional constants (one for each variable x_{i_j}, for $j \in \{1, 2, \ldots, p\}$), $\mathcal{A}'' \models \rho$ (the computation on such an expansion $\mathcal{A}''$ always starts with the queue initialized as empty). □

Definition 5 A program scheme $\rho' \in$ NPSQ($2m-1$), for some $m \geq 2$, over the signature σ, is defined exactly as is a program scheme of NPSQ(1) except that the test in any while instruction is a program scheme $\rho \in$ NPSQ($2m-2$). The bound variables of ρ' consist of the bound variables of any test in any while instruction; all free variables in any test in any while instruction are input-output or free variables of ρ'; and there may be other free and input-output variables (appearing in ρ' at the 'top level' but not in any test). Of course, any free variable never appears on the left-hand side of an assignment instruction, in a guess instruction or in a pop instruction in ρ' (at the 'top level').

Suppose that a program scheme $\rho' \in$ NPSQ($2m-1$) has s free variables. Then it takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining s constants as input and computes on $\mathcal{A}'$ in the obvious way; except that when some while instruction is encountered, the test, which is a program scheme $\rho \in \text{NPSQ}\ (2m-2)$, is evaluated according to the expansion of $\mathcal{A}'$ by the current values of any relevant input-output variables of ρ' (which may be free in ρ). In order to evaluate this test, the queue associated with ρ is initialized as empty and when the test has been evaluated the computation of ρ' resumes accordingly with its queue and input-output and free variables being exactly as they were immediately prior to the test being evaluated. In particular, queues can not be 'passed across' in the evaluation of tests: the values of variables can be but they are never amended in the process. $\square$

Consequently, we obtain a hierarchy of classes of problems:

$$\text{NPSQ}(1) \subseteq \text{NPSQ}(2) \subseteq \ldots \subseteq \cup\{\text{NPSQ}(i) : i = 1, 2, \ldots\} = \text{NPSQ}$$

(we use the inclusion relation between consecutive classes because this is how they are related as classes of problems). Similar hierarchies have been defined previously. In [5], the basic NPS hierarchy was considered (where the program schemes have no access to a queue or any other sort of additional storage), as was the NPSS hierarchy where the program schemes of NPS are given access to a stack; in [34], the NPSA hierarchy was considered where the program schemes of NPS are given access to arrays; and in [36], the NPSB hierarchy was considered where the program schemes of NPSA are restricted so that the arrays are, in a sense, 'binary write once'. The semantics of the program schemes of NPSS, NPSA and NPSB are similar to those of NPSQ in that neither stacks nor arrays can be 'passed across' in the evaluation of tests. We have that:

$$\text{NPS} \subseteq \text{NPSS} \subseteq \text{NPSB} \subseteq \text{NPSA},$$

with analogous inclusions holding at each level of the corresponding hierarchies. It is straightforward to simulate access to and from an array using a program scheme with a queue. Consequently, we have that NPSA $\subseteq$ NPSQ (and, again, inclusions hold at each level of the hierarchies).

The classes of program schemes NPS, NPSS, NPSB and NPSA have two important properties in common: they all have equivalent formulations as certain vectorized Lindström logics, and, as a consequence, have zero-one laws. Let us now define these concepts.

We start with the notion of a zero-one law. For the moment, fix the domain of any finite structure of size n as $\{0, 1, \ldots, n-1\}$. We say that a problem Ω has a *zero-one law* if the fraction defined as the number of σ-structures of size n in Ω divided by the total number of σ-structures of size n tends to either 0 or 1 as n tends to infinity. It is not difficult to see that if any (non-trivial) problem has a zero-one law then it must be over a relational signature, *i.e.*, a signature devoid of constant symbols. A logic (resp. class of program schemes) has a *zero-one law* if the problems over relational signatures defined by the sentences of the logic (resp. accepted by the program schemes of the class) all have a zero-one law. We now revert back to the domains of our structures being arbitrary finite sets.

Suppose that the problem Ω is over the signature σ, where $\sigma = \langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, with each relation symbol R_i having arity a_i and each C_i being a constant symbol. The logic $(\pm\Omega)^*[\mathrm{FO}]$ consists of those formulae built using the usual constructs of first-order logic and also the *operator* (or, more precisely, *vectorized sequence of Lindström quantifiers*) Ω, where the operator Ω is applied as follows.

- Suppose that $\psi_1(\mathbf{x}^1, \mathbf{y}), \psi_2(\mathbf{x}^2, \mathbf{y}), \ldots, \psi_r(\mathbf{x}^r, \mathbf{y})$ are well-formed formulae of $(\pm\Omega)^*[\mathrm{FO}]$ such that:
  - each $\mathbf{x}^i$ is a ka_i-tuple of distinct variables, for some fixed $k \geq 1$ and for each $i \in \{1, 2, \ldots, r\}$;
  - $\mathbf{y}$ is an m-tuple of distinct variables, for some $m \geq 0$, each of which is different from any variable of $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^r$; and
  - all free variables of any ψ_i are contained in either $\mathbf{x}^i$ or $\mathbf{y}$, for each $i \in \{1, 2, \ldots, r\}$.
- Suppose that $\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c$ are k-tuples of variables and constant symbols from the underlying signature (these variables and constant symbols need not be distinct).
- Then
$$\Omega[\lambda \mathbf{x}^1 \psi_1(\mathbf{x}^1, \mathbf{y}), \mathbf{x}^2 \psi_2(\mathbf{x}^2, \mathbf{y}), \ldots, \mathbf{x}^r \psi_r(\mathbf{x}^r, \mathbf{y})](\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c)$$
is a formula of $(\pm\Omega)^*[\mathrm{FO}]$ whose free variables are the variables of $\mathbf{y}$ together with any other variables appearing in $\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c$.

Note that applications of the operator Ω can be nested, appear within the scope of quantifiers, appear negated, and so on.

If Φ is a sentence of the form $\Omega[\lambda \mathbf{x}^1 \psi_1(\mathbf{x}^1), \mathbf{x}^2 \psi_2(\mathbf{x}^2), \ldots, \mathbf{x}^r \psi_r(\mathbf{x}^r)](\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c)$, as above, over some signature σ', then we interpret Φ in a σ'-structure $\mathcal{A}$ inductively as follows (note that as Φ is a sentence, the variables of $\mathbf{y}$, above, are absent and the tuples $\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c$, which are only there if there are constant symbols in σ, consist entirely of constant symbols of σ'). First, we build a σ-structure $\Phi(\mathcal{A})$.

- The domain of the σ-structure $\Phi(\mathcal{A})$ is $|\mathcal{A}|^k$.
- For each $i \in \{1, 2, \ldots, r\}$, the relation R_i of $\Phi(\mathcal{A})$ is defined via:

  - for any $\mathbf{u} \in |\Phi(\mathcal{A})|^{a_i} = |\mathcal{A}|^{ka_i}$, $R_i(\mathbf{u})$ holds in $\Phi(\mathcal{A})$ if, and only if, $\psi_i(\mathbf{u})$ holds in $\mathcal{A}$.

- For each $j \in \{1, 2, \ldots, c\}$, the constant C_j of $\Phi(\mathcal{A})$ is defined via:
  - C_j is the interpretation of the tuple of constants $\mathbf{d}_j$ in $\mathcal{A}$.

We define that $\mathcal{A} \models \Phi$ if, and only if, $\Phi(\mathcal{A}) \in \Omega$ (the situation where Φ has free variables is similar except that Φ is interpreted in expansions of σ'-structures by an appropriate number of constants).

Logics such as $(\pm\Omega)^*$[FO] are called *vectorized Lindström logics*. Vectorized Lindström logics play an important role in finite model theory. For example, there are a number of characterizations of complexity classes as the classes of problems definable, sometimes only on the class of ordered finite structures (which we discuss soon), by the sentences of (fragments of) certain vectorized Lindström logics (see [14, 26]); and Dawar [11] has shown that if the complexity class **P** can be captured on the class of all finite structures by *any* logic then it can be captured by a vectorized Lindström logic (we discuss later what it means to be a 'logic').

The realizations in [5, 34, 36] of NPS, NPSS, NPSB and NPSA as vectorized Lindström logics are through operators corresponding to the *transitive closure problem* (see, for example, [25]), the *path system problem* (see, for example, [20, 28, 33]) and two problems whose instances involve Petri nets, respectively.

The following lemma will be required later. Given operators Ω_1 and Ω_2 (corresponding to problems Ω_1 and Ω_2), we denote by $(\pm\Omega_1, \pm\Omega_2)^*$[FO] the vectorized Lindström logic built using the constructs of first-order logic and both operators Ω_1 and Ω_2. Obviously, we can use more than two such operators and the resulting logic is denoted similarly.

Lemma 6 *Let Ω_1 and Ω_2 be problems that are in $\mathcal{RE} \cap \mathcal{EXT}$. Then there exists a problem $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$ such that the class of problems defined by the sentences of $(\pm\Omega_1, \pm\Omega_2)^*[FO]$ is identical to the class of problems defined by the sentences of $(\pm\Omega)^*[FO]$.*

Proof Suppose, for simplicity, that the problem Ω_1 is over the signature $\sigma_1 = \langle R, C\rangle$, where R is a relation symbol of arity 1 and C is a constant symbol, and that the problem Ω_2 is over the signature $\sigma_2 = \langle T_1, T_2\rangle$, where T_1 and T_2 are relation symbols of arities 1 and 2, respectively (the general case is similar). Define the signature $\sigma = \langle R, T_1, T_2, C, D_0, D_1\rangle$, where D_0 and D_1 are additional constant symbols. Define the problem Ω over σ to consist of those σ-structures $\mathcal{A}$ such that either $D_0^{\mathcal{A}} = D_1^{\mathcal{A}}$ and $\mathcal{A}$ restricted to the symbols of σ_1 is in Ω_1, or $D_0^{\mathcal{A}} \neq D_1^{\mathcal{A}}$ and $\mathcal{A}$ restricted to the symbols of σ_2 is in Ω_2.

Any formula of $(\pm\Omega)^*$[FO] of the form:

$$\Omega[\lambda\mathbf{x}\psi, \mathbf{x}^1\psi_1, \mathbf{x}^2\psi_2,](\mathbf{y}, \mathbf{z}^1, \mathbf{z}^2)$$

(where ψ describes a relation corresponding to R, ψ_1 describes a relation corresponding to T_1 and ψ_2 describes a relation corresponding to T_2, with all tuples of variables of appropriate lengths) is logically equivalent to the formula:

$$(\mathbf{z}^1 = \mathbf{z}^2 \wedge \Omega_1[\lambda\mathbf{x}\psi](\mathbf{y})) \vee (\mathbf{z}^1 \neq \mathbf{z}^2 \wedge \Omega_2[\lambda\mathbf{x}^1\psi_1, \mathbf{x}^2\psi_2]).$$

The formula $\Omega_1[\lambda\mathbf{x}\psi](\mathbf{y})$ is logically equivalent to the formula:

$$\exists\mathbf{z}(\Omega[\lambda\mathbf{x}\psi, \mathbf{x}^1(\mathbf{x}^1 = \mathbf{x}^1), \mathbf{x}^2(\mathbf{x}^2 = \mathbf{x}^2)](\mathbf{y}, \mathbf{z}, \mathbf{z})),$$

and similarly for the formula $\Omega_2[\lambda\mathbf{x}^1\psi_1, \mathbf{x}^2\psi_2]$. The result follows as $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$. $\square$

Theorem 7

$$NPSQ = \bigcup\{(\pm\Omega)^*[FO] : \Omega \in \mathcal{RE} \cap \mathcal{EXT}\}.$$

Proof Fix $c > 0$. Assume, as our induction hypothesis, that every problem accepted by a program scheme of NPSQ(i), where $i < 2c + 1$, can be defined by a sentence of a vectorized Lindström logic $(\pm\Omega)^*$[FO] where the problem $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$.

Let ρ be a program scheme of NPSQ($2c + 1$) over the relational signature $\sigma = \langle R_1, R_2, \ldots, R_m\rangle$, where the relation symbol R_i has arity a_i (w.l.o.g. we may suppose that ρ does not contain if and if-then-else instructions, only while instructions). Suppose further that ρ has input-output variables $x_1, x_2, \ldots, x_p$ and q while instructions. Let the tests, *i.e.*, program schemes of NPSQ($2k$), in these instructions be $\rho_1, \rho_2, \ldots, \rho_q$, and w.l.o.g. assume that the free variables of ρ_i are $x_1, x_2, \ldots, x_p$, for $i \in \{1, 2, \ldots, q\}$.

Amend ρ so that it becomes a program scheme ρ_0 of NPSQ(1) over the signature $\sigma_0 = \sigma \cup \langle T_1, T_2, \ldots, T_q\rangle$, where $T_1, T_2, \ldots, T_q$ are new relation symbols of arity p, by replacing the test ρ_i with the atomic formula $T_i(x_1, x_2, \ldots, x_p)$, for each $i \in \{1, 2, \ldots, q\}$.

Define the problem Ω_0 over σ_0 as:

$$\{\mathcal{A} \text{ is a } \sigma_0\text{-structure} : \mathcal{A} \models \rho_0\}.$$

By Theorem 3, Ω_0 is in $\mathcal{RE} \cap \mathcal{EXT}$. Let σ' be the expansion of σ with p additional constant symbols and consider each ρ_i as accepting a problem over σ'. For each $i \in \{1, 2, \ldots, q\}$, the induction hypothesis applied to ρ_i yields that there is a problem $\Omega_i \in \mathcal{RE} \cap \mathcal{EXT}$ such that the problem over

σ' accepted by ρ_i is defined by a sentence Φ_i of the logic $(\pm\Omega_i)^*[\mathrm{FO}]$. For each $i \in \{1, 2, \ldots, q\}$, let the formula $\varphi_i(x_1, x_2, \ldots, x_p)$ be obtained from Φ_i by replacing the additional constant symbols by the variables $x_1, x_2, \ldots, x_p$. Then, for every σ-structure $\mathcal{A}$, $\mathcal{A} \models \rho$ if, and only if,

$$\mathcal{A} \models \Omega_0[\lambda \mathbf{x}^1 R_1(\mathbf{x}^1), \ldots, \mathbf{x}^m R_m(\mathbf{x}^m), \mathbf{y}^1 \varphi_1(\mathbf{y}^1), \ldots, \mathbf{y}^q \varphi_q(\mathbf{y}^q)],$$

where $\mathbf{x}^i$ is an a_i-tuple of variables, for each $i \in \{1, 2, \ldots, m\}$, and $\mathbf{y}^i$ is a p-tuple of variables, for each $i \in \{1, 2, \ldots, q\}$. By Lemma 6, the problem accepted by ρ can be defined by a sentence of a vectorized Lindström logic $(\pm\Omega)^*[\mathrm{FO}]$ where the problem $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$.

The base case of our induction is when $\rho \in \mathrm{NPSQ}(1)$. Let Ω be the problem accepted by ρ. By Theorem 3, the problem $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$ and it is trivially the case that Ω can be defined by a sentence of $(\pm\Omega)^*[\mathrm{FO}]$. Hence, by induction, any problem of NPSQ can be defined by a sentence of a vectorized Lindström logic $(\pm\Omega)^*[\mathrm{FO}]$ where $\Omega \in \mathcal{RE} \cap \mathcal{EXT}$.

Conversely, suppose, as our induction hypothesis, that for any formula $\varphi(\mathbf{x})$ of $\bigcup\{(\pm\Omega)^*[\mathrm{FO}] : \Omega \in \mathcal{RE} \cap \mathcal{EXT}\}$ of symbolic length less then m, there is a program scheme ρ of NPSQ, with free variables $\mathbf{x}$, accepting exactly the same class of structures (with the free variables considered as constant symbols) that φ defines. If the formula φ' of $\bigcup\{(\pm\Omega)^*[\mathrm{FO}] : \Omega \in \mathcal{RE} \cap \mathcal{EXT}\}$ is of the form: $\forall x_i \varphi$; $\exists x_i \varphi$; $\neg\varphi$; $\varphi_1 \wedge \varphi_2$; or $\varphi_1 \vee \varphi_2$, where the formulae φ_1 and φ_2 have symbolic length less than m, then there is a clearly a program scheme ρ' of NPSQ accepting the same class of structures that φ' defines (again, with any free variables regarded as constant symbols). The only interesting case is when φ' is formed from formulae of $\bigcup\{(\pm\Omega)^*[\mathrm{FO}] : \Omega \in \mathcal{RE} \cap \mathcal{EXT}\}$, of symbolic length less than m, by an application of an operator Ω_0, with the problem corresponding to Ω_0 being in $\mathcal{RE} \cap \mathcal{EXT}$.

Let φ' be over the signature σ' and let Ω_0 be over the signature σ_0. As $\Omega_0 \in \mathcal{RE} \cap \mathcal{EXT}$, there is a program scheme ρ_0 of NPSQ(1) accepting Ω_0. If $\mathcal{A}'$ is a σ'-structure then in order to ascertain whether $\mathcal{A} \models \varphi'$, we build a σ_0-structure $\Phi(\mathcal{A})$ (as we did when defining the semantics of a vectorized Lindström logic), with domain $|\mathcal{A}|^k$, and check whether $\Phi(\mathcal{A})$ is in Ω_0. We can amend the program scheme ρ_0 so that it works with k-tuples of elements, rather than single elements, and so that tests involving symbols of σ_0 are replaced with the appropriate sub-formulae of φ'. In this amended program scheme, assignments, for example, will be replaced k separate assignments, and a pop from the queue will be replaced with k separate pops. The result is a program scheme of NPSQ that takes a σ'-structure $\mathcal{A}$ as input and accepts if, and only if, $\mathcal{A} \models \varphi'$. The result follows (as the base case of the induction is trivial). $\square$

Essentially, Corollary 7 equates the 'syntactically-defined' class of problems NPSQ with the 'semantically-defined' class of problems definable using first-order constructs in tandem with 'recursively enumerable extension-closed' operators.

An immediate corollary of Theorem 7 is that NPSQ has a zero-one law.

Corollary 8 *The class of program schemes NPSQ has a zero-one law.*

Proof By [34] (a simple extension of a result in [13]), it was shown that any logic of the form $(\pm\Omega)^*[\mathrm{FO}]$, where $\Omega \in \mathcal{EXT}$, has a zero-one law. The corollary follows from Theorem 7. $\square$

Although Theorem 7 does not imply that the class of problems NPSQ can be realized as a vectorized Lindström logic $(\pm\Omega)^*[\mathrm{FO}]$, for some problem Ω (as can the classes of problems NPS, NPSS, NPSB and NPSA), it turns out that this is indeed the case. Let $\mathcal{C}$ be some class of problems and let Ω be some problem (as defined prior to the statement of Lemma 6). If $\Omega \in \mathcal{C}$ and any problem in $\mathcal{C}$ can be defined by a sentence of $(\pm\Omega)^*[\mathrm{FO}]$ of the form:

$$\Omega[\lambda \mathbf{x}^1\psi_1(\mathbf{x}^1), \mathbf{x}^2\psi_2(\mathbf{x}^2), \ldots, \mathbf{x}^r\psi_r(\mathbf{x}^r)](\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^c)$$

where each ψ_i is quantifier-free first-order and each $\mathbf{d}^j$ is a tuple of constant symbols, then we say that the problem Ω is *complete for $\mathcal{C}$ via quantifier-free first-order translations*.

Theorem 9 *There exists a problem Ω_q that is complete for NPSQ(1) via quantifier-free first-order translations. Consequently, NPSQ $= (\pm\Omega_q)^*[FO]$.*

Proof Let ρ be a program scheme of NPSQ(1), over some signature σ, which contains l instructions (we write the instructions of our program schemes sequentially, one per line, and we split while, if and if-the-else instructions so that the test is written on one line with the instructions within the while loop on the lines thereafter: the first instruction is the input instruction). W.l.o.g. we may assume that: ρ has no free variables; the input-output variables of ρ are $x_1, x_2, \ldots, x_k$, for some $k \geq 1$; any push onto the queue is done via the instruction '`push` x_1'; any pop from the queue is done via the instruction 'x_1 `:= pop`'; and the accept instruction appears once as the last instruction, *i.e.*, instruction number l.

Let $\mathcal{A}$ be a σ-structure of size $n \geq 2$. An element $\mathbf{u} = (u_0, u_1, \ldots, u_k)$ of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$ encodes a *partial ID* of ρ on input $\mathcal{A}$ via: a computation of ρ on input $\mathcal{A}$ is about to execute instruction u_0 and the variables $x_1, x_2, \ldots, x_k$ currently have the values $u_1, u_2, \ldots, u_k$, respectively. Henceforth, we identify partial IDs of ρ and the elements of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$. A *(full) ID* is a partial

ID augmented with a queue (that is, a list of elements of $|\mathcal{A}|$, possibly with repetitions).

We now build a digraph $G = (V, F)$ using ρ and $\mathcal{A}$. The vertex set V is $(\{1, 2, \ldots, l\} \times |\mathcal{A}|^k) \times (\{1, 2, \ldots, l\} \times |\mathcal{A}|^k)$ and there is an edge from $(\mathbf{u}, \mathbf{v})$ to $(\mathbf{u}', \mathbf{v}')$ if it is possible for ρ, on input $\mathcal{A}$, to move from partial ID $\mathbf{u}$ to partial ID $\mathbf{u}'$, and $\mathbf{v} = (1, u_1, u_1, \ldots, u_1)$ and $\mathbf{v}' = (1, u_1', u_1', \ldots, u_1')$ (the point of having the current value of the variable x_1 in a partial ID $\mathbf{u}$ duplicated in the second component of a vertex $(\mathbf{u}, \mathbf{v})$ will become apparent when we logically encode digraphs such as G). Of course, as to whether such a move can occur might depend upon the element currently at the head of the queue (in a computation of ρ on $\mathcal{A}$). Nevertheless, the edges of G reflect the possible moves of ρ on input $\mathcal{A}$. We now register possible dependence or influence on the queue by labelling some of the edges of F. If an edge of F corresponds to a push instruction then we label the edge with the symbol $+$, and if it corresponds to a pop instruction then we label it with the symbol $-$ (note that because all pushes and pops only involve x_1, we do not need to worry about detailing the variable involved in the push or pop). Given our labelled digraph G, we can now simulate exactly any computation of ρ on input $\mathcal{A}$ if we augment this digraph with a queue whose values correspond to elements of $|\mathcal{A}|$. All we have to remember is that: the traversal of a 'positive' edge $((\mathbf{u}, \mathbf{v}), (\mathbf{u}', \mathbf{v}'))$ of F pushes the second component $\mathbf{v}$ onto the queue, *i.e.*, the current value of the variable x_1; the traversal of a 'negative' edge $((\mathbf{u}, \mathbf{v}), (\mathbf{u}', \mathbf{v}'))$ of F is only possible if prior to the traversal of this edge, the value at the head of the queue is the value $\mathbf{v}'$, *i.e.*, the new value of the variable x_1; and after the traversal of a 'negative' edge of F, the head of the queue has to be thrown away.

It is trivial to see that $\mathcal{A} \models \rho$ if, and only if, there is a path (with the edge traversals obeying the above rules) from a *source vertex* of the form

$$((1, u_1, u_2, \ldots, u_k), (1, u_1, u_1, \ldots, u_1))$$

to a *sink vertex* of the form

$$((l, u_1', u_2', \ldots, u_k'), (1, u_1', u_1', \ldots, u_1')).$$

We can 'logically encode' our labelled digraph above over the signature $\sigma_q = \langle E, E^+, E^-, S, T\rangle$, where E, E^+ and E^- are relation symbols of arity 4 and S and T are binary relation symbols. The relation E details the edges without labels; the relation E^+ details the edges labelled $+$; the relation E^- details the edges labelled $-$; the relation relation S details the source vertices (of the form $((1, u_1, u_2, \ldots, u_k), (1, u_1, u_1, \ldots, u_1))$, above); and the relation details the sink vertices (of the form $((l, u_1', u_2', \ldots, u_k'), (1, u_1', u_1', \ldots, u_1'))$, above). (Note that an arbitrary σ_q-structure, when thought of as an edge-labelled digraph whose vertices are pairs of domain elements, might have multiple edges.

This is of no consequence.) Define the problem Ω_q as:

$$\begin{aligned}\Omega_q = \{\mathcal{G} \in \mathrm{STRUCT}(\sigma_q) \quad : \quad & \text{it is possible to traverse a path in the labelled} \\ & \text{digraph } \mathcal{G} \text{ from a vertex of } S \text{ to a vertex of } T\}\end{aligned}$$

(of course, by 'traverse a path' we mean according to the above rules, and starting with an empty queue).

When we consider the labelled digraph G, built above, as a σ_q-structure $\mathcal{G}$, it is straightforward to see that $\mathcal{G}$ can be described in terms of $\mathcal{A}$ by quantifier-free first-order formulae (although the encoding process may introduce additional isolated vertices which have no bearing on whether the resulting structure $\mathcal{G}$ is in Ω_q or not). Hence, as Ω_q can trivially be accepted by a program scheme of NPSQ(1), Ω_q is complete for NPSQ(1) via quantifier-free first-order translations. Proceeding as in the proof of Theorem 7 yields the result. $\Box$

Not only can we have the notion of a quantifier-free first-order translation but we can clearly also have the notion of a *quantifier-free first-order translation with a numeric universe*; that is, where the defining formulae (as detailed prior to Theorem 9) are interpreted over a structure $\mathcal{A}$, of size n, adjoined with a numeric universe $\{0, 1, \ldots, n-1\}$ and the binary relation add (for which $add(u, v)$ holds if, and only if, $v = u+1$). The definition of such a translation is made more complicated by having variables of both element type and numeric type, with the result that the domain of a structure $\Phi(\mathcal{A})$, as defined prior to Lemma 6 (when we defined the semantics of a vectorized Lindström logic), is of the form $|\mathcal{A}|^k \times \{0, 1, \ldots, n-1\}^m$; and so the size of such a structure is n^{k+m}. We do not explicitly define the notion of a quantifier-free first-order translation with a numeric universe as, given the above clarifying remark, it is analogous to that of a quantifier-free first-order translation. Theorem 2 and the proof of Theorem 9 immediately yield the following corollary.

Corollary 10 *The problem Ω_q is complete for $\mathcal{RE}$ = NPSQ$_+(1)$ via quantifier-free first-order translations with a numeric universe.* $\Box$

As far as we know, this is the first completeness result for the class of recursively enumerable problems, $\mathcal{RE}$, via very restricted logical reductions (of course, completeness via reductions such as quantifier-free first-order translations with a numeric universe implies completeness via more standard reductions such as log-space and polynomial-time reductions).

We can also define a hierarchy NPSQ$_+$, whose levels are NPSQ$_+(i)$, for $i \geq 1$, analogous to the hierarchy NPSQ but where we also have variables of numeric type. The only qualification we should add is that numeric variables can only appear as input-output variables in any program scheme of NPSQ$_+$ (and so, in particular, numeric values can not be 'passed across' to component program schemes via free variables). The question remains as to what exactly is the class of problems accepted by the program schemes of NPSQ_+. Whilst we can not give an exact answer to this question, we can provide a satisfactory answer to a related question concerning the program schemes of NPSQ when restricted to ordered structures.

We say that a problem Ω, over the signature σ, is defined by a logical sentence φ' *on ordered structures* (or, equivalently, *with a built-in successor relation*) if φ' is over the signature $\sigma' = \sigma \cup \langle succ, first, last\rangle$, where $succ$ is a binary relation symbol and $first$ and $last$ are constant symbols, and for every σ-structure $\mathcal{A}$:

- $\mathcal{A} \in \Omega$ if, and only if, every extension of $\mathcal{A}$ to a σ'-structure $\mathcal{A}'$ by a successor relation for which $first$ and $last$ are the least and greatest elements, respectively, satisfies φ'; and

- $\mathcal{A} \notin \Omega$ if, and only if, no extension of $\mathcal{A}$ to a σ'-structure $\mathcal{A}'$ by a successor relation for which $first$ and $last$ are the least and greatest elements, respectively, satisfies φ'.

Note that given some logic L, it might not be the case that every sentence of L is 'well-formed' on ordered structures (as implied by the above definition) as the satisfiability of a sentence of L in some appropriate structure might not be order-invariant, with respect to the 'built-in' successor relation (as the definition above demands). In fact, it is undecidable as to whether a first-order sentence is well-formed on ordered structures; that is, whether first-order logic on ordered structures has a recursive syntax (we say a bit more about this difficulty in the next section but the reader is referred to [30] for a full discussion). We denote the fact that we are considering some vectorized Lindström logic $(\pm\Omega)^*[\mathrm{FO}]$ on ordered structures by writing $(\pm\Omega)^*[\mathrm{FO}_s]$.

It should be clear that not just logical sentences can define problems on ordered structures but, in exactly the same way, program schemes can too. We define the hierarchy NPSQ_s, whose levels are $\mathrm{NPSQ}_s(i)$, for $i \geq 1$, analogously to the hierarchy NPSQ but where every program scheme defines a problem on ordered structures (note that we now face the problem of whether a program scheme of NPSQ over some signature $\sigma \cup \langle succ, first, last\rangle$ is well-formed). In the same way, we obtain the notion of a *quantifier-free first-order translation with a built-in successor relation.*

Proposition 11 *$\mathcal{RE} = NPSQ_+(1) = NPSQ_+(2) = NPSQ_s(1) = NPSQ_s(2)$. For $i \geq 3$, $NPSQ_+(i) \subseteq NPSQ_s(i)$; and so $NPSQ_+ \subseteq NPSQ_s$.*

Proof Let the problem Ω be such that it is accepted by the program scheme ρ in $\mathrm{NPSQ}_+(1)$. We simulate ρ with a program scheme ρ' of $\mathrm{NPSQ}_s(1)$ as follows. In ρ', use the built-in successor relation to simulate the numeric values of a computation in ρ. That is, we simulate ρ directly except that we model a numeric variable, t, say, of ρ having some value i as a new variable t', in ρ', having the value of the $(i+1)$th element of the linear order described by the built-in successor relation. We can 'increment' and 'decrement' the value of t' by setting it as the preceding or succeeding element of this linear order, respectively. The resulting program scheme ρ' is clearly well-formed and accepts Ω.

Conversely, let Ω be the problem accepted by some program scheme ρ in NPSQ$_s(1)$. We simulate ρ with a program scheme ρ' of NPSQ$_+(1)$ as follows. We begin by guessing, and checking (using our numeric variables), a successor relation which we store on the queue. We then use this successor relation to directly simulate a computation of ρ (of course, we manipulate the queue in order to gain access to our successor relation). The resulting program scheme ρ' accepts Ω. Given that, by Theorem 2, $\mathcal{RE} =$ NPSQ$_+(1) =$ NPSQ$_s(1)$, it is trivial that $\mathcal{RE} =$ NPSQ$_+(2) =$ NPSQ$_s(2)$.

Let Ω be the problem accepted by some program scheme ρ in NPSQ$_+(i)$, for some $i \geq 3$. A construction analogous to that above enables us to construct a suitable program scheme of NPSQ$_s(i)$ accepting Ω. $\square$

It is worth pointing out why we have been unable to extend the proof, above, that NPSQ$_+(1) \subseteq$ NPSQ$_s(1)$ to a proof that NPSQ$_s(i) \subseteq$ NPSQ$_+(i)$, for $i \geq 3$. Take $i = 3$, for example. Any computation of a program scheme ρ in NPSQ$_s(3)$ will, in general, involve a number of tests which are program schemes of NPSQ$_s(2)$. The built-in successor relation might be used in all of these tests; and note that it is *the same* successor relation that is used each time a test is evaluated. If we were to pursue the construction in the proof of Proposition 11 then we would, essentially, build our own successor relation every time we evaluated a test; and these successor relations could well be *different* (remember that our semantics are such that we can not pass across the contents of queues to program schemes involved in tests). At present, we do not know whether NPSQ$_s(i) \subseteq$ NPSQ$_+(i)$, for $i \geq 3$; and whether NPSQ$_s \subseteq$ NPSQ$_+$.

Proposition 11 and the proof of Theorem 9 immediately yield the following corollary.

Corollary 12 *The problem Ω_q is complete for $\mathcal{RE}$ = NPSQ$_s(1)$ via quantifier-free first-order translations with a built-in successor relation (or, equivalently, on ordered structures). Consequently, NPSQ$_s = (\pm\Omega_q)^*[FO_s]$.* $\square$

Unlike the situation for NPSQ$_+$, we can give an alternative characterization of the class of problems NPSQ$_s$. The class of problems accepted by log-space deterministic oracle Turing machines with access to an oracle from $\mathcal{RE}$ is denoted by $\mathbf{L}^{\mathcal{RE}}$.

Theorem 13 $\mathbf{L}^{\mathcal{RE}} = NPSQ_s(3) = NPSQ_s = (\pm\Omega_q)^*[FO_s]$. *Moreover, every problem in* $\mathbf{L}^{\mathcal{RE}}$ *can be defined by a sentence of the logic* $(\pm\Omega_q)^*[FO_s]$ *of the form:*

$$\begin{aligned}&\exists y_1\exists y_2\ldots\exists y_k(\Omega_q[\lambda\mathbf{x}\varphi(\mathbf{x},\mathbf{y}),\mathbf{x}\varphi_+(\mathbf{x},\mathbf{y}),\mathbf{x}\varphi_-(\mathbf{x},\mathbf{y}),\mathbf{z}\psi_S(\mathbf{z},\mathbf{y}),\mathbf{z}\psi_T(\mathbf{z},\mathbf{y})]\\&\qquad\wedge\neg\Omega_q[\lambda\mathbf{x}'\varphi'(\mathbf{x}',\mathbf{y}),\mathbf{x}'\varphi'_+(\mathbf{x}',\mathbf{y}),\mathbf{x}'\varphi'_-(\mathbf{x}',\mathbf{y}),\mathbf{z}'\psi'_S(\mathbf{z}',\mathbf{y}),\mathbf{z}'\psi'_T(\mathbf{z}',\mathbf{y})])\end{aligned}$$

where φ, φ_-, φ_+, ψ_S, ψ_T, φ', φ'_-, φ'_+, ψ'_S *and* ψ'_T *are quantifier-free first-order with a built-in successor relation.*

Proof We use results, due to Gottlob in [19], concerning the logical capture of classes of problems accepted by log-space deterministic oracle Turing machines. Let **CC** be a complexity class that is closed under **NP**-reductions, conjunctions, disjunctions and marked unions (see [19] for more details as regards these definitions); and let Ω be a problem such that Ω is complete for **CC** via quantifier-free first-order translations with a built-in successor relation. Gottlob proved in Theorem 5.2 of [19] (with reference to Remark 9 of Section 6 of [19]) that if this is the case then any problem in **CC** can be defined by a sentence of the logic $(\pm\Omega)^*[\mathrm{FO}_s]$ (of a particular normal form); and in Theorem 4.6 and Corollary 3.3 of [19] (again, with reference to Remark 9 of Section 6 of [19]) that the class of problems defined by the sentences of $(\pm\Omega)^*[\mathrm{FO}_s]$ is $\mathbf{L}^{\mathbf{CC}}$. The class of problems $\mathcal{RE}$ satisfies the above hypotheses and, by Corollary 10, Ω_q is complete for $\mathcal{RE}$ via quantifier-free first-order translations with a built-in successor relation. Hence, by Corollary 12 and [19], any problem in $\mathrm{NPSQ}_s = (\pm\Omega_q)^*[\mathrm{FO}_s]$ can be defined by a sentence, of $(\pm\Omega_q)^*[\mathrm{FO}_s]$, of the form:

$$\begin{aligned}&\exists y_1\exists y_2\ldots\exists y_k(\Omega_q[\lambda\mathbf{x}\varphi(\mathbf{x},\mathbf{y}),\mathbf{x}\varphi_+(\mathbf{x},\mathbf{y}),\mathbf{x}\varphi_-(\mathbf{x},\mathbf{y}),\mathbf{z}\psi_S(\mathbf{z},\mathbf{y}),\mathbf{z}\psi_T(\mathbf{z},\mathbf{y})]\\&\qquad\wedge\neg\Omega_q[\lambda\mathbf{x}'\varphi'(\mathbf{x}',\mathbf{y}),\mathbf{x}'\varphi'_+(\mathbf{x}',\mathbf{y}),\mathbf{x}'\varphi'_-(\mathbf{x}',\mathbf{y}),\mathbf{z}'\psi'_S(\mathbf{z}',\mathbf{y}),\mathbf{z}'\psi'_T(\mathbf{z}',\mathbf{y})])\end{aligned}$$

where φ, φ_-, φ_+, ψ_S, ψ_T, φ', φ'_-, φ'_+, ψ'_S and ψ'_T are quantifier-free first-order formulae; and the class of problems defined by the sentences of NPSQ_s is $\mathbf{L}^{\mathcal{RE}}$. The result follows. $\square$

Note that $\mathrm{NPSQ}_s(2) \subset \mathrm{NPSQ}_s(3)$ as $\mathbf{L}^{\mathcal{RE}}$ is closed under complementation whereas $\mathcal{RE}$ is not.

6 A simple application

It is a question of great importance in finite model theory and database theory as to whether there is a logic which captures exactly the class of polynomial-time solvable problems. Of course, one has to be precise about what one means by a logic and Gurevich [21] has formulated a definition which has been widely adopted. A *logic* L is given by a pair of functions (Sen, Sat) satisfying the following conditions. The function Sen associates with every signature σ a recursive set $Sen(\sigma)$ whose elements are called *L-sentences over σ*. The function Sat associates with every signature a recursive relation $Sat_\sigma(\mathcal{A}, \varphi)$, where $\mathcal{A}$ is a σ-structure and φ is a sentence of L. We say that $\mathcal{A}$ *satisfies* φ (and write $\mathcal{A} \models \varphi$) if $Sat_\sigma(\mathcal{A}, \varphi)$ holds. Furthermore, we require that $Sat_\sigma(\mathcal{A}, \varphi)$ if, and only if, $Sat_\sigma(\mathcal{B}, \varphi)$ when $\mathcal{A}$ and $\mathcal{B}$ are isomorphic σ-structures.

Whilst many complexity classes containing **NP** (including **NP** itself) can be captured by logics (on the class of all finite structures) as yet no-one has exhibited a logic (in Gurevich's sense) to capture any mainstream complexity class (for which the expectation is that it is properly) contained in **NP**. Logical characterizations of complexity classes on the class of ordered finite structures usually suffer from the difficulty that deciding whether a formula of the logic is order-invariant is undecidable. The reader is referred to, for example, [14] and [21] for a further discussion on capturing complexity classes by logics.

It is straightforward to verify that the classes of program schemes NPSQ (together with the classes of the hierarchy therein) and $\text{NPSQ}_+(1)$ are logics in the sense of Gurevich; and so, for example, Theorem 3 (resp. Theorem 2) implies that $\mathcal{RE} \cap \mathcal{EXT}$ (resp. $\mathcal{RE}$) can be logically captured. However, the characterization $\mathcal{RE} = \text{NPSQ}_s(1)$ is not a logical characterization of $\mathcal{RE}$ (as any first-order sentence can be effectively transformed into a program scheme of NPSQ(1)).

By imposing resource conditions upon our program schemes, we can logically capture other classes of problems. We begin by making explicit the resources used by a Turing machine to accept a problem. A problem Ω, over some signature σ, is accepted by a non-deterministic Turing machine M *in time* $f(x)$ (resp. *using space* $g(x)$) if M accepts Ω and for every encoding $e_\sigma(\mathcal{A})$ of any σ-structure $\mathcal{A}$ of size n in Ω, there is an accepting computation of M on input $e_\sigma(\mathcal{A})$ taking time at most $f(e_\sigma(n))$ (resp. using space at most $g(e_\sigma(n))$). In particular, the complexity of M on input $e_\sigma(\mathcal{A})$ is measured in terms of $e_\sigma(n)$ and not n, although the two measures are polynomially related (recall that σ contains at least one relation symbol).

In contrast, the time and space complexities of a program scheme of $\text{NPSQ}_+(1)$ or NPSQ(1) on some input structure are measured in terms of the size of the input structure. In order to evaluate the time taken or the space used in some computation of a program scheme of $\text{NPSQ}_+(1)$ or NPSQ(1) on some input structure of size n, we treat the execution of every assignment, quantifier-free test, pop, push, *etc.*, as taking one unit of time, and we regard a variable or a place on the queue as occupying one unit of space. As for a non-deterministic Turing machine, the complexity of a program scheme of $\text{NPSQ}_+(1)$ or NPSQ(1) is derived using accepting computations only.

For any function $f(x)$, let $\mathcal{POLY}(f(x))$ denote

$$\begin{aligned}\{p_0(x).(f(q_0(x)))^k + p_1(x).(f(q_1(x)))^{k-1} + \ldots \\ \ldots + p_{k-1}(x).f(q_{k-1}(x)) + p_k(x) \\ : \ k \geq 0, p_0, p_1, \ldots, p_k, q_0, q_1, \ldots, q_{k-1} \text{ are polynomials with} \\ \text{integer coefficients}\}.\end{aligned}$$

If $\mathcal{C}$ is a class of functions then $\mathcal{POLY}(\mathcal{C}) = \bigcup_{f(x)\in\mathcal{C}} \mathcal{POLY}(f(x))$.

For example, if $f(x)$ is the identity function then $\mathcal{POLY}(f(x))$ is the class of polynomials in x (with integer coefficients); if $f(x) = 2^x$ then $\mathcal{POLY}(f(x))$ is the class of polynomial combinations of exponentials of the form $2^{p(x)}$, where $p(x)$ is some polynomial; and if $f(x) = 2^{2^x}$ then $\mathcal{POLY}(f(x))$ is the class of polynomial combinations of exponentials of the form $2^{2^{p(x)}}$, where $p(x)$ is some polynomial. Many classes of functions of interest in complexity theory can be realized as $\mathcal{POLY}(f(x))$, for some function $f(x)$.

Let Ω be some problem over the signature σ. The proof of Theorem 2 yields the following.

Corollary 14 *(i) If Ω can be accepted by a non-deterministic Turing machine in time $f(x)$ (resp. using space $g(x)$) then there exists a program scheme ρ of NPSQ$_+(1)$ accepting Ω such that every σ-structure in Ω of size n is accepted in time $F(n)$ (resp. using space $G(n)$), for some $F(x)$ in $\mathcal{POLY}(f(x))$ (resp. $G(x)$ in $\mathcal{POLY}(g(x))$).*

(ii) If Ω can be accepted by a program scheme of NPSQ$_+(1)$ in time $f(x)$ (resp. using space $g(x)$) then there is a non-deterministic Turing machine M accepting Ω such that every encoding of a σ-structure in Ω of size n is accepted in time $F(e_\sigma(n))$ (resp. using space $G(e_\sigma(n))$), for some $F(x)$ in $\mathcal{POLY}(f(x))$ (resp. $G(x)$ in $\mathcal{POLY}(g(x))$). □

The proof of Theorem 3 yields the following.

Corollary 15 *(i) If Ω can be accepted by a non-deterministic Turing machine in time $f(x)$ (resp. using space $g(x)$) and Ω is in $\mathcal{EXT}$ then there exists a program scheme ρ of NPSQ(1) accepting Ω such that every σ-structure in Ω of size n is accepted in time $F(n)$ (resp. using space $G(n)$), for some $F(x)$ in $\mathcal{POLY}(f(x))$ (resp. $G(x)$ in $\mathcal{POLY}(g(x))$).*

(ii) If Ω can be accepted by a program scheme of NPSQ(1) in time $f(x)$ (resp. using space $g(x)$) then Ω is in $\mathcal{EXT}$ and there is a non-deterministic Turing machine M accepting Ω such that every encoding of a σ-structure in Ω of size n is accepted in time $F(e_\sigma(n))$ (resp. using space $G(e_\sigma(n))$), for some $F(x)$ in $\mathcal{POLY}(f(x))$ (resp. $G(x)$ in $\mathcal{POLY}(g(x))$). □

Denote the class of non-deterministic Turing machines that accept problems in time (resp. using space) $f(x)$ where $f(x)$ is a function from the class of functions $\mathcal{C}$ as $\mathrm{NTM}^t(\mathcal{C})$ (resp. $\mathrm{NTM}^s(\mathcal{C})$), with $\mathrm{NPSQ}^t(1)(\mathcal{C})$ and $\mathrm{NPSQ}^t_+(1)(\mathcal{C})$ (resp. $\mathrm{NPSQ}^s(1)(\mathcal{C})$ and $\mathrm{NPSQ}^s_+(1)(\mathcal{C})$) defined likewise (of course, these denotations also refer to the classes of problems accepted by such Turing machines or program schemes). The following is immediate from above.

Corollary 16

$$NTM^t(\mathcal{POLY}(\mathcal{C})) = NPSQ^t_+(1)(\mathcal{POLY}(\mathcal{C})),$$

$$NTM^t(\mathcal{POLY}(\mathcal{C})) \cap \mathcal{EXT} = NPSQ^t(1)(\mathcal{POLY}(\mathcal{C})),$$

$$NTM^s(\mathcal{POLY}(\mathcal{C})) = NPSQ^s_+(1)(\mathcal{POLY}(\mathcal{C})),$$

and

$$NTM^s(\mathcal{POLY}(\mathcal{C})) \cap \mathcal{EXT} = NPSQ^s(1)(\mathcal{POLY}(\mathcal{C})),$$

for any class of functions $\mathcal{C}$. □

We can regard $\mathrm{NPSQ}^t(1)(\mathcal{POLY}(\mathcal{C}))$, for example, as a logic (in Gurevich's sense) by identifying program schemes with pairs $(\rho, F(x))$, where ρ is a program scheme of NPSQ(1) and $F(x)$ is a function from $\mathcal{POLY}(\mathcal{C})$. Of course, this requires that the functions of $\mathcal{C}$ form a recursive set and that there is a Turing machine which when given a function $f(x)$ from $\mathcal{C}$ and a positive integer n, computes the value $f(n)$. Henceforth, we assume that this is the case for any of our classes of functions $\mathcal{C}$. Thus, Corollary 16 can be used to logically capture a whole host of complexity classes including **NP**, **PSPACE**, **NEXPTIME**, **EXPSPACE**, **2-NEXPTIME**, **2-EXPSPACE**, ..., **ELEMENTARY** (note that **k-NEXPSPACE** = **k-EXPSPACE**, for all $k \geq 1$). However, Corollary 16 can also be used to logically characterize 'fragments' of (the above) complexity classes obtained by intersecting the complexity class with the class of problems $\mathcal{EXT}$. Note that this is a 'semantic' rather than a 'syntactic' restriction and so it is not immediately obvious as to whether such a fragment of a complexity class can be logically captured. Furthermore, existing complexity-theoretic inequalities or hypotheses can be used to derive analogous inequalities or hypotheses for the respective fragments of the complexity classes in question obtained by intersecting with the class of problems $\mathcal{EXT}$.

Corollary 17 *Let $\mathcal{C}$ and $\mathcal{D}$ be classes of functions and let $\epsilon, \delta \in \{t, s\}$. The following are equivalent.*

(*i*) *$NPSQ^\epsilon_+(1)(\mathcal{POLY}(\mathcal{C})) = NPSQ^\delta_+(1)(\mathcal{POLY}(\mathcal{D}))$.*

(ii) $NTM^{\epsilon}(\mathcal{POLY}(\mathcal{C})) = NTM^{\delta}(\mathcal{POLY}(\mathcal{D}))$.

(iii) $NTM^{\epsilon}(\mathcal{POLY}(\mathcal{C})) \cap \mathcal{EXT} = NTM^{\delta}(\mathcal{POLY}(\mathcal{D})) \cap \mathcal{EXT}$.

(iv) $NPSQ^{\epsilon}(1)(\mathcal{POLY}(\mathcal{C})) = NPSQ^{\delta}(1)(\mathcal{POLY}(\mathcal{D}))$.

Proof Implications $(i) \Rightarrow (ii)$, $(ii) \Rightarrow (iii)$ and $(iii) \Rightarrow (iv)$ follow immediately from Corollary 16. As for implication $(iv) \Rightarrow (i)$, let Ω be some problem in $\mathrm{NPSQ}^{\epsilon}_{+}(1)(\mathcal{POLY}(\mathcal{C}))$, over the signature σ, where the witnessing program scheme is ρ. Let σ' be the signature σ augmented with a binary relation symbol L and two constant symbols C and D (we assume that these symbols do not already appear in σ). We shall now construct a program scheme ρ' of $\mathrm{NPSQ}^{\epsilon}(1)(\mathcal{POLY}(\mathcal{C}))$ which takes σ'-structures as input.

Let $\mathcal{A}'$ be a σ'-structure of size n and let $\mathcal{A}$ be the σ-structure obtained from $\mathcal{A}'$ by restricting to the symbols of σ. On input $\mathcal{A}'$, the program scheme ρ' interprets the relation $L^{\mathcal{A}'}$ as a linear order with minimum element $C^{\mathcal{A}'}$ and maximum element $D^{\mathcal{A}'}$, and writes this linear order onto the queue. Of course, the triple $(L^{\mathcal{A}'}, C^{\mathcal{A}'}, D^{\mathcal{A}'})$ might not encode a linear order at all. However, this does not concern us. All we are interested in is that an ordering is written onto the queue, with minimal element $C^{\mathcal{A}'}$, which may or may not be a linear ordering of the elements of $|\mathcal{A}'|$ with maximum element $D^{\mathcal{A}'}$. Of course, if, in the construction of this ordering, an ordering with maximum element $D^{\mathcal{A}'}$ can not be 'decoded' from $(L^{\mathcal{A}'}, C^{\mathcal{A}'}, D^{\mathcal{A}'})$ and written on the queue then this particular computation halts and rejects. In the case that $(L^{\mathcal{A}'}, C^{\mathcal{A}'}, D^{\mathcal{A}'})$ encodes a linear order, we insist that this linear order is duplicated on the queue. Next, the program scheme ρ' uses this ordering to simulate the computation of the program scheme ρ on the input $\mathcal{A}$ restricted to the domain of the linear ordering (as in the proof of Theorem 3). The resulting program scheme ρ' is a program scheme of $\mathrm{NPSQ}^{\epsilon}(1)(\mathcal{POLY}(\mathcal{C}))$. Consequently, $\mathcal{A} \models \rho$ if, and only if, there exists a σ'-structure $\mathcal{A}'$ that is an extension of $\mathcal{A}$ with a linear order (L, C, D) of $|\mathcal{A}|$ such that $\mathcal{A}' \models \rho'$.

By hypothesis, there is a program scheme ρ'_0 of $\mathrm{NPSQ}^{\delta}(1)(\mathcal{POLY}(\mathcal{D}))$ accepting exactly the same problem as ρ'. Consider the following program scheme ρ_0 of $\mathrm{NPSQ}^{\delta}_{+}(1)(\mathcal{POLY}(\mathcal{D}))$. On input a σ-structure $\mathcal{A}$, the program scheme ρ_0 begins by guessing a linear order and writing this linear order onto its queue (of course, it verifies that a guessed linear order is indeed a linear order). Then the program scheme ρ_0 simulates the computation of ρ'_0 on the σ'-structure $\mathcal{A}'$ obtained from $\mathcal{A}$ by augmenting it with the guessed (and verified) linear order. Consequently, $\mathcal{A} \models \rho_0$ if, and only if, there exists a σ'-structure $\mathcal{A}'$ that is an extension of $\mathcal{A}$ with a linear order (L, C, D) of $|\mathcal{A}|$ such that $\mathcal{A}' \models \rho'_0$. Thus, the result follows. $\square$

Let us end this section by showing how we apply Corollary 17. It is well-known that, for example, **NEXPTIME** $\neq$ **2-NEXPTIME**. Corollary 17 implies that **NEXPTIME** $\cap\, \mathcal{EXT} \neq$ **2-NEXPTIME** $\cap\, \mathcal{EXT}$ and that the class of problems accepted by program schemes of NPSQ(1) restricted to run in time $2^{2^{p(x)}}$, for some polynomial $p(x)$, properly contains the class of problems accepted by program schemes of NPSQ(1) restricted to run in time $2^{p(x)}$, for some polynomial $p(x)$. Also, it is open as to whether, for example, **NEXPTIME** = **EXPSPACE**. By Corollary 17, this question is equivalent to whether **NEXPTIME**$\cap\mathcal{EXT}$ = **EXPSPACE**$\cap\mathcal{EXT}$ and to whether the class of problems accepted by program schemes of NPSQ(1) restricted to run in time $2^{p(x)}$, for some polynomial $p(x)$, is the same as the class of problems accepted by program schemes of NPSQ(1) restricted to run using space $2^{p(x)}$, for some polynomial $p(x)$.

We feel that it is interesting that open complexity-theoretic questions are equivalent to analogous logical questions involving classes of problems with zero-one laws, and even closed under extensions. Of course, these latter classes of problems do not contain some trivial complexity-theoretic problems like PARITY; yet still they have a definitive role to play in the question of whether two complexity classes, such as **NEXPTIME** and **EXPSPACE** which have significant complexity-theoretic capacities, are identical. Moreover, a hypothesis such as **NEXPTIME**$\cap\mathcal{EXT}$ = **EXPSPACE**$\cap\mathcal{EXT}$, and its logical (program-schematic) realization, rather than the more expansive hypothesis **NEXPTIME** = **EXPSPACE**, might prove more amenable to attack.

7 Conclusions

There are numerous directions in which to continue the research of this paper. Perhaps the most obvious is a consideration of whether the hierarchy within NPSQ is proper. It was proven in [5] that the analogous hierarchies in NPS and NPSS are indeed proper although no such results are currently available for NPSB and NPSA. For the latter classes, all that is known is that NPSB(1) $\subset$ NPSB(2) $\subset$ NPSB(3) [36] (with an analogous statement for NPSA). In fact, the proof in [36] also yields that NPSQ(1) $\subset$ NPSQ(2) $\subset$ NPSQ(3). It may just be coincidence but NPS and NPSS can be realized as fragments of $\mathcal{L}^{\omega}_{\infty\omega}$ whereas NPSB, NPSA and NPSQ can not.

Given the pivotal role of inductive fixed-point logic in descriptive complexity theory, we would like to know whether there are problems definable by a sentence of inductive fixed-point logic which are not in NPSQ. We conjecture that there are such problems.

Whilst NPSQ is an extensive class of problems, there are recursively solvable problems which have a zero-one law but which are not in NPSQ. One such is the problem HC consisting of those digraphs, *i.e.*, finite structures over a signature consisting of one binary relation symbol, which have a Hamiltonian cycle. It has long been known that HC has a zero-one law [7]. However, if the problem HC is in NPSQ then the class of problems $(\pm\mathrm{HC})^*[\mathrm{FO}]$ is contained in NPSQ; which yields a contradiction by either [13], where it was shown that the logic $(\pm\mathrm{HC})^*[\mathrm{FO}]$ does not have a zero-one law, or by [10], where it was essentially shown that any problem in **NP** can be defined in $(\pm\mathrm{HC})^*[\mathrm{FO}]$. The problems HC and RIGIDITY, *i.e.*, the problem consisting of all those graphs with no non-trivial automorphisms, played a central role in [13]. It was proven there that whilst $(\pm\mathrm{HC})^*[\mathrm{FO}]$ does not have a zero-one law, $(\pm\mathrm{RIGIDITY})^*[\mathrm{FO}]$ does. It would be interesting to know whether the problem RIGIDITY is in NPSQ.

The questions as to the class of problems captured by the program schemes of NPSQ_+ and whether the hierarchy within NPSQ_+ is proper are interesting; especially given that $\mathbf{L}^{\mathcal{RE}} = \mathrm{NPSQ}_s$ and that the hierarchy within NPSQ_s collapses to the third level.

Finally, we remark that the results that $\mathrm{NPSQ}_+(1)$ consists of the class of problems $\mathcal{RE}$ and NPSQ(1) consists of the class of problems $\mathcal{RE} \cap \mathcal{EXT}$ can be regarded as a sort of preservation theorem, in the model-theoretic sense. Preservation theorems in finite model theory have hitherto revolved around first-order logic (see [4]) and we feel that the existence of such theorems should be considered for relevant extensions of first-order logic.

References

[1] S. Abiteboul, M.Y. Vardi and V. Vianu, Fixpoint logics, relational machines and computational complexity, *Journal of the Association for Computing Machinery* **44** (1997) 30–56.

[2] S. Abiteboul and V. Vianu, Generic computation and its complexity, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, ACM Press (1991) 209–219.

[3] S. Abiteboul and V. Vianu, Computing with first-order logic, *Journal of Computer and System Sciences* **50** (1995) 309–335.

[4] N. Alechina and Y. Gurevich, Syntax vs semantics on finite structures, *Lecture Notes in Computer Science Volume* 1261, Springer-Verlag (1997) 14–33.

[5] A.A. Arratia-Quesada, S.R. Chauhan and I.A. Stewart, Hierarchies in classes of program schemes, *Journal of Logic and Computation* **9** (1999) 915–957.

[6] A. Blass, Y. Gurevich and S. Shelah, Choiceless polynomial time, *Annals of Pure and Applied Logic* **100** (1999) 141–187.

[7] B. Bollobás, *Random Graphs*, Academic Press (1985).

[8] S. Brown, D. Gries and T. Szymanski, Universality of data retrieval languages, *Proceedings of* 6*th Annual ACM Symposium on Principles of Programming Languages*, ACM Press (1979) 110–117.

[9] R. Constable and D. Gries, On classes of program schemata, *SIAM Journal of Computing* **1** (1972) 66–118.

[10] E. Dahlhaus, Reduction to NP-complete problems by interpretations, *Lecture Notes in Computer Science Volume* 171, Springer-Verlag (1984) 357–365.

[11] A. Dawar, Generalized quantifiers and logical reducibilities, *Journal of Logic and Computation* **5** (1995) 213–226.

[12] A. Dawar, A restricted second-order logic for finite structures, *Information and Computation* **143** (1998) 154–174.

[13] A. Dawar and E. Grädel, Generalized quantifiers and 0-1 laws, *Proceedings of the* 10*th IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press (1995) 54–64.

[14] H.D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer-Verlag (1995).

[15] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: *Complexity of Computation* (ed. R.M. Karp), Volume 7 of SIAM-AMS Proceedings (1974) 43–73.

[16] H. Friedman, Algorithmic procedures, generalized Turing algorithms and elemenatry recursion theory, in: *Logic Colloquium* 1969 (ed. R.O. Gandy and C.M.E. Yates), North-Holland (1971) 361–390.

[17] M.R. Garey, A catalog of complexity classes, in: *Handbook of Theoretical Computer Science Volume A* (ed. J. van Leeuwen), Elsevier (1990) 67–161.

[18] R.L. Gault and I.A. Stewart, An infinite hierarchy in a class of polynomial-time program schemes, *submitted for publication*.

[19] G. Gottlob, Relativized logspace and generalized quantifiers over finite ordered structures, *Journal of Symbolic Logic* **62** (1997) 545–574.

[20] M. Grohe, Existential least fixed-point logic and its relatives, *Journal of Logic and Computation* **7** (1997) 205–228.

[21] Y. Gurevich, Logic and the challenge of computer science, in: *Current Trends in Theoretical Computer Science* (ed. E. Börger), Computer Science Press (1988) 1–57.

[22] Y. Gurevich, Evolving algebras 1993: Lipari guide, in: *Specification and Validation* (ed. E. Börger), Oxford University Press (1995) 9–36.

[23] D. Harel and D. Peleg, On static logics, dynamic logics, and complexity classes, *Information and Control* **60** (1984) 86–102.

[24] N. Immerman, Relational queries computable in polynonial time, *Information and Control* **68** (1986) 86–104.

[25] N. Immerman, Languages that capture complexity classes, *SIAM Journal of Computing* **16** (1987) 760–778.

[26] N. Immerman, *Descriptive Complexity*, Springer-Verlag (1998).

[27] N.D. Jones and S.S. Muchnik, Even simple programs are hard to analyse, *Journal of the Association for Computing Machinery* **24** (1972) 338–350.

[28] C. Lautemann, T. Schwentick and I.A. Stewart, Positive versions of polynomial time, *Information and Computation* **147** (1998) 145–170.

[29] D. Leivant, Descriptive characterizations of computational complexity, *Journal of Computer and System Sciences* **39** (1989) 51–83.

[30] M. Otto, *Bounded Variable Logics and Counting*, *Lecture Notes in Logic Volume* 9, Springer-Verlag (1997).

[31] M. Paterson and N. Hewitt, Comparative schematology, *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM Press (1970) 119–128.

[32] I.A. Stewart, Logical and schematic characterization of complexity classes, *Acta Informatica* **30** (1993) 61–87.

[33] I.A. Stewart, Logical description of monotone NP problems, *Journal of Logic and Computation* **4** (1994) 337–357.

[34] I.A. Stewart, Program schemes, arrays, Lindström quantifiers and zero-one laws, *Lecture Notes in Computer Science Volume* 1683, Springer-Verlag (1999) 374–388 (full version accepted for publication in *Theoretical Computer Science*).

[35] I.A. Stewart, Using program schemes to logically capture polynomial-time on certain classes of structures, *submitted for publication.*

[36] I.A. Stewart, Program schemes with binary write-once arrays and the complexity classes they capture, *submitted for publication.*

[37] J. Tiuryn and P. Urzyczyn, Some relationships between logics of programs and complexity theory, *Theoretical Computer Science* **60** (1988) 83–108.

[38] M. Vardi, Complexity of relational query languages, *Proceedings of* 14*th Annual ACM Symposium on Theory of Computing*, ACM Press (1982) 137–146.